UNIVERSIDADE D
COIMBRA

Filipe Guerreiro Assunção

Neuro-Evolution: Representations, Efficiency and Development

Doctoral thesis submitted in partial fulfillment of the Doctoral Program in Information Science and Technology supervised by Professor Fernando Jorge Penousal Martins Machado, co-supervised by Professor Bernardete Martins Ribeiro, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

December 2020

Financial support by Fundação para a Ciência e Tecnologia,
SFRH/BD/114865/2016.
Neuro-Evolution: Representations, Efficiency and Development

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

Abstract

Artificial Neural Networks (ANNs) are hard to design and optimise. First, we preprocess the dataset and design and extract the features. Second, we define the topology of the network, i.e., select and parameterise the type, sequence, and connectivity of the layers. Third, we train the network, which implies selecting the learning algorithm and setting its parameters. This iterative process is time-consuming, and each choice affects the others. Moreover, to address a new task, the model must be at least reviewed. In the worst case scenario, we may have to re-start from scratch.

The literature on approaches that aim at automating the generation of ANNs is vast. The focus of this Thesis is on NeuroEvolution (NE) – the set of methods that apply Evolutionary Computation to optimise ANNs. Based on the review of the state-of-the-art, it is our perception that the vast majority of NE approaches focus on the optimisation of a particular ANN structure or the generation of solutions to a specific problem. We hypothesise that using Grammar-based Genetic Programming approaches, we can propose and develop a flexible representation to design a general-purpose framework to optimise the deployment of Deep Artificial Neural Networks (DANNs). Additionally, this representation will support time-efficient search and the use of limited computational resources.

After examining the performance of Grammatical Evolution and Structured Grammatical Evolution, we conclude that these methods are not appropriate to optimise multi-layered networks. Therefore, we propose Dynamic Structured Grammatical Evolution (DSGE). DSGE is the first contribution of this Thesis and is the core of Deep Evolutionary Network Structured Representation (DENSER). DENSER is a novel grammar-based NE representation, which optimises DANNs using a two-level structure. This representation can encode all aspects of ANNs: from the layer types and sequencing to the learning strategy. It can also encode any other hyper-parameter, e.g., the data augmentation strategy.

One of the main limitations of the evolutionary engine used with DENSER is the time needed to generate effective ANNs. Fast Deep Evolutionary Network Structured Representation (Fast-DENSER) uses the representation of DENSER, but to speed up search it replaces the evolutionary engine by an Evolutionary Strategy. Besides, Fast-DENSER introduces a methodology to increase the training time of each individual as generations proceed. As the individuals are enabled to train for longer, by the end of evolution, the networks are fully-trained and ready to be deployed, i.e., after the end of evolution, no further training is required.

Finally, we propose incremental development to avoid restarting search from scratch each time we have to solve a new problem. Incremental development facilitates the search in target tasks by warm-starting evolution. Moreover, it allows knowledge from previously addressed source tasks to be incorporated throughout evolution. In other words, incremental development aids evolution by transferring topological information from source to target tasks.

Resumo

A otimização de Redes Neuronais Artificiais (RNAs) é uma tarefa difícil e complexa. Em primeiro, é necessário pré-processar os dados e extrair *features*. Em segundo, definir a topologia, ou seja, escolher e parametrizar o tipo, ordem e conectividade das camadas da rede. Em terceiro, treinar a rede, o que implica definir qual o algoritmo de aprendizagem e os seus parâmetros. Este processo é iterativo e demorado e as escolhas realizadas em cada etapa afetam as restantes. Não obstante, o modelo tem que ser pelo menos revisto para a resolução de uma nova tarefa. No pior cenário, poderá ser necessário refazer toda a otimização da rede.

A literatura é rica em abordagens que focam a otimização de RNAs. Esta Tese foca particularmente a NeuroEvolução (NE) – conjunto de métodos que utilizam Computação Evolucionária para otimizar RNAs. Na bibliografia, a maioria das abordagens de NE evidenciam a otimização de uma RNA específica, ou a geração de soluções para um único problema. A nossa hipótese é que, utilizando técnicas de Programação Genética baseada em Gramáticas, seja possível propor uma representação flexível. Esta representação é facilitadora de uma *framework* genérica para otimizar e passar a produção Rede Neuronais Profundas (RNPs). No entanto, o método tem que ser eficiente, quer a nível temporal, quer dos recursos computacionais necessários.

Após analisarmos o desempenho dos métodos *Grammatical Evolution* e *Structured Grammatical Evolution* concluímos que estes não são adequados à otimização de redes multi-camada. Desta forma, propomos uma nova abordagem, à qual chamamos *Dynamic Structured Grammatical Evolution*. Esta abordagem, além de ser a primeira contribuição da Tese, é também central para o *Deep Evolutionary Network Structured Representation (DENSER)*. O *DENSER* é proposto como uma representação de NE baseada em gramáticas, que codifica RNPs utilizando uma estrutura com dois níveis. Esta representação permite a codificação de todos os aspetos de uma RNA: tipo e sequenciamento de camadas, algoritmo de aprendizagem, estratégia de geração de dados, etc.

Uma das maiores limitações do *DENSER* é o tempo necessário para gerar uma solução eficaz. O *Fast Deep Evolutionary Network Structured Representation (Fast-DENSER)* tem como objetivo acelerar a procura de RNAs. No *Fast-DENSER* o motor evolucionário é substituído por uma Estratégia Evolucionária, mantendo a representação utilizada no *DENSER*. Por outro lado, no *Fast-DENSER* o tempo de treino é uma propriedade do indivíduo, que pode aumentar ao longo da evolução. Após o término da evolução, as redes não necessitam de ser treinadas durante mais tempo.

Por fim, propomos o desenvolvimento incremental de forma a evitar que a procura comece do início aquando da resolução de um novo problema. Para facilitar a procura, a população inicial é formada tendo em conta as topologias encontradas para problemas anteriores. É também possível incorporar conhecimento proveniente de outros modelos em qualquer fase da evolução. Por outras palavras, o desenvolvimento incremental transfere informação topológica de forma a acelerar a evolução.

Acknowledgements/Agradecimentos[1]

Terminada esta fase é altura de agradecer a todos os que contribuíram para a viagem que culmina com a elaboração do presente documento. Não foram poucos os percalços, dúvidas e desafios que tiveram que ser ultrapassados. A ajuda de todos foi fundamental e é parte integrante desta Tese.

Começo por agradecer aos meus orientadores: professor Penousal Machado e professora Bernardete Ribeiro. Agradeço o convite que me endereçaram em 2014 para iniciar um percurso na investigação, convite esse que contribuiu bastante para o meu desenvolvimento profissional. Agradeço por terem aceite, sem hesitar, o convite para percorrem comigo este percurso e toda a liberdade que me concederam na definição do que viria a ser a Tese. Por fim, agradeço-lhes todo o apoio nas múltiplas questões e linhas de investigação que foram surgindo.

Um agradecimento a todos os membros do Computational Design and Visualization Lab. Em particular, obrigado António Cruz, Catarina Maçãs, Evgheni Polisciuc, João Correia, João Cunha e Tiago Martins. Agradeço não só todas as discussões técnicas, mas também todo o espaço de convívio, incluíndo todos os jogos klask. Catarina Maçãs e Tiago Martins, não poderia deixar de agradecer os inúmeros gráficos que me fizeram. Não poderia deixar de mencionar o Nuno Loureço, o qual considero praticamente um orientador desta Tese. Nuno, obrigado pelos diversas discussões, troca de ideias e mentoria. Uma nota de agradecimento ao MAS-BioISI (em particular ao Nuno e ao Davide) e ao LASIGE pela abertura com que me receberam em Lisboa.

Obrigado aos meus pais e familiares pelo apoio incondicional. Obrigado a todos os meus amigos pelos momentos de convívio e descontração. Por fim, um especial agradecimento à Cátia Ferreira, minha namorada e futura esposa. Sem ti, muito do que está neste documento não teria sido possível. Obrigado por me teres apoiado nos bons e maus momentos, por celebrares as minhas vitórias e por não me deixares desistir perante as derrotas e frustrações. A todos vós, um agradecimento pela (enorme) quantidade de vezes que questionaram quando começava a trabalhar - este "massacre" ajudou-me a manter o foco.

Uma palavra especial para recordar o Romeu. Por todos os momentos em que estiveste deitado debaixo da cadeira enquanto escrevia. Por todas as correrias e demais brincadeiras. Também tu fazes parte do sucesso desta Tese.

Por fim, agradeço à Fundação para a Ciência e Tecnologia (FCT), e ao projeto GADgET (DSAIPA/DS/0022/2018) por me terem financiado. Um obrigado a todos os outros que certamente me estou a esquecer mas que direta ou indiretamente contribuíram para o sucesso desta viagem.

Filipe Assunção

Coimbra, 9 de Dezembro de 2020

[1] For personal reasons, the acknowledges are written in Portuguese. Apologies to the non-Portuguese speakers.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Grammars

List of Acronyms

AE Auto-Encoder. 10, 25, 39, 62–67, 69, 70, 79–81, 132

AE-CNN Automatically Evolving CNN. 28, 99, 100

AI Artificial Intelligence. 1, 7, 30

ANN Artificial Neural Network. vii, 1–3, 5, 8, 9, 11–13, 20–27, 30, 31, 36, 39–42, 46–49, 52, 56–59, 61, 62, 71, 79–81, 83, 84, 87, 89–92, 98, 101, 104–106, 110, 112, 121–123, 131–133, 157, 158, 165, 170, 171

ANNA ELEONORA Artificial Neural Networks Adaption: Evolutionary Leaning Of Neural Optimal Running Abilities. 24

AUCROC Area Under the Receiver Operating Characteristic Curve. 43, 45, 49, 54, 56

AutoML Automated Machine Learning. 6, 12, 13, 20, 36, 131, 133, 164

BEV Bird's Eye View. 33

BNF Backus-Naur Form. 166

BP Back-Propagation. 9, 20–24, 27, 47, 56, 61, 80, 94, 109, 132

CAE Convolutional Auto-Encoder. 10

CFG Context-Free Grammar. 17, 80, 85, 164

CGP Cartesian Genetic Programming. 17, 25, 27, 105

CMA-ES Covariance Matrix Adaptation Evolution Strategy. 13, 26, 27

CNN Convolutional Neural Network. 9, 10, 13, 21, 27–29, 33, 83, 85, 89–92, 94, 96–98, 100, 101, 103, 105, 107, 109–113, 115–118, 120, 121, 125, 127, 128, 132, 133, 167

CoSyNE Cooperative Synapse NeuroEvolution. 21, 26, 29

CPPF Compositional Pattern Production Function. 3, 71–75, 77, 79–81, 132

CPPN Compositional Pattern Production Network. 23, 25, 26, 33, 70, 71, 73, 75

CWR CopyWeights with Re-init. 34

DAE Denoising Auto-Encoder. 10, 26

DANN Deep Artificial Neural Network. vii, 2, 3, 6, 9–13, 21, 25–32, 35, 36, 39, 80, 83, 84, 89, 92, 96, 98, 101, 103, 104, 113, 123, 125–132, 165, 166, 171, 172

DBN Deep Belief Network. 10

DCT Discrete Cosine Transform. 26

DEN Dynamically Expandable Network. 35

DENSER Deep Evolutionary Network Structured Representation. vii, ix, 3, 5, 6, 81, 83, 86, 87, 89–94, 96–112, 115, 121, 123, 132, 133

DL Deep Learning. 9–11, 29, 92

DPPN Differentiable Pattern Producing Network. 26

DSGE Dynamic Structured Grammatical Evolution. vii, 3, 4, 6, 39, 49, 50, 52–54, 56, 58, 59, 61, 62, 71–73, 75–77, 79–81, 83, 84, 88, 108, 131–133, 157, 159, 160, 169

DT Decision Tree. 8, 66, 67

EA Evolutionary Algorithm. 91

EC Evolutionary Computation. 1, 7, 11, 13–16, 19–23, 26, 29, 31, 36, 65, 71, 100, 131, 133, 165

EP Evolutionary Programming. 26

ES Evolutionary Strategy. 3, 26, 103, 104, 121, 133, 165

ESP Enforced Subpopulations. 25, 29

EvoCNN Evolving Deep Convolutional Neural Networks. 27, 28, 99

EWC Elastic Weight Consolidation. 34

EXACT Evolutionary Exploration of Augmenting Convolutional Topologies. 29

EXAMM Evolutionary eXploration of Augmenting Memory Models. 33

Fast-DENSER Fast Deep Evolutionary Network Structured Representation. vii, ix, 3, 4, 6, 103–105, 107–113, 115, 116, 118, 120–126, 130–134, 165, 167, 171

FPR False Positive Rate. 117

GA Genetic Algorithm. 21, 25–29, 83, 84, 103, 104

GAN Generative Adversarial Network. 10

GE Grammatical Evolution. 3, 18–20, 23, 25, 28, 29, 39, 40, 42, 43, 45–49, 52–54, 56, 58, 61, 71–73, 75–77, 79–81, 83, 131, 132

GGP Grammar-based Genetic Programming. 1–3, 17, 18, 52, 83, 131, 132

GNARL Generalised Acquisition of Recurrent Links. 24

GP Genetic Programming. 3, 14, 16–18, 20, 23, 25, 26, 50, 71–81, 132

GPU Graphics Processing Unit. 3, 9, 24, 30, 89, 103, 104, 106–108, 111–113, 121, 126, 133

HTC Hypothesis Testing Plasticity. 35

JSON JavaScript Object Notation. 167

k-NN k-Nearest Neighbors. 13, 66, 67, 69

LDA Latent Dirichlet Allocation. 13

LDF Lateral Distribution Function. 120

LEAF Learning Evolutionary AI Framework. 29

LEEA Limited Evaluation Evolutionary Algorithm. 27

LSTM Long Short-Term Memory. 10, 32

MCTS Monte-Carlo Tree Search. 14

ML Machine Learning. 1, 7, 8, 12–14, 26, 36, 40, 65, 67, 123, 131, 157, 158, 161, 162, 164

MLP Multi-Layer Perceptron. 8, 9, 22, 23, 64–67, 69, 70, 97

MOSAIC Monte-Carlo Tree Search for Algorithm Configuration. 14, 36

MSE Mean Squared Error. 171

NE NeuroEvolution. vii, 1–3, 5–7, 12, 20–27, 29–32, 35, 36, 39, 40, 49, 63, 80, 83, 101, 121, 123, 131, 132, 165, 171

NEAT NeuroEvolution of Augmenting Topologies. 3, 23, 25, 26, 29, 32, 33, 70–81, 132

NLP Natural Language Processing. 9, 10, 33

NSGA Non-Dominated Sorting Genetic Algorithm. 28

NTM Neural Turing Machine. 35

PCA Principal Component Analysis. 8, 62

PDGP Parallel Distributed Genetic Programming. 24

PNAS Progressive Neural Architecture Search. 32

PNN Progressive Neural Network. 35

RECIPE Resilient Classification Pipeline Evolution. 14, 36, 157, 161–164

RF Random Forest. 66, 67

RL Reinforcement Learning. 27

RMSE Root Mean Squared Error. 40, 43, 45–49, 54, 56, 171

RNN Recurrent Neural Network. 10, 35

ROC Receiver Operating Characteristic. 117, 118, 120

SANE Symbiotic, Adaptive NeuroEvolution. 25, 29

SGE Structured Grammatical Evolution. 3, 19, 20, 39–50, 52–54, 56, 58, 61, 80, 131

SVM Support Vector Machine. 8, 12, 13, 66, 67, 158

TCDCN Tasks-Constrained Deep Convolutional Network. 33

TENN Topology-optimisation Evolutionary Neural Network. 23

TPOT Tree-based Pipeline Optimization Tool. 14, 36

TPR True Positive Rate. 117

UCI University of California Irvine. 3, 26, 40, 161

VAE Variational Auto-Encoder. 10

WCD Water-Cherenkov Detector. 116, 120, 121

Glossary

AE Auto-Encoder. 10, 25, 39, 62–67, 69, 70, 79–81, 132

AE-CNN Automatically Evolving CNN. 28, 99, 100

AI Artificial Intelligence. 1, 7, 30

ANN Artificial Neural Network. vii, 1–3, 5, 8, 9, 11–13, 20–27, 30, 31, 36, 39–42, 46–49, 52, 56–59, 61, 62, 71, 79–81, 83, 84, 87, 89–92, 98, 101, 104–106, 110, 112, 121–123, 131–133, 157, 158, 165, 170, 171

ANNA ELEONORA Artificial Neural Networks Adaption: Evolutionary Leaning Of Neural Optimal Running Abilities. 24

AUCROC Area Under the Receiver Operating Characteristic Curve. 43, 45, 49, 54, 56

AutoML Automated Machine Learning. 6, 12, 13, 20, 36, 131, 133, 164

BEV Bird's Eye View. 33

BNF Backus-Naur Form. 166

BP Back-Propagation. 9, 20–24, 27, 47, 56, 61, 80, 94, 109, 132

CAE Convolutional Auto-Encoder. 10

CFG Context-Free Grammar. 17, 80, 85, 164

CGP Cartesian Genetic Programming. 17, 25, 27, 105

CMA-ES Covariance Matrix Adaptation Evolution Strategy. 13, 26, 27

CNN Convolutional Neural Network. 9, 10, 13, 21, 27–29, 33, 83, 85, 89–92, 94, 96–98, 100, 101, 103, 105, 107, 109–113, 115–118, 120, 121, 125, 127, 128, 132, 133, 167

CoSyNE Cooperative Synapse NeuroEvolution. 21, 26, 29

CPPF Compositional Pattern Production Function. 3, 71–75, 77, 79–81, 132

CPPN Compositional Pattern Production Network. 23, 25, 26, 33, 70, 71, 73, 75

CWR CopyWeights with Re-init. 34

DAE Denoising Auto-Encoder. 10, 26

DANN Deep Artificial Neural Network. vii, 2, 3, 6, 9–13, 21, 25–32, 35, 36, 39, 80, 83, 84, 89, 92, 96, 98, 101, 103, 104, 113, 123, 125–132, 165, 166, 171, 172

DBN Deep Belief Network. 10

DCT Discrete Cosine Transform. 26

DEN Dynamically Expandable Network. 35

DENSER Deep Evolutionary Network Structured Representation. vii, ix, 3, 5, 6, 81, 83, 86, 87, 89–94, 96–112, 115, 121, 123, 132, 133

DL Deep Learning. 9–11, 29, 92

DPPN Differentiable Pattern Producing Network. 26

DSGE Dynamic Structured Grammatical Evolution. vii, 3, 4, 6, 39, 49, 50, 52–54, 56, 58, 59, 61, 62, 71–73, 75–77, 79–81, 83, 84, 88, 108, 131–133, 157, 159, 160, 169

DT Decision Tree. 8, 66, 67

EA Evolutionary Algorithm. 91

EC Evolutionary Computation. 1, 7, 11, 13–16, 19–23, 26, 29, 31, 36, 65, 71, 100, 131, 133, 165

EP Evolutionary Programming. 26

ES Evolutionary Strategy. 3, 26, 103, 104, 121, 133, 165

ESP Enforced Subpopulations. 25, 29

EvoCNN Evolving Deep Convolutional Neural Networks. 27, 28, 99

EWC Elastic Weight Consolidation. 34

EXACT Evolutionary Exploration of Augmenting Convolutional Topologies. 29

EXAMM Evolutionary eXploration of Augmenting Memory Models. 33

Fast-DENSER Fast Deep Evolutionary Network Structured Representation. vii, ix, 3, 4, 6, 103–105, 107–113, 115, 116, 118, 120–126, 130–134, 165, 167, 171

FPR False Positive Rate. 117

GA Genetic Algorithm. 21, 25–29, 83, 84, 103, 104

GAN Generative Adversarial Network. 10

GE Grammatical Evolution. 3, 18–20, 23, 25, 28, 29, 39, 40, 42, 43, 45–49, 52–54, 56, 58, 61, 71–73, 75–77, 79–81, 83, 131, 132

GGP Grammar-based Genetic Programming. 1–3, 17, 18, 52, 83, 131, 132

GNARL Generalised Acquisition of Recurrent Links. 24

GP Genetic Programming. 3, 14, 16–18, 20, 23, 25, 26, 50, 71–81, 132

GPU Graphics Processing Unit. 3, 9, 24, 30, 89, 103, 104, 106–108, 111–113, 121, 126, 133

HTC Hypothesis Testing Plasticity. 35

JSON JavaScript Object Notation. 167

k-NN k-Nearest Neighbors. 13, 66, 67, 69

LDA Latent Dirichlet Allocation. 13

LDF Lateral Distribution Function. 120

LEAF Learning Evolutionary AI Framework. 29

LEEA Limited Evaluation Evolutionary Algorithm. 27

LSTM Long Short-Term Memory. 10, 32

MCTS Monte-Carlo Tree Search. 14

ML Machine Learning. 1, 7, 8, 12–14, 26, 36, 40, 65, 67, 123, 131, 157, 158, 161, 162, 164

MLP Multi-Layer Perceptron. 8, 9, 22, 23, 64–67, 69, 70, 97

MOSAIC Monte-Carlo Tree Search for Algorithm Configuration. 14, 36

MSE Mean Squared Error. 171

NE NeuroEvolution. vii, 1–3, 5–7, 12, 20–27, 29–32, 35, 36, 39, 40, 49, 63, 80, 83, 101, 121, 123, 131, 132, 165, 171

NEAT NeuroEvolution of Augmenting Topologies. 3, 23, 25, 26, 29, 32, 33, 70–81, 132

NLP Natural Language Processing. 9, 10, 33

NSGA Non-Dominated Sorting Genetic Algorithm. 28

NTM Neural Turing Machine. 35

PCA Principal Component Analysis. 8, 62

PDGP Parallel Distributed Genetic Programming. 24

PNAS Progressive Neural Architecture Search. 32

PNN Progressive Neural Network. 35

RECIPE Resilient Classification Pipeline Evolution. 14, 36, 157, 161–164

RF Random Forest. 66, 67

RL Reinforcement Learning. 27

RMSE Root Mean Squared Error. 40, 43, 45–49, 54, 56, 171

RNN Recurrent Neural Network. 10, 35

ROC Receiver Operating Characteristic. 117, 118, 120

SANE Symbiotic, Adaptive NeuroEvolution. 25, 29

SGE Structured Grammatical Evolution. 3, 19, 20, 39–50, 52–54, 56, 58, 61, 80, 131

SVM Support Vector Machine. 8, 12, 13, 66, 67, 158

TCDCN Tasks-Constrained Deep Convolutional Network. 33

TENN Topology-optimisation Evolutionary Neural Network. 23

TPOT Tree-based Pipeline Optimization Tool. 14, 36

TPR True Positive Rate. 117

UCI University of California Irvine. 3, 26, 40, 161

VAE Variational Auto-Encoder. 10

WCD Water-Cherenkov Detector. 116, 120, 121

Chapter 1

Introduction

The possibility of creating intelligent machines has been a philosophical dilemma for years and has been moving the Artificial Intelligence (AI) community since its genesis. In Turing's seminal paper [252] the author discusses the question "can machines think?". However, he writes that this is an ill-suited question, that would lead to an unproductive debate. It is more important that we learn about the cognitive power of computers. Therefore, it is crucial to understand "how can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes" [177].

In a broad sense, Machine Learning (ML) seeks to propose methods that improve with experience. There is a wide set of ML methods, and thus selecting the most appropriate one for a specific problem is challenging. In the current Thesis, we focus on algorithmic approaches inspired by the phenomena that happen in nature and biology. In particular, we focus on Artificial Neural Networks (ANNs). For an ANN to perform a single task we need to define its structure (i.e., number and type of layers, and connectivity of the layers/neurons), and learning strategy (i.e., learning algorithm and its hyper-parameters). Additionally, the parameterisation despite well suited for a given problem is not necessarily able to solve others. In other words, ANNs although inspired in neurosciences and in the human brain are still not able to match its versatility, adaptability, and robustness.

By combining Evolutionary Computation (EC) and ANNs we hypothesise that the gain is twofold. On the one hand, EC has the ability to promote the automatic search for adequate topologies and/or parameters of ANNs, which when accomplished by trial-and-error is a difficult and time-consuming task. On the other hand, EC introduces a means to incremental development by enabling the identification of existing knowledge that can be re-used to solve a new and possibly more complex task. In addition, incremental development may also speed up the search for a model to address a new task.

In Section 1.1 we set the research hypothesis; Section 1.2 enumerates the main contributions of the current Thesis; and Section 1.3 describes the structure of the remainder of the document.

1.1 Research Hypothesis

Motivated by the difficulty in designing an ANN, we investigate approaches to automate its deployment, i.e., methods that automatically set the topology, learning, and any other hyper-parameters of ANNs. In particular, we apply EC to optimise ANNs, a field known as NeuroEvolution (NE). There are various branches of EC, but the focus of this Thesis is on Grammar-based Genetic Programming (GGP) approaches. This decision takes into account that we need an easy

to adapt methodology, so that we may develop a general-purpose framework which is ultimately able to address diverse types of network topologies, and can be applied to a wide set of domains and fields without changes to the framework. It is our perception that the definition of the search space using a grammar, and the search for flexible representations are key-aspects to such a framework. One of the limitations of NE methods is the time required for obtaining an effective network. A flexible representation is crucial to facilitate the identification of evolutionary units (or blocks) that can be incorporated later in, to solve new problems. The idea is to speed up the search by building the networks incrementally.

Therefore, and based on the aforementioned, our research hypothesis is that GGP methods can be applied to develop a general-purpose, flexible, and efficient framework for the automatic optimisation of ANNs that can be built incrementally. Without loss of generality, we conduct the majority of our experiments in image classification problems. To tackle the research hypothesis we formulate four research questions:

Which GGP method is effective in NE? There are various GGP methods, and they all share the same background, i.e., the search space is defined by means of a grammar. However, the way in which the individuals are represented and the genotype decoded to the phenotype differs. It is our objective to investigate which of the methods is able to effectively optimise ANNs. The most adequate method is the one that attains the best results, but without compromising the search time.

Which representation scheme is adequate to a flexible encoding of ANNs? When optimising ANNs we need to take several aspects into account. First, it is important to consider if the approach will be applied only to small-scale networks, or if it will require the emergence of potentially deep architectures. Second, the representation has to be flexible. There are diverse types of network architectures, and the networks are required to solve various problems. Thus, the representation has to encompass all these possibilities without the need for changes when there is the need to search for a different network type and/or a solution to a different task.

How can the ANNs be evaluated and evolved efficiently? It is mentioned in the first research question that the best approach is not the one that reports the best results, but the one that reports a good trade-off between performance and search time. In NE, especially when searching for Deep Artificial Neural Networks (DANNs), the proposal of efficient methods is of paramount importance. To generate solutions in acceptable time frames, the majority of the approaches limit the evaluation time of the individuals and, consequently, the ANNs' training time. However, it is our opinion that this hinders the generation of properly trained and ready-to-deploy ANNs.

Can the ANNs be developed incrementally? The automatic generation of ANNs resorting to NE is time-consuming. This is even more striking when we consider that, for each new problem, we have to re-start the search from scratch. Therefore, we aim at developing a methodology that enables the re-use of useful knowledge from source tasks, which can be transferred to yet unsolved target tasks.

1.2 Contributions

The main contributions of this Thesis to the field of NE are highlighted and detailed next. These contributions are in-line with the research hypothesis.

Literature review – a comprehensive review of the NE state of the art (with particular focus on approaches for optimising DANNs), and survey of transfer, multi-task, and incremental learning applied to ANNs (Chapter 2).

Review of Genetic Programming (GP) methods applied to NE – a comparison of the performance of GGP methods on the evolution of the topology and connection weights of ANNs. In particular, we compare Grammatical Evolution (GE) to Structured Grammatical Evolution (SGE) on the evolution of one-hidden-layered ANNs, and conclude that the performance of SGE is statistically superior to GE, but that none of the methods is able to optimise multi-layered ANNs (Section 3.1). A comparison between various GGP methods, tree-based GP and NeuroEvolution of Augmenting Topologies (NEAT) is also conducted on the optimisation of weight policies based on Compositional Pattern Production Functions (CPPFs); the GGP methods are outperformed by the remaining ones (Section 3.4).

Proposal of a GGP approach – a novel extension to SGE, referred to as Dynamic Structured Grammatical Evolution (DSGE), is proposed as a means to grow the genotype continuously as more grammatical expansions are required. DSGE outperforms both GE and SGE. The combination of DSGE with dynamic production rules allows the optimisation of the topology and connection weights of multi-layered ANNs by enabling the grammar to keep track of the placement of the neurons in the networks (Section 3.2). In addition to optimising multi-layered ANNs, we also apply DSGE to the evolution of Scikit-Learn classification pipelines (Appendix A).

Proposal of a grammar-based NE representation – a grammar-based NE representation based on DSGE that enables the evolution of effective DANNs. The method is called Deep Evolutionary Network Structured Representation (DENSER) (Chapter 4) and is able to optimise the topology, learning strategy, and any other network's hyper-parameters. The representation is based on a novel two-level schema where the outer level encodes the macrostructure of the network (i.e., layers, learning, or any other block), and the inner level stores the associated hyper-parameters. Fast Deep Evolutionary Network Structured Representation (Fast-DENSER) (Chapter 5) replaces the evolutionary engine previously used in DENSER by a $(1+\lambda)$-Evolutionary Strategy (ES); the objective is to speed up evolution. Additionally, the networks are trained for a maximum Graphics Processing Unit (GPU) training time, which contrasts with the typical evaluation policy that trains the networks for a maximum number of epochs. Moreover, the training time increases as evolution proceeds so that more complex networks, that may benefit from longer learning sessions, have access to more training time. By the end of evolution, the models require no further training.

Application to a wide set of problems – the proposed methods are tested in various benchmark problems (e.g., University of California Irvine (UCI) benchmarks), visual recognition tasks (e.g., MNIST, SVHN, CIFAR), and a real-world problem of the physics domain (detection of gamma-rays based on the ground impact patterns). The results reported by the proposed methods are compared to the state of the art.

Release of classification models – the fittest models that are automatically generated by DENSER and Fast-DENSER to classify the considered benchmarks are made available and can be used without the need for training. The trained models can be found in the GitHub's repository `https://github.com/fillassuncao/denser-models`. The models generated by the automatic methods are, typically, structurally different from human-designed ones.

Release of open-source software – the last stable version of Fast-DENSER (Appendix B) is made available as open-source software and can be found in the GitHub repository `https://github.com/fillassuncao/fast-denser3`. We also release the code for DSGE, and AutoML-DSGE, which are respectively available at `https://github.com/nunolourenco/sge3` and `https://github.com/fillassuncao/automl-dsge`.

Proposal of an incremental development strategy – the extension of Fast-DENSER to enable the incorporation of network structures that are discovered throughout the search for networks for previous source tasks. The introduced incremental development approach speeds up the search on target tasks, and generates higher-performing models (Chapter 6).

The previous contributions resulted in a series of publications in national (2) and international (8) peer-reviewed conferences and journals (3), and book chapters (1). Next, we enumerate the publications related to this Thesis, with reference to the section/chapter/appendix where they are reflected (within brackets).

- Filipe Assunção et al. "Evolutionary Machine Learning: An Essay on Experimental Design". In: *Proceedings of the 23rd Portuguese Conference on Pattern Recognition (RecPad)*. 2017 (Chapter 2)
- Filipe Assunção et al. "Evolutionary Machine Learning: An Essay on Benchmarking". In: *Proceedings of the 23rd Portuguese Conference on Pattern Recognition (RecPad)*. 2017 (Chapter 2)
- Filipe Assunção et al. "Automatic Generation of Neural Networks with Structured Grammatical Evolution". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. June 2017, pp. 1557–1564. DOI: 10.1109/CEC.2017.7969488 (Section 3.1)
- Filipe Assunção et al. "Towards the Evolution of Multi-layered Neural Networks: A Dynamic Structured Grammatical Evolution Approach". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 393–400. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071286. URL: http://doi.acm.org/10.1145/3071178.3071286 (Section 3.2)
- Nuno Lourenço et al. "Structured Grammatical Evolution: A Dynamic Approach". In: *Handbook of Grammatical Evolution*. Ed. by Conor Ryan, Michael O'Neill, and JJ Collins. Cham: Springer International Publishing, 2018, pp. 137–161. ISBN: 978-3-319-78717-6. DOI: 10.1007/978-3-319-78717-6_6. URL: https://doi.org/10.1007/978-3-319-78717-6_6 (Section 3.2)
- Filipe Assuncao et al. "Automatic Evolution of AutoEncoders for Compressed Representations". In: *2018 IEEE World Congress on Computational Ingelligence (WCCI)*. July 2018, pp. 1–8. DOI: 10.1109/CEC.2018.8477874 (Section 3.3)
- Filipe Assunção et al. "Using GP Is NEAT: Evolving Compositional Pattern Production Functions". In: *Genetic Programming*. Ed. by Mauro Castelli et al. Cham: Springer International Publishing, 2018, pp. 3–18. ISBN: 978-3-319-77553-1. DOI: 10.1007/978-3-319-77553-1_1 (Section 3.4)
- Filipe Assunção et al. "Evolving the Topology of Large Scale Deep Neural Networks". In: *Genetic Programming*. Ed. by Mauro Castelli et al. Cham: Springer International Publishing, 2018, pp. 19–34. ISBN: 978-3-319-77553-1. DOI: 10.1007/978-3-319-77553-1_2 (Chapter 4)

- Filipe Assunção et al. "DENSER: Deep Evolutionary Network Structured Representation". In: *Genetic Programming and Evolvable Machines* 20.1 (Mar. 2019), pp. 5–35. ISSN: 1573-7632. DOI: 10.1007/s10710-018-9339-y. URL: https://doi.org/10.1007/s10710-018-9339-y (Chapter 4)

- Filipe Assunção et al. "Fast DENSER: Efficient Deep NeuroEvolution". In: *Genetic Programming*. Ed. by Lukas Sekanina et al. Cham: Springer International Publishing, 2019, pp. 197–212. ISBN: 978-3-030-16670-0. DOI: 10.1007/978-3-030-16670-0_13 (Chapter 5)

- Filipe Assunção et al. "Automatic Design of Artificial Neural Networks for Gamma-Ray Detection". In: *IEEE Access* 7 (2019), pp. 110531–110540. DOI: 10.1109/ACCESS.2019.2933947 (Chapter 5)

- Filipe Assunção et al. "Incremental Evolution and Development of Deep Artificial Neural Networks". In: *European Conference on Genetic Programming (EuroGP)*. Cham: Springer International Publishing, 2020 (Chapter 6)

- Filipe Assunção et al. "Evolution of Scikit-Learn Pipelines with Dynamic Structured Grammatical Evolution". In: *International Conference on the Applications of Evolutionary Computation*. Cham: Springer International Publishing, 2020 (Appendix A)

- Filipe Assunção et al. "Fast-DENSER: Fast Deep Evolutionary Network Structured Representation". In: *SoftwareX* 14 (2021), p. 100694. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2021.100694 (Appendix B)

In addition to the publications, the work carried out in this Thesis was distinguished three times. The nominations and awards are described next.

- Finalist, Human Competitive Awards (Humies), Genetic and Evolutionary Computation Conference (GECCO), Kyoto, Japan, 2018
  - The Human Competitive "Humies" award seeks to recognise results that are obtained by any form of evolutionary computation, and that are competitive with the human performance. DENSER was nominated finalist because of the highly human competitive results reported on object recognition tasks.

- The Best of Technological Portugal, Exame Informática, Lisbon, Portugal, 2018
  - Award granted to the project DENSER by the Portuguese magazine Exame Informática, in the category of software.

- Best Student Paper Runner-up, European Conference on Genetic Programming, Parma, Italy, 2018
  - Paper: Using GP is NEAT: Evolving Compositional Pattern Production Functions.

1.3 Document Structure

The remainder of the document is structured as follows. In Chapter 2 we introduce all background concepts required to the understanding of the Thesis; this Chapter also surveys related works and the state of the art of NE. In Chapter 3 we devise exploratory work to better understand the behaviour of grammar-based methods on the automatic optimisation of ANNs. In

Chapter 4, based on the conclusions of Chapter 3, we propose DENSER, a novel grammar-based NE representation that is capable of generating high performing DANNs. In Chapter 5 we introduce Fast-DENSER, a new evolutionary engine adapted to the representation of DENSER that speeds up search, and enables the evolution of ready-to-deploy models, i.e., DANNs that require no further training after evolution. In Chapter 6 we propose incremental development as a means to enable the DANNs to continuously grow to solve new tasks. Therefore, the knowledge that is gathered when addressing previous source tasks aids in solving a target task, potentially speeding up search. In Chapter 7 we summarise the Thesis conclusions and address future work.

This document is also composed by two appendices. In Appendix A, we introduce AutoML-DSGE, an Automated Machine Learning (AutoML) framework based on DSGE for the optimisation of Scikit-Learn pipelines. In Appendix B, we detail the Fast-DENSER framework.

Chapter 2

Background and Related Work

The objective of the current Chapter is to discuss background concepts required to the understanding of the remainder of this Thesis. The Chapter starts by the definition of ML terminology (Section 2.1), which is followed by the introduction of methods for automating ML (Section 2.2). Next, we detail EC (Section 2.3) because it is the base for NE (Section 2.4). Finally, we briefly survey transfer, multi-task, and incremental learning methodologies (Section 2.5).

2.1 Machine Learning

ML is a sub-field of AI. In particular, in the current Thesis we focus on example-based ML, where the goal is to develop models that automatically learn how to solve a particular task based on a dataset.Formally, a dataset is a set of instances that are related to a specific task or problem. In supervised learning, a dataset (D) of size n is defined as D=$\{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $\{x_1, \ldots, x_n\}$ are the instances of the dataset, and $\{y_1, \ldots, y_n\}$ are the labels. The i-th label is the output of a function (f) that receives as input the i-th instance, i.e., $y_i = f(x_i)$. The instances are vectors of m features (i.e., real, categorical or discrete data characteristics), and thus D=$\{(\{x_{1,1}, \ldots, x_{1,m}\}, y_1), \ldots, (\{x_{n,1}, \ldots, x_{n,m}\}, y_n)$. The labels map the instances to a set of k classes, i.e., $y_i \in \{0, \ldots, k\}$. An overview of the dataset structure is depicted in Figure 2.1. Briefly, a dataset can be seen as a matrix of features, where each row is an instance, and the columns except for the last are the features; the last column is the label.

Learning can be supervised or unsupervised. In supervised learning, as aforementioned, the dataset is labelled: the instances are categorised into different classes. On the other hand, in unsupervised learning, the instances are unlabelled, i.e., we have instances but no classes. Therefore, whereas in supervised learning, the objective is to learn a function that maps the instances to the classes, in unsupervised learning, the goal is to learn the hidden structure and patterns of the dataset. Supervised learning is used in classification and regression tasks, and unsupervised learning in clustering, or dimensionality reduction. The focus of this Thesis is mainly on supervised classification tasks. Supervised learning is either generative or discriminative. Generative methods predict the probability of the target class y of a sample x, i.e., the model captures the joint probability P(x, y) of an observation x with a target variable y. Discriminative models learn to model the decision boundary between classes, and therefore, instead of the joint probability, they tend to model the conditional probability. The decision boundary is a hyper-surface that defines the separation frontier between the different classes of the problem

The dataset, in supervised learning, is often divided into three disjoint partitions: (i) train – subset to train the model; (ii) validation – optional subset to tweak the model during training;

$x_{1,1}$	$x_{1,2}$	$x_{1,...}$	$x_{1,m}$		y_1
$x_{2,1}$	$x_{2,2}$	$x_{2,...}$	$x_{2,m}$		y_2
$x_{...,1}$	$x_{...,2}$	$x_{...,...}$	$x_{...,m}$		$y_{...}$
$x_{n,1}$	$x_{n,2}$	$x_{n,...}$	$x_{n,m}$		y_n

Figure 2.1: Representation of a dataset of size n, focusing on the instances x (composed by m features), and the labels y.

and (iii) test – subset to test the trained model. The performance of the models should always be evaluated on the test set, and the instances of the test partition should never be applied neither to train nor to tweak the model. Otherwise, we would not be able to perform an unbiased analysis of the model's results.

The ML literature comprises multiple approaches; for example, for classification and regression we can use Support Vector Machines (SVMs), Decision Trees (DTs) or ANNs, for clustering the k-Means, and for dimensionality reduction the Principal Component Analysis (PCA). For a comprehensive in-depth review and explanation of the several ML methods refer to [34]. The focus of this Thesis in on ANNs, which are detailed next.

ANNs [273] are a bio-inspired ML model with roots in the neurosciences. ANNs are composed of several interconnected units, known as neurons, that form a network that simulates the human brain. The connection between any two neurons has a weight associated with it which, together with activation functions, confer ANNs the ability to approximate non-linear functions of the input feed to the network. The weights allow networks to mimic biological synapses. To do so, the inputs of the neurons (x), are multiplied by the weights (w) and may be summed with a bias term (b). The result is passed through an activation function (act) and then possibly to another neuron. The simplest structure of an ANN is the perceptron [208]. The perceptron is a fully-connected single hidden-layered feed-forward ANN, with a single output neuron and where, similarly to the stated before, the output (out) is computed by $\text{out} = act(\sum_{i=1}^{N}(x_i \cdot w_i) + b)$, where b is the bias, and N is the input size, that matches the number of features of the instances. The perceptron is a linear classifier that, historically, uses the unit step function as the activation function, and thus, the mapping function can be written as:

$$\text{out} = \begin{cases} 1, \text{if} \sum_{i=1}^{N}(x_i \cdot w_i) + b > 0, \\ 0, \text{otherwise.} \end{cases}$$

The perceptron is only able to distinguish linearly separable data. The Multi-Layer Perceptron (MLP) extends the perceptron to an unconstrained number of hidden-layers and output neurons. Together with the use of non-linear activation functions (e.g., tanh, or sigmoid) these changes make the MLP able to deal with non-linearly separable data. Figure 2.2 represents an example of a MLP with two hidden-layers. To compute the output of the network the weighted sum of inputs has to be divided into three steps: (i) first we compute the output of the first hidden-layer (out_1) given the input, $\text{out}_1 = act(W_1 \cdot X + b_1)$; (ii) next, we compute the output of the second hidden-layer (out_2) given the input of the first hidden-layer, $\text{out}_2 = act(W_2 \cdot \text{out}_1 + b_2)$; and (iii) finally we compute the output of the network (out_{pred}) given the output of the second hidden-layer, $\text{out}_{\text{pred}} = act(W_3 \cdot \text{out}_2 + b_3)$. The previous equations are defined in the form of

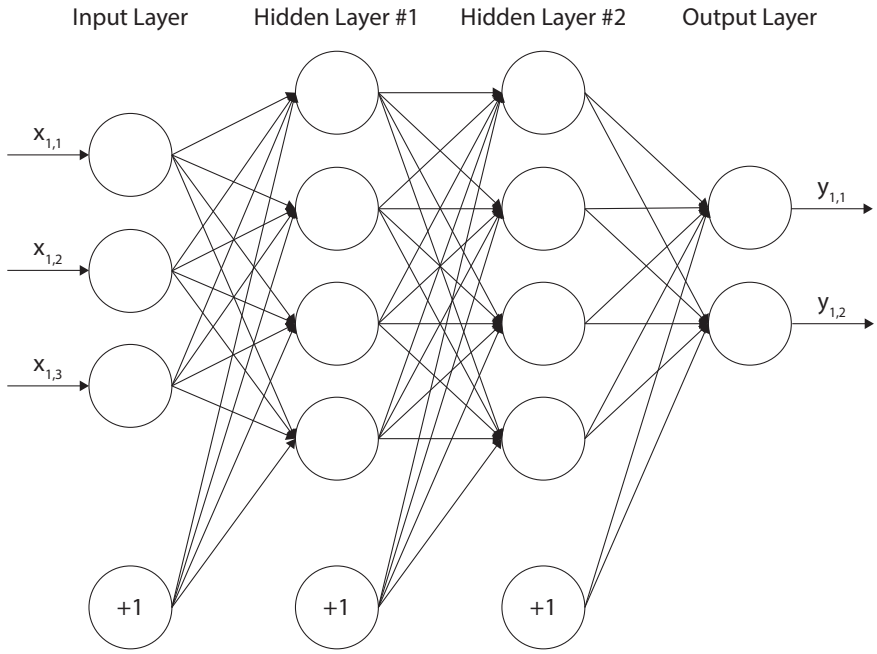


Figure 2.2: Example of a Multi-Layer Perceptron with two hidden-layers.

matrices. If the MLP has more (or less) than two hidden-layers the same rationale is applied. The connection weights (W_1, W_2, and W_3), and the bias (b_1, b_2, b_3) are initially set at random. To approximate the output of the network to the target value, the network must be trained, which is normally accomplished using a gradient descent strategy, e.g., Back-Propagation (BP). For an in-depth explanation of the BP algorithm we refer the reader to Section 5.3 of [34].

The recent technological advances in computational power, in particular, the widespread availability of GPUs, have potentiated Deep Learning (DL) and the deployment of more complex networks, that require further resources to be trained. From an architectural point of view, the main difference between shallow and deep ANNs concerns the number of hidden-layers. Typically, a shallow network has up to two hidden-layers, and beyond that, the network is considered deep [27]. However, the key-point of DL is the ability of the models to learn the data representation and features. While, to design shallow approaches, someone with domain expertise is often required, in DL the feature engineering stage is avoided [144]. In particular, to perform feature engineering, we need to have an in-depth comprehension of the problems's domain, so that we can create and synthesize features (e.g., for credit card fraud detection possible features are the average transaction amount, or the average number of daily transactions). This per se is a time-consuming process that involves many impactful decisions such as how to handle missing values, outliers or different feature's scales. In addition, after having a list of features, we have to select those that work best depending on the model we are using. Based on the performance of the model, we may need to revisit the features to improve them, or even create new ones.

Convolutional Neural Networks (CNNs) [76, 145] are an example of a DANN that has been widely applied to Computer Vision and Natural Language Processing (NLP) tasks [33]. An overview of the topology of a CNN is depicted in Figure 2.3. The network structure of CNNs divides the layers into two separate, but related groups. The first group learns features, and for that it makes use of convolutional and pooling layers; the second performs classification based on

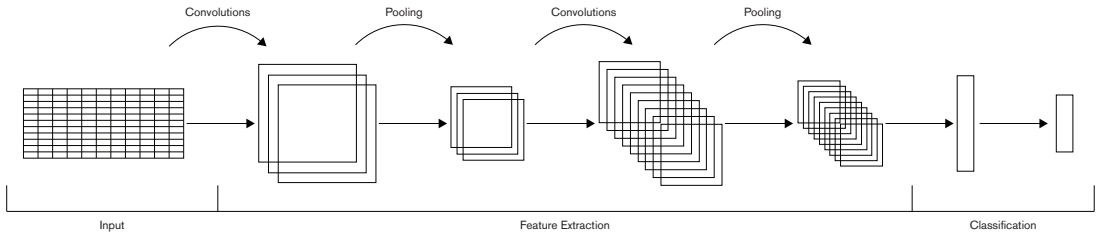


Figure 2.3: Overview of a Convolutional Neural Network.

the extracted features, and for that purpose makes use of fully-connected layers. Convolutional layers are composed of several filters, each one responsible for learning a different feature; pooling layers promote data down-sampling, by applying the maximum or average functions. Convolutional and pooling layers are connected to a small region of the previous layer, known as the receptive field, and then to apply the convolution/pooling to the whole input data, the operation must be systematically applied to all the different regions of the input layer. CNNs explore the spatial correlation present in the data, and the receptive fields allow for weight-sharing, given that the filters are applied to the entire input, with the same parameters. This provides CNNs with translation invariance and training efficiency.

Auto-Encoders (AEs) [105] are an unsupervised generative DANN that has the main objective of rebuilding the input data, i.e., whereas the previous models predict the classes (y) from the input data (x), AEs predict x from x. The goal of AEs is to learn a compact representation of the input. Notwithstanding, AEs can also be used to remove noise from data, in what is known as a Denoising Auto-Encoder (DAE) [258]. Figure 2.4 depicts the structure of a vanilla AE. The structure of the layers in AEs is divided into two distinct phases: (i) encoding and (ii) decoding. The encoding phase maps the input of the network into its compressed lower-dimensionality version, normally known as latent space. The decoding phase maps the latent space into x', which has the same dimensionality of the input x. The objective is to train the AE so that x' reconstructs x with the least amount of error. AEs are feed-forward DANNs that are usually symmetric, i.e., the number of neurons and layers of the encoding layers are the same as the number of neurons and layers of the decoding layers, but inverted. The layer that encodes the latent space is known as the chokepoint. The latent space can be used in a classification task by feeding the output of the encoder to a discriminative model (e.g., [274]).

CNNs and AEs are two examples of feed-forward DANNs. Recurrent Neural Networks (RNNs) contrast with feed-forward DANNs. Whereas, in feed-forward networks, the signal flows in a single direction from the input to the output, without loops, in RNNs the signal can go in any direction, and there can be loops. One of the key-advantages of RNNs is that they introduce the notion of state, and thus create a sense of memory. Consequently, the decisions taken by the network are influenced by the past. That is the reason why RNNs are widely used with temporally-related data, such as NLP (where the context of an instance changes its meaning), or video (where consecutive frames are related). An example of an RNN is the Long Short-Term Memory (LSTM) [107]. There are many other DL approaches and variants of the previous ones. For example Convolutional Auto-Encoders (CAEs) [166], Variational Auto-Encoders (VAEs) [126], Deep Belief Networks (DBNs) [104], Generative Adversarial Networks (GANs) [84]. Notwithstanding, in the current Thesis, we will focus on the optimisation of CNNs, and AEs. A more in-depth analysis of other DL approaches can be found in [71, 85]. The focus goes mainly to CNNs because they are one of the most challenging network topologies in terms of design: in

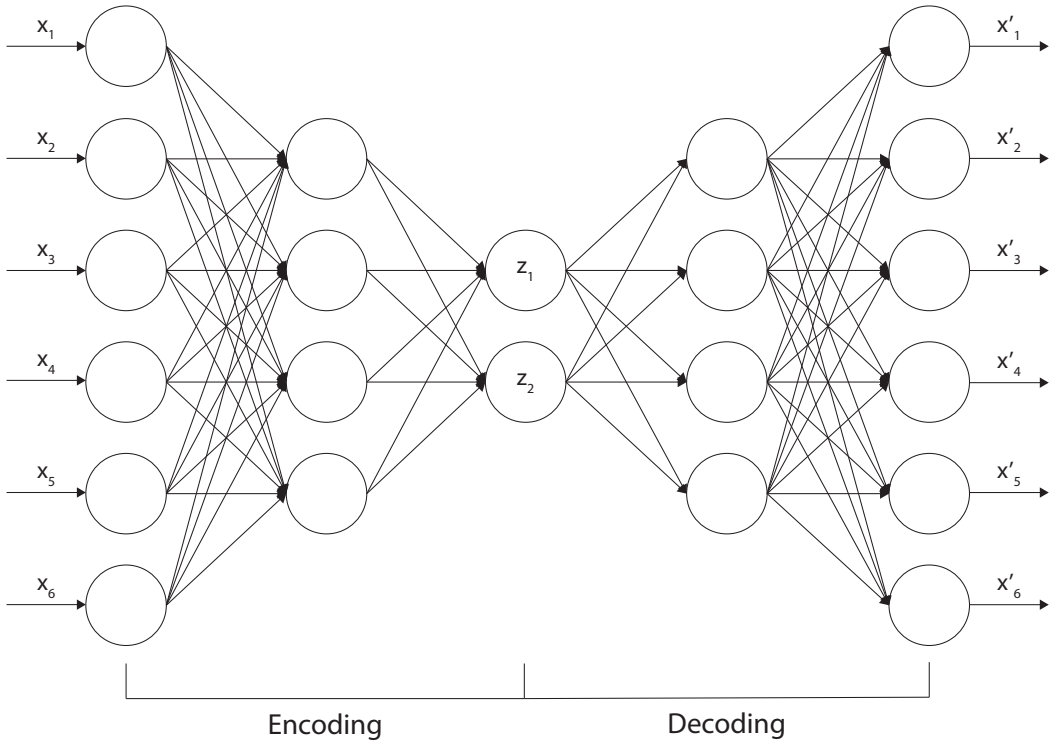


Figure 2.4: Overview of an Auto-Encoder.

addition to having multiple layer types, each layer has specific parameters that largely impact the effectiveness of the network.

Despite the widespread use of ANNs, their application to different tasks still faces limitations and/or challenges. The majority of the challenges are even more striking when working with DANNs. Next, we briefly enumerate them.

Topology definition – the search for adequate topologies that are able to solve specific problems is a hard and time-consuming task [249]. As mentioned above, there are several different DL approaches and architectures. Moreover, the performance of the networks is directly impacted by the chosen architecture and topology. To tackle this problem, some heuristics to aid the design of DANNs have been proposed (see, e.g., [259]). Additionally, approaches for the automatic search of adequate topologies (and often the weights) of ANNs resorting to EC have also emerged. These will be presented in Section 2.4.

Difficulty to interpret – ANNs are known for being black-box models, i.e., given an input they generate an output, but justifying why that output was generated is difficult. That is the reason why DANNs are not widely used in domains such as banking or trading, where a clear explanation for the action to be taken is often required. As such, there is the need for researching methodologies to identify and understand which blocks of a given DANN contribute to a decision, and what is their role in the taken decision.

Training – training an ANN for solving a complex task is time-consuming, possibly requiring a big dataset of labelled instances that is to be iterated for a large number of training

epochs. As above stated, an approach for finding adequate topologies for DANNs is to use heuristics. However, if the training of the DANN under optimisation is slow, the search process will take a long time to run, and consequently, the search for appropriate methods (or even heuristics) to efficiently train and assess the quality of an ANN is needed.

Consistency – one of the most important properties of ANNs is their generalisation ability, i.e., their capacity to obtain consistent results when presented with new and unseen data. Szegedy et al. [241] conducted experiments where noise is added to the images that were used to train the networks. They concluded that applying noise in a way to maximise the network's prediction error leads the networks to misclassify instances that they were previously capable of correctly labelling. Therefore, the challenge is how to efficiently train robust DANNs capable of coping with unseen instances and small perturbations of the original signals.

Bias-Variance dilemma [78] – bias and variance are two aspects that learning models aim at minimising. However, as two competing objectives, it is not possible to minimise both, and thus a trade-off between them must be established. Whereas bias is related to the errors caused by the assumptions that are made by the learning model, the variance is concerned with the errors caused by the sensitivity of the model. In other words, a high bias can cause under-fitting, leading the model to fail to capture the relationship between the observations and the target variable, and a high variance leads the model into learning the noise contained in the training data, potentially losing the generalisation ability.

Incremental learning – ANNs are usually developed and trained to solve a specific problem. Consequently, when we try to use the same network for tackling a different task the results will likely be far from the desired ones. More recently, works focusing modularity and the re-use of parts of previously trained models have been published [21, 46, 204]. These works allow learning different tasks in an incremental fashion, i.e., without losing previously acquired knowledge.

The enumerated limitations and challenges indicate as a promising research direction methods that automatically search for the best network configuration, and that promote the re-use of knowledge to solve incrementally distinct tasks. Next, we discuss approaches that automate the parameterisation of ML methods, focusing then on NE.

2.2 Automated Machine Learning

AutoML seeks to automate learning in the broad sense. The majority of ML methods require human intervention to effectively learn from the data, in a trial-and-error time-consuming process that requires data pre-processing, feature engineering, model selection, and model parameterisation (see Figure 2.5). AutoML can be seen as a sort of a black-box that, given the dataset, provides the best ML approach and its appropriate parameterisation.

The most common and widely used approach to optimise decisions and to hyper-parameterise ML is grid search: the best parameterisation of a ML model is discovered by an exhaustive search of all the combinations of a grid of parameters. Duan and Keerthi [56], and Min and Lee [175] use grid search to optimise the soft margin and kernel parameters of SVMs; Cortez [48] applies grid search to tune the hyper-parameters of ANNs. The main disadvantage of grid search is the curse of dimensionality, i.e., the explosion in the number of parameters drastically increases the number of setups that have to be tested. To deal with the previous, we can instead use grid search methods that seek to narrow the number of setups, for example, by adapting the

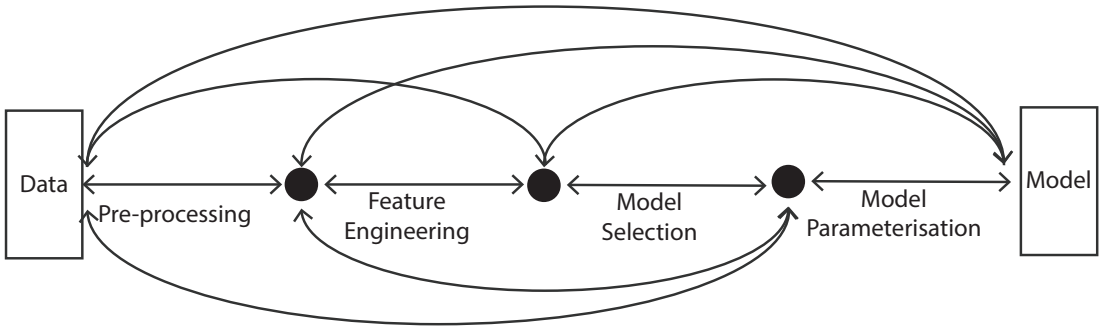


Figure 2.5: Human-designed ML: from the data to the tuned model.

resolution of the grid in run-time [113]. Nonetheless, grid search has the advantage that it is highly parallelisable.

To overcome the issue of exploring the entire grid of hyper-parameters, we may instead resort to random search. As the name suggests, while grid search performs an exhaustive enumeration of the search space, random search selects the combinations of the hyper-parameters in a stochastic manner. According to Bergstra et al., given the same computational time, random search is able to discover better parameterisations for ANNs than grid search [29, 30]. Random search is as parallelisable as grid search. On the other hand, it is not-adaptive [277], and with very high dimensional search spaces, it struggles to find near-optimal solutions.

Bayesian methods [178, 220] have the objective to model probabilistically the behaviour of the system to drive search towards regions of the domain that are prone to generate good parameterisations. Snoek, Larochelle, and Adams applied Bayesian optimisation to tune the parameters of the Branin-Hoo function, Logistic Regression, Online Latent Dirichlet Allocation (LDA), Latent Structured SVMs, and CNNs [225]. It was demonstrated by Bergstra, Yamins, and Cox that statistical methods can perform better than manual tuning or random search at hyper-parameter optimisation [31].

In addition to random and grid search, and to statistical methods, we have another class of heuristic that is often used to automate ML-related decisions, which is EC. A couple of examples include the application of EC to define the weights of the attributes of the samples for the k-Nearest Neighbors (k-NN) [114], and the tuning of the parameters of SVMs [44], or ANNs [271]. On the broader sense, we can consider general-purpose EC numerical optimisers as hyper-parameter optimisers, e.g., Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [99], that has already been successfully applied to the optimisation of SVMs [74] or DANNs [155].

The majority of the aforementioned AutoML methods focus on the optimisation of a specific ML model. Nonetheless, the ultimate goal of AutoML is to fully automate the entire process: from the data pre-processing, and feature design and selection, up to the model choice, and parameterisation. In other words, the goal is to automate the entire process of Figure 2.5. Recently, there have been competitions seeking to promote such systems, e.g., the ChaLearn competition [96, 97]. The challenge is organised into 6 increasingly difficult levels (preparation, novice, intermediate, advanced, expert, and master), where the ultimate goal is to "create the perfect black box eliminating the human in the loop" [97].

Weka [72] and Scikit-learn [194] are examples of two ML libraries that enable users to explore their data and easily deploy ML models. They make available stable implementations of

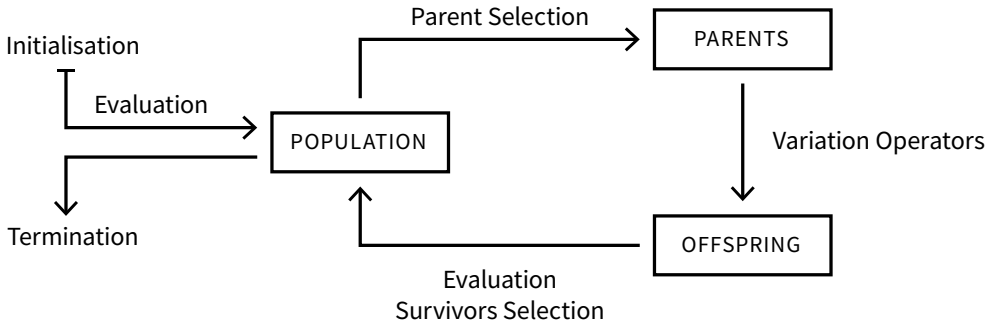


Figure 2.6: Flow-chart of Evolutionary Computation.

the vast majority of ML methods but, despite providing default parameterisation, they are not adequate to effectively solve all problems. Auto-WEKA [131, 246], Tree-based Pipeline Optimization Tool (TPOT) [187], Hyperopt-Sklearn [129, 130], Auto-Sklearn [65], Resilient Classification Pipeline Evolution (RECIPE) [213], and Monte-Carlo Tree Search for Algorithm Configuration (MOSAIC) [200] are examples of methods that aim at optimising the pipelines for the Weka and Scikit-learn libraries from the pre-processing of the raw data to the parameterisation of the model to be used. Except for TPOT, RECIPE, and MOSAIC all the previous methodologies are based on Bayesian optimisation; TPOT and RECIPE use GP; MOSAIC is based in Bayesian optimisation and Monte-Carlo Tree Search (MCTS) [128]. The goal is to search for Weka or Scikit-Learn pipelines, i.e., sequences of the libraries' primitives that perform feature selection and classification. These optimisation frameworks are not only responsible for selecting the primitives but also promote their parameterisation. Auto-Weka, Hyperopt-Sklearn, Auto-Sklearn, and RECIPE generate pipelines of fixed size. TPOT allows the generation of pipelines of unrestricted size, i.e., it does not have a fixed number of pre-processors, and multiple copies of the dataset can be used simultaneously so that multiple methods are applied to it, and then the features combined. The majority of these approaches target the maximisation of the classification performance. In addition, TPOT also seeks for compact pipelines.

The multiple optimisation methods pointed so far have advantages and disadvantages: (i) grid search allows a thorough analysis of the space of the hyper-parameters and is highly parallelisable, but on the other hand, it suffers from the curse of dimensionality, making it too expensive for scenarios where large amounts of choices and hyper-parameters have to be tuned; (ii) random search partially solves the curse of dimensionality issue of grid search, but it is still inefficient, as it fails to guide search towards feasible regions of the search space; (iii) on the other hand, statistical methods like Bayesian optimisation seek to capture and model the relations between the various hyper-parameters, making an informed exploration of the search domain; however, Bayesian optimisation makes the choice of the related parameters necessary (e.g., kernel); (iv) EC is capable of capturing the relations between the multiple hyper-parameters too, and it is highly parallelisable; it also requires the definition of hyper-parameters, which are easier to understand. Further details on EC can be found in Section 2.3.

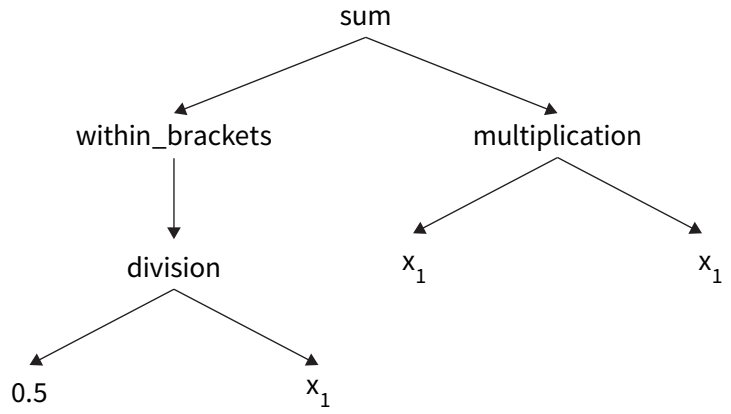


Figure 2.7: Example of a GP tree that encodes the phenotype (0.5 / x_1) + x_1 * x_1.

2.3 Evolutionary Computation

EC [32, 59, 69, 98, 136] comprises a set of computational models that draw inspiration on the natural-selection theory introduced by Charles Darwin [50]: those individuals most suited to the environment are more likely to survive and reproduce, and thus, have a higher chance of passing their characteristics to the offspring. Consequently, as time passes, individuals incorporate characteristics that make them better adapted to the environmental conditions.

The core of all evolutionary engines is the same. To start, an initial population of individuals is created. Individuals are possible solutions for the problem that is to be solved. Additionally, as in nature, there is a genotype which encodes the candidate solution, and a phenotype resulting from the mapping of the genotype into the domain space of the problem. Next, the quality of each candidate solution is assessed, and parents are probabilistically selected, as a function of their quality. At this point, the offspring is generated by applying variation operators to the selected parents, and the new population is created, by choosing, from the pool containing the parents and the offspring, which ones should pass onto the next generation. This process is repeated until a stop criterion is met.

A flow-chart of the main components of EC methods is depicted in Figure 2.6. Next, we detail each of the components.

Population – set of individuals that is evolved throughout the generations. There are two key-aspects when defining the population: (i) the representation, which, as a rule of thumb, should be the encoding scheme that matches the given problem best (e.g., binary or real vectors, graphs, or trees); and (ii) the population size (number of evolved individuals) – a large population allows for a greater diversity but may increase the evaluation time.

Evaluation – determines the fitness of an individual, usually by assessing how well an individual solves the problem at hand and, as such, enables the comparison of the candidate solutions.

Parent selection – concerns the process of choosing the **parents**, i.e., the individuals that will generate the **offspring**. The selection pressure defines the ratio between the likelihood of selecting high and low fitted individuals. On the one hand, high selection pressures reduce diversity. On the other hand, a low selection pressure approximates random search and thereby slows down evolution.

$$
\begin{aligned}
\text{<start>} &::= \text{<expr> <op> <expr>} && (1)\\
&\mid \text{<expr>} && (2)\\
\text{<expr>} &::= \text{<term> <op> <term>} && (3)\\
&\quad (\ \text{<term> <op> <term>}\ ) && (4)\\
\text{<op>} &::= + && (5)\\
&\quad - && (6)\\
&\quad / && (7)\\
&\quad * && (8)\\
\text{<term>} &::= x_1 && (9)\\
&\quad 0.5 && (10)
\end{aligned}
$$

Grammar 2.1: Example of a Context-Free-Grammar. This grammar will be used throughout the remainder of this chapter to illustrate representation schemes and decoding procedures of the various grammar-based methods.

Variation operators – generate the offspring based on the chosen parents. The two most common variation operators are: (i) recombination, which normally receives two parents as input and recombines their genetic material, giving origin to new individuals that should inherit characteristics from both parents; and (ii) mutation, that takes a single parent and applies small changes to it, possibly introducing new genetic material.

Survivors selection – selects the individuals that pass to the next generation. The population size is often kept fixed during evolution. The choice of which individuals proceed evolution is based on a given criterion, such as, fitness (e.g., the fittest individuals move to the next generation), or age (e.g., the offspring seeds the next generation). When the selection of the survivors is based on the age, we can use elitism to keep a percentage of the best solutions between generations.

Initialisation – creates the genetic material that will seed the first generation. When there is a-priori knowledge about the problem, it can be incorporated into the initial individuals. Otherwise, the initial population should be generated in a random and unbiased fashion.

Termination condition – defines the criterion that halts evolution. When the solution to the problem is known, the stop condition can be defined as reaching it or reaching a solution that is within a defined margin of error of the target solution. However, in many problems, the optimal is not known, or it is not possible to guarantee that it will be achieved in an acceptable time-frame. Therefore, we need to define other stop conditions, such as the maximum number of generations or evaluations.

In the current Thesis, we focus on GP: the set of EC methods that evolves executable structures. While, in most EC methods, the solutions to the problem correspond to the phenotypes of the individuals, in GP, the solutions arise from executing the evolved programs. To solve a problem using GP, the set of primitives that can be used by the method must be defined, i.e., the problem is set at a higher-level, stating what needs to be done, but not how it should be accomplished. Then, evolution is guided by assessing how well an individual solves the problem. To measure the quality of the individuals, the programs are executed, and fitness is computed as a function of the output of the programs

Table 2.1: Derivation steps followed to obtain $(0.5/x_1)+x_1{*}x_1$. We consider the production rules of Grammar 2.1, and the axiom start.

Derivation step
<start>
<expr> <op> <expr>
(<term> <op> <term>) <op> <expr>
(0.5 <op> <term>) <op> <expr>
(0.5 / <term>) <op> <expr>
(0.5 / x_1) <op> <expr>
(0.5 / x_1) + <expr>
(0.5 / x_1) + <term> <op> <term>
(0.5 / x_1) + x_1 <op> <term>
(0.5 / x_1) + x_1 * <term>
(0.5 / x_1) + x_1 * x_1

The most widely used form of GP was introduced by Koza [136]. Individuals are encoded as trees where inner nodes represent functions (i.e., non-terminal symbols), leaves stand for terminals, and the connections between different nodes denote the flow of control and passing of parameters. Solutions are obtained by depth-first in-order traversing the tree. Traditionally, to avoid the generation of invalid solutions, non-terminal symbols were required to accept any node as input. Later, typed nodes were defined. As an example, consider the set of functions: sum, subtraction, division, multiplication, within_brackets, and the terminals x_1 and 0.5. The arity of all functions except the within_brackets function is 2. The arity of the within_brackets function is 1. Figure 2.7 represents a tree that encodes the phenotype (0.5 / x_1) + x_1 * x_1.

Several other types of GP algorithms are found in the literature. Brameier and Banzhaf [36] introduce linear GP, where each individual in the population is a sequence of executable instructions. Poli [195] proposed the representation of the individuals using graphs, and later Miller introduced Cartesian Genetic Programming (CGP) [173]. A drawback that is common to all the GP types pointed so far is that their use is problem-specific, i.e., to apply the approaches to different domains, there is the need to define how should the problem solutions be mapped to the method's representation. Therefore, from this point onwards, we will focus on GP algorithms that represent the individuals based on grammatical formulations, and therefore are easy to apply to different domains and problems. In Section 2.3.1, we detail grammatical formulations, followed by GGP methods (Sections 2.3.2 and 2.3.3).

2.3.1 Grammars

Context-Free Grammars (CFGs) are rewriting systems that can be formally defined by a 4-tuple, $G = (N, T, P, S)$, where: (i) N is the set of non-terminal symbols; (ii) T is the set of terminal symbols; (iii) P is the set of production rules of the form $x ::= y$, $x \in N$ and $y \in \{N \cup T\}^*$; and (iv) S is the start symbol (or axiom). Additionally, a grammar (G) defines a language $L(G)$, i.e., all sequences that can be obtained starting with the initial symbol and recursively applying the production rules. Because production rules often have more than one non-terminal symbol, it is common to replace, in each derivation step, the leftmost non-terminal symbol by its expansion.

An example of a CFG designed to encode symbolic regression expressions is shown in Grammar 2.1. The CFG is composed by 4 non-terminal symbols (start, expr, op, and term), 6 terminal symbols (+, -, /, *, x_1, and 0.5), 4 production rules, and the axiom is start. An example of a sequence that can be derived from the example grammar is (0.5 / x_1) + x_1 * x_1. This sequence

Table 2.2: Example of the mapping procedure of GE. Each row represents a derivation step. We used the production rules of Grammar 2.1.

Derivation step	Genotype	Modulo
<start>	[20, 115, 17, 67, 200, 140, 20, 84, 95, 254, 10]	20 % 2 = 0
<expr> <op> <expr>	[115, 17, 67, 200, 140, 20, 84, 95, 254, 10]	115 % 2 = 1
(<term> <op> <term>) <op> <expr>	[17, 67, 200, 140, 20, 84, 95, 254, 10]	17 % 2 = 1
(0.5 <op> <term>) <op> <expr>	[67, 200, 140, 20, 84, 95, 254, 10]	67 % 4 = 3
(0.5 / <term>) <op> <expr>	[200, 140, 20, 84, 95, 254, 10]	200 % 2 = 0
(0.5 / x_1) <op> <expr>	[140, 20, 84, 95, 254, 10]	140 % 4 = 0
(0.5 / x_1) + <expr>	[20, 84, 95, 254, 10]	20 % 2 = 0
(0.5 / x_1) + <term> <op> <term>	[84, 95, 254, 10]	84 % 2 = 0
(0.5 / x_1) + x_1 <op> <term>	[95, 254, 10]	95 % 4 = 3
(0.5 / x_1) + x_1 * <term>	[254, 10]	254 % 2 = 0
(0.5 / x_1) + x_1 * x_1	[10]	–

is obtained following the derivation steps of Table 2.1.

2.3.2 Grammatical Evolution

The key-advantage of GGP over other GP methods is flexibility, which lies in the fact that the search space is defined through a grammar, i.e., it is set in a human-readable format, that enables non-experts to use the method. One of the most widely-used GGP methods is GE. Other forms of GGP exist, and a complete survey can be found in [168]. In particular, the readers interest in variants of GE is redirected to [212].

GE was proposed by Ryan et al. [186, 211], and is a GGP method that has an indirect encoding scheme, where the genotype is separated from the phenotype. More precisely, in the original GE proposal, the genotype is a linear and ordered sequence of bits, where each group of 8 bits encodes an integer that is later used in the genotype to phenotype mapping. The current approach to GE uses the integers directly, and each integer is called a codon.

To map the genotype to the phenotype, the codons are read sequentially from the left to the right. Starting from the axiom, the mapping procedure iteratively decides which production rule should be applied to expand the leftmost non-terminal symbol. To select the production rule, we compute the modulo (%), i.e., the remainder after the division of the codon by the number of possibilities for expanding the leftmost non-terminal symbol. The remainder defines the expansion possibility, that should be applied to the leftmost non-terminal symbol. When there is only one possibility for expanding a non-terminal symbol, no codon is read. On the other hand, grammars can be recursive, and thus the number of codons may be insufficient. When this occurs, the sequence of codons is re-used from the start, a technique known as wrapping. To avoid entering an infinite wrapping loop, or generating solutions that are too complex to be evaluated, we define a maximum number of wrappings. If the maximum number of wrappings is reached, the mapping procedure is halted, and the individual is assigned the worst fitness value.

Consider that we have the genotype [20, 115, 17, 67, 200, 140, 20, 84, 95, 254, 10], the productions rules of Grammar 2.1, and that the axiom is start. Therefore, with this genotype the mapping procedure generates the phenotype (0.5 / x_1) + x_1 * x_1, following the steps enumerated in Table 2.2. The axiom is the start non-terminal symbol, which has 2 expansion possibilities. The first codon is 20, and because 20%2 is 0, we select the first expansion possibility, which rewrites <start> as <expr><op><expr>. This re-writing procedure is repeated until we are left with no more non-terminal symbols to expand. After the mapping procedure, as in any other form of GP, to evaluate the quality of the individual, the generated structure must be executed.

Table 2.3: Example of the mapping procedure of SGE. Each row represents a derivation step. We have used the production rules of Grammar 2.1. The sub-sequences of the genotype encode the expansion possibilities of the start, expr, op, and term non-terminal symbols, respectively.

Derivation step	Genotype
<start>	[[0], [1, 0], [2, 0, 3], [1, 0, 0, 0]]
<expr> <op> <expr>	[[], [1, 0], [2, 0, 3], [1, 0, 0, 0]]
(<term> <op> <term>) <op> <expr>	[[], [0], [2, 0, 3], [1, 0, 0, 0]]
(0.5 <op> <term>) <op> <expr>	[[], [0], [2, 0, 3], [0, 0, 0]]
(0.5 / <term>) <op> <expr>	[[], [0], [0, 3], [0, 0, 0]]
(0.5 / x_1) <op> <expr>	[[], [0], [0, 3], [0, 0]]
(0.5 / x_1) + <expr>	[[], [0], [3], [0, 0]]
(0.5 / x_1) + <term> <op> <term>	[[], [], [3], [0, 0]]
(0.5 / x_1) + x_1 <op> <term>	[[], [], [3], [0]]
(0.5 / x_1) + x_1 * <term>	[[], [], [], [0]]
(0.5 / x_1) + x_1 * x_1	[[], [], [], []]

To promote evolution, GE applies single point recombination and point mutation. Briefly, the recombination operator selects a cutting point from each of the parents and swaps the genetic material. Point mutation probabilistically replaces codons for other valid ones (integers or bits depending on the used representation).

2.3.3 Structured Grammatical Evolution

SGE, proposed by Lourenço et al. [156, 159], is an extension to GE that defines a new genotypic representation. The main objective of SGE is to overcome two key-issues of GE: low locality and high redundancy [117, 245]. The locality measures how the changes in the genotype impact the phenotype. In GE, the genotype is a linear sequence of codons, where there is not a one-to-one mapping between the codons and the non-terminal symbols. Therefore, it is easy for a change in one of the codons to affect many derivation steps from that point onwards (low locality[1]). On the other hand, the redundancy is concerned with the fact that, in GE, different genotypes may generate the same phenotype. This is a result of the modulo operation that is used in the decoding procedure, e.g., consider that a non-terminal symbol has 4 expansion possibilities then, either the codon 20 or 40 lead to the same derivation step (20%4=40%4=0).

To overcome the low locality and high redundancy issues of GE, the new genotypic representation of SGE defines a one-to-one mapping between the codons and the non-terminal symbols. Instead of a single ordered sequence of codons, the genotype is composed of multiple independent ordered sequences of codons, one for each non-terminal symbol. The size of each sequence of codons is of the maximum number of possible expansions for the non-terminal symbol it encodes, and thus there is no need for wrapping. The use of the modulo operation is not required as we know exactly which non-terminal symbol the codon represents, and therefore the codons are constrained to values in the interval $[0, \text{max\_expansions}_{\text{non\_term}} - 1]$, where $\text{max\_expansions}_{\text{non\_term}}$ sets the maximum number of expansion possibilities for the non-terminal symbol.

The genotype encoding of SGE assumes that the sequences have enough codons to derive the largest possible phenotype. To that end, we need to pre-process the grammar to set the maximum number of expansions of each non-terminal symbol, i.e., the maximum size of each sequence of codons. When the grammar has recursive production rules, they are replaced by new

[1]In EC the locality measures the impact a small change in the genotype has in the phenotype: ideally when changing a minor fraction of the genotype (e.g., a bit), we want the phenotype to be only slightly different, and with binary coding for example whilst 0000 represents the decimal 0, 1000 represents the decimal 8.

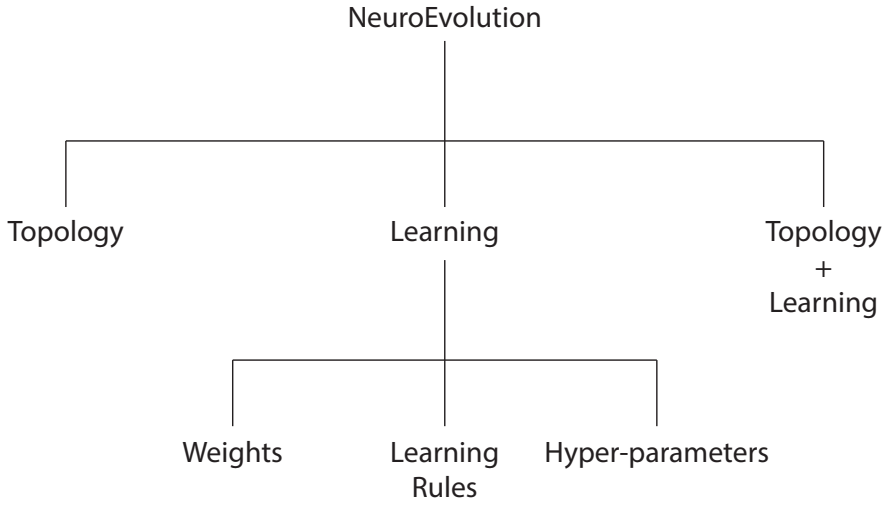


Figure 2.8: Overview of the taxonomy of previous surveys on NeuroEvolution.

production rules, which are no more than the recursive production rule replicated a pre-defined number of times, but where the last level has no calls to another recursive level. This procedure mimics the behaviour of tree-based GP, where a maximum tree-depth is defined.

To establish a comparison to the mapping procedure of GE we derive $(0.5 \ / \ x_1) + x_1 * x_1$ using SGE. We consider the genotype [[0], [1, 0], [2, 0, 3], [1, 0, 0, 0]], and Grammar 2.1. The derivation steps are described in Table 2.3, and clarify how the codons are consumed from the sequences of each specific non-terminal symbol.

The variation operators are point mutation and uniform crossover. The mutation operator, despite the same as in GE, has different constraints on the new generated codon, that is randomly chosen within $[0, \text{max\_expansions}_{\text{non\_term}} - 1]$. The crossover swaps sets of codons, i.e., all the codons of the non-terminal symbols are swapped between the parents according to the bit-mask.

The new representation scheme of SGE overcomes the locality limitation of GE by introducing a one-to-one mapping between the codons and the non-terminal symbols. In addition, as the codons are associated with specific non-terminals, we no longer require the modulo to decode the genotype, and thus SGE reduces redundancy. These advantages result in better evolvability, as demonstrated in Lourenço [156]. Consequently, we have chosen SGE over GE to conduct the first experiments of the current Thesis.

2.4 NeuroEvolution

NE is a sub-field of AutoML that refers to the automatic optimisation of ANNs by means of EC. The gains are three-fold: (i) in terms of learning, gradient-descent algorithms (e.g., BP) are limited to optimising differentiable functions, but EC can optimise any function, as long as we can measure improvement through the fitness function; (ii) NE can be applied to generate ANNs for any problem; and (iii) it is a general-purpose tool to search for any type of ANN, able to optimise all the ANN's aspects: architecture, and learning parameters; we only need to adapt the representation scheme to deal with the search space of the type of ANNs we want to generate.

The majority of the seminal surveys in the area of NE (e.g., [67, 214, 271]) group the works according to the aspects of the networks that are the target of optimisation: evolution of the

weights and/or any other learning parameters, evolution of the architectures, or simultaneous evolution of both the learning and architecture (see Figure 2.8). In addition, the weight representation schemes tend to be divided into binary or real representations, and the architecture representation into direct, developmental, or implicit. Baldominos, Saez, and Isasi [24] focus on NE for optimising CNNs. Nonetheless, similarly to previous works, the authors also survey the fundamental works of NE. Risi and Togelius [206] survey NE applied to games. More recently, Stanley et al. [230], and Galván and Mooney [77], in addition to surveying the field's seminal works, also explore the most recent trends and challenges, such as the scaling of NE to DANNs.

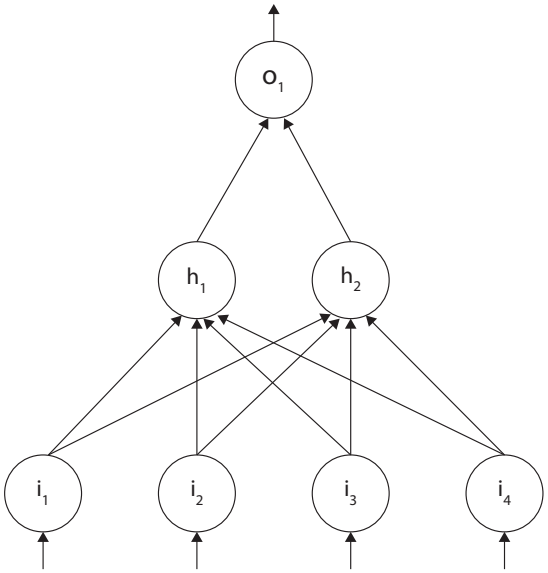
The key-points when applying EC to the optimisation of ANNs are the representation of the candidate solutions and the fitness function. The representation must encode the parameters one seeks to optimise, such as the number of neurons, their connectivity patterns, weights, and activation functions, or the learning parameters. The evaluation of the networks that are found throughout generations is demanding, not because it is hard to implement[2], but due to the time that is needed for assessing the quality of the networks represented by the candidate solutions. On the one hand, we can use the evolutionary process to our advantage and sample, from generation to generation, a given percentage of the dataset (few data translates into a faster evaluation) [122, 183, 233]. On the other hand, we can compress the dataset, reducing its dimensionality [5]. Nonetheless, to tackle the burden of the training time, the majority of NE approaches impose a low maximum number of training epochs [171, 236, 238]. It is important to mention that the evaluation of the candidate solutions is not a problem when evolution considers the optimisation of the connection weights, as the optimised values are used to measure the effectiveness of the network. However, it is clear that when we want to optimise both the topology and learning, the deeper the network the more challenging the task is. In line with the aforementioned, in the current section we group the NE works according to their ability to cope with small (Section 2.4.1) or large scale ANNs (Section 2.4.2). Next, we introduce the fundamental principles and seminal works, and thus we follow the structure of previous NE surveys.

2.4.1 NeuroEvolution for Small-Scale Networks

To the best of our knowledge, the first works on NE date back to 1989, with the application of Genetic Algorithms (GAs) to the optimisation of the weights (and biases) of feed-forward ANNs using binary [262] and real encodings [179]. The architecture of the network is specified a-priori, with a one-to-one mapping between the synaptic weights and the binary (or real) values that are the target of evolution (see Figure 2.9). EC replaces the BP algorithm, overcoming the possibility of the gradient-descent algorithm to become stuck in local optima. Further, such learning algorithms require differentiable error functions, which EC does not [240]. Many other approaches that follow the same principles are found in the literature (e.g., [264]). From the example of Figure 2.9, it is perceivable that the precision of the weights is related to the number of bits used to encode each real value. The more bits we use the higher the precision, but the slower the evolution is. Besides, in binary representations, the variation operators can drastically change a candidate solution, making it more difficult to promote a smooth evolution (low locality). Whitley, Starkweather, and Bogart [263] apply GENITOR to the optimisation of the connection weights using binary and real representations and conclude that, in the tested benchmarks, the binary and real-representations generate optimised sets of weights, but the real encoding converges faster.

Cooperative Synapse NeuroEvolution (CoSyNE) [81] evolves each weight in a separate sub-

[2]To implement the evaluation procedure of the candidate solutions one can map them onto trainable models using well-documented libraries, such as, Scikit-learn [194], Caffe [112], Tensorflow [1], Theano [244], Keras [42], or DeepLearning4J [58].



From	To	Binary	Real
i_1	h_1	0 100 0011	4.1875
i_1	h_2	1 010 0001	-2.0625
i_2	h_1	0 011 0000	3.0
i_2	h_2	0 001 1000	1.5
i_3	h_1	0 100 1100	4.75
i_3	h_2	1 010 1110	-2.875
i_4	h_1	0 000 0010	0.125
i_4	h_2	0 111 0101	7.3125
h_1	o_1	1 100 0111	-4.4375
h_2	o_1	0 010 1111	2.9375

Figure 2.9: Encoding of the weights of a fully-connected feed-forward ANN using binary and real representations. The binary representation uses 8 bits for each value, where the most significant bit encodes the signal (1 for negative numbers, and 0 for positive), the next 3 bits encode the integer, and the remaining bits the floating part; this representation can encode real values in the interval $[-7.9375, 7.9375]$. The individual in the binary representation would then consist of a bit-string of size 80 (10 connections $\times$ 8 bits), and in the real representation would be made of 10 real values.

population. The genotype can be interpreted as a $n \times m$ matrix, where n is the number of synaptic weights of the network (i.e., the number of sub-populations), and m is the size of each sub-population. Each network is then formed by evaluating each row of the matrix and mapping the weights to the corresponding connections.

The previous works apply EC to replace the use of an established learning algorithm. In addition, some methods combine the evolution of the weights with well-established learning methods, where EC is used to generate the initial set of weights. Sedki, Ouazar, and Mazoudi [217], and Wang, Zeng, and Chen [261] develop NE methods that generate sets of weights that after evolution are used to seed the network that is further trained using BP. Another option is to develop hybrid methods that combine evolution with learning algorithms, i.e., in each generation the candidate solutions are trained for a given number of epochs using a learning algorithm, and then the learned characteristics (in this case the weights tuned by the learning algorithm) can be passed to the offspring (Lamarckian [141] evolution), or not (Baldwinian [25] evolution). In both Lamarckian and Baldwinian evolution the objective is to promote the learning ability, but only in Lamarckian evolution the learned characteristics are passed to the next generations. Lamarckism and Baldwinism have successfully been applied to training ANNs with NE [37, 54, 63, 92, 103, 139, 192], not only for the optimisation of the weights but also the structure. Valdivieso et al. [255] compare these hybrid approaches applied to the evolution of ANNs and conclude that MLPs trained with them generate lower average error than when trained with other learning methodologies (evolutionary or not). The same work concludes that the networks generated by Baldwinian evolution tend to be larger and report smaller errors than those generated by

Lamarckian evolution.

NE approaches focusing learning are not constrained to the direct optimisation of the connection weights, which is particularly difficult when the networks have thousands or even millions of connections. There are also works reporting on the tuning of the parameters of well-established learning algorithms. For example, the application of EC to the optimisation of the hyper-parameters of BP and its variants [123, 193]. Alternatively, we can evolve the rules that are used by the learning algorithms [26, 199]. Instead of evolving rules that iteratively tune the weights, some authors address the search for functions that given the position of two nodes in an ANN return the corresponding connection weight. HyperNEAT [231] applies NEAT (further discussed next) to the evolution of Compositional Pattern Production Networks (CPPNs): ANNs that seek to capture the regularities in the structure of the network that is to be trained[3]. However, the same can be achieved by simply evolving a function with tree-based GP that replaces the CPPN. Buk, Koutník, and Šnorek [38] used this approach, and the results have proven to be comparable to those reported by HyperNEAT.

The problem when targeting only the network's learning is that the topology is kept fixed. According to Turner and Miller, "the choice of topology has a dramatic impact on the effectiveness of NE when only evolving weights; an issue not faced when manipulating both weights and topology" [254]. The authors also state that it may be more beneficial to search for an adequate topology rather than weights. One of the first works on the optimisation of the architecture of ANNs was proposed in 1989, by Geoffrey Miller et al. [172] – innervator – where the structure of the networks is represented using a connectivity matrix where each cell (i, j) represents if the i-th neuron is connected to the j-th neuron. The matrix is evolved as a bit-string: a bit set to 1 denotes a connection. The connection weights of each candidate solution are optimised using BP, over a defined number of epochs. A different approach is introduced by Kitano [127]: as in innervator [172] the network structure is encoded by a connectivity matrix, and the candidate solutions are trained using BP. The difference is that instead of encoding the matrix as a binary bit-string, Kitano develops an L-system to generate it (indirect encoding). On the tested problems (N-M-N encoder/decoder), the representation used by Kitano outperformed the direct encoding. The representation at the level of the connectivity matrix has one main issue: before evolution, there is the need to set an upper bound for the number of neurons (which sets the connectivity matrix size). Other more recent approaches seek to solve this by encoding the networks at the neuron's level, e.g., Rocha, Cortez, and Neves propose Topology-optimisation Evolutionary Neural Network (TENN) [207]: an approach for tuning the number of hidden-nodes and connections of MLPs; Soltanian et al. introduce GE-BP [228], a grammar-based approach based on GE to optimise the same aspects of the network's as previous methods.

The application of EC to the evolution of the topology of the ANNs without the evolution of the weights requires training the networks using another learning strategy. From the works listed above, one can conclude that it seems to be a common practice to train the candidate solutions with BP. Nonetheless, it is not trivial to set up the hyper-parameters of BP that work well on a wide range of the topologies that are generated throughout evolution. Mandischer [162, 163] structures the candidate solutions in a network parameter area, where the learning parameters are encoded (learning rate and momentum), and a sequence of layers area, that is responsible for keeping the input and output connections, and the layer size. Jung and Reggia [115] propose an indirect representation scheme that is based on an encoding language, with the layer as the basic unit, and where it is easy to set what parts of the network are to be evolved, and which ones are kept fixed during evolution. This way, the learning parameters (if unknown) can be easily optimised. This encoding mechanism enables the introduction of a-priori knowledge by defining non-optimisable parts of the network.

[3]NEAT as a NE approach is adequate for evolving CPPNs.

The previous works tune hyper-parameters of the learning algorithm for each network. The problem is that the training of each candidate solution may be a long and costly process, that even aided by GPU computing can be time-consuming. On the other hand, as previously mentioned, to apply gradient-descent learning methods, we need the activation functions to be differentiable, and even then, we are likely to get trapped in local optima. That is the motivation many practitioners argue for evolving both the network topology and weights simultaneously when optimising small-scale ANNs. There are several examples of approaches of this type [68, 207, 278]. Next, we will discuss the ideas behind the most influential works on NE for the simultaneous evolution of the structure and weights of ANNs.

Artificial Neural Networks Adaption: Evolutionary Leaning Of Neural Optimal Running Abilities (ANNA ELEONORA) [164] addresses the problem of optimising ANNs using a linear bit-string where each set of bits encodes the connection between a given pair of nodes, i.e., despite evolving the connectivity of the network (and respective weights), the number of hidden-nodes is not tuned. One of the key points of ANNA ELEONORA is that the granularity (number of bits for representing each weight) is a property of each candidate solution, and optimised during evolution. For each candidate solution, the first byte encodes the granularity, followed by at most $1+g\times n\_nodes$ bits, where g represents the granularity, and n_nodes the number of hidden-nodes of the network. The first bit represents if there is a connection between the pair of nodes or not; when there is not a connection, the bits for encoding the weight are not used. The granularity solves the problem of defining the ranges of the weights on binary encodings (previously discussed in Figure 2.9). However, there is a trade-off: high granularity values slow evolution and small values may be insufficient to find effective weights. Further, ANNA ELEONORA does not enable free exploration of the search space as it requires the number of hidden-nodes to be set. Generalised Acquisition of Recurrent Links (GNARL) [6] solves the previous issues by encoding the weights as real-values and encoding the networks at the node's level. It has the particularity of proposing search operators that preserve the behaviour of the network, thus promoting a smooth evolution. The initial random networks have their weights changed by a Gaussian mutation, and nodes and connections may be added without changing the behaviour of the network: nodes are added without connections, and connections are added with an initial weight of zero. The removal of a node is not that smooth because all the connections from and to that node are removed. A similar strategy is followed by EPNet [272], where the mutations are designed to smooth evolution. EPNet makes use of the principles of Lamarckian evolution where, from one generation to the next, the candidate solutions are trained using a variant of BP. If the train of the network is successful (i.e., if the performance of the ANN improves), no mutation is applied. Otherwise, the candidate solution is sequentially mutated from pruning to constructing, until one of the mutation operators is successful, i.e., the mutations that are first applied remove nodes and connections, and only the last mutation operator creates new nodes and/or connections. By the end of evolution, the best performing network is further trained.

Cellular Encoding [90, 91] is a developmental system by Gruau et al., that is inspired by the biological process of cell division. Each cell represents a neuron of the network, and a set of operators divide the cell to form others and establish the corresponding connections, forming the entire network. For example, sequential division splits one cell into two, where the first gets the input from the initial cell and connects to the second cell that inherits the output link from the original cell. Further details about the operators used to manipulate the cell division can be found in [90].

At a higher level of abstraction, ANNs are directed graphs, and thus the application of graph-based methods to their evolution is promising. Pujol and Poli apply Parallel Distributed Genetic Programming (PDGP) [195] to the evolution of ANNs [198]. In PDGP, the nodes of the graph are placed in a fixed-size two-dimensional grid, which thus enables the evolution of

layered structures. With the same rationale numerous works report the application of CGP [174] to ANNs [3, 120, 121, 165, 253]. In addition to the evolution of the structure and weights of the ANNs, CGP allows the optimisation of the activation function of each node (which in the previous approaches is not target of evolution). The nodes can get their inputs from previous layers (up to a defined threshold of levels back). Similarly, linear GP can also be adapted to evolving multi-output networks, such as AEs [167].

One of the problems commonly pointed-out to such type of evolutionary procedures is that they are tailored to solving very specific tasks, and adapting them to different network structures and/or to the resolution of other problems is hard. Grammar-based approaches solve this issue by requiring the user to change only the grammar so that the system is able to explore the search space that encompasses the desired network topologies and respective parameterisation; it also eases the incorporation of domain-specific knowledge [168]. Tsoulos et al. [251] evolve the weights and topology of one-hidden-layered networks with GE. Because of the locality issues of GE it is argued that it underperforms in the optimisation of the weights, and for that reason, Ahmadizar et al. apply GE to optimise only the topology, and the weights are evolved using a GA [4].

A different paradigm is used in Symbiotic, Adaptive NeuroEvolution (SANE) [180, 181, 182]. Whereas in the above approaches, each individual is responsible for encoding a complete ANN (a set of nodes, connections, and weights) and the fitness is the performance on a given task, in SANE a population of neurons is evolved (each individual represents a neuron with its incoming and outgoing connections). A fixed defined number of neurons forms a network and the fitness of each individual is the average quality of the networks it participates in. The idea behind SANE is that the population maintains diversity, but neurons specialise differently, providing SANE with implicit niching. Enforced Subpopulations (ESP) [80, 82] further extends SANE by explicitly creating a separate population for each neuron. This speeds up evolution by allowing the individuals to recombine within the sub-population, with neurons that share the same characteristics. The previous facilitates the evolution of recurrent networks.

One of the most well known NE approaches is NEAT, introduced by Stanley and Miikkulainen [232]. While the initial population of other NE approaches is formed at random, NEAT starts evolution from minimal structures with the inputs directly connected to the outputs (and no hidden nodes). This allows the generation of minimal structures without the need to incorporate a penalisation term in the fitness function (e.g., number of nodes, connections, or trainable parameters). The genotype grows by adding nodes and/or connections. To smooth evolution, when a node is added an existing connection is replaced by two new ones: from the input of the connection that is replaced to the new node, and from the new node to the output of the replaced connection (the connection that is replaced is not removed but rather deactivated); new connections are established between non-connected nodes. One of the problems that often occurs when dealing with solutions where slight structural changes on the phenotype highly affect the performance is that during evolution, when new parts are added to the network, their parameterisations are not yet optimised, and thus are penalised compared to the remaining population solutions. To overcome this issue, NEAT introduces historical markings: a global innovation number that keeps track of the structural changes. The crossover operator only changes genes that have the same historical marking. To protect innovation the population is divided into niches, and all the individuals within the same niche are given the same fitness (fitness sharing). Based on the initial NEAT formulation, multiple variants have been proposed: Schrum and Miikkulainen [215], and Künzel and Meyer-Nieberg [140] describe multi-objective versions of NEAT; as mentioned above HyperNEAT [231] is the application of NEAT to the generation of CPPNs; Reisinger et al. [204] extend NEAT to the evolution of modular networks; Miikkulainen et al. [171] change the paradigm and tackle the generation of DANNs (this last work will be further discussed in the upcoming sub-section).

To sum up, the literature on NE approaches is large and continuously increasing. The evolutionary goals vary: (i) weights and/or learning parameters; (ii) structure; or (iii) simultaneous evolution of the weights and/or learning parameters and structure. A wide set of EC methods have been applied to the evolution of ANNs: from standard GAs, to ESs, Evolutionary Programming (EP), and GP. There are approaches where all the information is directly encoded in the genotype, and approaches that specify rules for constructing the phenotype (developmental approaches). It is also possible to group the different methods according to the basic unit of evolution: (i) connection; (ii) node; or (iii) layer-based methods. Whereas the first allows a finer level of detail, layer-based methods encode the networks at a macro-level. The question concerns what type of networks we are trying to evolve, more precisely, what is the depth needed for solving the task we have at hand. When we need deep networks, with hundreds of nodes and thousands of connections, it is inefficient to evolve them at the level of the connection and node. As will be evident during the upcoming section, the majority of NE approaches that target the evolution of DANNs are layer-based.

2.4.2 NeuroEvolution for Deep Learning

The works surveyed in the previous section were designed mainly targeting the evolution of small-scale networks for addressing (nowadays) easy to solve problems, e.g., XOR, n-bit odd parity, double pole balancing, boolean functions, or benchmarks from the UCI ML repository [55]. Such networks tend to require few neurons, and consequently, a low number of connections. Thus, the application of these techniques to the evolution of DANNs is hard: the search space and the necessary solutions are more complex, i.e., in addition to optimising the weights, neurons, and connections, we often need to choose between different types of layer. Another problem is related to the dimensionality of the dataset instances which, when considering real-world problems, is often large. Large dataset instances, associated with a high number of instances, make the evaluation of the candidate solutions slow, and thus new strategies are necessary to cope with the evolution of DANNs for real-world problems.

We start by enumerating works that target the automatic training of DANNs. Koutník et al. extended CoSyNE to the optimisation of the weights of large-scale networks [133, 134]. Instead of directly evolving the real-values of the connection weights, they propose the evolution of Discrete Cosine Transform (DCT) coefficients with CoSyNE. An extra step is necessary to evaluate the candidate solutions: each row encodes a DCT coefficient matrix that using the inverse DCT is mapped to the weight matrix. The authors refer to this encoding scheme as a compressed representation because the number of DCT coefficients needed to encode the weights is much lower than when using the direct encoding. For example, in [132], with 20 DCT coefficients, it is possible to encode networks with about 3000 weights. Such encoding schemes seek to evolve large sets of weights using indirect representations that reduce the size of the target of evolution. As the reader may recall, this is somewhat close to the rationale behind HyperNEAT (and similar methods), where the goal is to find spatial relationships on the weight patterns and generate functions able to output the appropriate weights for the synaptic connections. Verbancsics and Harguess [256] adapt HyperNEAT to image classification and demonstrate that despite the lack of competitive results in training models for classification tasks, HyperNEAT can effectively generate feature extractors, that are later feed to another ML model. Fernando et al. [62] replace the CPPNs by Differentiable Pattern Producing Networks (DPPNs), i.e., only the topology is evolved, and the weights learned (with Lamarckian evolution). DPPNs have proven effective in compressing the weights of DAEs: 157684 parameters mapped to about 200. Hausknecht et al. [100] compare several NE methods (CMA-ES, NEAT, and HyperNEAT) in Atari game playing, and conclude that indirect methods scale better to higher-dimensional inputs

than direct methods. An indirect weight encoding scheme is also proposed by Such et al. [235], who suggest the use of GAs to train DANNs for Reinforcement Learning (RL) tasks; the trained DANN has over 4 million parameters, which are indirectly represented by lists of random seeds that are used by random Gaussian number generators.

One of the main issues when searching for deep networks for large datasets is that the evaluation of the candidate solutions is too costly. Limited Evaluation Evolutionary Algorithm (LEEA) [183] addresses this problem by evaluating the population against a small set of the training instances, which are selected to be diverse, according to the network outputs they are expected to provide. To avoid fluctuating fitness values (due to the variability of the selected instances) the fitness is computed considering the performance in the current set of instances, and in the previous sets. LEEA enabled Morse and Stanley to optimise the weights of networks ranging from 1170 to 1470 connections.

To overcome the large search space often associated to DANNs, some approaches tackle the optimisation of the learning and network parameters of fixed topologies, i.e., in addition to optimising the weights or learning parameters, the topology is kept fixed throughout evolution (e.g., number and type of hidden-layers), but the specific parameters of each layer are optimised (e.g., number of neurons, activation function). Young et al. [277], and Lorenzo et al. [154] optimise the structural parameters of CNNs. The first work only focuses on the parameters of the convolutional layers; the second also searchers for the best settings for the pooling layers. The work of Lorenzo et al. was successfully applied to the tuning of the parameters of the well-established LeNet-4 [145]. The simultaneous optimisation of learning and structure parameters is carried out by Loshchilov and Hutter in [155]. The authors compare CMA-ES to Bayesian optimisation methods, and conclude that CMA-ES should be taken into account for parameter optimisation as it can generate good results, with moderate computational resources.

The evolution of the parameters of a-priori defined networks allows the optimisation of models that we already know to work well, e.g., it enables the fine-tuning of a single model to different problems, or even the adaptation of a network to specific constraints, like a maximum amount of training time or trainable parameters. The main limitation is that the previous methods make it impossible to automatically generate novel structures, which is the key-point of the simultaneous evolution of the topology and learning: generate novel models in terms of architecture and learning strategies for problems we are unsure how to handle.

As aforementioned, there are plenty of approaches to promote the evolutionary optimisation of the weights of ANNs (even for deep networks). Nonetheless, it is not time efficient to search directly for the best set of weights for each of the candidate topologies generated throughout evolution. The process of optimising, for each network, thousands or even millions of trainable parameters takes too long, making evolution unfeasible. This way, the majority of NE works that seek to optimise the learning and topology of DANNs focus on the optimisation of the learning parameters and not directly the weights, i.e., in addition to optimising the topology, the objective is to fine-tune the best learning algorithm for each candidate topology (e.g., Adam, BP) and the respective parameterisation (e.g., learning rate, momentum).

CGP-CNN [236] extends CGP to the optimisation of CNNs: each position of the genotype instead of representing a neuron is a layer. Evolution optimises the structure of the network and the connectivity of the layers (i.e., a layer can receive the output of multiple layers as input). The generated solutions are highly effective, and competitive with the state of the art. Nonetheless, the authors define macro-blocks, and the evolutionary approach places them and optimises its parameters. The blocks have specific layers, and thus, up to a certain point, evolution is biased towards the author's knowledge of models that are known to work well. CGP-CNN does not optimise any learning parameter and/or policy. Similarly to CGP-CNN, Evolving Deep Convolutional Neural Networks (EvoCNN) [238] focuses on optimising the structure of DANNs, but

where the evolutionary unit is a layer instead of a macro-block. The network topology is divided into two parts: the first part is used for feature extraction and includes convolutional and pooling layers, and the second part encodes fully-connected layers. The initial weights of the networks are also target evolved (mean and standard deviation parameters for a Gaussian distribution). The authors of EvoCNN later propose Automatically Evolving CNN (AE-CNN) [239], which optimises the architecture of CNNs by combining and stacking ResNet [101] and DenseNet [108] blocks. These blocks represent interconnected sequences of convolutional layers that are known to work well, i.e., the method is biased by knowledge from hand-crafted CNNs. The use of a-priori knowledge speeds up evolution and facilitates the generation of highly effective CNNs.

Real et al. [202] use the search space of NASNet [280] to optimise the architectures of image classifiers. The networks are composed of two different cell types that are repeated in specific positions of a fixed topology: normal and reduction cells. All normal and all reduction cells have the same structure, but normal and reduction cells are different and independent. The goal of evolution is to optimise the architecture of the normal and reduction cells. Each cell has several interconnected nodes. The operations of the nodes can be either convolution of pooling. To reduce the size of the signal, the reduction cells are followed by a stride of two. Another particularity of the search space of NASNet is that the normal cells are stacked in blocks of cells with the objective that, after optimisation, the stack can be enlarged to increase the performance of the network. To facilitate evolution, Real et al. introduce "aging evolution": the individual that is discarded is not the worst but the oldest one. At the time, the best-generated model, referred to as AmoebaNet-A, set a new state of the art in ImageNet. Following the same rationale, Liu et al. [150] introduce hierarchical representations to optimise a cell that is placed multiple times in specific positions of a model with a fixed topology. Hierarchical representations contrast with flat representations. While, in flat representations, the architecture of DANNs is encoded as a single directed acyclic graph, in hierarchical representations the idea is to have different levels of representation where lower-level blocks are combined to form higher-level blocks.

GeNet [270] also follows a block-based approach, but where the blocks/cells are not defined, they are evolved. Each individual is composed of S stages, and each stage has N nodes, i.e., the individuals are formed by stages, which are graphs of nodes. Each node is a convolutional layer (with a fixed number of filters), where batch normalisation is applied to the output and passed through a ReLU layer. Between stages, there is a spatial pooling layer. The initial population is formed by individuals that derive from the LeNet [145] architecture. The individuals are trained with a fixed learning strategy. Despite the automatic design of the stages (i.e., blocks), there is also a-priori knowledge when defining the operations that can occur in the stages, and between stages. NSGA-Net [160] is based on GeNet: the same encoding scheme is used, but the search procedure that guides evolution is Non-Dominated Sorting Genetic Algorithm (NSGA)-II [52] – a well known multi-objective optimisation approach. In NSGA-Net the encoding considers skip connections, and the multiple objectives that are considered to assess the quality of the networks are classification error and computational complexity (measured by the number of floating point operations). Another novel aspect of the approach is that the search procedure considers exploration and exploitation phases. The first explores different connectivity patterns in the nodes through crossover and mutation, and the latter takes into account past information to generate candidate solutions using Bayesian optimisation.

The previous works, despite generating highly efficient networks, start evolution from the definition of what a block/cell is and where it should be placed in the network, i.e., they require the definition of a-priori knowledge, and thus, it is unlikely that the generated architectures are novel and different from those typically assembled by human designers. Baldominos et al. [23] compare a GA to GE in the evolution of the topology and general optimiser parameters of CNNs for the MNIST dataset. The search space contains DANNs with up to 4 convolutional layers and

2 dense layers. To accelerate the train of the CNNs, the networks are evaluated for 5 epochs and, in each epoch, only half of the training instances are considered. To increase diversity, in the initial population the individuals are randomly created until there are no invalid solutions, and to avoid premature convergence to a single solution, niching is used. The results show that the CNNs optimised by GE surpass in performance those obtained by the GA. A similar approach is considered by Lima and Pozo [148], who apply GE to the optimisation of CNNs too.

Miikkulainen et al. [171] adapt NEAT to the evolution of the topology and hyper-parameters of DANNs. They propose DeepNEAT, which is similar to NEAT, but where each node of the chromosome represents a layer instead of a neuron. The type of each layer (e.g., convolutional, or recurrent) and the hyper-parameters (e.g., number and shape of the filters in the convolutional layers) are optimised and, whereas, in NEAT, the values of the weights are directly evolved, in DeepNEAT the authors evolve the connectivity between layers, and optimise general hyper-parameters (e.g., learning rate or learning algorithm). The evolved DANNs are evaluated for a fixed number of epochs. DeepNEAT is extended to CoDeepNEAT, that draws inspiration in SANE, ESP, and CoSyNE. CoDeepNEAT co-evolves two populations of modules and blueprints; the modules' population evolves small (or parts) of DANNs; the blueprints' population defines what modules should be assembled to form the DANNs. The module's population is divided into sub-populations (speciation) and, because the blueprints point to sub-populations of modules, when building the network, a module is chosen at random from a sub-population. When the blueprint refers more than once to the same sub-population of modules, the same module is chosen. To evaluate the individuals the DANNs are trained similarly to DeepNEAT, and the fitness is propagated back to the blueprints, and to the modules, computed as the average fitness of all the DANNs it participates in.

Learning Evolutionary AI Framework (LEAF) [147] adapts CoDeepNEAT to cloud computing infrastructures (e.g., Amazon AWS or Microsoft Azure). The system is composed by three layers: (i) the algorithm layer, which is essentially CoDeepNEAT and thus, enables LEAF to optimise DANNs; (ii) the system layer, which distributes the training of the generated DANNs; and (iii) the problem-domain layer, which introduces multi-objective optimisation. The networks are evolved taking into account two objectives: performance (i.e., accuracy), and complexity (i.e., number of network parameters). Similarly, to scale, Evolutionary Exploration of Augmenting Convolutional Topologies (EXACT) [53] is based on volunteer computing. In particular, EXACT uses approximately 4500 computing nodes from the Citizen Science Grid to generate and train roughly 120000 CNNs in 2 months.

The vast majority of the above-mentioned methods were proposed considering the automatic optimisation of CNNs. The main reason for that is the complexity associated with the deployment of this type of DL architecture: multiple layer types must be selected, placed, and interconnected; further, each layer type has specific parameters. The hand-design of CNNs has shown that they are highly effective in solving real-world problems. However, their success is intrinsically connected to the design and train of the networks. Notwithstanding, most of the principles discussed on the automation of the tuning of CNNs are generalisable to other DL architectures. The current Thesis focuses mainly on the optimisation of CNNs. For examples of large-scale NE approaches targeting the optimisation of other architectures refer to [49, 201, 222].

2.5 Transfer, Multi-Task and Incremental Learning

The main limitation when addressing the automatic search for DANNs is the time and resources that are required to attain effective solutions. NE is based on EC, and thus we must assess the quality of the population. As discussed in the previous section, and especially when optimising

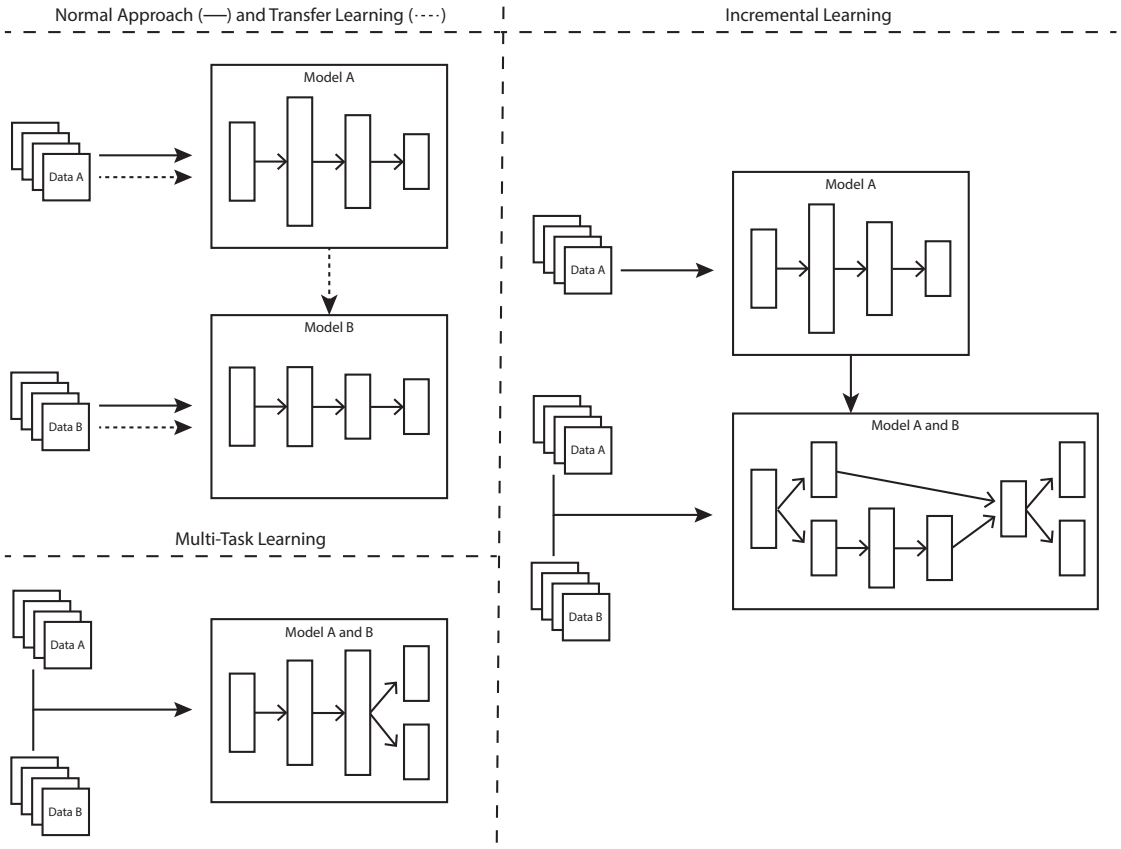


Figure 2.10: Differences between transfer, multi-task, and incremental learning.

DANNs, the evaluation of the individuals requires their training, which is time-consuming and computationally-demanding. For example, the training of candidate solutions in CoDeepNEAT is parallelised on 100 GPUs, and each generation takes around 1 hour; Real et al. use 450 GPUs, and each run takes about 7 days to perform. Other methods are computationally cheaper, e.g., the approach by Lorenzo and Nalepa [153] only requires about 120 minutes to obtain results, but the speedup is obtained at the cost of the performance of the optimised model.

The default approach to NE starts evolution from scratch each time we require a network to solve a specific problem, i.e., the initial population is composed of individuals that are formed at random, or derivations of a specific network structure. Consequently, the evolutionary knowledge gained from the optimisation of ANNs for previous tasks is not considered. This knowledge can speed up evolution and, as such, in this section, we survey important works on transfer, multi-task, and incremental (or cumulative/lifelong) learning applied to ANNs. In the broader picture, the ability to transfer knowledge and/or solve multiple-tasks incrementally is a key-point to the emergence of robust and adaptable AI systems that pave the roadmap to general AI.

Transfer [189, 242] and multi-task learning [40] are similar in the sense that both are approaches to aid learning multiple problems. The difference lies in how learning proceeds. In multi-task learning, the model is simultaneously trained on several problems, and, by the end, it must be able to perform well on all of them. In transfer learning, the knowledge gained when solving primary tasks is ported to secondary tasks and, consequently, the model trained on the

last task may be unable to work on the previous task – a problem known as catastrophic forgetting [73]. Incremental/continual/lifelong learning [205, 247] merges concepts from the two learning paradigms: the objective is to start with a model able to solve a single task (or set of tasks) and to continuously expand it, enabling the re-use of past knowledge, and ultimately being able to solve all observed tasks, i.e., incremental learning mimics the behaviour of the human brain, a modular structure that evolves and adapts to new problems. Figure 2.10 depicts the flow that can be expected of transfer, multi-task, and incremental learning. We are particularly interested in transfer and incremental learning for their ability to build upon past knowledge, and therefore speed up the search for adequate DANNs. Multi-task learning principles can also be of use to large-scale NE, but their main advantage is to provide inductive bias, i.e., training in multiple tasks forces the exploration of latent spaces that generalise more easily.

The application of transfer, multi-task, and incremental learning to NE can occur at two different levels. On the one hand, the EC engine can be adapted to search for candidate solutions through transfer or multi-task learning (e.g., [95, 184, 216]). For example, instead of evolving ANNs that can solve multiple-tasks, we can resort to EC approaches that simultaneously evolve solutions to different problems. This way, we will likely end up with a different solution for each problem. On the other hand, the EC engine can be adapted to optimise ANNs that incorporate principles from transfer, multi-task, and incremental learning. Considering the previous example, we want to optimise a single network that tackles multiple problems simultaneously. The scope of this Thesis is on the latter hypothesis, and thus, next, we will survey transfer, multi-task, and incremental learning ANNs, both generated by manual and automatic approaches.

2.5.1 Transfer Learning

Transfer learning [189, 242] aims to leverage the knowledge from previous source tasks (e.g., architecture or weights) to smooth the learning in a new target task. The key-problem is the compatibility and representation of the tasks. It is expected that the underlying principles are related. Otherwise, there is no gain in transferring from one task to the other. In particular, a negative transfer may occur when the transfer is performed between two highly distinct tasks, i.e., instead of helping to create a more effective model, transfer hinders performance. When the representation of the tasks is different, there must be a mapping from one domain to the next. The effectiveness of transfer learning approaches is often measured according to two criteria: time and performance – the objective is that when transferring from a task A to a task B the time taken to solve the two tasks is inferior when transferring from one task to the other than when addressing both independently, with at least the same degree of performance.

The easiest and likely most used form of transfer learning consists of fine-tuning the weights of a network pre-trained on a source task [93, 109, 237]. This is the methodology followed by Ciresan, Meier, and Schmidhuber [45], with the difference that they do not fine-tune all the weights but rather freeze the first n layers, and only train (from randomly initialised weights) the last layers. The weights are partially transferred between different character recognition tasks and show that transfer learning enhances the performance and speeds up learning. It is also demonstrated that it is best to train a DANN in part of the dataset and then continue training with all the instances. The decision of which layers should be frozen, and which ones should be re-trained, is likely specific for each architecture and dataset. Yosinski et al. [276] investigate transfer learning in a well-known architecture that, by the time it was proposed, set a new state-of-the-art result in the ImageNet 2012 challenge [138]. The authors define two types of features: (i) general, i.e., high-level features that are widely applicable to different tasks; and (ii) specific features, i.e., features that are tailored to a particular dataset. The learning of the general and specific features happens at the first and last layers of the architecture, respectively.

The work investigates transferring and freezing the features, or transferring and fine-tuning the features. The results indicate that, independently of the number of layers that are transferred, there is a trend for higher-performing results when the features are fine-tuned to the target dataset. Further, it is concluded that transferring weights (even from dissimilar source tasks) is better than initialising the weights at random.

The majority of the fine-tuning approaches optimise the weights from the source to the target task, and then all instances are predicted using the fine-tuned parameters. A different approach is followed by SpotTune [94], which introduces a transfer learning method that performs an instance-based fine-tuning, i.e., the instances of the target dataset can be either passed by the pre-trained or the fine-tuned layers. To accomplish this, a binary variable is coupled to each layer to indicate whether or not, according to a specific instance, that layer should be frozen or fine-tuned. This new variable is trained together with the classification task, using the Gumbel Softmax sampling method [110].

Although transfer learning is often based on the fine-tuning of the models from previously related tasks, there are also works that have the goal of evolving network topologies that can be adapted to other problems. Zoph et al. [280] propose the NASNet search space that allows for the design of architectures that can be ported from simple problems to more difficult ones. The objective of the optimisation is to find normal and reduction cells (discussed in Section 2.4.2) that are placed in a fixed topology. The normal cells are repeated N times, and thus the architecture can be optimised for a small problem (e.g., CIFAR-10) with a small N, and then applied to a larger problem (e.g., ImageNet) with a greater value of N. A similar rationale can be followed to deploy the architectures in devices with low computational resources by decreasing N, i.e., the optimised architecture can be adapted to different problems by adjusting the value of N. Liu et al. [149] propose Progressive Neural Architecture Search (PNAS), which has as main objective to speed up the search conducted by Zoph et al. over the NASNet domain. For the same setup, PNAS requires 5 times fewer evaluations than the standard implementation [280] to obtain results with similar performance. To speed up evolution Liu et al. initialise the search from minimal cells that complexify as search proceeds (similar to NEAT). Additionally, they use a surrogate model (in particular, a LSTM) to estimate the performance of the cells without the need for training the models; only the top surrogate choices are trained. Similarly to the work of Zoph et al., the generated cells are successfully ported to more complex network structures. The same methodology can be taken by previously discussed large-scale NE approaches, e.g., Real et al. [202] also optimise DANNs in the NASNet search space, and Liu et al. [150] optimise a cell that is placed in different base models (with a fixed architecture) depending on the problem.

In what regards NE, the most straightforward approach to transfer learning is to resume the optimisation and extend the models to the target task based on the last population (or single best individual) of the source task. This is the methodology followed by Cardamone, Loiacono, and Lanzi [39], who investigate the performance of online NEAT applied to transfer learning between tracks of a car racing simulator, i.e., transfer learning in tasks where the domain is kept, and thus, the input of the network has the same shape for the different tasks. The proposed approach uses the best solution found for a previous track to warm-start evolution. The results make it evident that this transfer learning strategy speeds up evolution (the initial performance on the secondary tasks is much higher when transfer learning is used), and that, by the end of evolution, the performance is at least as good as without transfer learning. Furthermore, the best results are obtained when passing from a more complex track to an easier one. The same rationale is followed by Wong et al. [266].

The previous methodology works smoothly because it is applied to transfer learning between tasks where the domain representation is the same. In what regards works that deal with the representation for transfer learning, Verbancsics and Stanley [257] investigate a static representation

scheme that can, without any modification, be applied to different tasks. Hence, the transfer from one task to the next is transparent and requires no changes to the representation. The authors propose an indirect encoding method, called Bird's Eye View (BEV), which is based on the HyperNEAT approach. The BEV represents the state of the problem in a geometric space, and thus it is natural that it is mapped into a 2-dimensional substrate, where CPPNs can be used to compute the connectivity, and therefore explore the spatial relationship of the data. The work of Verbancsics and Stanley contrasts with the work of Taylor, Whiteson, and Stone [243], who present an approach where transfer learning can be applied to incompatible source and target tasks. To this end, they propose inter-task mappings, which translate the last population of a source task (evolved using NEAT) into the initial population of a target task (that is also optimised using NEAT). The inter-task mappings can be fully or partially defined by the user or automatically learned. Similarly, Elsaid et al. [61] extend Evolutionary eXploration of Augmenting Memory Models (EXAMM) [188] to transfer learning. The method performs transfer learning from source to target tasks that may have different input and/or output shapes. Identically to other approaches, the optimisation for the target task is resumed from the source task. However, a series of mutation operations that add/remove inputs or outputs (and connections) is conducted from the source to the target task.

2.5.2 Multi-Task Learning

The main objective of multi-task learning is different than the one of transfer learning. Whereas, in transfer learning, the ultimate goal is to speed up evolution, reducing the time that is required to solve the target task, in multi-task learning, the goal is to leverage multiple tasks. Multi-task learning approaches are normally grouped according to two different objectives: (i) simultaneously train and learn to predict different tasks; or (ii) simultaneously train on a primary task and on one or more secondary tasks, to enhance the performance on the primary task. In the second scenario, the secondary tasks are only used to increase the generalisation ability of the primary task, and therefore the performance on the secondary tasks is not considered. Nonetheless, in the two scenarios, the objective is to learn a representation that is shared among tasks. As an example, Collobert and Weston [47] describe a multi-task CNN that is simultaneously trained to address 6 NLP tasks, and Zhang et al. [279] address the problem of facial landmark detection using multi-task learning; the authors propose a Tasks-Constrained Deep Convolutional Network (TCDCN) which, instead of maximising the performance of all tasks, focuses only on the main task, assisted by the auxiliary tasks. TCDCNs can use different loss functions for each of the tasks and, therefore, it is straightforward to combine regression and classification tasks. Because the different tasks are unlikely to converge at the same time, a task-specific early stop method is introduced, i.e., the contribution of each task to the loss function halts at different training phases. The results show that TCDCN layers capture a shared representation of the main task, and that task-specific early stopping increases the training performance.

There are examples of hand-designed multi-task learning models in a wide range of domains: from NLP [47, 227, 269] to computer vision [152, 118, 176], and speech processing [124, 218, 267]. The above-enumerated approaches develop architectures and training methodologies to tackle problems from the same domain. Contrarily, Kaiser et al. [116] propose a multi-task model that can simultaneously learn tasks of different domains, e.g., image classification, image captioning, text translation, and speech recognition. To deal with the different domains and, consequently, different types of inputs, the authors propose modality networks for text, image, audio, and categorical data. The data is processed using different blocks. In particular, the authors introduce the use of convolution, attention, and mixture of expert blocks. The architecture of the multi-task network is composed of an encoder, a mixer, and a decoder, that are formed using the blocks.

The proposed architecture shows that it is possible to simultaneously solve multiple problems of different domains with performances that are competitive with the state of the art, and, under some circumstances, training with multiple problems even reports results that are superior to single-task learning.

Meyerson and Miikkulainen [170] propose an automatic deep multi-task learning method based on soft layer ordering, which contrasts with parallel ordering methods. Whereas, in parallel ordering, the structure of the networks is assembled in a way that aligns feature extractors across tasks, in soft ordering methods, the features can be used more flexibly at different depths. One of the main limitations of this multi-tasking structure is that the layers are set in a fixed-topology. To expand the acquisition of sharing relationships Liang, Meyerson, and Miikkulainen [146] extend CoDeepNEAT to automatically optimise the type of multi-task networks introduced by Meyerson and Miikkulainen [170]. The layers in soft ordering are mapped to CoDeepNEAT modules, which are evolved and combined to form adequate topologies. Different CoDeepNEAT-based methods are proposed and show that the best results are obtained by co-evolving the modules and task-specific routings, i.e., specific topologies are evolved for each task, and modules and components shared between them. The results of the CoDeepNEAT-based method surpass the performance of the standard soft ordering implementation.

2.5.3 Incremental Learning

The main difference between transfer and multi-task learning lies in how the multiple tasks are acquired. In multi-task learning, there is only one training stage, i.e., all the tasks are learned simultaneously, and after training, the model can address the multiple tasks. On the other hand, transfer learning enables the sequential learning of multiple tasks, in multiple stages (one stage per task) but, the model, at each stage, only recalls the target task where it was last trained on [73]. The purpose of incremental learning [190] (also known as cumulative or lifelong learning) is to learn continuously, without forgetting the source tasks. Consequently, incremental learning may benefit from combining the advantages of the transfer and multi-task learning paradigms. The different tasks should be learned sequentially and using the knowledge of the source tasks (when related). In any stage, the model should be able to solve all the tasks it was trained on. A comparison study of different incremental learning methods is found in [119].

A key-question to incremental learning is the balance between the previously learned and the new to be solved tasks – known as the stability-plasticity dilemma [169]. On the one hand, the networks must be plastic so that the new knowledge can be integrated. On the other hand, they have to be stable to avoid forgetting the previously acquired knowledge. One of the most common approaches to cumulative learning is to always train the network considering all the data from the previous and new tasks. However, the amount of memory required to train the network largely increases with the number of addressed tasks. Elastic Weight Consolidation (EWC) is a regularisation method that balances stability and plasticity by updating the weights at different speeds, i.e., EWC first identifies the parameters that are most important to previous tasks, and then reduces its plasticity, so that previously acquired knowledge is not disrupted. CopyWeights with Re-init (CWR) [151] keeps two sets of weights for the output classification layer: (i) a set of consolidated weights (cw) that are used for prediction; and (ii) a set of temporary weights (tw) that are randomly re-initialised in each training session. By the end of each training session, the weights in tw are used to update cw. In CWR the layers of the intermediate layers are frozen after the first learning session. AR1 [161] extends CWR to enable the adaptation of the weights of the intermediate layers.

An alternative to regularisation methods are ensembles formed by incrementally generated classifiers, where each classifier solves a task. Again, this type of approach does not scale well

because an additional classifier is needed for each task. An alternative is to limit the maximum size of the model. For example, PathNet [64] optimises pathways in a large fixed-structure DANN that consists of a pre-defined set of layers, each with a given number of modules. In each layer, a maximum number of modules can be used, and the outputs of the modules are merged (in this case summed), before passing to the next layer. The best pathway for each dataset is frozen, and therefore, when solving new tasks, the modules can be re-used, but its parameters (including weights) cannot be altered. The results demonstrate that PathNet is able to cumulatively solve tasks, and speeds up the emergence of effective DANNs when prior-tasks have been solved.

On the one hand, the definition of the network grid provides PathNet with a way to deal with memory limitations. On the other hand, as more problems are solved, there is the risk of exhausting the network and freezing all available pathways. With this rationale, Progressive Neural Networks (PNNs) [210] introduce a dynamic architecture. PNNs are defined by a multi-column architecture where each column is a network for solving a specific task. The model starts with a single column, and to address a secondary task the parameters of the first column are frozen, and a new column with random parameters is added to the model. The new column can establish connections to previous columns, this way enabling transfer and continuous learning, and avoiding catastrophic forgetting. Therefore, the model is always able to solve previous tasks, but it is the user's responsibility to know which output layer (i.e., column) should be considered. Gideon et al. [79] compare PNNs to the standard pre-training and fine-tuning transfer learning strategy, and conclude that, in emotion recognition, the transfer of learning based on PNNs consistently obtains results that are statistically superior to the standard strategy. As aforementioned, this is a computationally expensive method, and similarly to the ensemble approaches, it scales poorly. Dynamically Expandable Network (DEN) [275] addresses lifelong learning by sharing the architecture of the networks across multiple tasks and expanding the architectures when there is the need to learn new problem features. To address a secondary task, DEN selectively re-trains important parts of the network. However, when this is not sufficient, the network is expanded by adding new nodes. When a semantic drift threshold is surpassed, the nodes are duplicated, to prevent the loss of features.

Another dual-memory system is proposed by Soltoggio [229]. The authors propose a novel plasticity rule referred to as Hypothesis Testing Plasticity (HTC). The learning is separated into short-term and long-term memories, and according to the HTC, the consolidation of new knowledge is not based on time, but rather on the degree of certainty of the stimulus-action pair. Further examples of methods that apply memory systems to address lifelong learning exist. For example, Neural Turing Machines (NTMs) augment the networks with an external read/write memory, which can be trained using NE [88]. Parisi et al. [191] propose a dual-memory system, based on two RNNs: one is responsible for the episodic memory (representation learning), and the other for semantic memory (structural plasticity modulation).

Inspired by the configuration of the human brain, some authors advocate that the road to incremental learning is through the evolution of modular networks, i.e., networks whose architecture is formed by tightly coupled clusters of neurons/layers that are sparsely connected. The rationale is that the clusters are responsible for well-defined tasks (or sub-problems of the main task), and that knowledge can be shared between the clusters so that learning proceeds hierarchically and continuously. Clune, Mouret, and Lipson [46] investigate the impact that evolving networks based on the maximisation of the performance and minimisation of the computational cost (i.e., connections) have on the modularity of the generated networks. The results make it evident that modularity emerges as a byproduct of multiple optimisation objectives; further, there is a trend for each cluster of nodes to tackle a sub-problem of the main task. The networks evolved considering the computational cost adapt faster when transferred to another task than the networks optimised only considering performance, i.e., modularity eases transfer learning.

Notwithstanding, the continuous expansion of the modular structure is not explored. The work of Clune et al. was extended by Ellefsen, Mouret, and Clune [60] to understand whether or not the modular networks mitigate catastrophic forgetting. The goal was to develop topologies where learning selectively affects only specific clusters of the network. To this end, the networks were optimised to address, interchangeably, two setups of the same domain; the results show that considering the performance and connectivity cost not only increases the performance and modularity of the evolved solutions, but also reduces catastrophic forgetting.

2.6 Summary

ML is a vast field, with numerous approaches, each very specific in its way of application and parameterisation. This Thesis focuses on ANNs, which have many parameters that require optimisation: (i) architecture of the network, i.e., the sequence and number of layers, the type and parameterisation of each layer, and the connectivity between layers; and (ii) the learning strategy, i.e., which algorithm should be used to tune the connectivity weights, and the parameterisation of the algorithm.

To facilitate the selection of the most appropriate ML model, AutoML seeks to automate the cyclic process of having to pre-process/extract features, select the model, and parameterise it. These decisions are even more challenging considering that they are interdependent, and thus one affects the others. In particular, we introduce grid, and random-search methods, Bayesian optimisation, and EC. Particular focus is given to methods that optimise pipelines, i.e., sequences of ML methods (from the pre-processing to the model parameterisation) that are applied to the raw data. We highlight Auto-WEKA, TPOT, Hyperopt-Sklearn, Auto-Sklearn, MOSAIC and RECIPE.

From the above methods, we focus on EC. It is a search method that is highly scalable and parallelisable. It is a bio-inspired methodology where a population of candidate solutions is continuously evolved throughout generations, and where, from one generation to the next, the fittest candidate solutions have a higher chance of reproducing, and thus of passing their characteristics to the offspring. In particular, we are interested in GGP. The domain is specified in a human-readable format, and consequently easily understandable and adaptable to search for solutions for different domains.

We focus on the application of EC to the optimisation of ANNs – a field known as NE. Traditionally, the bibliography is grouped according to the aspects of the ANNs that are optimised: (i) topology; (ii) learning; or (iii) topology and learning simultaneously. We follow a different approach, dividing the literature into NE approaches for shallow and deep architectures. The main difference between shallow and deep architectures goes beyond the number of hidden-layers, the key-difference is that DANNs are able to automatically extract features from the raw data. Notwithstanding, the larger depth of DANNs means that the networks have a higher amount of parameters, and thus are more challenging to optimise and train. That is the reason why the majority of the approaches that tackle the evolution of DANNs encode the networks at a layer-level and do not optimise the weights directly but, at most, focus on the tuning of the learning algorithm and its parameters.

There are plenty of NE approaches that successfully generate competitive ANNs. The problem is that these methods are often designed considering the optimisation of a single solution to a specific task. There is no advantage in solving multiple tasks, as the knowledge that is possibly acquired when evolving ANNs for source problems is not re-used for new target tasks. That is why we analyse transfer, multi-task, and incremental learning approaches. The different learning paradigms differ essentially on the methodology followed to learn multiple tasks. Transfer learn-

ing identifies and re-uses parts of the source network that may speed up the learning of the target task but, by the end, the network is only able to address the last task (catastrophic forgetting). Multi-task learning learns multiple tasks simultaneously, but the learning is not continuous, i.e., the network cannot be later expanded to work with another task it was not trained on. Finally, incremental learning integrates principles from transfer and multi-task learning, so that a system that is initially trained on a single (or set of) task(s) can learn to address new tasks, without forgetting the old ones.

Chapter 3

Small-Scale NeuroEvolution

This Chapter introduces exploratory work based on the ideas discussed in the related work (Chapter 2). In particular, we investigate the application of grammar-based approaches to the automatic optimisation of ANNs. Section 3.1 applies SGE to optimise single hidden-layered networks. Section 3.2 introduces DSGE as a means to promote the evolution of multi-layered networks. These first two works focus on the optimisation of the topology and weights of ANNs. The training of the networks is only possible because they tend to be small, and consequently have few connections to be optimised. In a different research direction, in Section 3.3, we set to scale the evolution of (potentially) deep AEs to generate compressed representations; the networks are trained using the Adam optimiser. Contrarily, in Section 3.4, we fix the topology and compare different approaches to optimise the learning policy. The results of these initial experiments pave the way to the development of a grammar-based NE approach able to optimise DANNs. The results and next steps are summarised in Section 3.5

3.1 Neural Networks Structured Grammatical Evolution

Our first proposal to the automatic optimisation of ANNs is based on SGE. It is a first step towards the exploration of the potential of grammar-based approaches applied to NE. In particular, we focus on the use of the recently proposed SGE to the evolution of one-hidden-layered ANNs. We hypothesise that, due to the higher locality and lower redundancy, SGE will be more efficient than GE on the exploration of the search space of the ANNs.

The results show that for the considered benchmarks, SGE consistently reports performances that are statistically superior to those obtained by GE. The main limitation of GE and SGE-based approaches is that they are limited to optimising one-hidden-layered networks. This happens because of the inability of the evolutionary engines to deal with expandable production rules, and thus it is not possible to store the placement of the neurons in different layers.

The remainder of the current section is organised as follows. Section 3.1.1 details the components of the evolutionary engine that differ from the standard SGE implementation (previously described in Section 2.3.3). Section 3.1.2 reports the experimental setup and results. Section 3.1.3 summarises the framework conclusions and next steps.

3.1.1 Evolutionary Engine

The core evolutionary engine of the proposed methodology is the same as in canonical SGE. We adapt the mutation and crossover genetic operators. In addition, we define a fitness function for

the evaluation of ANNs.

3.1.1.1 Mutation

In SGE (Section 2.3.3) the probability of mutating each gene, i.e., changing one of the integers from the list of integers associated with a non-terminal symbol is the same, and equal to $1/n$, where n is the number of genes (which equals the number of non-terminal symbols). However, the number of integers in each gene is not the same. To overcome this drawback, we propose a roulette-like approach for selecting the gene to be mutated, with the probability of choosing each gene equal to:

$$p_i = \frac{len(gene_i)}{\sum_{j=1}^{n} len(gene_j)},$$

where $gene_i$ is the i-th gene, n is the total number of genes, and $len(gene)$ is the number of integers of the gene that are being used to map the individual from the genotype to the phenotype. Next, we randomly select one of the integers from the chosen gene and change its value to a new valid possibility. Mutations are not applied to non-expressed integers, i.e., those that are not used in the genotype to phenotype mapping procedure.

3.1.1.2 Crossover

We rely on a one-point crossover to combine two parents. We start by choosing a random cutting point in the genotype, and then we swap the genetic material between the two parents. In SGE the genetic material that is swapped corresponds to genes and as such not directly to the integers, i.e., the genetic information that is swapped consists of lists of integers that encode the expansions of the non-terminal symbols.

3.1.1.3 Fitness Evaluation

The performance of each ANN is measured by the Root Mean Squared Error (RMSE), obtained while solving a classification task. To avoid overfitting and to tackle unbalanced datasets, we consider the RMSE per class, and the fitness function is the minimisation of the multiplication of the exponential values of the multiple RMSEs per class, as follows:

$$\text{fit} = \prod_{c=1}^{m} exp\left(\sqrt{\frac{\sum_{i=1}^{n_c}(o_i - t_i)^2}{n_c}}\right),$$

where m is the number of classes of the problem, n_c is the number of instances of the problem that belong to class c, o_i is the confidence value predicted by the evolved network, and t_i is the target value. To ensure that higher errors are more penalised than lower ones, we use the exponential function. This also avoids the classification function from being too constrained to the error in one of the classes, failing to learn the overall problem.

3.1.2 Experimentation

The proposed methodology is used to evolve the topology and hyper-parameters of ANNs for 4 different binary classification problems from the UCI ML Repository [55]. The results are analysed and compared with others obtained using standard GE, and GE-based NE approaches. More precisely, we compare to NNC [251], GE-BP [228], and GEGA [4].

<sigexpr> ::= <node> (1)
| <node> + <sigexpr> (2)
<node> ::= <weight> * sig(<sum> + <bias>) (3)
<sum> ::= <weight> * <features> (4)
| <sum> + <sum> (5)
<features> ::= x_1 (6)
| ... (7)
| x_n (8)
<weight> ::= <number> (9)
<number> ::= <digit>.<digit><digit> (10)
| - <digit>.<digit><digit> (11)
<digit> ::= 0 | 1 | 2 | 3 | 4 (12)
| 5 | 6 | 7 | 8 | 9 (13)

Grammar 3.1: Grammar used by SGE to search for ANNs for the Flame, Wisconsin Breast Cancer Detection, Ionosphere, and Sonar datasets.

3.1.2.1 Datasets

We select 4 binary classification problems from the UCI Machine Learning repository [55]. The problems have an increasing number of features. In the next paragraphs we present a brief description of each of the benchmarks.

Flame [75] – This dataset contains artificially generated data for clustering purposes. It has 240 instances with two attributes each that are to be separated into two different classes, the first one containing 87 instances and the second one with 153 instances.

Wisconsin Breast Cancer Detection (WDBC) [234] – The WDBC is comprised of 30 features extracted from digitalised images of breast masses. The dataset has 569 instances, where 212 are malign and 357 are benign.

Ionosphere [221] – This benchmark is used for the classification of ionosphere radar returns, where the returns are classified into two different classes: good (225 instances) if it returns evidence of structure, and bad (126 instances) otherwise. 34 features are provided.

Sonar [86] – The sonar dataset contains 60 properties of sonar signals that allow a classification model to separate between signals that are bounced off a metal cylinder (111 instances), or off a rock cylinder (97 instances).

3.1.2.2 Grammar

The grammar used to evolve ANNs is depicted in Grammar 3.1 and is based on the ones used in the works we later compare the results of SGE to [4, 228, 251]. The grammar allows the evolution of both the topology and hyper-parameters (including connection weights) of one-hidden-layered ANNs. In brief words, it is capable of representing a series of neurons (<*sigexpr*>) as well as their connections to the previous layer. Each neuron is represented by <*node*> and is no more than a weight multiplied by the result of the activation function, which takes as input a weighted

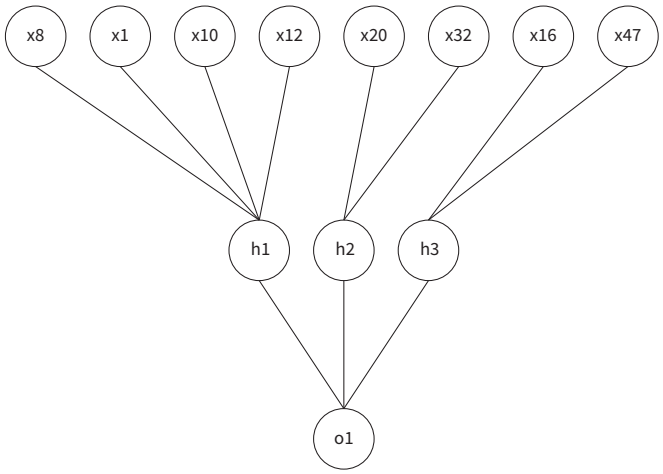


Figure 3.1: Example of the topology of an ANN that SGE can generate. The values of the weights and bias are omitted for simplicity. x, h and o represent the input, hidden and output neurons, respectively.

sum of the multiple input nodes: <*features*>, where n denotes the number of attributes of the problem. All neurons use the sigmoid (sig) activation function, and therefore the output of the ANNs is sig(<*sigexpr*>). By encoding connections, the evolved networks are not fully connected and feature selection is performed. An example of an ANN that can be formed by using the detailed grammar is: $-9.99 * sig(-6.93 * x8 + 9.93 * x1 + 9.40 * x10 + 6.56 * x12 + -1.99) + -6.98 * sig(8.99 * x20 + -4.69 * x32 + -4.68) + 9.99 * sig(1.84 * x16 + -7.60 * x47 + 0.99)$ (see Figure 3.1). The network is composed by three hidden-neurons, where each neuron is connected to 4, 2, and 2 features, respectively.

3.1.2.3 Experimental Setup

The experimental parameters used for conducting the search for ANNs with SGE and GE are detailed in Table 3.1. To make the comparison as fair as possible, we apply just one mutation to 95% of the population individuals, instead of defining a per-gene mutation probability. By doing this we ensure that only one change occurs in each individual, making the two methodologies similar in terms of the way they explore the search space. Additionally, in both engines, the mutation operator is only allowed to change the integers that are used in the genotype to phenotype mapping and populations are initialised at random.

All datasets are partitioned the same way: 70% of each class instances are used for training, and the remaining 30% for testing. Only the training data is used to assess the fitness of the individuals. The test data is kept aside from the evolutionary process and is used exclusively for validation purposes, i.e., to evaluate the networks in the classification of unseen instances. No pre-processing or data augmentation methodologies are applied to the datasets, and thus the datasets are used as they were obtained.

3.1.2.4 Experimental Results

To evaluate the ability of SGE to evolve models adequate for the classification of the chosen datasets, we focus our attention on the analysis of several network properties, namely: (i) fit-

Table 3.1: SGE experimental parameters.

Parameter	**Value**
Number of runs	30
Population size	100
Number of generations	500
Total number of evaluations	50000
Crossover rate	95%
Mutation rate	1 mutation in 95% of the individuals
Tournament size	3
Elite size	1%
SGE Parameter	**Value**
Recursion level	6
GE Parameter	**Value**
Individual size	200
Wrapping	0
Dataset Parameter	**Value**
Training percentage	70%
Testing percentage	30%

ness; (ii) RMSE; (iii) Accuracy; (iv) Area Under the Receiver Operating Characteristic Curve (AUCROC); (v) F-measure; (vi) number of neurons; and (vii) number of used features. All properties, except for the fitness, are analysed in the training and test sets.

3.1.2.5 Evolutionary Results

The evolution of the fitness of the best individuals across 50000 evaluations (100 individuals during 500 generations) is depicted in Figure 3.2. Each plot compares the evolution of the fitness of SGE to GE. The results are averages of 30 independent runs. SGE consistently presents mean fitness values inferior to those reported by GE. The goal of evolution is to minimise the per class exponential RMSE, and consequently, lower values stand for better performances.

A one-by-one analysis of the plots shows that the differences between SGE and GE are greater in the flame and ionosphere datasets than in the WDBC and sonar. Nonetheless, the general trend is that SGE and GE do not converge to the same results, and GE becomes stuck earlier than SGE. This is explained by the fact that SGE is able to explore the search space more efficiently (high locality and low redundancy), and thus takes longer to converge due to the number of feasible solutions it encounters.

In all datasets except for the WDBC the differences between SGE and GE are statistically significant (see Section 3.1.2.6). In WDBC the SGE approach is able to obtain better results in terms of means, but the differences are not statistically significant. In the initial population, the average quality of the best individuals is close to 2.2; in the remaining problems, it is around 2.5. As such, the degree for improvement is smaller, and with the given number of evaluations SGE and GE end having similar results. In the sonar results, we have the opposite, the problem is more difficult because of the higher number of attributes (60), which leads to a larger search space. For that reason, we performed this experiment again, with a larger number of evaluations. In concrete, we multiply the maximum number of evaluations by 5, i.e., 250000 evaluations (500

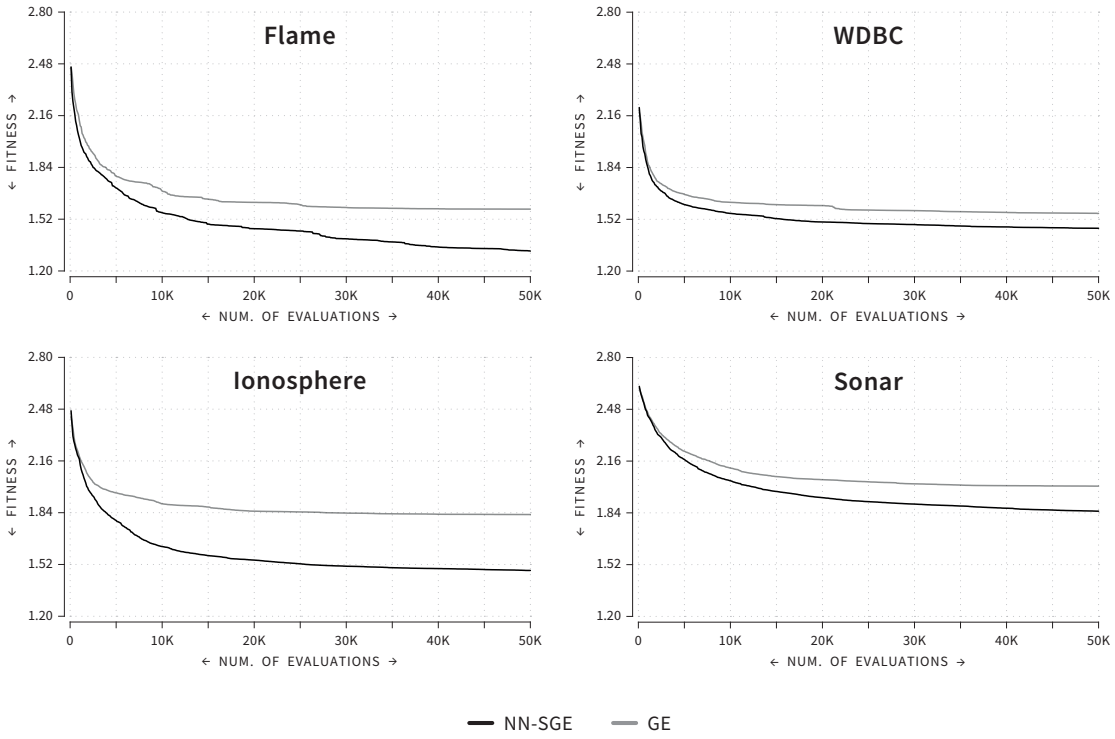


Figure 3.2: Evolution of the fitness of the best individuals on the training set across 50000 evaluations for the flame, WDBC, ionosphere and sonar datasets. The results are averages of 30 independent runs, and compare SGE to GE.

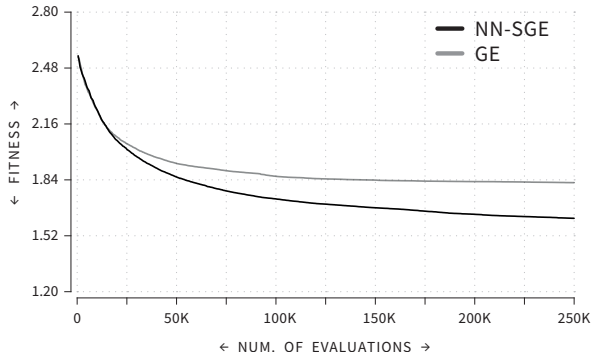


Figure 3.3: Evolution of the fitness of the best individuals on the training set across 250000 evaluations for the sonar dataset. Results are averages of 30 independent runs, and compare SGE to GE.

individuals over 500 generations). The results with the larger number of evaluations are depicted in Figure 3.3. The difference becomes larger and statistically significant.

So far, the results presented focus only on the analysis of the fitness over the training set, and therefore we have not yet analysed the generalisation ability of networks found by SGE,

Table 3.2: SGE experimental results: fitness, RMSE, accuracy, AUCROC, f-measure, number of neurons and number of used features. Results are based on the 30 best networks in terms of the fitness on the training set, one from each independent run.

			Flame	WDBC	Ionosphere	Sonar 50k	Sonar 250k
Fitness		SGE	**1.32 ± 0.25**	**1.46 ± 0.08**	**1.48 ± 0.18**	**1.85 ± 0.18**	**1.62 ± 0.16**
		GE	1.58 ± 0.36	1.55 ± 0.18	1.82 ± 0.28	2.01 ± 0.23	1.82 ± 0.16
Train	RMSE	SGE	**0.16 ± 0.13**	**0.19 ± 0.03**	**0.21 ± 0.07**	**0.34 ± 0.06**	**0.27 ± 0.04**
		GE	0.28 ± 0.15	0.24 ± 0.09	0.33 ± 0.10	0.38 ± 0.08	0.33 ± 0.06
	Accuracy	SGE	**0.96 ± 0.08**	**0.95 ± 0.02**	**0.93 ± 0.11**	**0.84 ± 0.12**	**0.92 ± 0.04**
		GE	0.90 ± 0.10	0.92 ± 0.12	0.79 ± 0.20	0.76 ± 0.16	0.86 ± 0.08
	AUCROC	SGE	**0.98 ± 0.04**	**0.99 ± 0.01**	**0.94 ± 0.04**	**0.91 ± 0.04**	**0.93 ± 0.04**
		GE	0.96 ± 0.05	0.98 ± 0.02	0.86 ± 0.18	0.88 ± 0.06	0.91 ± 0.04
	F-measure	SGE	**0.96 ± 0.08**	**0.93 ± 0.03**	**0.93 ± 0.18**	**0.78 ± 0.23**	**0.90 ± 0.05**
		GE	0.91 ± 0.09	0.88 ± 0.18	0.78 ± 0.32	0.70 ± 0.29	0.82 ± 0.16
Test	RMSE	SGE	**0.22 ± 0.13**	**0.23 ± 0.04**	**0.32 ± 0.05**	**0.44 ± 0.04**	**0.42 ± 0.05**
		GE	0.31 ± 0.15	0.27 ± 0.09	0.38 ± 0.07	0.45 ± 0.06	0.45 ± 0.04
	Accuracy	SGE	**0.93 ± 0.09**	**0.93 ± 0.02**	**0.87 ± 0.10**	**0.73 ± 0.09**	**0.78 ± 0.05**
		GE	0.88 ± 0.11	0.90 ± 0.12	0.76 ± 0.18	0.68 ± 0.12	0.73 ± 0.06
	AUCROC	SGE	**0.96 ± 0.08**	**0.98 ± 0.02**	**0.90 ± 0.05**	**0.82 ± 0.05**	**0.84 ± 0.05**
		GE	0.93 ± 0.08	0.97 ± 0.03	0.83 ± 0.09	0.81 ± 0.05	0.81 ± 0.05
	F-measure	SGE	**0.94 ± 0.09**	**0.91 ± 0.03**	**0.89 ± 0.17**	**0.64 ± 0.20**	**0.74 ± 0.07**
		GE	0.89 ± 0.09	0.86 ± 0.18	0.76 ± 0.31	0.61 ± 0.25	0.68 ± 0.14
Num. Neurons		SGE	4.87 ± 1.83	3.73 ± 1.53	3.53 ± 1.36	3.07 ± 1.39	4.23 ± 1.33
		GE	3.33 ± 1.40	3.13 ± 1.53	2.50 ± 1.41	2.53 ± 1.20	3.93 ± 1.78
Num. Features		SGE	2.00 ± 0.00	12.0 ± 6.51	12.1 ± 5.79	13.3 ± 6.42	21.93 ± 0.53
		GE	1.97 ± 0.18	8.40 ± 3.81	7.33 ± 5.33	9.40 ± 5.73	11.37 ± 4.45

i.e., the ability to evolve models that also perform well on the test data. Table 3.2 reports the RMSE, accuracy, AUCROC, and f-measure of the best networks for the training and test sets. We complement this information with the fitness of the best individuals (only measured in the training set), the number of neurons, and the number of used features. Each cell of the table is formatted as follows: mean ± standard deviation; bold values indicate the methodology that reports the best results for that dataset. Looking at the results summarised in the table, it is possible to acknowledge that SGE is consistently superior to GE. The fitness and RMSE values in SGE are lower than the ones attained by GE, and the accuracy, AUCROC, and f-measure values are higher in the experiments performed with SGE. This behaviour is consistent in the training and test sets. Moreover, standard deviation values are lower in SGE, which indicates that it consistently finds good results.

The difference between the training and test results in GE are, on average, 0.06, 0.05, 0.05 and 0.05 for the RMSE, accuracy, AUCROC, and f-measure, respectively. For SGE the differences are, on average, 0.08, 0.06, 0.05, 0.06 for the RMSE, accuracy, AUCROC, and f-measure, respectively. Even though the differences in SGE are slightly superior to the ones found in GE, it is our perception that this is not an indicator of overfitting, but rather a result of the SGE superior results. This is also supported by the complexity of the networks. SGE is capable of finding and optimising the weights and bias of topologies with a higher number of neurons and that use a larger number of problem features, proving that SGE performs a better exploration of the problems domain reaching solutions that perform better.

Table 3.3: Graphical overview of the statistical results of the comparison between SGE and GE. +, ++, and + + + correspond to statistically significant differences respectively with low, medium, and large effect sizes; ∼ stands for no statistical difference.

		Flame	WDBC	Ionosphere	Sonar 50k	Sonar 250k
Fitness		+++	∼	+++	++	+++
Train	RMSE	++	∼	+++	++	+++
	Accuracy	++	∼	+++	++	+++
	AUCROC	++	∼	+++	++	++
	F-measure	+++	∼	+++	∼	+++
Test	RMSE	∼	∼	+++	∼	++
	Accuracy	++	∼	++	∼	++
	AUCROC	++	∼	+++	∼	++
	F-measure	++	∼	+++	∼	++

SGE surpasses the results achieved by previous GE-based approaches that optimise ANNs to the same datasets. NNC [251] reports an accuracy of 0.9544 in the WDBC dataset and of 0.9034 in the ionosphere. The results are incomplete since they only show the results attained by the best networks. SGE mean accuracy values for WDBC and ionosphere are 0.93 and 0.87, respectively. However, the best-found networks have test accuracies up to 0.97 and 0.96 for the WDBC and ionosphere, respectively. GE-BP [228] reports an average test accuracy of 0.899 in the ionosphere benchmark. Despite lower than SGE, the authors use backpropagation to fine-tune the weights of the evolved networks and longer experiments (in terms of the number of performed evaluations). Later, in Section 3.1.2.7, we show that by fine-tuning the best evolved networks SGE reports an average test accuracy of 0.89 ± 0.03. More recently, GEGA [4] combines a GA with GE, obtaining a RMSE of 0.4398 and an accuracy of 0.7201 in the sonar benchmark, and 0.8694 in the ionosphere dataset. The proposed evolutionary approach takes into account the generalisation ability of the evolved networks, by measuring the classification accuracy on the test set, which is considered in the fitness function. By doing so, the evolutionary engine is provided with all the dataset instances, and no data is kept aside from the evolutionary process[1]. Thus, the results should be compared with our training ones, which are superior; even if compared to the test results, our approach is superior. Additionally, we also use less computational resources than previous approaches to train the networks, since we perform less evaluations. NNC report having used 1 million evaluations (500 individuals, 2000 generations), GE-BP used from 92000 to 250000 evaluations, and GEGA used 250000 evaluations (500 individuals, 500 generations).

3.1.2.6 Statistical Analysis

To verify if the differences between the approaches are meaningful, we perform statistical analysis. We start by checking whether the samples follow a Normal Distribution using the Kolmogorov-Smirnov and Shapiro-Wilk tests, with a significance level $\alpha = 0.05$. The tests reveal that we cannot say that SGE does not follow a Normal Distribution. However, for GE, the test revealed that it does not follow a Normal Distribution. Based on the results of these tests, we assume that our data does not follow a Normal Distribution, and will use non-parametric tests to perform the pairwise comparison for each of the recorded metrics. We used the Mann–Whitney U test with the same level of significance, $\alpha = 0.05$.

[1] By considering both the training and test sets to compute the fitness, there is no notion of a "true" test set, i.e., there is no partition of the dataset that can be used to measure generalisation by the end of evolution.

Table 3.3 presents a graphical overview of the statistical results: $\sim$ indicates no statistical difference between SGE and GE, and + indicates that SGE is statistically superior to GE. The effect size [66] is denoted by the number of + signs, where +, ++, and + + + correspond respectively to low ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$), and large ($r \geq 0.5$) effect sizes.

The results show that GE is never is never statistically superior to SGE. Moreover, the approaches are statistically equivalent in 15 occasions: on the test RMSE of the flame dataset, for all comparisons on the WDBC, and on the test and f-measure results of the sonar 50k dataset. For all the other comparisons SGE outperforms GE, and the effect size is medium in 16 occasions, and large in 14 occasions. These results confirm the viability of the proposed approach.

3.1.2.7 Analysis of the Evolved Networks

To better understand the impact of evolving the weights and topology of an ANN we fine-tune the networks built by SGE using BP in two different scenarios: (i) considering the evolved topology (T) and using the evolved weights (W) to seed the training (BP T+W); and (ii) considering the topology but not the evolved weights (BP T). In addition, we also compare the evolved networks with the results obtained by training one-hidden-layered Fully-Connected Neural Networks (FCNNs), with the hidden-layer having the number of nodes equal to the ones used by the best networks evolved using SGE. The results are depicted in Table 3.4 and are averages of 30 networks, one for each evolutionary run. For convenience, the first row (SGE) shows the results prior to the application of the BP algorithm (copied from Table 3.2). The BP algorithm is applied until convergence, up to a maximum of 1000 epochs, with the default parameters of pybrain [250]: a learning rate of 0.01, a learning rate decay of 1, and no momentum.

Focusing on the results obtained by applying BP to the evolved networks (BP T+W and BP T), it is clear that the evolution of both the topology and weights is important for achieving better results. When we do not consider the weights, the performance of the networks is inferior to when the evolved weights are considered. This difference is confirmed by the statistical analysis, which shows that the BP T+W setup is always statistically superior to the BP T setup ($\alpha = 0.05$). Moreover, the advantage of evolving the weights is also noticeable in the number of training epochs: when the evolved weights are used, the number of epochs that are required for the learning algorithm to converge is lower than when weights are initialised at random. This behaviour is observable in all the conducted experiments.

The results obtained by the handcrafted FCNNs are consistently worse than those obtained by BP T+W. In fact, BP T+W is statistically superior to the handcrafted 30 times (marked with two asterisks), and there is no statistical difference in the remaining cases. These results show that evolution is helpful and eases the process of finding effective networks to solve the considered problems. The comparison of the BP T+W setup with the baseline shows that, in the flame and WDBC datasets, the results achieved without fine-tuning are slightly superior to those resulting from further training using BP. After a careful examination, we conclude that this is a result of the implementation of the BP algorithm that splits the training data into two disjoint sets. Therefore it is not guided taking into account all data instances, and since the flame and WDBC problems have lower complexity by comparison with the ionosphere and sonar, evolution is able to attain a near-perfect tuning of the weights.

3.1.3 Overview

In this section, we have adapted SGE to the automatic search of the topology, and weights of one-hidden-layered ANNs. In particular, we introduce the following changes to the SGE evolutionary engine: (i) the mutation probabilities of each gene are different, and proportional to the number

Table 3.4: Comparison between the evolutionary results of SGE, the evolutionary results trained for longer by Back-Propagation considering the weights (BP T+W), and not considering the weights (BP T), and Fully Connected Neural Networks (FCNNs). * means that BP T+W is statistically superior to BP T and ** that it is statistically superior to BP T and FCNN.

			Flame	WDBC	Ionosphere	Sonar 50k	Sonar 250k
Train	RMSE	SGE	0.16 ± 0.13	0.19 ± 0.03	0.21 ± 0.07	0.34 ± 0.06	0.27 ± 0.04
		BP T+W	**0.19 ± 0.14****	**0.26 ± 0.07****	**0.20 ± 0.05****	**0.33 ± 0.06****	**0.26 ± 0.06****
		BP T	0.43 ± 0.08	0.36 ± 0.09	0.31 ± 0.06	0.48 ± 0.04	0.45 ± 0.05
		FCNN	0.40 ± 0.08	0.46 ± 0.06	0.23 ± 0.03	0.39 ± 0.06	0.40 ± 0.07
	Accuracy	SGE	0.96 ± 0.08	0.95 ± 0.02	0.93 ± 0.11	0.84 ± 0.12	0.92 ± 0.04
		BP T+W	**0.93 ± 0.13****	**0.90 ± 0.08****	**0.95 ± 0.03****	**0.84 ± 0.12****	**0.92 ± 0.04****
		BP T	0.70 ± 0.11	0.79 ± 0.12	0.88 ± 0.08	0.61 ± 0.11	0.66 ± 0.14
		FCNN	0.74 ± 0.11	0.66 ± 0.09	0.94 ± 0.02	0.77 ± 0.12	0.74 ± 0.16
	AUCROC	SGE	0.98 ± 0.04	0.99 ± 0.01	0.94 ± 0.04	0.91 ± 0.04	0.93 ± 0.04
		BP T+W	**0.94 ± 0.13****	**0.95 ± 0.09****	0.94 ± 0.04*	**0.91 ± 0.05****	**0.93 ± 0.03****
		BP T	0.63 ± 0.27	0.85 ± 0.19	0.87 ± 0.14	0.67 ± 0.15	0.70 ± 0.20
		FCNN	0.68 ± 0.30	0.56 ± 0.14	**0.96 ± 0.02**	0.84 ± 0.13	0.79 ± 0.18
	F-measure	SGE	0.96 ± 0.08	0.93 ± 0.03	0.93 ± 0.18	0.78 ± 0.23	0.90 ± 0.05
		BP T+W	**0.94 ± 0.10****	**0.82 ± 0.22****	**0.97 ± 0.02****	**0.79 ± 0.23****	**0.91 ± 0.04****
		BP T	0.81 ± 0.06	0.55 ± 0.40	0.91 ± 0.05	0.42 ± 0.32	0.50 ± 0.31
		FCNN	0.81 ± 0.08	0.11 ± 0.27	0.95 ± 0.01	0.70 ± 0.25	0.72 ± 0.16
Test	RMSE	SGE	0.22 ± 0.13	0.23 ± 0.04	0.32 ± 0.05	0.44 ± 0.04	0.42 ± 0.05
		BP T+W	**0.23 ± 0.14****	**0.28 ± 0.06****	**0.31 ± 0.05**	**0.43 ± 0.04***	**0.41 ± 0.05***
		BP T	0.43 ± 0.08	0.36 ± 0.09	0.35 ± 0.06	0.49 ± 0.03	0.47 ± 0.04
		FCNN	0.40 ± 0.08	0.46 ± 0.06	0.31 ± 0.04	0.43 ± 0.04	0.44 ± 0.05
	Accuracy	SGE	0.93 ± 0.09	0.93 ± 0.02	0.87 ± 0.10	0.73 ± 0.09	0.78 ± 0.05
		BP T+W	**0.91 ± 0.14****	**0.88 ± 0.08****	**0.89 ± 0.03***	**0.74 ± 0.09***	**0.78 ± 0.05****
		BP T	0.69 ± 0.11	0.78 ± 0.12	0.83 ± 0.08	0.59 ± 0.10	0.62 ± 0.12
		FCNN	0.75 ± 0.11	0.66 ± 0.09	0.88 ± 0.03	0.70 ± 0.11	0.69 ± 0.13
	AUCROC	SGE	0.96 ± 0.08	0.98 ± 0.02	0.90 ± 0.05	0.82 ± 0.05	0.84 ± 0.05
		BP T+W	**0.92 ± 0.16****	**0.94 ± 0.09****	0.90 ± 0.05*	**0.82 ± 0.05***	**0.85 ± 0.05****
		BP T	0.62 ± 0.25	0.84 ± 0.19	0.84 ± 0.16	0.63 ± 0.14	0.67 ± 0.18
		FCNN	0.71 ± 0.28	0.56 ± 0.14	**0.91 ± 0.05**	0.78 ± 0.11	0.76 ± 0.16
	F-measure	SGE	0.94 ± 0.09	0.91 ± 0.03	0.89 ± 0.17	0.64 ± 0.20	0.74 ± 0.07
		BP T+W	**0.93 ± 0.13****	**0.80 ± 0.23****	**0.91 ± 0.02***	**0.67 ± 0.19***	**0.75 ± 0.07****
		BP T	0.80 ± 0.06	0.54 ± 0.39	0.89 ± 0.05	0.40 ± 0.31	0.46 ± 0.28
		FCNN	0.81 ± 0.08	0.11 ± 0.27	0.91 ± 0.02	0.63 ± 0.22	0.67 ± 0.11
Epochs		SGE	-	-	-	-	-
		BP T+W	558 ± 482	343 ± 425	259 ± 307	99 ± 160	118 ± 223
		BP T	633 ± 438	697 ± 408	863 ± 274	519 ± 477	624 ± 472
		FCNN	749 ± 406	657 ± 358	818 ± 233	775 ± 391	654 ± 425

of integers of each gene; and (ii) the mutations are only applied to expressed genes, i.e., those that are used in the genotype to phenotype mapping. The goal of evolution is to minimise the error in the classification task. To avoid overfitting, and to deal with unbalanced datasets, we consider the RMSE per class, which is combined with the exponential function. By doing so, low errors have less impact than greater ones.

The proposed methodology is tested in four binary classification datasets, with increasing number of features: flame, WDBC, ionosphere, and sonar. Results show that SGE creates ANNs that are more effective than those generated by GE. The performance of the evolved ANNs is

better in terms of fitness, RMSE, accuracy, AUCROC, and f-measure in the training and test sets, which proves that SGE is capable of evolving consistent ANNs, that perform well beyond the training data. Notwithstanding, we performed a statistical analysis to assess the significance of the results. The analysis revealed that SGE is consistently statistically superior to GE. The results are also superior to those reported by other GE-based NE methods.

As the reader may have noticed, we only tackled binary problems and the optimisation of the neurons of a single layer. With the current approach, to implement a network that has more than a single output neuron, the only possibility would be to add a production rule that creates multiple sigexpr (see Grammar 3.1). For example, for a network with two output neurons, we would have: <network> ::= <sigexpr> <sigexpr>. The problem is that this would make it impossible to re-use any of the hidden-layer neurons. This is a limitation of standard grammar-based approaches: the inability to generate dynamic production rules, i.e., it is impossible to keep track of how many neurons there are in the network, and where they are located (when we have multiple layers). Next, we introduce an approach to overcome this issue.

3.2 Evolution of Multi-Layered Artificial Neural Networks

As demonstrated in the previous section, GE and SGE are not adequate for the generation of multi-layered ANNs. This happens because both approaches are unable to promote the re-use of the evolutionary units, i.e., if we have multiple hidden-layers, it is impossible to track down how many neurons there are in each specific layer so that multiple connections can be established to it. To create the neurons in grammar-based approaches, typically, a recursive production rule is applied. In the previous example of Grammar 3.1, the sigexpr rule creates an ordered sequence of neurons (lines 1 and 2). However, this set of neurons encodes a specific hidden-layer and is not able to encode multiple layers. To overcome the difficulties in optimising multi-layered ANNs, we need to devise a strategy to adapt the grammar as evolution proceeds, so that we can keep track of the number of hidden-layers and neurons in each layer.

To adapt the grammar as evolution proceeds, we introduce dynamic production rules – grammar production rules that change according to mutations in the individual's genotype. However, as a side-effect, we need an evolutionary engine that can cope with production rules that are created and/or modified during evolution. To that end, we propose DSGE.

To facilitate the introduction of the above concepts, we start by detailing DSGE (Section 3.2.1), and comparing DSGE to GE and SGE on the evolution of one-hidden-layered networks (Section 3.2.2). Next, we introduce dynamic production rules (Section 3.2.3), and apply DSGE together with dynamic production rules to the evolution of multi-layered ANNs (Section 3.2.4).

3.2.1 Dynamic Structured Grammatical Evolution

Dynamic Structured Grammatical Evolution (DSGE) is an extension to SGE. SGE, despite able to overcome the GE low locality and high redundancy issues, still has three main limitations: (i) to deal with recursion the grammar is pre-processed so that the maximum tree-size of each non-terminal symbol is determined, and intermediate grammar derivation rules are created to mimic the recursion process; (ii) the genotype size is a-priori defined, and the individuals initialised to the maximum size, i.e., the maximum number of required integers for expanding the non-terminals are generated, and even when they are not used in the grammar expansion (non-coding genes) they can suffer mutations; and (iii) the evolutionary engine is limited to a fixed grammar that does not change across generations. The rationale in DSGE is to encode only the integers that are used in the grammatical expansion, and thus, the genetic operators only act upon

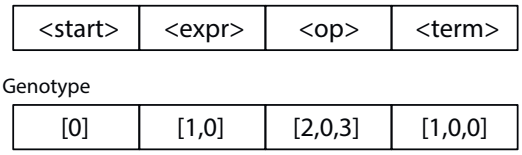


Figure 3.4: Example of a DSGE genotype considering Grammar 2.1.

encoding genes. Consequently, when more genes are necessary they are created on the fly in run-time, which as we will see later, is vital to deal with evolving grammars. On the other hand, the growth of the genotype enables DSGE to better deal with recursion because the genotype is expanded when successive recursive production rules are applied.

The code for DSGE is publicly available under the GNU GPL license in the following GitHub repository: `https://github.com/nunolourenco/sge3`.

3.2.1.1 Representation

The representation of the individuals is close to the proposed in SGE. Each candidate solution is represented by a set of ordered derivation steps, where each specific sequence stores the expansions of the non-terminal symbol it encodes. Unlike SGE, in DSGE the size of the genotype is variable. The integers encoding the expansions of the non-terminals are generated as they are required in the genotype to phenotype mapping (discussed in Section 3.2.1.2). To prevent the genotype from reaching untrackable sizes, we introduce a new variable – tree-depth – that similarly to standard GP defines the maximum depth of the recursion.

The new representation eases the expansion of recursive rules. While, in SGE, the recursive rules are unfolded into a fixed number of non-recursive rules (that match the maximum tree-depth), in DSGE this pre-processing step is not needed because the genotype can grow continuously (but limited by the tree-depth that defines the maximum recursion level). Therefore, in DSGE, the recursive rules are encoded as any other non-recursive rule.

To create the initial population, each individual is created using Algorithm 1. The procedure receives as input the grammar (grammar), the maximum tree-depth (max_depth), the current genotype (genotype), the non-terminal symbol to be expanded (symbol), and the current sub-tree depth (depth). The genotype variable is initially empty, the non-terminal symbol to be expanded is set to the grammar axiom, and the current sub-tree depth is set to 0. Then, for the non-terminal symbol given as input, one of the possible derivation rules is selected (lines 2-11), and the non-terminal symbols of the chosen derivation rule are recursively expanded (lines 12-15). However, when selecting the expansion rule, there is the need to check whether or not the maximum sub-tree depth has already been reached (lines 3-5). When that happens, only non-recursive derivation rules can be selected for expanding the non-terminal symbol (lines 6-7).

An example of the genotype of a candidate solution is depicted in Figure 3.4. The example is based in Grammar 2.1, and thus we need expansion possibilities for the non-terminal symbols start, expr, op and term. The genotype encodes the phenotype $(0.5/x_1)+x_1*x_1$. The genotype to phenotype mapping procedure is described next, in Section 3.2.1.2.

3.2.1.2 Genotype to Phenotype Mapping Function

To map the candidate solutions genotype into the phenotype, we use Algorithm 2. The algorithm is similar to the one used to initialise the genotype but, instead of randomly selecting the derivation rule to use in the expansion of the non-terminal symbols, we use the choices that are

Algorithm 1 DSGE initialisation procedure.

```
1: procedure CREATE_INDIVIDUAL(grammar, max_depth, genotype, symbol, depth)
2:     expansion = randint(0, len(grammar[symbol])-1)
3:     if is_recursive(symbol) then
4:         if expansion in grammar.recursive(symbol) then
5:             if depth ≥ max_depth[symbol] then:
6:                 non_rec = grammar.non_recursive(symbol)
7:                 expansion = choice(non_rec)
8:     if symbol in genotype then
9:         genotype[symbol].append(expansion)
10:    else
11:        genotype[symbol] = [expansion]
12:    expansion_symbols = grammar[symbol][expansion]
13:    for sym in expansion_symbols do
14:        if not is_terminal(sym) then
15:            create_individual(grammar, max_depth, genotype, symbol, depth+1)
```

Algorithm 2 DSGE genotype to phenotype mapping procedure.

```
1: procedure MAPPING(genotype, grammar, max_depth, read_integers, symbol, depth)
2:     phenotype = ""
3:     if symbol not in read_integers then
4:         read_integers[symbol] = 0
5:     if symbol not in genotype then
6:         genotype[symbol] = []
7:     if read_integers[symbol] ≥ len(genotype[symbol]) then
8:         if depth ≥ max_depth[symbol] then
9:             generate_terminal_expansion(genotype, symbol)
10:        else
11:            generate_expansion(genotype, symbol)
12:    gen_int = genotype[symbol][read_integer[symbol]]
13:    expansion = grammar[symbol][gen_int]
14:    read_integers[symbol] += 1
15:    for sym in expansion do
16:        if is_terminal(sym) then
17:            phenotype += sym
18:        else
19:            phenotype += mapping(genotype, grammar, max_depth, read_integers, sym,
    depth+1)
20:    return phenotype
```

encoded in the individual's genotype (lines 12-19). During evolution, the genetic operators may change the genotype in a way that requires a larger number of integers than the ones available. To deal with it, the following genotype's repair procedure is applied: new derivation rules are selected at random and added to the genotype of the individual (lines 3-11).

Table 3.5 shows an example of the application of the mapping procedure to the genotype of Figure 3.4. The example shreds evidence on how the integers encoding the derivation steps are

Table 3.5: Example of the mapping procedure of DSGE. Each row represents a derivation step. The list of genes represents the integers needed for expanding the start, expr, op, and term non-terminals, respectively. The genotype is based in Grammar 2.1 and in the genotype of Figure 3.4, but where the last codon from the term non-terminal was removed.

Derivation step	DSGE
<start>	[[0], [1, 0], [2, 0, 3], [1, 0, 0]]
<expr><op><expr>	[[], [1, 0], [2, 0, 3], [1, 0, 0]]
(<term><op><term>) <op><expr>	[[], [0], [2, 0, 3], [1, 0, 0]]
(0.5 <op><term>) <op><expr>	[[], [0], [2, 0, 3], [0, 0]]
(0.5 / <term>) <op><expr>	[[], [0], [0, 3], [0, 0]]
(0.5/ x_1) <op><expr>	[[], [0], [0, 3], [0]]
(0.5/ x_1) + <expr>	[[], [0], [3], [0]]
(0.5/x_1)+ <term><op><term>	[[], [], [3], [0]]
(0.5/x_1)+x_1 <op><term>	[[], [], [3], []]
(0.5/x_1)+x_1* <term>	[[], [], [], []]
new expression gene (random(0, 1))	[[], [], [], [0]]
(0.5/x_1)+x_1*x_1	[[], [], [], []]

consumed. The phenotype that corresponds to the genotype is $(0.5/x_1)+x_1*x_1$.

3.2.1.3 Genetic Operators

Mutation is restricted to the integers that are used in the genotype to phenotype mapping and changes a randomly selected expansion to another valid one, constraint to the maximum sub-tree depth. To do so, we first select one gene. The probability of selecting the i-th gene (p_i) is proportional to the number of integers of that non-terminal symbol that are used in the genotype to phenotype mapping ($read\_integers$):

$$p_i = \frac{read\_integers_i}{\sum_{j=1}^{n} read\_integers_j},$$

where n is the total number of genes. Additionally, genes, where there is just one possibility for expansion, are not considered for mutation purposes. After selecting the gene to be mutated, we randomly select one of its integers and replace it with another valid possibility.

Crossover is used to recombine two parents (selected using tournament selection) to generate two offspring. We use the one-point crossover. As such, after selecting the cutting point (at random), the genetic material is exchanged between the two parents. The choice of the cutting point is done at the gene level and not at the integers level, i.e., what is exchanged between parents are genes and not the integers.

3.2.2 Comparison between GE, SGE, and DSGE

To compare GE, SGE and DSGE, we focus on the optimisation of one-hidden-layered ANNs. We will still not tackle the evolution of multi-layered ANNs because, for that, we need dynamic production rules, which will be presented next (in Section 3.2.3). To ease the comparison of the different GGP methods, we follow a setup similar to the one previously described in Section 3.1. Consequently, we use Grammar 3.1 to evolve network topologies that are limited to one hidden-layer and one output neuron. The experiments are conducted in the flame, WDBC, ionosphere

Table 3.6: Experimental parameters of the comparison between GE, SGE, and DSGE.

Parameter	**Value**
Num. runs	30
Population size	100
Num. generations	500
Crossover rate	95%
Mutation rate	1 mutation in 95% of the individuals
Tournament size	3
Elite size	1%
GE Parameter	**Value**
Individual size	200
Wrapping	0
SGE Parameter	**Value**
Recursion level	6
DSGE Parameter	**Value**
Max. depth	{sigexpr: 6, sum: 3}
Dataset Parameter	**Value**
Training percentage	70%
Testing percentage	30%

and sonar datasets (check Section 3.1.2.1). Despite not the core of this Thesis, we also investigate the performance of DSGE on the optimisation of Scikit-Learn classification pipelines. The results are reported in Appendix A.

Next, in Section 3.2.2.1, we detail the experimental setup. The experimental results are presented and discussed in Section 3.2.2.2.

3.2.2.1 Experimental Setup

Table 3.6 details the parameters used for the experiments conducted to compare GE, SGE and DSGE. To make the exploration of the search space similar, we only allow one mutation in 95% of the population individuals. Otherwise, if we define per-gene mutation probabilities, and as the definition of genes is different between the considered methods, we would have very different domain exploration approaches. To ensure that all methods can generate similar network topologies, we restrict the domains in terms of the neurons and connectivity: GE, SGE and DSGE are constrained to generated networks with up to 7 neurons and 8 connections in each neuron.

The datasets are all randomly partitioned in the same way: 70% of each class instances are used for training, and the remaining 30% for testing. We assign fitness using, exclusively, training data, and thus test data is never used during evolution. As such, performance on test data is a fair indication of the model's performance. Similar to the previous experiments, the datasets are used as they are obtained.

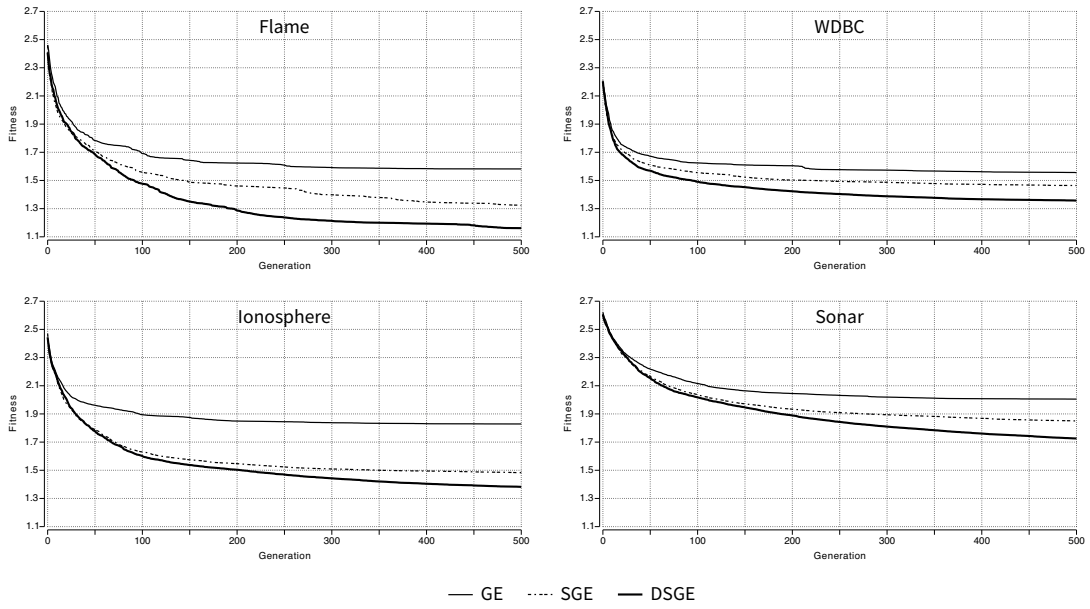


Figure 3.5: Comparison between GE, SGE, and DSGE on the evolution of one-hidden-layered networks. Fitness evolution of the best individuals across generations for the flame, WDBC, ionosphere, and sonar datasets. The results are averages of 30 independent runs.

3.2.2.2 Experimental Results

Figure 3.5 shows the evolution of the fitness across generations for the GE, SGE, and DSGE on the flame, WDBC, ionosphere, and sonar datasets. Results are averages of 30 independent runs. The figure shows that DSGE outperforms the other methods in all the considered problems, indicating that the new representation promotes the efficient exploration of the search space.

The results are presented in more detail in Table 3.7. Results are averages of the best network (in terms of fitness) of each of the evolutionary runs, and are formatted as follows: mean $\pm$ standard deviation; bold highlights the best result. For all experiments we record the fitness, RMSE, AUCROC and f-measure. All metrics are computed for the training and test sets, except the fitness. In addition, we also report the average number of neurons and used features.

An analysis of the results shows that DSGE performs better than the other approaches in all the datasets, and in all the considered metrics. Regarding the structure of the generated networks, DSGE is capable of finding solutions that are more complex in terms of the number of used neurons and input features. As the complexity of the networks grows, more real-values have to be tuned. Nonetheless, DSGE is capable of performing such tuning, reaching solutions that, despite being more complex, perform better in the considered classification problems.

Table 3.7: Evolution of one-hidden-layered ANNs with GE, SGE, and DSGE. Results are averages of 30 independent runs; bold highlights the best result. The tags +, ++ and + + + correspond to statistically significant differences respectively with low, medium, and large effect sizes; $\sim$ stands for no statistical difference. The results of GE and SGE are the same as in Table 3.2.

			Flame	WDBC	Ionosphere	Sonar
Fitness		GE	1.58 ± 0.36	1.55± 0.18	1.82 ± 0.28	2.01 ± 0.23
		SGE	1.32 ± 0.25	1.46 ± 0.08	1.48 ± 0.18	1.85 ± 0.18
		DSGE	**1.16 ± 0.13**$^{+++}$	**1.36 ± 0.06**$^{+++}$	**1.38 ± 0.13**$^{+++}$	**1.73 ± 0.12**$^{+++}$
Train	RMSE	GE	0.28 ± 0.15	0.24 ± 0.09	0.33 ± 0.10	0.38 ± 0.08
		SGE	0.16 ± 0.13	0.19 ± 0.03	0.21 ± 0.07	0.34 ± 0.06
		DSGE	**0.08 ± 0.06**$^{+++}$	**0.15 ± 0.03**$^{+++}$	**0.17 ± 0.05**$^{++}$	**0.30 ± 0.05**$^{++}$
	Accuracy	GE	0.90 ± 0.10	0.92 ± 0.12	0.79 ± 0.20	0.76 ± 0.16
		SGE	0.96 ± 0.08	0.95 ± 0.02	0.93 ± 0.11	0.84 ± 0.12
		DSGE	**0.99 ± 0.03**$^{+++}$	**0.97 ± 0.01**$^{+++}$	**0.97 ± 0.03**$^{+++}$	**0.90 ± 0.04**$^{++}$
	AUCROC	GE	0.96 ± 0.05	0.98 ± 0.02	0.86 ± 0.18	0.88 ± 0.06
		SGE	0.98 ± 0.04	0.99 ± 0.01	0.94 ± 0.04	0.91 ± 0.04
		DSGE	**0.99 ± 0.01**$^{+++}$	**0.99 ± 0.00**$^{+++}$	**0.96 ± 0.03**$^{+++}$	**0.93 ± 0.04**$^{\sim}$
	F-measure	GE	0.91 ± 0.09	0.88 ± 0.18	0.78 ± 0.32	0.70 ± 0.29
		SGE	0.96 ± 0.08	0.93 ± 0.03	0.93 ± 0.18	0.78 ± 0.23
		DSGE	**0.99 ± 0.02**$^{+++}$	**0.96 ± 0.02**$^{+++}$	**0.98 ± 0.02**$^{+++}$	**0.88 ± 0.05**$^{++}$
Test	RMSE	GE	0.31 ± 0.15	0.27 ± 0.09	0.38 ± 0.07	0.45 ± 0.06
		SGE	0.22 ± 0.13	0.23 ± 0.04	0.32 ± 0.05	0.44 ± 0.04
		DSGE	**0.14 ± 0.08**$^{++}$	**0.20 ± 0.03**$^{++}$	**0.28 ± 0.04**$^{+++}$	**0.43 ± 0.04**$^{\sim}$
	Accuracy	GE	0.88 ± 0.11	0.90 ± 0.12	0.76 ± 0.18	0.68 ± 0.12
		SGE	0.93 ± 0.09	0.93 ± 0.02	0.87 ± 0.10	0.73 ± 0.09
		DSGE	**0.97 ± 0.05**$^{+++}$	**0.95 ± 0.02**$^{+++}$	**0.90 ± 0.03**$^{++}$	**0.76 ± 0.05**$^{\sim}$
	AUCROC	GE	0.93 ± 0.08	0.97 ± 0.03	0.83 ± 0.09	0.81 ± 0.05
		SGE	0.96 ± 0.08	0.98 ± 0.02	0.90 ± 0.05	0.82 ± 0.05
		DSGE	**0.99 ± 0.03**$^{++}$	**0.98 ± 0.01**$^{++}$	**0.93 ± 0.04**$^{++}$	**0.83 ± 0.04**$^{\sim}$
	F-measure	GE	0.89 ± 0.09	0.86 ± 0.18	0.76 ± 0.31	0.61 ± 0.25
		SGE	0.94 ± 0.09	0.91 ± 0.03	0.89 ± 0.17	0.64 ± 0.20
		DSGE	**0.98 ± 0.04**$^{+++}$	**0.93 ± 0.03**$^{+++}$	**0.93 ± 0.02**$^{++}$	**0.72 ± 0.06**$^{\sim}$
Num. Neurons		GE	3.33 ± 1.40	3.13 ± 1.53	2.50 ± 1.41	2.53 ± 1.20
		SGE	4.87 ± 1.83	3.73 ± 1.53	3.53 ± 1.36	3.07 ± 1.39
		DSGE	6.47 ± 1.20	6.23 ± 1.58	5.97 ± 1.78	6.13 ± 1.69
Num. Features		GE	1.97 ± 0.18	8.40 ± 3.81	7.33 ± 5.33	9.40 ± 5.73
		SGE	2.00 ± 0.00	12.0 ± 6.51	12.1 ± 5.79	13.3 ± 6.42
		DSGE	2.00 ± 0.00	14.5 ± 3.52	13.3 ± 4.74	17.6 ± 4.66

To verify if the differences between the tested approaches are significant, we perform a statistical analysis. In Section 3.1.2.6, we have already demonstrated that SGE is consistently statistically superior to GE. As such, we will now focus in comparing DSGE to SGE. To check if the samples follow a Normal Distribution, we use the Kolmogorov-Smirnov and Shapiro-Wilk tests, with a significance level $\alpha = 0.05$. The tests reveal that the data does not follow a Normal Distribution and, as such, a non-parametric test (Mann-Whitney U, $\alpha = 0.05$) will be used to perform the pairwise comparison for each recorded metric. Table 3.7 uses a graphical overview to present the results of the statistical analysis: the tag $\sim$ indicates no statistical difference between DSGE and SGE, and the tag + represents that DSGE is statistically superior to SGE. The effect size is denoted by the number of + tags, where +, ++ and +++ correspond respectively to low ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$) and large ($r \geq 0.5$) effect sizes. The statistical analysis reveals that DSGE is consistently statistically superior to SGE: DSGE is never worse than SGE and is only equivalent in 5 situations. In all the remaining 31 comparisons, DSGE is statistically superior, with a medium effect size in 11 occasions, and a large effect size in 20 occasions.

Focusing on the comparison between the training and test results, in DSGE the differences between training and test performance are, on average, 0.09, 0.06, 0.04 and 0.06, for RMSE, accuracy, AUCROC and f-measure, respectively. For SGE the differences are, on average, 0.08, 0.06, 0.04 and 0.06 for the RMSE, accuracy, AUCROC and f-measure, respectively. Finally, for GE the differences are, on average, 0.05, 0.04, 0.04 and 0.04 for RMSE, accuracy, AUCROC and f-measure, respectively. Despite the fact that the differences in DSGE are superior to the ones in SGE and GE, it is our perception that this is a result of the superior DSGE results and not an indicator of overfitting.

In Section 3.1 we have shown that the results obtained by SGE are superior to those of other grammar-based approaches (namely, the ones described in [4, 228, 251]). Additionally, we demonstrated that it is beneficial to evolve both the topology and weights of a network since this setup reported results that are superior to hand-crafting the ANNs and then training using BP. DSGE is statistically superior to SGE, and consequently superior to previous methods.

3.2.3 Dynamic Production Rules

As discussed previously, to enable the optimisation of multi-layered ANNs, we need to keep track of the overall structure of the networks. This is the core rationale behind the definition of dynamic production rules. The goal is to adapt the grammar to each individual in order to incorporate the number of layers and the number of neurons in each layer. If we know the placement of the neurons in each layer we can then re-use them, i.e., a neuron in a given layer can serve as input to others in subsequent layers.

As in standard production rules, dynamic production rules encode the expansion possibilities for a given non-terminal symbol. The main difference is that, in order to turn each neuron accessible and re-usable, each production rule is split into indexable sub production rules. In our use-case of multi-layered networks, we have a dynamic production rule that specifies the inputs that the neurons in a hidden-layer can receive. This dynamic production rule is split into multiple production rules, where each encodes the inputs of a specific hidden-layer. The number of splits is proportional to the number of hidden-layers of the network. More precisely, consider Grammar 3.2, which is an extension of Grammar 3.1 to the evolution of multi-layered ANNs: a network is formed by a set of hidden-layers and an output layer. The layers are formed by a set of neurons (nodes), where each neuron can establish connections to multiple others (encoded by the sum production rule that connects to multiple features with a given weight). The grammar is biased towards recursion, i.e., the recursive production rules that deal with the growth of the network in terms of layers and neurons (hidden-layers, and nodes, respectively) have more

<network> ::= <hidden-layers>-- <output-layer> (1)
<hidden-layers> ::=<hidden-layers> -- <hidden-layers> (2)
| <layer> -- <hidden-layers> (3)
| <layer> (4)
<output-layer> ::= sig(<sum> + <float>) (5)
<layer> ::= <nodes> (6)
<nodes> ::= <nodes> - <nodes> (7)
| <node> - <nodes> (8)
| <node> - <node> (9)
<node> ::= sig(<sum> + <float>) (10)
<sum> ::= <float> * <features> (11)
| <sum> + <sum> (12)
| <sum> + <sum> (13)
<float> ::= <digit>.<digit><digit> (14)
| - <digit>.<digit><digit> (15)
<digit> ::= 0 | 1 | 2 | 3 | 4 (16)
| 5 | 6 | 7 | 8 | 9 (17)
<features> ::= x_1 (18)
| ... (19)
| x_n (20)

Grammar 3.2: Grammar used by DSGE for evolving multi-layered ANNs. The n represents the number of features of the problem; - and -- are used as neuron and layer separators, respectively.

expansion possibilities that call itself. Based in Grammar 3.2, and considering we want to encode an ANN with two hidden-layers, with 4 and 3 nodes, respectively, the dynamic production rule <features> is split into three indexable rules:

<features-1> ::= $x_1 \mid \ldots \mid x_n$

<features-2> ::= $h_{1,1} \mid h_{1,2} \mid h_{1,3} \mid h_{1,4}$

<features-3> ::= $h_{2,1} \mid h_{2,2} \mid h_{2,3}$,

where x, and n respectively represent the input features, and the total number of features (defined by the user in the input grammar), and $h_{k,j}$ the output of the j-th neuron of the k-th layer. Therefore, <features-1>, <features-2>, and <features-3> encode the input possibilities of the first hidden-layer, second hidden-layer, and output layer, respectively.

From the above example it is clear that different individuals can have a different network structure (number of layers and neurons in each layer). For example, if in addition to two hidden-layers (with 4 and 3 neurons, respectively), we aim at encoding a network with a third hidden-layer (with 2 neurons), we would require another production rule <features-4> ::= $h_{3,1}$ | $h_{3,2}$. From these examples, it is clear that the use of dynamic production rules makes the grammar a property of the individual.

The two above examples expose the challenge that is to promote evolution using dynamic production rules. For example, the addition of a new hidden-layer indirectly implies that there

Table 3.8: Experimental parameters for the experiments on the evolution of multi-layered ANNs with dynamic production rules and DSGE.

Parameter	**Value**
Num. runs	30
Population size	100
Num. generations	1500 / 3500
Crossover rate	95%
Mutation rate	1 mutation in 95% of the individuals
Tournament size	3
Elite size	1%
DSGE Parameter	**Value**
Max. depth	{hidden-layers: 3, nodes: 5, sum: 4}
Dataset Parameter	**Value**
Training percentage	70%
Testing percentage	30%

is a new production rule in the grammar, which we were previously not considering. Adding new production rules in run-time has a drastic impact in both GE and SGE: in GE it affects all derivation steps from the decoding of the dynamic production rule onwards; in SGE there is no standard procedure to expand the genotype. This is the reason why we pair dynamic production rules with DSGE. The genotype in DSGE only keeps the encoding genes, and the genotype to phenotype mapping is responsible for generating new genetic material if needed, and thus is natively capable of dealing with new production rules.

The main challenge that remains when combining dynamic production rules with DSGE is that different individuals can have a varying number of genes. This is a consequence of a different number of splits of the dynamic production rules across individuals. Therefore, the crossover operator is changed to a bit-mask operator that only allows the exchange of genes that are common to both parents. Nonetheless, the application of the genetic operators can generate invalid connections (e.g., if in the first hidden-layer of two individuals one has 2 neurons and the other 10), and thus in the genotype to phenotype procedure, we must check if the connection we are decoding is valid or not. We may be faced with a connection to a neuron in a previous layer that has been erased, or a connection to a neuron that does not exist. In such a scenario, the genotype is fixed by replacing the faulty connection with a valid one. The dynamic production rule is fixed too.

3.2.4 Dynamic Production Rules to Evolve Multi-Layered Artificial Neural Networks

To investigate the potential of the combination of dynamic production rules with DSGE on the evolution of multi-layered ANNs, we address the same datasets (flame, WDBC, ionosphere, and sonar), but using Grammar 3.2. In the production rules <hidden-layers>, <nodes> and <sum> a higher chance is given to the recursive expansion of the non-terminal symbols due to the higher difficulty of tuning parameters in deeper and more complex topologies; this was decided after preliminary experimentation.

The experimental setup is detailed in Section 3.2.4.1, and the experimental results are pre-

sented and discussed in Section 3.2.4.2.

3.2.4.1 Experimental Setup

Table 3.8 details the experimental parameters used for the tests conducted with dynamic production rules and DSGE on the optimisation of multi-layered networks. The experiments are performed using Grammar 3.2, where the production rule <features> is dynamic.

When evolving feed-forward multi-layered ANNs, we have two possibilities regarding the inputs of each layer, i.e., we can restrict a layer to connect only to the previous layer, or we can let the neurons of a layer to connect to all previous layers. In the first case, the i-th layer must connect to neurons from the <features-i> dynamic production rule. In the latter case, the i-th layer can connect to neurons from the <features-1> to the <features-i> dynamic production rules. In the current experiments, we have decided to enable the connections of the neurons to be established to any of the neurons in previous hidden-layers. The exception is the output layer that can only connect to hidden-neurons, and not to the input features. When establishing the connections between neurons, the probability of choosing a neuron in the previous layer is proportional to the number of previous hidden-layers:

$$P(neuron_{i-1}) = \sum_{j=1}^{j=i-2} P(neuron_j),$$

i.e., when establishing the connections of a neuron in the i-th layer, the probability of linking to a neuron in the previous layer $(i-1)$ is equal to the probability of linking to a neuron in the remaining layers $(1, \ldots, i-2)$. The rationale is to reduce the likelihood of the emergence of deep networks with neurons that are of no use, that is, neurons that are not connected (directly or indirectly) to output neurons.

3.2.4.2 Evolutionary Results

The results of the experiments combining DSGE and dynamic production rules to the optimisation of multi-layered ANNs considering Grammar 3.2 are presented in Table 3.9. The first row of each metric shows the same results of Table 3.7 with DSGE on the evolution of one-hidden-layered networks (1 H-L). The second row shows the results using the new grammar formulation of the networks that have potentially more than one-hidden-layer ($\geq$ 1 H-Ls). Additionally, on the last row, we restrict the analysis of the results to the networks that are evolved with the new grammar but only considering those that have more than one-hidden-layer ($>$ 1 H-L), thus discarding the evolutionary runs that resulted in networks with just one hidden-layer.

Results of 1 H-L and $\geq$ 1 H-Ls are averages of 30 independent runs. In the $>$ 1 H-Ls setup, the results are averages of 21, 19, 18 and 21 independent runs for the flame, WDBC, ionosphere, and sonar datasets, respectively. That is, for the flame, WDBC, ionosphere, and sonar datasets in 9, 11, 12, and 9 runs the best network is composed of only one hidden-layer. For the flame and WDBC, we perform runs with 1500 generations, and for the ionosphere and sonar, we use 3500 generations. The rationale behind the different number of generations is related to the number of features of each problem and the possible progression margin: in flame and WDBC the training and test metrics already report close-to-optimum values, thus suggesting that one hidden-layer is likely enough to solve the classification task.

Table 3.9: Evolution of multi-layered ANNs based on the combination of dynamic production rules with DSGE. Results are averages of 30 independent runs; bold highlights the best result. The tags + and ∼ symbols represent the result of statistical tests, and have the same meaning as in Table 3.7. The 1 H-L row reports the same results of Table 3.7 for the DSGE setup; the ≥ 1 H-Ls row reports the results of combining production rules with DSGE (all evolutionary runs); and in the > 1 H-Ls row we only focus on the evolutionary runs that generate networks with more than one hidden-layer.

			Flame	WDBC	Ionosphere	Sonar
Fitness		1 H-L	1.16 ± 0.13	1.36 ± 0.06	1.38 ± 0.13	1.73 ± 0.12
		≥ 1 H-Ls	1.15 ± 0.18$^{\sim}$	1.35 ± 0.13$^{\sim}$	1.36 ± 0.19$^{\sim}$	1.57 ± 0.18$^{\sim}$
		> 1 H-Ls	**1.08 ± 0.13**$^{+++}$	**1.32 ± 0.11**$^{\sim}$	**1.30 ± 0.11**$^{+++}$	**1.53 ± 0.15**$^{+++}$
Train	RMSE	1 H-L	0.08 ± 0.06	0.15 ± 0.03	0.17 ± 0.05	0.30 ± 0.05
		≥ 1 H-Ls	0.07 ± 0.08$^{\sim}$	0.15 ± 0.05$^{\sim}$	0.17 ± 0.08$^{\sim}$	0.26 ± 0.07$^{\sim}$
		> 1 H-Ls	**0.07 ± 0.05**$^{+++}$	**0.07 ± 0.04**$^{\sim}$	**0.14 ± 0.05**$^{+++}$	**0.25 ± 0.06**$^{+++}$
	Accuracy	1 H-L	0.99 ± 0.03	**0.97 ± 0.01**	0.97 ± 0.03	0.90 ± 0.04
		≥ 1 H-Ls	0.99 ± 0.02$^{\sim}$	0.97 ± 0.02$^{\sim}$	0.97 ± 0.03$^{\sim}$	0.92 ± 0.05$^{\sim}$
		> 1 H-Ls	**0.99 ± 0.02**$^{\sim}$	0.97 ± 0.02$^{\sim}$	**0.98 ± 0.02**$^{\sim}$	**0.93 ± 0.04**$^{+++}$
	AUCROC	1 H-L	0.99 ± 0.01	0.99 ± 0.00	0.96 ± 0.03	0.93 ± 0.04
		≥ 1 H-Ls	0.99 ± 0.01$^{\sim}$	0.99 ± 0.01$^{\sim}$	0.96 ± 0.04$^{\sim}$	0.93 ± 0.05$^{\sim}$
		> 1 H-Ls	**1.00 ± 0.01**$^{\sim}$	**0.99 ± 0.00**$^{\sim}$	**0.97 ± 0.02**$^{\sim}$	**0.93 ± 0.04**$^{\sim}$
	F-measure	1 H-L	0.99 ± 0.02	0.96 ± 0.02	0.98 ± 0.02	0.88 ± 0.05
		≥ 1 H-Ls	0.99 ± 0.02$^{\sim}$	0.96 ± 0.03$^{\sim}$	0.97 ± 0.02$^{\sim}$	0.91 ± 0.06$^{\sim}$
		> 1 H-Ls	**0.99 ± 0.01**$^{\sim}$	**0.97 ± 0.03**$^{\sim}$	**0.98 ± 0.01**$^{+++}$	**0.92 ± 0.05**$^{+++}$
Test	RMSE	1 H-L	**0.14 ± 0.08**	**0.20 ± 0.03**	**0.28 ± 0.04**	**0.43 ± 0.04**
		≥ 1 H-Ls	0.17 ± 0.08$^{\sim}$	0.20 ± 0.04$^{\sim}$	0.31 ± 0.05$^{\sim}$	0.44 ± 0.05$^{\sim}$
		> 1 H-Ls	0.16 ± 0.08$^{\sim}$	0.20 ± 0.04$^{\sim}$	0.29 ± 0.05$^{\sim}$	0.43 ± 0.06$^{\sim}$
	Accuracy	1 H-L	0.97 ± 0.05	0.95 ± 0.02	0.90 ± 0.03	0.76 ± 0.05
		≥ 1 H-Ls	0.96 ± 0.04$^{\sim}$	0.95 ± 0.02$^{\sim}$	0.89 ± 0.03$^{\sim}$	0.77 ± 0.05$^{\sim}$
		> 1 H-Ls	**0.97 ± 0.04**$^{\sim}$	**0.95 ± 0.02**$^{\sim}$	**0.90 ± 0.03**$^{\sim}$	**0.78 ± 0.06**$^{\sim}$
	AUCROC	1 H-L	0.99 ± 0.03	0.98 ± 0.01	0.93 ± 0.04	**0.83 ± 0.04**
		≥ 1 H-Ls	0.98 ± 0.03$^{\sim}$	0.99 ± 0.01$^{\sim}$	0.91 ± 0.06$^{\sim}$	0.82 ± 0.07$^{\sim}$
		> 1 H-Ls	**0.99 ± 0.02**$^{\sim}$	**0.99 ± 0.01**$^{\sim}$	**0.93 ± 0.03**$^{\sim}$	0.82 ± 0.07$^{\sim}$
	F-measure	1 H-L	**0.98 ± 0.04**	0.93 ± 0.03	**0.93 ± 0.02**	0.72 ± 0.06
		≥ 1 H-Ls	0.97 ± 0.03$^{\sim}$	0.93 ± 0.03$^{\sim}$	0.92 ± 0.02$^{\sim}$	0.73 ± 0.07$^{\sim}$
		> 1 H-Ls	0.97 ± 0.03$^{\sim}$	**0.93 ± 0.03**$^{\sim}$	0.92 ± 0.02$^{\sim}$	**0.74 ± 0.08**$^{\sim}$
Num. H-Ls		1 H-L	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
		≥ 1 H-Ls	2.27 ± 0.98	2.23 ± 1.04	1.90 ± 0.84	2.37 ± 0.96
		> 1 H-Ls	2.81 ± 0.60	2.95 ± 0.52	2.50 ± 0.51	2.95 ± 1.12
Num. Neurons		1 H-L	3.33 ± 1.40	3.13 ± 1.53	2.50 ± 1.41	2.53 ± 0.38
		≥ 1 H-Ls	10.6 ± 6.49	9.73 ± 7.03	7.17 ± 4.98	9.27 ± 4.55
		> 1 H-Ls	12.5 ± 6.64	12.8 ± 6.70	9.33 ± 5.24	10.9 ± 4.12
Num. Features		1 H-L	2.00 ± 0.00	14.5 ± 3.52	13.1 ± 6.87	19.8 ± 8.39
		≥ 1 H-Ls	2.00 ± 0.00	16.3 ± 7.03	13.8 ± 6.66	21.8 ± 8.48
		> 1 H-Ls	2.00 ± 0.00	18.2 ± 6.91	15.8 ± 5.48	24.0 ± 7.54

The experimental results show that by using DSGE it is possible to evolve effective multi-layered ANNs. By comparing the first two rows (1 H-L and $\geq$ 1 H-Ls) it is also clear that the results are equivalent, which is confirmed by statistical analysis (Mann-Whitney, $\alpha = 0.05$) that shows no statistical differences. Focusing on the comparison between one-hidden-layered ANNs and those that have more than one hidden-layer ($>$ 1 H-Ls) a small, but noticeable difference exists. That difference is statistically significant in some of the training metrics, but in none of the test metrics. When there is a statistical difference the effect size is always large. Moreover, the difference is larger in the datasets that have more input features and a greater margin for improvement. On the contrary, almost no improvement is observable in the flame dataset, which is expected as the problem only has two features, and thus an ANN with one hidden-layer already performs close to optimal. In a nutshell, although there are no statistical differences between the performance of 1 H-L and $\geq$ 1 H-Ls runs, when the evolutionary process results in multi-layered ANNs the differences begin to emerge. This result shows that DSGE is able to cope with the higher dimensionality of the search space associated with the evolution of multi-layered topologies and indicates that in complex datasets, where there is a clear advantage in using deeper topologies, DSGE is likely to outperform other grammar-based approaches.

The complexity of the generated ANNs (in terms of the number of used neurons and hidden-layers) is greater when allowing the generation of multi-layered ANNs. The difference is even larger when we consider only those networks that have more than one hidden-layer. More neurons mean more weights and bias values that have to be optimised, making the evolutionary task more difficult. ANNs with fewer layers tend to have less neurons, and consequently are benefited from an evolutionary point of view, as less real values need to be tuned. Thus, a good proxy can be the use the evolutionary approach to evolve the initial set of weights and then apply a fine-tuning stage (e.g, backpropagation or resilient backpropagation) during a maximum number of epochs proportional to the number of neurons and/or hidden-layers. As we allow hidden-nodes to connect to nodes in previous layers (including input features) the number of used features is also higher.

3.2.5 Overview

To evolve multi-layered networks, we introduce dynamic production rules, i.e., production rules that in the use-case of multi-layered networks are split according to the number of hidden-layers of the individual. The splits generate indexable production rules that keep track of the network's structure, i.e., number of hidden-layers and neurons in each layer. To allow the use of dynamic production rules, the evolutionary engine needs to be capable of dealing with production rules that can be created and/or removed as evolution proceeds. To tackle this issue, we propose DSGE: a new genotypic representation for SGE. The gain is two-fold: (i) while in previous grammar-based representations the genotype encodes the largest allowed sequence, in DSGE the genotype grows as needed; and (ii) the maximum depth is defined for each sub-tree, and as such, there is no need to pre-process the grammar to compute the largest size of the recursive production rules. Therefore, DSGE enables the use of dynamic production rules.

The conducted experiments first compare the performance of GE, SGE, and DSGE on the evolution of one-hidden-layered ANNs. After assessing the performance of DSGE, we combine it with dynamic production rules on the optimisation of (potentially) multi-layered ANNs. The results show that DSGE can evolve one-hidden-layered ANNs (topology and weights) that perform statistically superior to those evolved by the other considered grammar-based approaches. Moreover, the results are also better than those obtained by hand-crafted ANNs fine-tuned using BP, and than the ones generated using other GE-based approaches [4, 228, 251]. Concerning the evolution of multi-layered ANNs, although the differences are not statistically significant, the

results show that DSGE combined with dynamic production rules is suitable for the evolution of multi-layered ANNs.

3.3 AutoEncoders for Compressed Representations

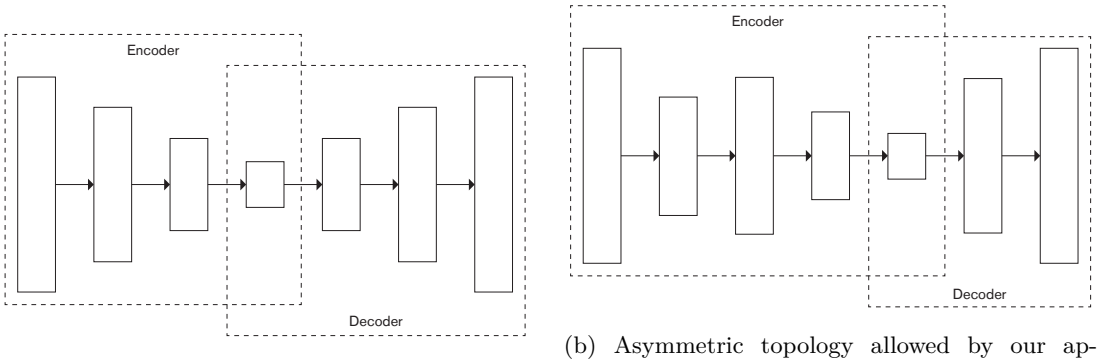
The work discussed in the previous sections focused on the optimisation of feed-forward fully-connected networks. Furthermore, the data to be classified consisted of features designed by experts. When addressing real-world applications the dimensionality (number of instances, and number of features) of the input signals tends to be high (e.g., images or sound), and it is not always straightforward for a non-expert user to identify the parts of the instances that are relevant or the features that help to solve the task. For these reasons, we investigate an automatic way to compress large data instances, which takes into account the compromise between compression and data loss.

To reduce the impact of the high number of available instances, we can resort to instance selection methods [89, 111]. Although this methodology results in faster learning times, it disregards many instances that could lead to superior results. In algorithms where learning is iterative (such as batch-learning in ANNs) a solution that selects few instances of the dataset to learn from in every iteration is preferable [122, 233]. One advantage of this strategy is that it allows for informed decisions regarding which instances of the dataset are more representative of the domain, as learning progresses. For example, instead of selecting instances that the algorithm quickly learns from, we can use those that are hard to classify, thus driving learning towards regions of the space where performance is poor.

Since this section aims to develop a method that uses as much of the available data as possible, our focus is on approaches that reduce the dimensionality of the datasets [35]. These methods search for the minimum amount of data characteristics that are necessary for learning. Examples of traditional methods include linear transformation techniques, such as PCA [265]. However, linear approaches cannot cope with the non-linearity present in the majority of real-world problems, and thus non-linear methods are preferred. For example, Koutník et al. [135] reduce the dimensionality of the input signal using Max-Pooling Convolutional Neural Networks. In the current section, we focus on AEs: an example of a topology of ANNs that can be used to reduce the data dimensionality, which according to Blum et al. [35] have a good performance on real-world datasets. An example of the application of AEs for dimensionality reduction is shown in [106]. The authors describe a method for initialising the weights of AEs so that they can effectively learn compressed representations of the raw data. Lander and Shang [142] propose EvoAE, a method that aims at tuning the neurons in a single hidden-layer and their input and output-weights. David and Greental [51] describe a similar approach, but that is capable of dealing with the optimisation of AEs that have more than one hidden-layer.

To acknowledge whether or not it is possible to automatically compress dataset instances without compromising the classification performance, we evolve AEs. Differently than in previous approaches, the evolved AEs will be tested in an image classification task, where the evolutionary process is guided by the quality of each AE in the task that is to be solved. Therefore, the performance of the AE is measured as the classification accuracy that is obtained when the classification is performed based on the compressed instances of the dataset. That is distinct from the usual metrics used to assess the quality of AEs, which often measure the ability to reconstruct the input signal.

The remainder of the section is organised as follows. Section 3.3.1 describes the method developed for evolving AEs for compressed representations. Section 3.3.2 analyses and discusses the experimental results. Section 3.3.3 summarises the contributions of the current section.



(a) Usual topology of an AE.

(b) Asymmetric topology allowed by our approach.

Figure 3.6: Different topologies of AutoEncoders (AEs). The encoder is the part of the AE that compresses the original signal; the decoder maps the compressed signal onto the original signal.

3.3.1 Approach

We propose a NE approach to search for effective AE topologies. The goal of the generated AEs is to find the most compact representation of the original data, i.e., the one that reduces the dataset dimensionality the most. Moreover, we also aim at evolving AEs that reduce the dimensionality without compromising the performance of the classifier that learns from the compressed data.

The structure of the AEs is not constrained, i.e., we do not restrict the search space to AEs that have the usual funnel structure: a decreasing number of neurons in the encoding part; a chokepoint (the layer with the least number of nodes); and an increasing number of neurons in the decoding part. Moreover, the number of neurons in the layers in the encoder tends to be the same as in the decoder, but inverted (see Figure 3.6a). Our evolutionary method allows for asymmetric structures (see Figure 3.6b). A network is said to be asymmetric when: (i) the layers do not need to have a decreasing/increasing number of neurons in the encoder/decoder; and (ii) the number of layers in the encoder and decoder can be different.

3.3.1.1 Representation

The candidate solutions are encoded as variable-length ordered lists of integers, where each integer represents the number of neurons of a specific layer. This evolutionary engine works as a proof of concept for the evolution of AEs to compress signals that are used in classification tasks. Therefore, we disregard the optimisation of other parameters such as the activation functions (fixed to ReLU); the extension of the representation to deal with other parameters is straightforward.

3.3.1.2 Genetic Operators

Mutation operators are designed to act upon the layers and neurons of the evolved AEs. In particular, we develop 5 mutation operators: (i) add layer – verifies that the individual has not already reached the maximum number of layers, and when that is not the case, a new layer with a random number of neurons (within the defined limits) is created and placed in a random position; (ii) remove layer – randomly deletes a layer after checking whether or not removing the layer violates the restrictions on the minimum number of layers of the evolved AEs; (iii) reset layer – chooses a layer and replaces its number of neurons by a new valid possibility (as in

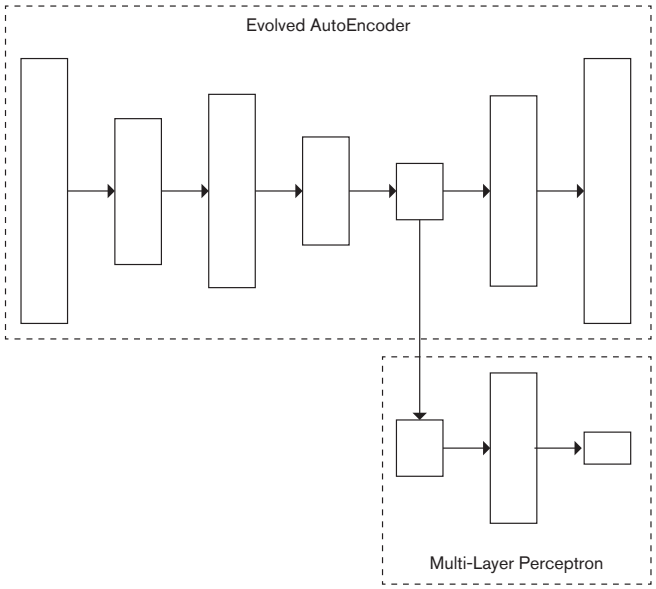


Figure 3.7: Fitness evaluation scheme: interaction between the AE and the MLP. The number of neurons of the input layer of the MLP is of the size of the chokepoint, and the output has a number of neurons equal to the number of problem classes.

the initialisation); (iv) number of neurons – selects a layer and applies a Gaussian perturbation (mean of 0 and standard deviation of 5) to the number of neurons, verifying if the number of neurons after the mutation is within the valid range; and (v) swap – picks two layers (at random), and swaps their positions. The mutation operators (iii) and (iv) are used to smooth evolution; we are aware that it is possible to attain the same effect by applying sequentially some of the remaining mutation operators.

Crossover is based on a bit-mask and takes two parents to generate the offspring. The parents are selected using tournament selection. Because the candidate solutions can have a different number of layers (due to the variable-length encoding), care must be taken when applying crossover. The bit-mask is of the size of the parent that has the highest number of layers. Only valid swap operations are performed, i.e., when a layer from an individual whose index is greater than the maximum number of layers of that individual is to be swapped, the crossover of that layer is not performed.

When swapping the layers based on the bit-mask, when an invalid swap of layers is to be done (i.e., copy a layer from an individual whose index is greater than the maximum number of layers of that individual), the copy is not performed.

3.3.1.3 Evaluation

The evaluation of each candidate solution is accomplished by measuring the usefulness of the low dimensionality data representation on the chokepoint. To that end, we divide the training of each individual into two steps: (i) unsupervised training of the AE; and (ii) supervised training of a classifier on the data from the chokepoint, i.e., the data is passed through the encoder of the AE and feed to the classifier. In the current section, the classifier is a MLP, with one hidden-layer

with 100 nodes, and ReLU as the activation function. The classifier is not evolved but rather set empirically. Figure 3.7 shows how evaluation proceeds.

The usual aim of the AEs is to reconstruct the input signal, with the minimum error. Instead, we reconstruct the mean signal of each class. Our goal is to learn the class representation, i.e., a compressed version of the features that represent the class; the objective is not to reconstruct the input of the network. As such, to train the AEs, we use the error between the reconstructed signal and the mean signal of the class of the instance that is being reconstructed.

To train the AEs, we use Keras [42], running on top of Tensorflow [1]. The training of the MLPs is performed with Scikit-Learn [194]. Usually, in ML, we only need two dataset partitions to assess the performance of a model. However, when combining EC and ML three disjoints sets are needed: one to train the evolved AE and the MLP (training set), another to assess the quality of the individual (validation set), and another that is kept out of the evolutionary cycle and used to perform an unbiased analysis of the results (test set). We do not use cross-validation due to the time it would require, making the evolutionary process unfeasible.

In addition to the maximisation of the classification performance (acc), the fitness function also considers the size of the chokepoint (h), and the percentage of layers in the decoder (r). The rationale behind the minimisation of the size of the chokepoint is to obtain compact representations of the input. The last part of the fitness function, i.e., the minimisation of the number of layers in the decoder, comes from the fact that we want to find representations that are easy to decode, thus focusing the evolutionary effort on the discovery of good encoders. The fitness function is formalised as:

$$\text{fitness} = \alpha(1 - acc) + \beta h + \gamma r,$$

where α, β, and γ are the weights of the fitness components. The goal of evolution is to minimise the fitness function.

3.3.2 Experimentation

To assess the performance of the proposed approach, we divide the experiments into two distinct parts. First, we evolve AEs for compressing the MNIST dataset [145]. During evolution, the performance of the compression is tested with a fixed-topology MLP with 100 neurons. Then, after evolution, we conduct experiments with different classifiers on the classification of the low dimensionality data representations provided by the evolved AEs. We compare the performances obtained using the raw data, and the compressed data representations. The experimental setup is detailed in Section 3.3.2.1; the dataset is described in Section 3.3.2.2; and the experimental results are analysed in Section 3.3.2.3.

3.3.2.1 Experimental Setup

The parameters used to perform the evolutionary runs on the automatic evolution of AEs are detailed in Table 3.10. Although the maximum number of generations is fixed to 100, we also use convergence as the stop criterion, i.e., when the fitness value of the best candidate solution does not improve for 7 generations, the evolutionary run is halted. Otherwise, when the stop criterion is never met, the evolutionary run stops when the maximum number of generations is reached. The values of the fitness function weights are very different due to the difference in the scale of the multiple coefficients; the weight values were set empirically.

No pre-processing or data augmentation techniques are applied to the dataset; we use the raw data as we obtain it. This is done because the goal of evolving AEs is to discover a function

Table 3.10: Experimental parameters for the evolutionary experiments on the automatic search of AEs for feature discovery.

EA Parameter	**Value**
Number of runs	30
Max. number of generations	100
Population size	100
Min. number of hidden-layers	2
Max. number of hidden-layers	6
Min. number of neurons	1
Max. number of neurons	784
Crossover rate	80%
Mutation rate	60%
Add layer mutation rate	15%
Remove layer mutation rate	15%
Reset layer mutation rate	5%
Gaussian mutation rate	60%
Gaussian mutation mean	0
Gaussian mutation stdev	5
Swap mutation rate	5%
Fitness Parameter	**Value**
α	20000
β	2
γ	10
Tournament size	3
Elite size	2%
Dataset Parameter	**Value**
Training set	60000
Validation set	5000
Test set	5000

Table 3.11: Experimental parameters of the different classifiers using the compressed representations as input.

MLP Parameter	**Value**
Number of hidden-layers	1 (100 neurons)
Activation function	ReLU
Solver	Adam
Learning rate	0.001
Beta 1	0.9
Beta 2	0.999
Epsilon	$1 \exp -8$
Number of epochs	200
DT Parameter	**Value**
Max. Depth	5
RF Parameter	**Value**
Number of estimators	10
Max. features	1
Max. depth	5
SVM Parameter	**Value**
Kernel	Linear
C	1
KNN Parameter	**Value**
Number of neighbours	3
Distance	Minkowski

capable of processing the dataset so that we can potentially get better performances in less time. In other words, we want to find the pre-processing function.

During evolution, the AEs are trained using the Adam [125] optimiser with learning rate, beta 1, and beta 2 of 0.001, 0.9, and 0.999, respectively. The training sessions are conducted for 7 epochs, with a batch size of 250. Longer training sessions could have been performed, but the longer the train, the more time is required for evaluation. We set this number of epochs after conducting preliminary experiments where we analysed the quality of the reconstructed images.

At the end of the evolutionary search of AEs, we focus on the analysis of the best 30 AEs (one from each of the runs). We conduct experiments to analyse the quality of the generated compressed representation, i.e., we pass the dataset instances through the AEs and classify them with different models. More precisely, we test MLPs, DTs, Random Forests (RFs), Naive Bayes, SVMs, and the k-NN classifiers. The parameters used for each of the classifiers are reported in Table 3.11. During evolution, and to assess the fitness of the AEs, we only use the MLP classifier, with the same parameters as the ones presented in Table 3.11.

Figure 3.8: Mean image of each class of the MNIST dataset.

3.3.2.2 Dataset

The experiments are conducted on the MNIST dataset [145]. The dataset consists of hand-written digits, centred in 28×28 grayscale images. Therefore, we are solving an image classification task, where the goal is to identify the correct digit that is contained in each of the images: from 0 to 9 (10 independent classes).

The AEs are trained to reconstruct the mean image of the class rather than the input image. Figure 3.8 depicts the mean image of each class, which is compute by pixel-wise averaging.

3.3.2.3 Experimental Results

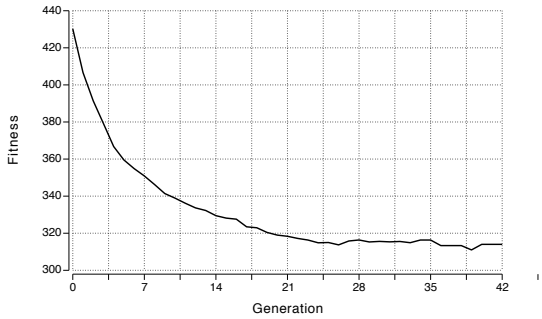
The evolution of the fitness of the best individuals across generations is shown in Figure 3.9a. The analysis of the results indicates that fitness is evolving, and stagnates around the 26th generation. The evolution stop criterion is convergence, and only ultimately a fixed number of generations; therefore, in each generation, the fitness value can be the average of a different number of runs. The evolution of the number of runs across generations is shown in Figure 3.9e. The first and last runs converge at the 17th, and 43rd generations, respectively.

The fitness function is composed of multiple components that take into account the different aspects of the AEs. The charts of Figures 3.9b, 3.9c, and 3.9d show the evolution across generations of the accuracy of the MLP, chokepoint size, and decoder ratio, respectively. Recall that the goal is to maximise the accuracy, and minimise the chokepoint size and the number of decoder layers. From the charts, it is possible to acknowledge that all the components contribute to the evolution of the fitness value, i.e., across the generations the accuracy of the MLP increases, and the number of neurons in the chokepoint and the ratio of the decoder decrease. The proportion of hidden-layers that are in the decoder tends to decrease.

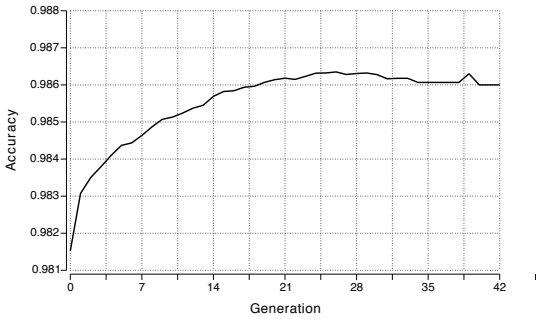
To better understand the relationship between the various fitness components, we compute the Pearson correlation between them. The only pair of components that reveal a strong (negative) correlation is the accuracy with the chokepoint size (-0.8343). This is an interesting result as it proves evolution is promoting the emergence of the desired AEs: we are able to find effective compressions (with a lower dimensionality) that report better performances (higher accuracies).

The structure of the best performing AE is depicted in Figure 3.10. The network is composed of 5 hidden-layers: 3 in the encoder, and 1 in the decoder. The chokepoint has 32 neurons, i.e., the compression is of approximately 99.96%. This AE allows a classification accuracy of 98.65%.

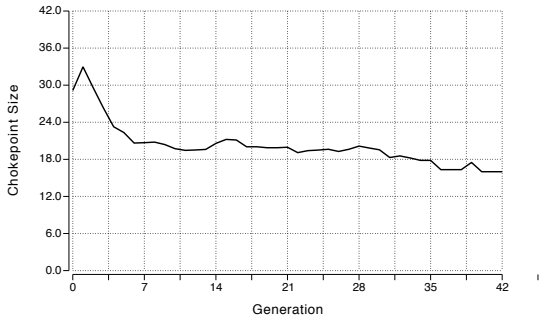
To further investigate the effectiveness of the generated AEs, we test the classification performance of multiple models using the found topologies. More precisely, besides the MLP classifier, we also use the following ones: DTs, RFs, Naive Bayes, SVMs, and k-NN. Those classifiers were selected to cover a large spectrum of the most used classifiers in ML tasks. The reported results are the average of the 30 best AEs, one from each evolutionary run. The weights of the AEs are the ones resulting from the training during evolution. In addition, when the classifiers have stochastic components, we perform 30 independent training sessions. The results are detailed in



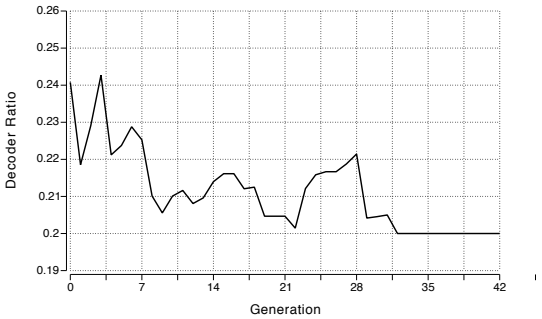
(a) Combined fitness.



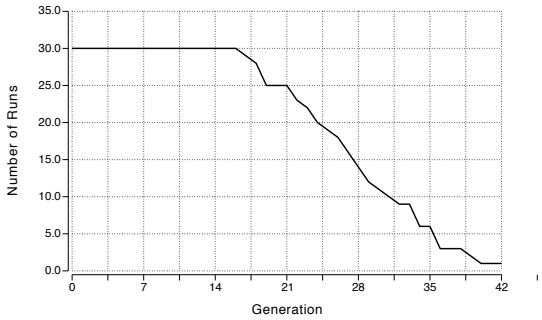
(b) Accuracy of the MLP.



(c) Size of the chokepoint.



(d) Decoder size / number of hidden-layers.



(e) Number of runs.

Figure 3.9: Evolutionary results of the evolution of AEs for compressed representations. The analysis focuses on the combined fitness, on the multiple fitness components, and on the number of runs. Results are the average of the best individuals across generations.

Table 3.12, and focus on the comparison of the performance of the classification of the raw data against the low dimensionality data representation.

An analysis of the results shows that using the compressed representation instead of the raw data leads to superior results using all classifiers, except for the decision trees. On average, the classification accuracy of the models using the raw data is 78.86%, with the compressed representation it is of 92.08%. These experiments are conducted using the test set, i.e., those data instances that are kept out of the evolutionary process. Therefore, it is possible to conclude

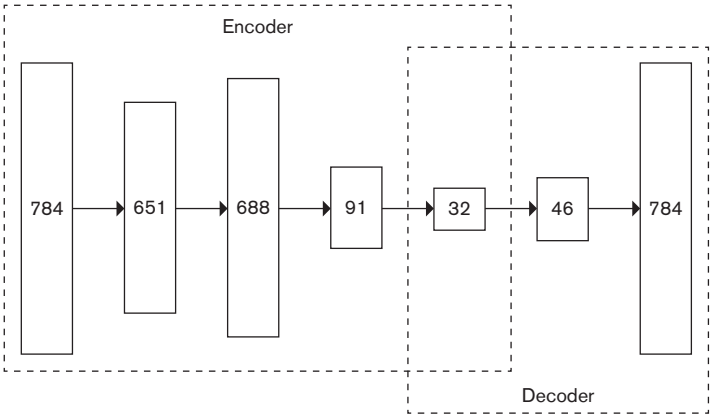


Figure 3.10: Topology of the best performing AE found by evolution. The numbers in each layer indicate the number of neurons.

that the evolved compression functions generalise, because they perform well when used by different classifiers, and consistently provide better results than when using the raw data.

We conduct a statistical study to check whether or not the different representations (original vs. compressed) have statistically significant differences. To check if the samples follow a Normal Distribution, we use the Kolmogorov-Smirnov and Shapiro-Wilk tests with a significance level $\alpha = 0.05$. The tests reveal that the data does not follow a normal distribution and, as such, a non-parametric test (Mann-Whitney U, $\alpha = 0.05$) is used to perform the pairwise comparison of the approaches (with Bonferroni correction). The p-value and the effect size are presented in Table 3.12. The effect size is represented using a graphical overview where $+$, $++$ and $+++$ correspond, respectively, to low ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$), and large ($r \geq 0.5$) effect sizes. The statistical tests show that the results reported when using the compressed data representation are consistently statistically superior to those reported by the use of the raw data. Except for the decision tree classifier, the statistical difference is in favour of the compressed data representation for all classifiers. Moreover, the effect size is strong in all pairwise comparisons.

In addition to analysing the accuracy of the MLPs in the classification of the low dimensionality compressed data from the best-found AEs, we examine the training times too. The main conclusion is that the time required to train a classification model using the compressed data is far lower than when using the raw data. For example, when training a MLP on the raw data, we need, on average, 25.7s; with the compressed data only 12.2s are necessary to train the classifier (2.1× faster). The results are even more striking when training with the k-NN: there is a speedup of 400 (from 561s to just 1.4s, on average). Nonetheless, this is an expected result as the average size of the chokepoint is roughly 17, i.e., we map the original raw data with a dimensionality of 784 to just 17 (dimensionality reduction of approximately 99.98%).

The above results show that it is possible to evolve asymmetric AEs capable of reducing the data dimensionality, with high compression rates. Further, the compression of the original data does not generate a decrease in the quality of the classification results. Instead, it is demonstrated that, when using the low dimensionality data, the results are statistically superior in the vast majority of the tested classifiers. The reduction of the input dimensionality also leads to a significant decrease in the training time of the tested classification models.

Table 3.12: Performance of multiple classifiers using the raw and compressed data representations. The statistical analysis reports the p-value and the effect size, where +, ++ and + + + correspond to low, medium, and large effect sizes, respectively.

		MLP	Decision Tree	Random Forest	Naive Bayes	SVM	KNN
Accuracy (%)	Raw Data	95.81	67.14	63.11	55.58	94.45	97.05
	Compressed Data	98.30	61.77	97.70	98.07	98.33	98.28
p-value		0.00	0.00	0.00	0.00	0.00	0.00
Effect Size		+++	+++	+++	+++	+++	+++

3.3.3 Overview

The increasing dimensionality of the datasets makes the application of approaches to reduce their dimensionality necessary, so that it is possible to process the data and extract knowledge from it in an acceptable amount of time. To achieve this, we seek to automate the process of discovering effective functions for compressing the raw data. More precisely, we apply an evolutionary approach to the optimisation of the structure of deep AEs. The AEs are trained in an unsupervised fashion, where the goal is to reconstruct the mean image of each of the problem classes. This way, instead of learning to reconstruct the original image, the goal is to learn a compressed representation of the features that encode each of the classes. To evaluate the performance of the AEs, we consider multiple aspects: (i) the accuracy of a classifier on the compressed data representation; (ii) the size of the compressed representation; and (iii) the ratio between the number of layers in the decoder and the total number of layers.

The results demonstrate that the approach can discover AEs that successfully reduce the original dimensionality of the problem, leading in the MNIST dataset to representations that have, on average, a size of 17, which is much lower than the original 784. Additionally, the evolved low dimensionality representations generalise: they do not only perform well when the classification is done by the MLP structure that assesses the quality of the compression during evolution, but good results are also reported using other classifiers.

The most important conclusion is that the classification performance considering the compressed representation is almost always statistically superior to when using the original data. The time needed for training the classifiers on the compressed representations is also considerably lower than when using the raw data.

3.4 Evolution of Weight Policies

The previous section tackled a very limited optimisation of a potentially deep network, where only the number of neurons in each layer was tuned. Because we were dealing with a simple problem, the training of the networks was considerably fast. However, the total number of neurons and connections makes the direct evolution of the weights unfeasible, and thus the AEs were trained using the Adam optimiser. In the current section, we do the opposite: the topology of the network is given, and we optimise the learning policy. There are many approaches to that end: evolve the learning algorithm parameters, the actual learning rules, or directly tune the weights and bias values of the networks. The approach adopted in this section is based on the premise that the network's weights follow some pattern that can be learned. Assuming that this pattern exists, it may be easier to evolve a function that outputs the weights of the connections between nodes than evolving the weights directly.

HyperNEAT [231] is based on the use of NEAT [232] to evolve CPPNs, which are structurally

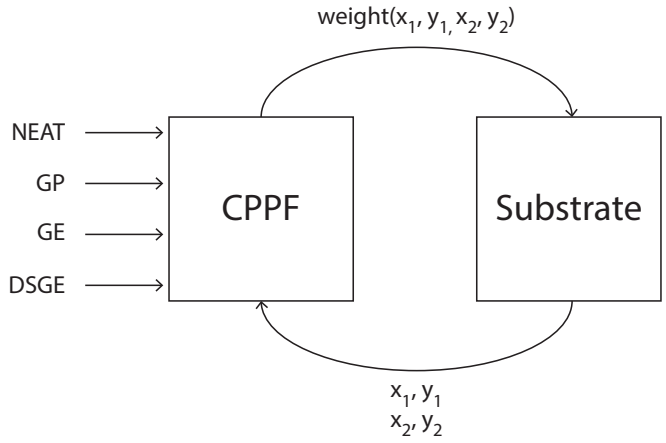


Figure 3.11: Overview of the evolution of weight policies with focus on the interaction between the CPPFs and the substrate.

similar to ANNs, and are used as a means to encode the weights of a network. In this context, CPPNs are a function that, given the position of two neurons, outputs the synaptic weight of the connection between the two given neurons. As such, and since a CPPN can be seen as a function mapping the coordinates of a pair of nodes into a weight, instead of evolving CPPNs one can use conventional EC techniques, such as GP, to evolve functions to the same task. Such functions are known as CPPFs [38]. Since, in essence, CPPNs are CPPFs that use the representation adopted by NEAT, from here on we will refer to both as CPPFs except when it is necessary to make a distinction. Thus, a CPPF evolved by NEAT is a CPPN.

In the current section, we apply different methods to the optimisation of CPPFs. More precisely, we optimise CPPFs using NEAT, GP, GE, and DSGE. We test these approaches in a visual discrimination and a line following task. Section 3.4.1 describes CPPFs and the used optimisation methods; Section 3.4.2 analyses the comparative performance of the considered methods; and Section 3.4.3 summarises the conclusions.

3.4.1 Generation of CPPFs

The main goal of the current section is to compare the performance of different approaches in the evolution of CPPFs for the training of ANNs. Figure 3.11 depicts the interaction between evolution, the generated CPPFs, and the substrate, which is the network that is being trained to solve a specific problem. Four different approaches are going to be tested in the evolution of CPPFs: NEAT, GP, GE, and DSGE, which are briefly described in Sections 2.4, 2.3, 2.3.2, and 2.3.3, respectively. CPPFs are functions of 4 inputs: x_1, y_1, x_2, y_2, which similarly to CPPNs are the positions of the neurons in the substrate, and one output, z, which is interpreted as the weight of the connection between neuron (x_1, y_1) and neuron (x_2, y_2). More specifically, the substrate is the ANN that is used to solve the problem that is being tested. It works as a grid of neurons, with a number of inputs and outputs that vary according to the problem that it must solve. To know the connections between the neurons in the substrate, we query the evolved CPPF; an output value above a defined threshold means that there is a connection between those neurons and that the synaptic weight is the value returned by the CPPF.

3.4.2 Experiments

We conduct experiments with the four evolutionary methodologies described above: NEAT, GP, GE, and DSGE. The objective of the experiments is the evolution of CPPFs for the training of neural controllers for solving specific tasks, which are described next, in Section 3.4.2.1. Section 3.4.2.2 details the experimental setup. Section 3.4.2.3 analyses and discusses the experimental results.

3.4.2.1 Description of the Problems

The experiments described in this section are conducted in two different environments: (i) a visual discrimination; and (ii) a line following tasks. Each environment is tested with two different setups that vary in complexity. In the upcoming sub-sections, we briefly describe the problems and the structure of the substrates that are used to solve them.

Visual Discrimination

The objective of this visual computation task is to distinguish two different objects: (i) a target; from (ii) a distractor, independently of their position in the field. We test the evolution of CPPFs for two different setups. The two setups have the same input dimensions: an 11×11 image, but where the targets and distractors differ. In the big-little setup, the target is a 3×3 square, and the distractor is a 1×1 square:

$$\text{target}_{\text{square}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{bmatrix}, \text{distractor}_{\text{square}} = \begin{bmatrix} 1 \end{bmatrix}.$$

To complexify the task we then conduct experiments with a triangular target and distractor, which have the same dimensions, but are mirrored (triup-down setup):

$$\text{target}_{\text{triangle}} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & \underline{1} & 0 \\ 1 & 1 & 1 \end{bmatrix}, \text{distractor}_{\text{triangle}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

The identification of the target shape is performed by a neural controller, which is trained using a CPPF. To identify the shape, the trained network generates the highest activation at the center of the target shape. Therefore, the goal of evolution is to minimise the distance between the response provided by the network and the target's center (underlined 1 in the target's matrices). To further complexify the problem, as in [38], the positions of the distractor are set at random. This makes the fitness function non-deterministic, as two consecutive evaluations can lead to different fitness values.

For this task, the CPPFs receive 6 inputs: additionally to the usual coordinates of the nodes x_1, y_1, x_2, y_2, there are also two delta values $x_1 - x_2$, and $y_1 - y_2$. The substrate consists of a sandwich network [231], i.e., the input layer is directly connected to the output layer, and both have the same size (in this case 121 neurons, one for each pixel of the image).

Line Following

In the line following task, the goal is to evolve the controllers of an agent so that it can efficiently navigate a road, i.e., follow a line with the maximum speed. The map is made of regions with different friction rates, and thus the agent should strive to steer in those with the lowest resistance. Two setups are tested: in the first one, all regions except the road have the same

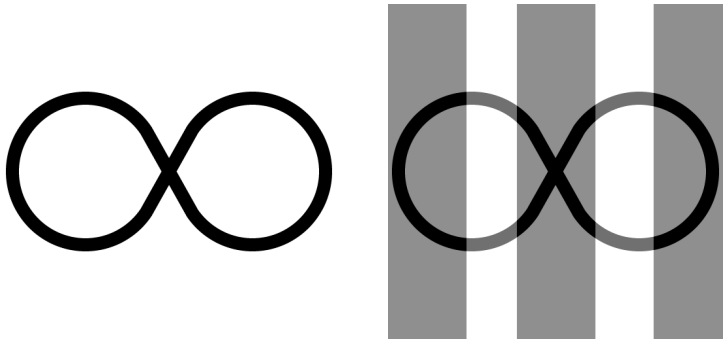


Figure 3.12: Line following task. On the left, the road that the agent must follow. On the right, the same road path with different friction areas.

friction rate (Figure 3.12, left). On the other setup, there are regions of different friction (marked as the darker stripes on the right side of Figure 3.12). The regions outside the road have friction that is 5 times higher than on the road.

The agent is a robot with two wheels and 5 sensors, each with a "vision" range of 3. The sensors are placed to the front of the agent and provide a read of the characteristics of the field within range. As such, the substrate is a feedforward network with 15 inputs (which are feed with the sensors data), a hidden-layer, and 2 outputs (that control each of the wheels).

The evolved CPPFs receive the standard 4 inputs but have 3 outputs, each responsible for encoding a function that represents the weights of a specific part of the substrate: input to output, hidden-connections, and output layer bias. When evolving the CPPNs, there is no problem in having three different functions, each represented by a different output neuron. However, to accomplish the same with CPPFs, authors typically evolve three different functions simultaneously (as in [38]). Instead, we decided to change the constants to vectors of size 3, where each of the values in the vector is to be used by each of the outputs. Consequently, the input instead of x_1, y_1, x_2, y_2 is also a vector, where x_1 is $[x_1, x_1, x_1]$, and the same for the remaining inputs.

3.4.2.2 Experimental Setup

To conduct experiments with the above benchmarks we adapt the vanilla implementations of each of the evolutionary engines. These are easily found in public repositories[2]. Table 3.13 details the parameterisation of the different algorithms. For the two benchmarks and respective setups the parameters are the same, except for the number of generations of each run, which is 250 for the visual discrimination task setups, and 100 in the line following. The codon_size in the GE parameters is the total number of integers used in each genotype, i.e., on average there is a mutation per individual.

For the grammar-based approaches (GE and DSGE), we use Grammar 3.3. This grammar is capable of generating CPPFs that encode the weights for the substrate based on multiple inputs. For the visual discrimination task x_1, y_1, x_2, y_2, d_1, and d_2: the first 4 inputs are the positions of the neurons in the substrate and the last are the deltas, $x_1 - x_2$ and $y_1 - y_2$, respectively. In the line following task the deltas are not considered.

When evolving CPPFs with tree or grammar-based GP, we consider a simple function set: sin, addition, subtraction, multiplication, and division. The terminals are the ones used in

[2]NEAT – `https://github.com/noio/peas` [28]; GP – `https://github.com/DEAP/deap` [70]; GE – `https://github.com/jmmcd/ponyge`; and DSGE – `https://github.com/nunolourenco/dsge`

Table 3.13: Experimental parameters for the experiments on the evolution of CPPFs.

Parameter	Value
Number of runs	30
Number of generations	250 / 100
Population size	100
Elite size	1%
Tournament size	3

NEAT Parameter	Value
Weight range	(-3, 3)
Minimum weight	0.3
Add node probability	0.03
Add connection probability	0.1
Mutate weight probability	0.8
Reset weight probability	0.1
Reenable connection probability	0.01
Disable connection probability	0.01
Mutate bias probability	0.2
Mutate node type probability	0.2
Std weight mutation	0.2
Std bias mutation	0.5

GP Parameter	Value
Crossover probability	0.9
Mutation probability	0.1
Maximum tree-depth	17

GE Parameter	Value
Codon size	127
Wrapping	2
Crossover probability	0.9
Mutation probability	0.05

DSGE Parameter	Value
Crossover probability	0.9
Mutation probability	$1/\text{codon\_size}$
Maximum recursion	17

Grammar 3.3, i.e., the inputs of the CPPF, and a float value that may range between -3 to 3. In NEAT, the nodes of the network can use the following activation functions: sin, bound, linear, Gaussian, sigmoid, and absolute value.

3.4.2.3 Experimental Analysis

For each experiment, we conduct 30 independent evolutionary runs so that we can understand the behaviour of the methods in each of the tested problems and setups. We analyse fitness evolution, convergence speed, and complexity of the evolved functions. In addition, statistical tests are performed to assess if any of the approaches is statistically superior to the others.

Visual Discrimination

In the visual discrimination task, the goal is to reduce the distance to the center of the target shape, and consequently, fitness is to be minimised. Although there is elitism, recall that the fitness function is not deterministic, and thus the quality of the same individual evaluated multiple times can vary. Figure 3.13 depicts, for both setups, the evolution of the fitness across generations. The results are averages of the 30 evolutionary runs. These charts show that with any of the evolutionary engines evolution is promoted, and there is convergence. The difference in the complexity of the setups is noticeable by the analysis of the differences in the fitness scales: in both setups, the average fitness of the best solutions starts approximately from the same point, but in the big-little setup it is capable of reaching much lower values than in the triup-down setup. Having the target and distractor shapes with the same size, but mirrored, makes the problem too challenging for an appropriate network to be found in the given number of generations.

Nonetheless, for both setups, in terms of fitness, the results reported by NEAT and GP are

$$
\begin{aligned}
\text{<expr>} &::= \text{<expr>}\ \text{<op>}\ \text{<expr>} && (1)\\
&\mid \text{<var>} && (2)\\
&\mid \text{<preop>}\ (\text{<expr>}) && (3)\\
\text{<var>} &::= x1 \mid y1 \mid x2 \mid y2 && (4)\\
&\mid d1 \mid d2 && (5)\\
&\mid \text{<float>} && (6)\\
\text{<preop>} &::= + \mid - \mid * \mid / && (7)\\
\text{<preop>} &::= sin \mid - && (8)\\
\text{<float>} &::= - \text{<first>}.\text{<number><number>} && (9)\\
&\mid \text{<first>}.\text{<number><number>} && (10)\\
\text{<first>} &::= 0 \mid 1 \mid 2 && (11)\\
\text{<number>} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 && (12)\\
&\mid 5 \mid 6 \mid 7 \mid 8 \mid 9 && (13)
\end{aligned}
$$

Grammar 3.3: Grammar used for evolving CPPFs with GE and DSGE. For the line following task in the <var> rule the terminal symbols d1 and d2 are removed from the grammar, and the last expansion possibility of <var> is replaced by [<float>, <float>, <float>].

superior to those of the grammar-based methods. To better analyse the quality of the generated solutions, we use box plots focusing on the distribution of the quality of the best individuals from each evolutionary run on the last generation (see Figure 3.14). The box plots show that for the current problem GE underperforms when compared to the remaining approaches. In the big-little setup, the NEAT and GP have a close median, with GP having a slightly superior dispersion; vice-versa for the triup-down setup, i.e., NEAT and GP have approximately the same median, but the dispersion is lower in GP. GP has more outliers, but these are a good indicator as they stand for better-performing solutions. DSGE in the big-little setup is worse than NEAT or GP, and in the triup-down setup has similar performance, but larger dispersion.

Focusing on the generated solutions, for the big-little setup, NEAT, GP, GE, and DSGE generate perfect solutions in 5, 8, 3, and 0 out of the 30 runs, respectively. So, despite GP having a slightly higher median and dispersion of the fitness values, it is the approach that most often finds solutions that are always able to identify the target shape. In the triup-down setup, no approach is capable of finding a perfect solution.

For all the above, we focus the analysis of the complexity of the approaches that are most competitive in both setups: NEAT and GP. On average, in the big-little setup, in the last population the candidate solutions of NEAT use 19.54 functions, while in GP 122.44 are used. On the triup-down setup, NEAT evolves structures that use 21.86 functions, and GP structures that use 132.74 functions. Although the differences seem big, we need to consider that in NEAT we are evolving CPPNs, and in GP we are evolving CPPFs. Thus, in NEAT, it is much easier to re-use the same function multiple times than in GP.

To better understand if any of the approaches is superior to the others, we conduct a statistical study. To check if the samples follow a Normal Distribution, we use the Kolmogorov-Smirnov and Shapiro-Wilk tests, with a significance level of $\alpha = 0.05$. The tests revealed that the data does not follow any distribution and, as such, a non-parametric test (Mann-Whitney U, $\alpha = 0.05$) will be used to perform the pairwise comparison of the approaches. Because we are comparing four methods we have to use Bonferroni correction, and thus $\alpha \approx 0.008$. Table 3.14 uses a graphical

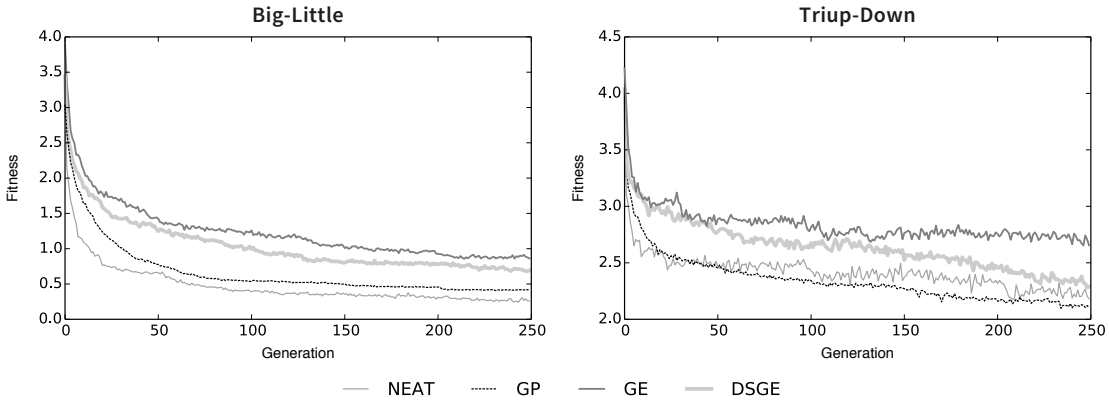


Figure 3.13: Evolution of the best individuals across generations in the visual discrimination task for the big-little (left) and triup-down (right) setups. Results are averages of 30 independent runs.

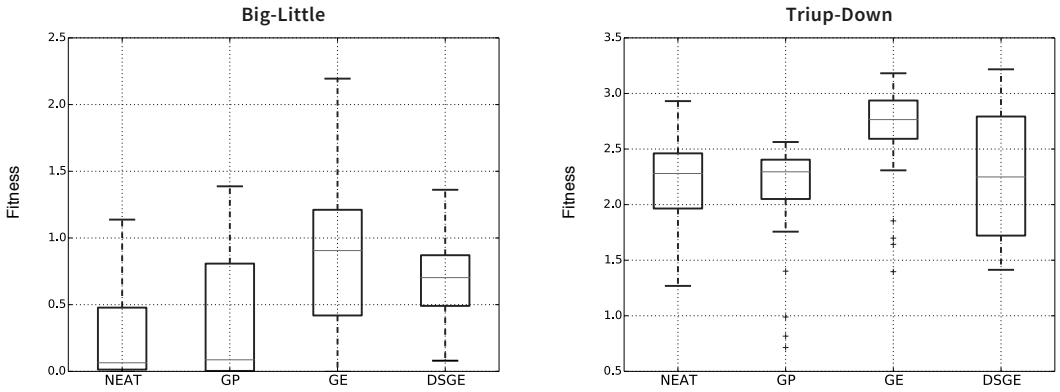


Figure 3.14: Analysis of the fitness of the best solutions of the visual discrimination task using box plots. On the left the big-little setup, and on the right the triup-down.

overview to present the results of the statistical analysis: $\sim$ indicates no statistical difference, and + signals that the approach in the row is statistically better than the one in the column. The effect size is denoted by the number of + signals, where +, ++ and + + + correspond respectively to low ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$) and large ($r \geq 0.5$) effect sizes. The statistical results show that it is not possible to point out a single approach as the best one. There is no difference between NEAT and GP. GE is outperformed by NEAT and GP, and DSGE is outperformed in the big-little setup by NEAT and GP.

Figure 3.15 presents examples of one of the best solutions (for each setup) regarding the activations that are generated for the identification of the target shapes: darker colours mean higher activation values. As perceptible, in the big-little setup, the target shape is correctly identified; in the triup-down, the trained network fails to identify the target shape and is activated by parts of the target and distractor shapes.

Table 3.14: Graphical overview of the statistical results of the visual discrimination experiments with effect sizes for the big-little (left) and triup-down (right) setups.

	NEAT	GP	GE	DSGE
NEAT		∼	+++	+++
GP	∼		+++	++
GE	∼	∼		∼
DSGE	∼	∼	∼	

	NEAT	GP	GE	DSGE
NEAT		∼	+++	∼
GP	∼		+++	∼
GE	∼	∼		∼
DSGE	∼	∼	++	

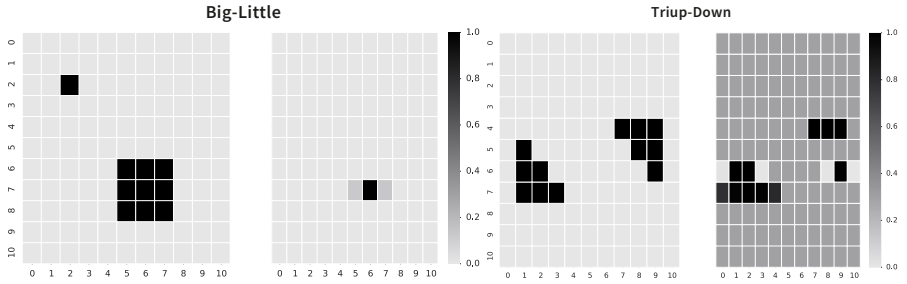


Figure 3.15: The left and right figures represent the activations generated by one of the best solutions (discovered using GP), for each of the setups: big-little and triup-down, respectively.

Line Following

The goal in the line following is to maximise the average speed of a robot in a road navigation task, i.e., maximise the distance travelled in a fixed amount of time. As previously, we start by analysing the evolution of the fitness across generations (see Figure 3.16). GE is the approach that takes the largest number of generations to converge, reaching the lowest results. On the other hand, NEAT, GP, and DSGE continue being the methods that generate the best solutions, with GP performing slightly better than the other two in the easy and hard setups. The box plots allow stronger conclusions than before (see Figure 3.17). While in the visual discrimination task NEAT or GP were not superior in the two setups, in the line following task it is perceptible that GP performs better than the remaining approaches in the easy and hard setups, i.e., despite having a few outliers GP has a median that is higher than those of the remaining approaches, and the interquartile range is smaller, meaning that the results are more consistent.

Like in the previous task, we analyse the complexity of the two most competitive approaches. On the easy setup, the average number of nodes of the CPPFs evolved by NEAT is 18.71, and by GP is 90.39. In the hard setup, the average number of nodes of the CPPFs evolved by NEAT is 18.36, and by GP is 99.69. There are two possibilities for this increase in complexity. On the one hand, more complex functions are required to effectively solve the hard setup. On the other hand, the high number of nodes in GP can be a consequence of bloat.

To better assess whether or not any of the methods is superior to the others, we perform statistical analysis, with the same conditions as before. The results are reported in Table 3.15, using the same graphical representation of the statistical analysis of the visual discrimination task. A perusal of the results shows that GP outperforms the other approaches in the easy and hard setups; NEAT outperforms the grammar-based approaches in the hard setup; DSGE outperforms GE; and GE is outperformed by all other approaches.

An example of the best models navigating in each of the setups is depicted in Figure 3.18. The purple line marks the path followed by the robot. In the easy setup, it is clear that the

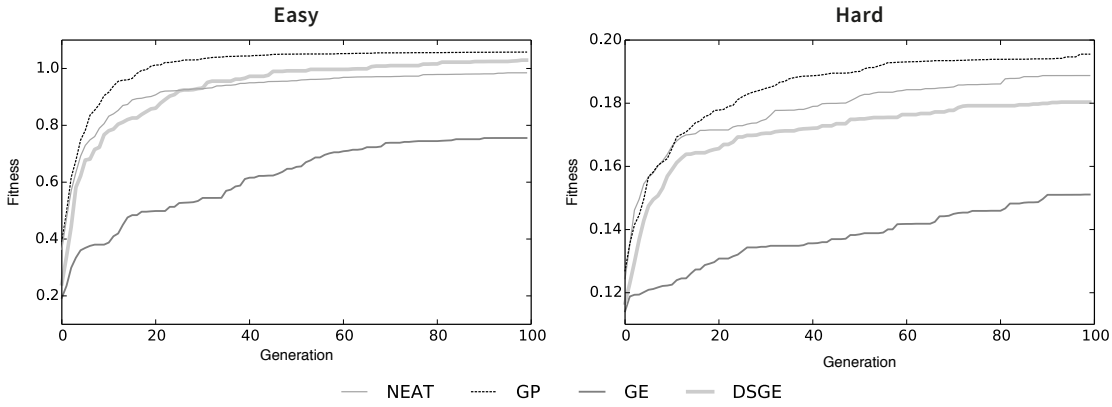


Figure 3.16: Evolution of the best individuals across generations in the line following task for the easy (left) and hard (right) setups. Results are averages of 30 independent runs.

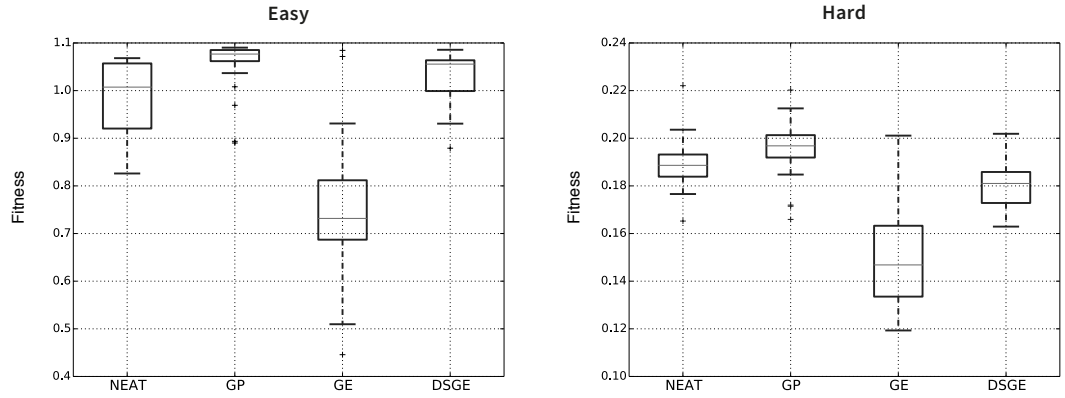


Figure 3.17: Analysis of the fitness of the best solutions of the line following task using box plots. On the left the easy setup, and on the right the hard.

robot is capable of travelling through the road without difficulty, never leaving the road. The same cannot be stated for the hard setup, where the regions of different friction make learning more challenging to the point that none of the evolved models can complete an entire lap in the given execution time. The vast majority of the best individuals depict the behaviour presented in the example, i.e., they leave the marked path and are unable to get back on track

Discussion

Summing up, we conclude that in the visual discrimination task no approach is superior to all the remaining ones. The analysis of the results, in both the evolution and box plots, shows that GP and NEAT have a similar performance. The statistical tests also confirm that there are no meaningful differences between these two approaches. Nevertheless, GP discovers perfect solutions most often. In the line following task, GP outperforms the remaining approaches.

The performance of the grammar-based approaches is fairly worse than the performance of the other methods. The reason for that can be explained by the way used to create the float constants, which makes the search space larger. While in NEAT and GP the floats are just one

Table 3.15: Graphical overview of the statistical results of the line following experiments with effect sizes for the easy (left) and hard (right) setups.

	NEAT	GP	GE	DSGE
NEAT		∼	+++	∼
GP	+++		+++	++
GE	∼	∼		∼
DSGE	+++	∼	+++	

	NEAT	GP	GE	DSGE
NEAT		∼	+++	+++
GP	+++		+++	+++
GE	∼	∼		∼
DSGE	∼	∼	+++	

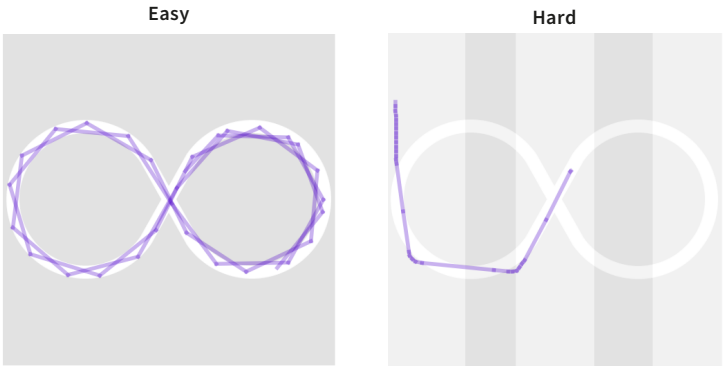


Figure 3.18: The left and right figures represent an example of one of the best solutions (found using GP) for the easy and hard setups, respectively.

terminal, in the grammar-based methods there is the need to associate an expansion possibility to each of the integers of the floats (which have a fixed precision). Nonetheless, it is still possible to state that DSGE performs better than GE.

Regarding the number of functions used by the methods that perform best, NEAT evolves structures that use fewer functions than the ones generated by GP. We are well aware that comparing their complexity in terms of the number of used functions might not be entirely fair. In NEAT, the functions correspond to activation functions of a neuron in the evolved ANN, and the same neuron can be used multiple times. On the other hand, in GP, the tree-structure makes the re-use of the non-terminals hard, leading to a consequent increase in the number of used functions. As such, even though the number of used functions is different, it is our perception that the generated solutions are of similar complexity.

Based on the experiments conducted, it is possible to state that GP has an overall better performance than NEAT, and it does so using a simpler function set. Another point that makes GP preferable for evolving CPPFs is that it requires far less parameterisation, without compromising the end results.

3.4.3 Overview

Whereas in the previous sections we addressed the evolution of the topology and weights of small-scale networks, and the evolution of the topology of AEs, in the current section we address the optimisation of weight policies. In particular, we compare different evolutionary methods for the generation of CPPFs. Our research hypothesis is that it is possible to replace NEAT in HyperNEAT by a simpler method, that requires far less parameters, without compromising the overall quality of the obtained results. To validate our hypothesis, we apply NEAT, GP, GE, and

DSGE to the evolution of CPPFs in two benchmarks commonly used in HyperNEAT experiments: visual discrimination and line following, each having two setups, that vary in complexity.

The experimental results, supported by statistical analysis, confirm our research hypothesis, i.e., they show that using tree-based GP it is possible to evolve effective CPPFs that, for the considered tasks, outperform the CPPFs discovered by the other methods, including NEAT. This result is somewhat surprising, especially taking into consideration that we used a vanilla implementation of GP and resorted to a basic and generic function set. As such, we consider that the results pave the way for applying GP approaches to the evolution of CPPFs, creating opportunities for the application of more sophisticated GP approaches to this type of tasks.

3.5 Summary

The current chapter addressed a set of preliminary experiments that have been conducted to understand the performance of grammar-based NE approaches. This type of approach has the main advantage that is easy to set because the only parameter that needs to be adapted to deal with different domains and/or network topologies is the input CFG. In addition, the grammar is defined in a text human-readable format, and thus easy to define by non-expert users. The first experiment (Section 3.1) compared the performance of SGE and GE on the optimisation of one-hidden-layered feed-forward ANNs for the classification of the flame, WDBC, ionosphere and sonar datasets. The results demonstrate the superior performance of SGE compared to previous grammar-based approaches. Further, the ANNs generated by SGE are compared to hand-designed networks, and the results show that the models generated by evolution when fine-tuned with BP (considering the evolved weights) are better than the hand-designed ANNs and better than when considering only the evolved topologies (with the weights set at random), i.e., there is an advantage in simultaneously evolving the network's topology and weights.

The main drawback of the application of grammar-based approaches to the evolution of dynamic structures (such as ANNs) is that it is hard to keep track of the topology that the successive expansions are creating. To mitigate the previous limitation we introduce DSGE and dynamic production rules (Section 3.2). DSGE proposes a novel genotypic scheme that avoids the SGE step of pre-processing the grammar to define the maximum recursion levels. Instead, in DSGE the genotype grows as needed and only encoding genes are stored. A first set of experiments in the evolution of one-hidden-layered ANNs shows that DSGE outperforms SGE and GE in the classification of the flame, WDBC, ionosphere, and sonar datasets. However, DSGE alone does not enable the automatic optimisation of multi-layered ANNs. The genotype can grow continuously, but that is not enough to encode the structure of the network so that the neurons of a hidden-layer can be re-used by the neurons in later layers. Dynamic production rules are sets of indexable production rules, which are created in run-time, and together with DSGE, allow the search for multi-layered networks. The results of the experiments show that DSGE, combined with dynamic production rules, can generate multi-layered ANNs. The networks that have more than one hidden-layer tend to provide better results than those that only have one hidden-layer.

The multi-layered networks evolved resorting to DSGE and dynamic production rules, despite a step towards the automatic evolution of DANNs are still relatively shallow. The search space allows up to 8 layers, with a maximum of 32 neurons per layer. However, the number of neurons of the best-found solutions have approximately 6 neurons. To seek the potential of NE to evolve deep networks, and simultaneously investigate the evolution of a different type of network, we address the evolution of AEs for data compression (Section 3.3). The performance of the AEs is not measured as their ability to reconstruct the input signal, but rather as their capability to

generate a compressed representation space that enables the input data to be correctly classified, i.e., the goal is to discover an accurate compression of the input, with features that can be used to classify the dataset. We test the approach with the MNIST dataset, and the results show that with a compression degree of 99.98% we can classify the MNIST dataset with an average accuracy of 92.08%. The average accuracy when using the raw data is of 78.86%.

The evolution of AEs is a step towards the evolution of deep networks, where we base the evolution on layers instead of neurons (i.e., the base unit of evolution is the layer). The problem is that the approach is over-simplistic and only considers a sequence of layers where the target of evolution is the number of neurons in each of the layers. It is easy to include, for example, the activation functions, but to include different layer types we require changes to the approach. On the other hand, it only targets the evolution of the topology and disregards the learning policy, which is kept fixed throughout evolution. In a different direction, we investigate the evolution of the learning strategy (with the topology of the network kept fixed). In Section 3.4, we compare for the visual discrimination and line following problems the performance of NEAT, GP, GE, and DSGE in the evolution of CPPFs: functions that, given two neuron positions (indexed in a grid), return their synaptic weight. The results show that the performance of NEAT and GP is above that of the grammar-based approaches. The grammar-based approaches are not adequate for the evolution of real values due to the need for multiple expansion rules.

The results of the experiments conducted in this chapter pave the way for the remainder of the Thesis. We have concluded that grammar-based approaches are well-suited for evolving the topology of ANNs. However, they are not the most adequate approach to the tuning of real values (despite working on considerably shallow networks). Their main advantage lies in the ease with which a non-expert user can change the grammar and adapt the method to deal with another problem. On a different line of research, we have confirmed that AEs are an effective tool to compress data, and as such, they can be of extreme importance when trying to develop large systems of ANNs to deal with multiple tasks simultaneously. These experiments are of importance to the proposal of a grammar-based representation that is capable of automatically generating deep networks. This representation scheme, called DENSER, is introduced and thoroughly discussed in the upcoming chapters, and incorporates the conclusions of the experiments conducted in the current chapter.

Chapter 4

Deep Evolutionary Network Structured Representation

DENSER is a novel NE general-purpose representation for the evolution of ANNs: it facilitates the automatic generation of ANNs of different types and/or to different problems. The approach draws inspiration in the experiments performed in Chapter 3, and seeks to extend them to the evolution of DANNs. The representation is based on DSGE, but the real-values are encoded directly rather than based on the expansion of multiple derivation rules. To enable the generation of deep models the evolutionary unit is the layer. All network parameters (e.g., layer types, layer parameters, learning strategy) can be optimised, and to this end we only have to adapt the grammar that defines the search space.

Section 4.1 describes the representation scheme of DENSER. Section 4.2 reports the results of the evolution of CNNs for CIFAR-10, and the analysis of the generalisation, robustness, and scalability of the best-evolved networks. The results show the ability of DENSER to search for effective DANNs that achieve an accuracy of 95.22% on CIFAR-10, and that without further evolution get an accuracy of 78.75% on the classification of CIFAR-100. To the best of our knowledge, this is the first grammar-based NE method to enable the generation of DANNs. Section 4.3 points out the limitations of DENSER. Section 4.4 summarises the conclusions.

4.1 Representation

From a technical point-of-view, DENSER combines components of GA and DSGE to enable the evolution of DANNs. This is similar to the rationale of GEGA but, while in GEGA [4], GE is used to optimise the neurons and the GA to optimise the weights, the motivation in DENSER is different. In DENSER both the GA and DSGE components encode the evolutionary units. However, whereas the GA component stores an ordered sequence of the evolutionary units that compose the search space of the networks, the DSGE portion facilitates the optimisation of the parameters of each of the evolutionary units. Therefore, in the context of DENSER the evolutionary units stand for all the components that require optimisation (and that we want to optimise), e.g., layers, learning policies or data augmentation strategies.

Furthermore, the results obtained in Section 3.4 raise the question that DSGE (as other GGP methods) underperform on the optimisation of real-values. For this reason, we adapt DSGE to facilitate the encoding of real values: instead of encoding the real values as the outcome of the application of several production rules adapted to generate integers/floats, we encode them

```
<features> ::= <convolution>                                                    (1)
             | <pooling>                                                        (2)
<convolution> ::= layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,1,5] (3)
                  [stride,int,1,1,3] <pad> <activation> <bias>                 (4)
                  <batch-norm> <merge-input>                                   (5)
<batch-norm> ::= batch-normalisation:True                                      (6)
               | batch-normalisation:False                                     (7)
<merge-input> ::= merge-input:True                                             (8)
                | merge-input:False                                            (9)
<pooling> ::= <pool-type> [kernel-size,int,1,2,5][stride,int,1,1,3] <pad>      (10)
<pool-type> ::= layer:pool-avg                                                 (11)
              | layer:pool-max                                                 (12)
<pad> ::= padding:same                                                         (13)
        | padding:valid                                                        (14)
<classification> ::= <fully-connected>                                         (15)
<fully-connected> ::= layer:fc <activation> [num-units,int,1,128,2048] <bias>  (16)
<activation> ::= act:linear                                                    (17)
               | act:relu                                                      (18)
               | act:sigmoid                                                   (19)
<bias> ::= bias:True                                                           (20)
         | bias:False                                                          (21)
<softmax> ::= layer:fc act:softmax num-units:10 bias:True                      (22)
<learning> ::= learning:gradient-descent lr:0.01 momentum:0.9 batch-size:125   (23)
```

Grammar 4.1: Example grammar for the encoding of CNNs.

directly, and apply standard genetic operators. In addition, the real-values are stored similarly to the expansion rules, and in the decoding procedure, they are used sequentially, without repetition.

The GA and DSGE parts of the representation are mapped into the outer and inner levels of the genotype, which are detailed in Section 4.1.1. The developed genetic operators, and evaluation procedure are described in Sections 4.1.2 and 4.1.3, respectively.

4.1.1 Encoding Scheme

Each individual encodes a single DANN through an ordered sequence of evolutionary units and the associated parameters. The representation facilitates the encoding of: (i) any type of layers and their respective parameters; (ii) the learning algorithm and parameters; (iii) data augmentation strategy and parameters; and/or (iv) any other network blocks that require tuning (e.g., data pre-processing). The representation of the individuals has two independent levels:

Outer Level – encodes the macrostructure of the ANNs, and represents the sequence of evolutionary units that form the network. Each unit in the sequence later serves as the starting non-terminal symbol for the expansion of the inner level genotype. This representation level requires the definition of the valid structure of the genotype, i.e., the goal of evolu-

tion: layers (which types of layers can be used, and in what order), learning, and data augmentation or any other required parameterisation of the network. This structure is important when we want to include prior knowledge into the search space (e.g., sequences of layers that are known to work well).

Inner Level – encodes the parameters, i.e., while the outer level encodes the macrostructure of the genotype, the inner level encodes the hyper-parameters required by the evolutionary units on the outer level. The parameters are codified in a CFG, and are represented as ranges, or closed sets of values. The symbols in the outer level must match one and only one of the production-rules of the inner levels' grammar. To encode real parameters, we use a 5-tuple with the following format: variable name, variable type (i.e., float or int), number of values to generate, minimum and maximum range.

An example of an outer and inner level is now provided. For example, CNNs are usually composed of convolution and/or pooling layers (responsible for feature learning), and fully-connected layers (that perform classification based on the learned representation). A possible outer level structure for the evolution of CNNs is: [(features, 1, 10), (classification, 1, 2), (softmax, 1, 1), (learning, 1, 1)]. The first parameter of each tuple stands for the evolutionary unit type (must match a grammatical non-terminal symbol), and the numbers represent the minimum and the maximum number of evolutionary units of that type, respectively. The previous structure is based in Grammar 4.1 and is able to encode CNNs with a minimum of 3 and a maximum of 14 layers: from 1 to 10 convolution and pooling (features) layers, followed by 1 or 2 fully-connected (classification) layers, and one softmax layer. The learning rule is for encoding the learning algorithm and its parameters. Looking at the grammar, we get the inner level structure. It is now clear that the features of the outer level rule are either a convolution or a pooling layer (first two lines of Grammar 4.1). The pooling layer has 4 parameters: pool type, kernel size, stride, and padding (line 10). The pool type and padding have closed values (e.g., the pool type is average or maximum) (lines 11 and 12), and the remaining parameters are real-valued. The same rationale can be applied to the remaining of the outer level structure. From the example outer level structure, it may seem that the definition of this input biases the generated networks towards structures we a-priori know to work well. However, we can enable a completely unrestricted search using the following outer-level structure: [(layer, 1, 100)], that defines that the network has between 1 and 100 layers; then, at the grammar, the layer non-terminal symbol can map to all existing layer types.

Figure 4.1 depicts an example of the genotype of an individual, based in Grammar 4.1 and in the outer level structure provided above. The figure depicts the complete genotype of the following phenotype: layer:conv num-filters:64 filter-shape:3 stride:2 padding:valid act:relu bias:True batch-normalisation:False merge-input:True layer:pool-max kernel-size:4 stride:2 padding:same layer:fc act:sigmoid num-units 256 bias:True layer:fc act:softmax num-units:10 bias:True learning: gradient-descent lr:0.01 momentum:0.9 batch-size:125. To decode the genotype, the outer level of each individual is traversed linearly. For each position of the outer level, we decode the corresponding inner level (see Section 3.2.1.2), with the difference that the real-values are encoded directly, and thus read sequentially without reusing. A step-by-step of the decoding of the first position of the outer level is provided next. The starting symbol is the non-terminal symbol <features> and, for its expansion, we select the first possibility because the inner level for the <features> non-terminal encodes for the DSGE genotype the integer 0, and thus <features> is mapped into layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,1,5] [stride,int,1,1,3] <pad> <activation> <bias> <batch-norm> <merge-input>. The next non-terminal symbol left for expansion is the block [num-filters,int,1,32,256], which is decoded based on the <features> inner level, where we read that the num-filters integer is to be mapped to 64, and thus our phenotype

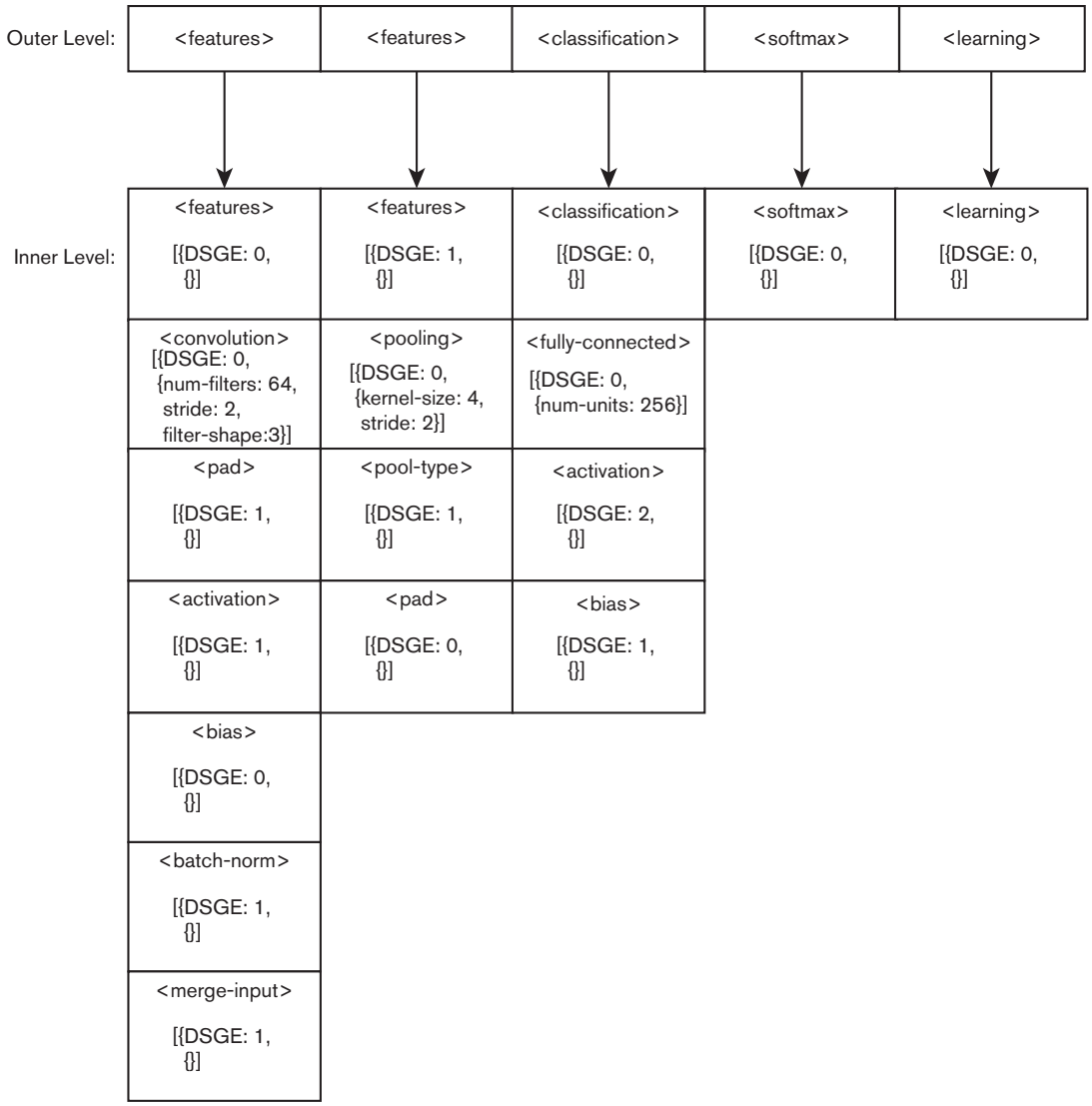


Figure 4.1: Example of the genotype of a CNN encoded by DENSER. Only the genes required in the genotype to phenotype mapping are encoded.

becomes layer:conv num-filters:64 [filter-shape,int,1,1,5] [stride,int,1,1,3] <pad> <activation> <bias> <batch-norm> <merge-input>. This process is repeated until we are left with no terminal blocks or non-terminals left for expansion. At this point, we move to the following position of the outer level genotype, until all the outer level positions have been traversed.

The initial population is generated at random, i.e., we generate a number of evolutionary units within the allowed range (respecting the defined outer level structure). Then, for each evolutionary unit, we set their parameters stochastically.

The novelty of DENSER relies in the combination of the two genotypic levels. Without the outer level, it would be impossible to encapsulate the genetic material, which facilitates

the application of the genetic operators and thus eases evolution. Without the inner level, it would be unfeasible to make DENSER a general-purpose representation: we only need to adapt the grammar to apply DENSER to the evolution of different network types, pertaining different layers, or to solve different tasks. Further, The grammar-based representation eases the incorporation of domain-specific knowledge. The grammar is defined in a human-readable text format, and therefore changing it does not require domain expertise.

4.1.2 Genetic Operators

To promote the evolution of the candidate solutions, we rely on crossover and mutation operators specifically designed to manipulate ANNs represented with DENSER. These operators act on the two levels of the genotype.

4.1.2.1 Crossover

One of the advantages of having two genotypic levels is that the outer level encodes each evolutionary unit (layer, learning, data augmentation, or other) separately. Since the genetic material is encapsulated, devising efficient crossover operators is facilitated. We develop two operators, which are probabilistically applied. In the context of DENSER, the term module refers to a set of layers that belong to the same index of the outer level genotype structure. For instance, in the above example, the symbol features has between 1 and 10 layers; therefore the feature's module is composed of between 1 and 10 layers (all those that have their derivation starting with the same symbol features). In particular, in Figure 4.1, the feature's module is composed by the first three evolutionary units.

One of the crossover operators, denoted as Single Module 1-Point Crossover, exchanges layers within the same module. To do so, we first select a given module from the two parents, and then we apply a one-point crossover to the module, without modifying the remaining modules. Note that the parents can have a distinct number of layers; therefore, the cutting point is randomly generated considering the individual that has the least number of layers in the module.

The other crossover operator is loosely based on the uniform operator for binary representations, and thus we call it Uniform Module Crossover. It acts upon the modules, swapping them between individuals. Whilst the Single Module 1-Point Crossover acts inside a given module, the Uniform Module Crossover exchanges entire modules between individuals.

An example of the application of the two crossover operators is shown in Figure 4.2.

4.1.2.2 Mutation

We design specific operators for each of the genotypic levels. The mutations that act upon the outer level aim at manipulating the structure (in terms of evolutionary units):

Add unit – a new evolutionary unit is generated at random, and the initial symbol for the grammatical derivation is set to the non-terminal symbol of the module where the layer will be placed. This operator can only be applied in modules where the maximum number of layers has not been yet reached;

Replicate unit – similar to the previous mutation operator but, instead of generating a new random unit, uses one that is already in the genotype and copies it into another random valid position of the module. The copy is done by reference, meaning that if at any given time the layer or some of its parameters are changed, the modifications are propagated to the copies. The same occurs if any of the copies is changed.

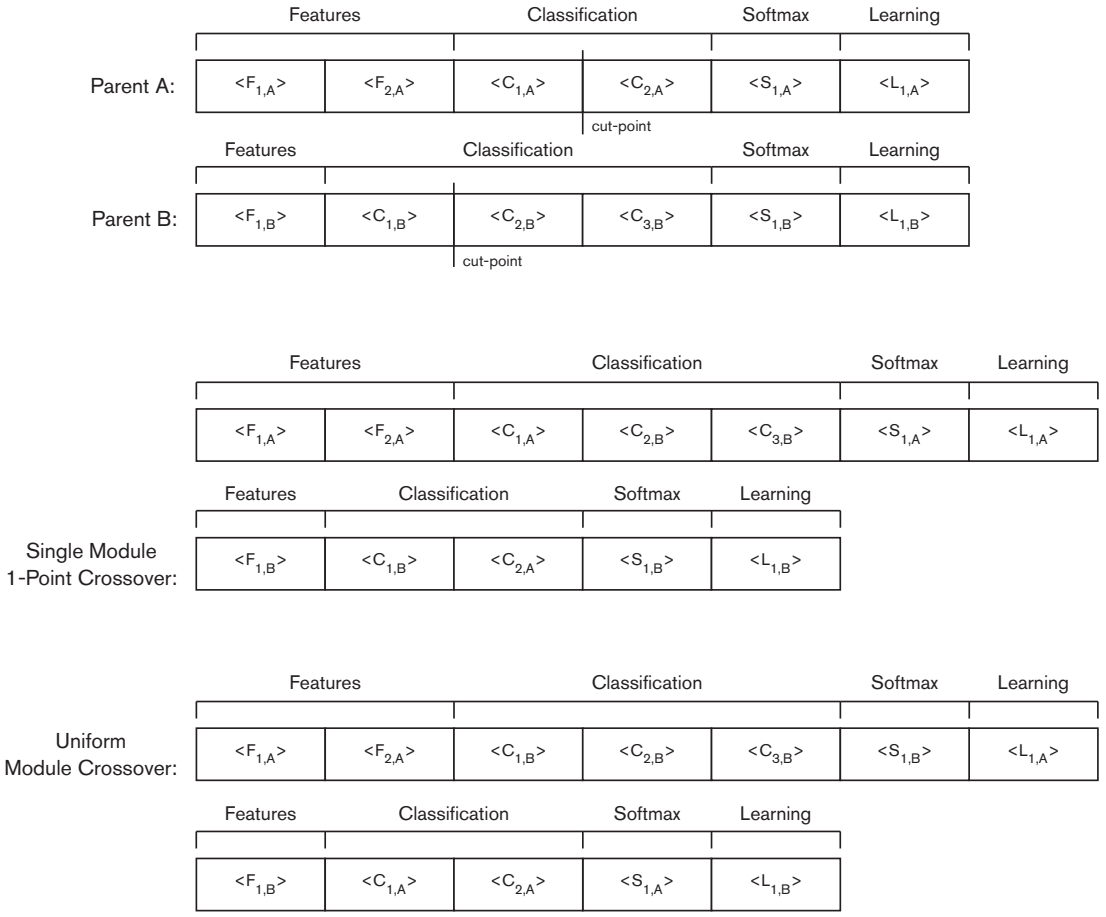


Figure 4.2: Example of the crossover operators of DENSER. The example focuses on the outer level of the genotype. For the bit-mask crossover the mask is 1001, which is associated to the features, classification, softmax and learning modules, respectively.

Remove unit – deletes a random unit from a given module. It is only possible to remove an evolutionary unit when, after removal, the number of units of the module it belongs to is still above the minimum threshold.

The previous operators act only at a macro level and thus do not change the parameters of the layers. This is accomplished by the inner level mutations:

Grammatical mutation – as in standard DSGE, an expansion possibility is replaced by another valid one;

Integer mutation – an integer block is replaced by a new one, where the integers are generated at random, within the allowed range;

Float mutation – similar to the integer mutation, but where instead of randomly generating new values, a Gaussian perturbation is applied.

Table 4.1: Numerical overview of the datasets used in the experiments conducted with DENSER.

Dataset	#Train	#Test	Input Size	#Classes
MNIST [145]	60000	10000	784	10
MNIST variants	50000	12000	784	10
Fashion-MNIST [268]	60000	10000	784	10
SVHN [185]	73257	26032	3072	10
Rectangles	12000	50000	784	2
CIFAR-10 [137]	50000	10000	3072	10
CIFAR-100 [137]	50000	10000	3072	100

4.1.3 Evaluation

To evaluate each individual, i.e., each ANN, we have to perform 4 sequential steps: (i) map the genotype to the phenotype; (ii) map the phenotype into a trainable model; (iii) train the ANN; and finally (iv) assess the performance of the model, which will determine the fitness of the candidate solution. To facilitate the evaluation, we use Keras [42]: an API with GPU support that aids in the training of ANNs; Keras runs on top of TensorFlow [1]. The GPU support is vital given the need to train every individual, and the time and computational cost associated with it.

Independently of the dataset, to use DENSER, and in order to report unbiased results, we need three disjoint data partitions:

Train – to train the network using the defined or evolved learning parameters;

Validation – to evaluate the performance (fitness) of the network during evolution;

Test – kept aside from the evolutionary search, and used to assess the performance of the best models on unseen data, so we can measure the generalisation ability of the evolved networks.

4.2 Evolution of Convolutional Neural Networks

To evaluate DENSER, we conduct experiments on the automatic search of CNNs for the classification of the CIFAR-10 dataset. To assess the generalisation, robustness, and scalability of the ANNs discovered by DENSER, we investigate the performance of the CNNs that perform best on CIFAR-10 on a wide set of computer vision benchmarks: MNIST, Fashion-MNIST, SVHN, Rectangles, and CIFAR-100. The benchmarks are further detailed in Section 4.2.1. The experimental setup is described in Section 4.2.2, and the analysis of the results is carried out in Sections 4.2.3 and 4.2.4. The best-trained models have been released at `http://github.com/fillassuncao/denser-models`.

4.2.1 Description of the Datasets

The CIFAR-10 [137] dataset is commonly used by state-of-the-art approaches, allowing a comparison between the ANNs found using DENSER and those generated by other evolutionary and non-evolutionary approaches. Furthermore, it is a moderately complex benchmark composed of 32×32 RGB real-world images. The dataset is large enough to allow an accurate understanding of the time it takes to find high performing DANNs. For the above reasons, we decided to use CIFAR-10 to evaluate the effectiveness of DENSER on automatically discovering DANNs.

Figure 4.3: From top to bottom, each row depicts randomly selected instances of the MNIST, MNIST Rotated, MNIST Background, MNIST Rotated Background, Fashion-MNIST, SVHN, Rectangles, Rectangles Background, and CIFAR, respectively. Each instance represents one of the classes of the dataset. The Rectangles dataset only has 2 classes, the first five belong to one of the classes, and the last five to the other.

To test the generalisation and robustness of the best models generated by DENSER, we use MNIST [145], which was previously used and detailed in Section 3.3. To investigate the robustness of the evolved ANNs, in addition to the standard MNIST, we also test several variants[1]:

MNIST Rotated – the digits are rotated between 0 and 360º;

MNIST Background – instead of a clean white background, a real-world image is used as background of the digit;

MNIST Rotated + Background – combines the previous two modifications, i.e., the digits are rotated, and an image is used in the background.

The standard MNIST is widely used as a dataset to establish comparisons among approaches. Nonetheless, it is known to be an easy task, where simple CNNs can achieve accuracies close to 99% [223]. For that reason, we have decided to also test on Fashion-MNIST [268] and SVHN [185]. Fashion-MNIST has the same format as MNIST (i.e., 28×28 grayscale images), but digits are replaced by clothing items. SVHN has an objective similar to MNIST (i.e., identify digits between 0 and 9), but the images are from Google Street View, and thus harder to distinguish.

[1]For more information about the MNIST variants check `https://sites.google.com/a/lisa.iro.umontreal.ca/public_static_twiki/variations-on-the-mnist-digits` (last accessed March 20, 2020).

Table 4.2: Parameters of the experiments conducted with DENSER.

Evolutionary Engine Parameter	**Value**
Number of runs	10
Number of generations	100
Population size	100
Crossover rate	70%
Mutation rate	30%
Tournament size	3
Elite size	1%
Dataset Parameter	**Value**
Train set	42500 instances
Validation set	7500 instances
Test set	10000 instances
Training Parameter	**Value**
Number of epochs	10
Loss	Categorical Cross-entropy
Batch size	125
Learning rate	0.01
Momentum	0.9
Data Augmentation Parameter	**Value**
Padding	4
Random crop	4
Horizontal flipping	50%

The scalability of the models evolved by DENSER is tested in the Rectangles [143], and CIFAR-100 [137] datasets. In the Rectangles, the goal is to distinguish between tall and wide rectangles placed in 28×28 images. Two setups are tested: (i) without any background image; and (ii) with a background image (similar to MNIST Background Image). CIFAR-100 is composed of the same images as CIFAR-10, but they are separated into 100 disjoint classes.

Table 4.1 details the numeric properties of the used datasets, namely, the number of training and test instances, the input size, and the number of classes. As mentioned previously, the evolutionary search for CNNs is carried out using CIFAR-10. For that reason, we partition CIFAR-10 into 3 disjoint sets: train, validation, and test. In all the other cases, since we use the ANNs found for CIFAR-10, only the training and test sets are needed. Figure 4.3 depicts example instances of each of the datasets.

4.2.2 Experimental Setup

Table 4.2 details the experimental parameters used for conducting the evolutionary runs on the automatic discovery of ANNs able to effectively classify CIFAR-10. The parameters are divided into 4 categories: (i) evolutionary engine – associated with the Evolutionary Algorithm (EA); (ii) dataset – partitioning of CIFAR-10 (recall that the test set is kept out of evolution); (iii) training – backpropagation algorithm parameters; and (iv) data augmentation – used to generate more data and prevent overfitting. The data is augmented using padding, horizontal flips, and random crops.

To encode CNNs we use Grammar 4.1 (except for the learning production rule that is not considered), and the following outer level structure: [(features, 1, 30), (classification, 1, 10), (softmax, 1, 1)]. Our search space encompasses CNNs with up to 41 hidden-layers: at most 30 convolution or pooling layers followed by up to 10 fully-connected and one softmax layers. When merging the output of the convolution layers with the input (merge-input parameter), the shape of the signals must be the same. When the number of channels is not equal, we pad the smallest of the signals; if the dimensionality of the signals is different, we down-sample the largest one using max-pooling. In this set of experiments, we decided not to optimise the learning parameters. That is why there is no production rule associated with learning (the parameters of Table 4.2, section training parameter are used). To evaluate the performance of the networks, we use their accuracy. We decided for this metric to enable comparison with other works, and because we will not be dealing with unbalanced datasets; in any case, it is straightforward to change accuracy to any other metric (e.g., f-measure).

The longer the ANNs are trained for the better grasp we have regarding their long term behaviour. However, the training of DANNs is a computationally expensive task, mostly due to the burden of the evolutionary process, since each network needs to be evaluated for its fitness. Likewise, similarly to the works of Miikkulainen et al. [171] and Suganuma et al. [236], we perform a short training of each DANN. In particular, we use 10 epochs. The fitness is the best performance on the validation set on the 10 epochs. Longer learning cycles, i.e., more epochs, would enable higher validation performances, and a better grasp of the long term training behaviour. On the other hand, longer training sessions would slow the evolutionary process, as the evaluation stage would be more time-consuming. After the evolutionary runs, we take the fittest network and perform a longer training, with 400 epochs, and the same learning policy. In the final learning scenario, the training and validation sets are merged, and the performance measured on the test set. For the same reason, only 10 evolutionary runs are conducted.

During evolution, invalid solutions can be generated. This occurs because some of the layers change the input shape. More specifically, pooling layers down-sample the input space. The same can happen with convolution layers when padding is not used. To avoid invalid solutions, the layers that generate invalid shapes[2] are not considered to build the model that is later trained and evaluated, i.e., these layers are non-coding genotype. We do not repair the genotype because to fix a layer we would have to change its parameters, and the layer can be a copy by reference, and thus we would possibly be changing too much of the genotype.

4.2.3 Evolution of CNNs for CIFAR-10

We start by analysing the ability of DENSER to search for and generate DANNs able to solve a classification task. We focus on object recognition, since it is an active research area in DL and thus, we target the automatic evolution of CNNs for the CIFAR-10 dataset.

The evolution of the average fitness (i.e., classification accuracy) and the number of hidden-layers of the fittest CNNs across generations is depicted in Figure 4.4. A brief perusal of the fitness evolution indicates that DENSER is working properly, as solutions tend to be fitter as time passes. The evolution converges around the 80th generation, with a classification accuracy of approximately 85%. The behaviour of the number of hidden-layers is more erratic, and two different and contradictory patterns are observable. From the start of evolution, and until the 60th generation, an increase in performance is accompanied by a decrease in the number of hidden-layers. This changes from the 60th generation until the last generation where, an increase in performance, is accompanied by an increase in the number of hidden-layers.

[2] An invalid shape is generated when the down-sampling creates a negative size for the shape of the output of the layer.

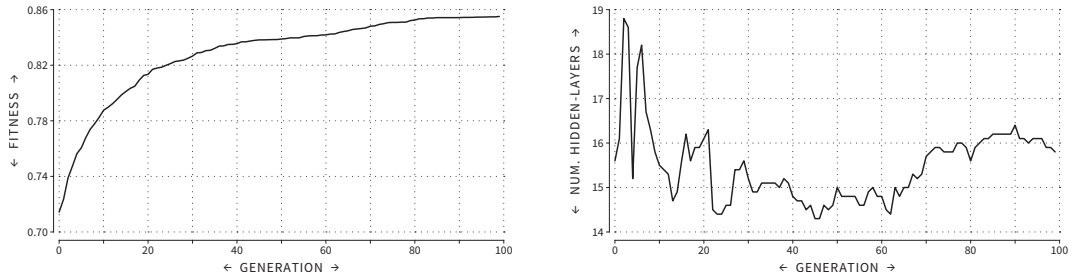


Figure 4.4: Experiments conducted with DENSER in the CIFAR-10 dataset. Evolution of the fitness (left) and number of hidden-layers (right) of the best individuals across generations. Results are averages of 10 independent runs.

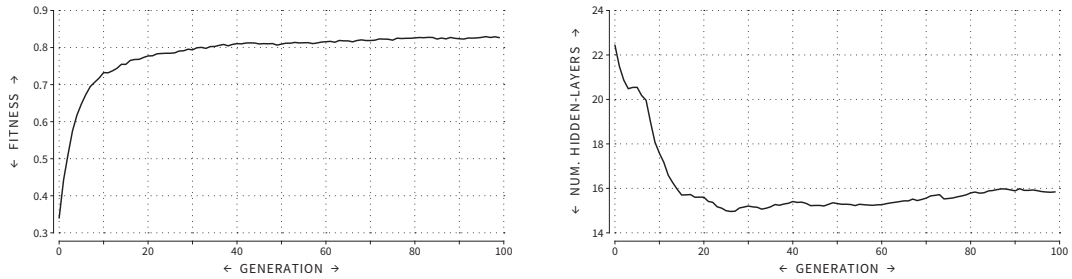


Figure 4.5: Experiments conducted with DENSER in the CIFAR-10 dataset. Evolution of the fitness (left) and number of hidden-layers (right) of the overall population across generations. Results are averages of 10 independent runs.

To support the previous result, we compute the correlation between the average fitness values of the best individuals and the average number of hidden-layers, per generation. Before the 60th generation, the Pearson correlation reports a coefficient of -0.7166 (moderate negative correlation) for the correlation between the two metrics. After the 60th generation, the coefficient is 0.9204 (strong positive correlation). This result is explained after the fact that, in the first generation, the randomly generated solutions have a large number of hidden-layers (approximately 15.6), which correspond to very deep networks. However, since the numeric parameters of each layer are set at random, they hardly provide any meaningful parameterisation. As evolution proceeds, and optimises the numeric values, the best solutions can steadily increase the number of hidden-layers, to improve performance.

The previous explanation suggests that it may be advantageous to start evolution from shallower networks, i.e., constrain the initialisation procedure to a maximum number of hidden-layers. Although this is common practice in many approaches, we strive to provide the least amount of knowledge and bias to the system. Therefore, it is noticeable that starting from very deep networks, DENSER finds out that trimming is necessary during the first generations.

In addition to analysing the best-evolved solutions, we also inspect the overall quality of the population. Figure 4.5 depicts the evolution of the fitness, and the number of hidden-layers across generations, at the population level. The conclusions are in line with those reported for the analysis of the best solutions, but the change in behaviour occurs earlier (around the 25th generation). Before the 25th generation, the Pearson correlation between the fitness and number of hidden-layers reports a coefficient of -0.89 (strong negative correlation), and a coefficient of

0.8801 after the 25th generation (strong positive correlation). The change in behaviour happens earlier than when considering only the best solutions because, in the first generations, the population has many low performing solutions that are quickly discarded.

By the end of evolution, the best solutions and the overall population have very similar fitness values of approximately 85% and 83%, respectively. The same happens with the number of hidden-layers, which is close to 16 for the best solutions and overall population analyses. This reinforces the idea that DENSER is working properly, and that the population is converging towards fit solutions.

The fittest network found during evolution is represented in Figure 4.6. The network is selected based on the accuracy in the validation set[3]. The analysis of the structure of the evolved network shows that it is different from the majority of the manually optimised networks: (i) convolutional and pooling layers do not form a block that is stacked multiple times; (ii) from the first to the last convolutional layer the filter shape does not follow any particular rule, e.g., in the ResNet [101] and VGG [224] all the filters have the same shape (3x3), and in the AlexNet [138] the filter shapes decrease from 11x11, to 5x5, and finally to 3x3; the same analysis can be done regarding the number of filters; and (iii) the evolved networks seem to be more prone to stack multiple fully-connected layers before the classification head. In the example of Figure 4.6 there is only one fully-connected layer before the classification head, but an analysis of the remaining of the best-found topologies shows that there is a tendency for more layers. Thus, it is fair to state that evolution promotes the emergence of network topologies that a human would unlikely think of, which makes this outcome novel and unexpected.

4.2.3.1 Further Training

Once the evolutionary process is complete, the fittest network found in each run, i.e., the ones that attain the highest accuracy value on the validation set, are re-trained 5 times. The networks are trained multiple times because the BP algorithm is stochastic (due to the different random weight initialisations), and thus different training sessions can provide significantly different performances. The accuracy results are averaged over 5 training sessions.

First, we train the networks with the same learning rate policy that is used in evolution (see Table 4.2), but during 400 epochs instead of 10. From this point onward, the classification accuracy results concern the performance obtained on the test set. With this setup, we obtain, on average, a classification accuracy of 88.41%. We also experiment with different test policies, namely the one described by Snoek et al. in [226]: for each instance of the test set, we generate 100 augmented images. The label assigned by the model to the instance is the class that gives the maximum average confidence value, on the 100 augmented images. With this methodology, the average classification accuracy of the fittest networks increases to 89.93%.

Notwithstanding the increase in performance by using the methodology followed by Snoek et al., the results still seem far from the state of the art. This happens because the results of the state of the art only focus on the performance reported by a single CNN. The classification accuracy of the best performing CNN found by DENSER (using the methodology by Snoek et al.) is of 92.70%. This result, despite competitive with the state of the art, does not surpass it. To avoid a biased choice of the best network, we base the selection on the network that attains the highest training accuracy. This also happens to be the network that provides the highest test accuracy, which proves that the networks generalise well.

To investigate if it is possible to increase the performance of the fittest networks, we re-train them with a different learning rate policy. In particular, we follow the methodology of CGP-

[3]It is later shown that this network reports as one of the best performing in terms of test accuracy, and thus it generalises well.

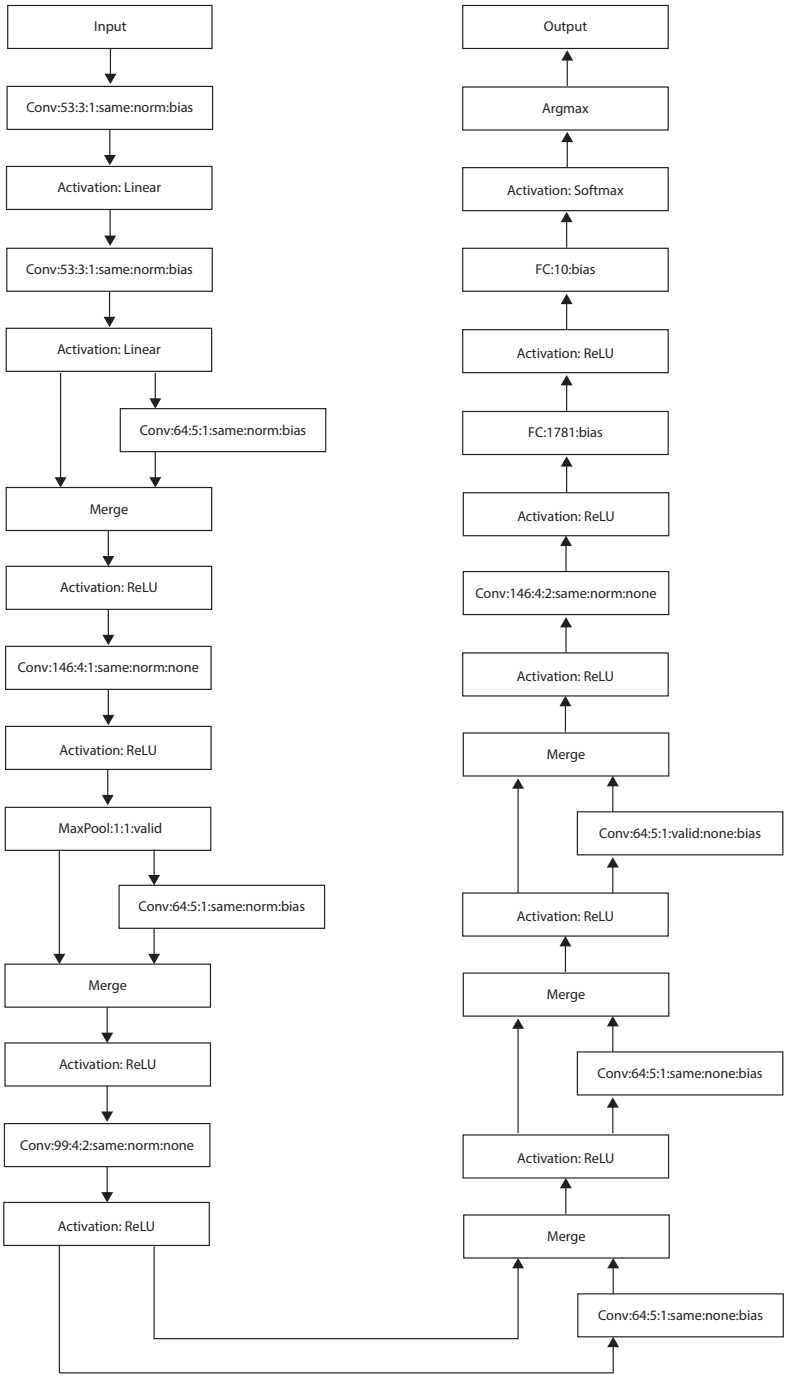


Figure 4.6: Topology of the fittest network found by DENSER during evolution. The network has 5 layer types: convolution (Conv), max-pooling (MaxPool), fully-connected (FC), merge, and activation. The layer blocks are formatted as: Conv:num-filters:filter-shape:stride:padding:batch-normalisation:bias; MaxPool:kernel-size:stride:padding; FC:num-units:bias; Activation:type. For more information about the padding type refer to the Keras documentation. When batch-normalization or bias are not used, we set the parameter to none.

Table 4.3: Overview of the results reported by DENSER on CIFAR-10.

Networks	Test Data Augmentation	Epochs	LR Policy-1	LR Policy-2
AVG best (evolution)	No	10	85.51%	–
AVG best	No	400	88.41%	92.51%
AVG best	Yes	400	89.93%	93.29%
Best	No	400	91.75%	93.38%
Best	Yes	400	92.70%	94.13%

CNN [236]: a varying learning rate that starts with 0.01; on the 5th epoch is increased to 0.1; by the 250th epoch is decreased to 0.01; and finally at the 375th epoch reduced to 0.001. With this learning policy, the mean accuracy of the best networks increases from 88.41% to 92.51%, without using data augmentation on the test set; and to 93.29% when data augmentation is applied to the test set. When we consider only the network that reports the highest accuracy on the training set, the best test classification accuracy is 93.38% without data augmentation; and 94.13% with data augmentation. This last result is highly competitive and is aligned with the state-of-the-art approaches for the automatic search of DANN to classify CIFAR-10.

The experimental results indicate that it is possible to obtain competitive results using evolutionary approaches and that it is possible to do so with limited computational resources, using a low number of training epochs (10) during evolution. Table 4.3 summarises the results obtained so far with CIFAR-10. The learning rate (LR) Policy-1 refers to the learning policy used during evolution (with 400 epochs); the LR Policy-2 refers to the variable learning rate methodology followed by CGP-CNN.

4.2.3.2 Ensembling

The previous results indicate that the best performing DANNs are obtained when using the varying learning rate policy from Suganuma et al. [236], and applying data augmentation to the test set. For each network, we performed 5 independent training sessions, where the initial conditions are different (due to the initial weights). Another source of stochastic behaviour is data augmentation. Therefore, despite having the same topology, the networks may have slightly different behaviours. For that reason, we investigate whether or not an ensemble composed of the 5 different training sessions performs better than the individually trained networks. The ensemble decision is taken as when applying data augmentation to the test set, i.e., we average the confidence values reported by all voters, and the decision is the class that reports a maximum average confidence value. The ensemble formed by the 5 training sessions of the best performing network achieves an accuracy of 95.14% (superior to the previous 94.13%).

In addition to testing the ensembles formed from different training sessions of the same network, we also investigate the performance of the ensembles formed by different networks. The decision of the ensemble is taken as before. In particular, we ensemble the two fittest networks found by DENSER, and for each of them, we consider the 5 independent training sessions, i.e., the ensemble is formed by 10 voters. This setup obtains an accuracy of 95.22%, which is superior to the one obtained when considering only the fittest network.

4.2.4 Generalisation, Robustness and Scalability

To better understand the quality of the CNNs generated by DENSER we investigate their: (i) generalisation – the ability to solve multiple tasks; (ii) robustness – the resilience to small modifications of the original input; and (iii) scalability – the capacity to solve problems with less

and more classes. To that end, we take the fittest CNNs and apply them to the classification of other benchmarks, which were not used for finding the topology. In particular, we test with MNIST and its variants (Section 4.2.4.1), with Fashion-MNIST and SVHN (Section 4.2.4.2), and with Rectangles and CIFAR-100 (Section 4.2.4.3).

4.2.4.1 MNIST and Variants

To investigate the robustness of the best performing CNN discovered by DENSER, we re-train the network in the MNIST dataset and apply it to the classification of the MNIST variants: MNIST Rotated, MNIST Background, and MNIST Rotated + Background. In this second step, the networks are not re-trained, i.e., the weights tuned for the standard MNIST are directly used for prediction. We train the network with LR Policy-2, the learning rate policy that obtained the best results. Trained on the standard MNIST, the network reports an average classification accuracy of 99.65%, without data augmentation on the test set, and an average classification accuracy of 99.70%, using data augmentation.

For the MNIST Rotated, MNIST Background, and MNIST Rotated + Background, using the previously trained CNN the classification accuracies are 46.63%, 42.48%, and 22.27%, respectively. These results were obtained without applying data augmentation to the test set instances because the images that are fed to the network are already "augmented" versions of MNIST. Compared to the 99.65% performance in MNIST, the results may be considered underperforming. Looking at the first 4 rows of Figure 4.3, it is clear that the dataset instances of the MNIST dataset (first row), and its variants (second to fourth rows) are very different. In addition, our data augmentation method does not consider image rotation.

However, when we re-train the network for the MNIST variants, we obtain average accuracies of 97.71%, 98.62%, and 92.66%, for the MNIST Rotated, MNIST Background, and MNIST Rotated + Background, respectively. These accuracies show that the evolved topology is perfectly able to cope with the variants of MNIST. Thus, the poor performance reported earlier is explained by the lack of adequate examples in the standard MNIST dataset, even when augmented, to allow the network to generalise to these new circumstances.

Finally, we re-train the network using instances from all MNIST variants, i.e., instead of training the network 4 times, one with each of the variants plus the standard dataset, we conduct a single training session, with 15000 instances from the standard dataset and from each one of the variants. The training instances are randomly selected, in a stratified way, meaning that the dataset is balanced. This setup yields an average classification accuracy of 95.67%.

4.2.4.2 Fashion-MNIST and SVHN

MNIST, despite important as a baseline, is nowadays considered a problem that is easy to solve, where a simple MLP can achieve performances close to 90%. That is the reason why we look into the performance in other more challenging datasets to better assess the generalisation ability of the CNNs generated by DENSER.

We take the best performing network and re-train it in the Fashion-MNIST, and SVHN datasets, with LR Policy-2. In the Fashion-MNIST, we get average classification accuracies of 94.23% (without data augmentation on the test set), and of 94.70% (with data augmentation). In the SVHN dataset, we get average classification accuracies of 95.44% (without data augmentation on the test set), and of 96.23% (with data augmentation).

As in previous experiments, we test ensembling the different training sessions of a single model, or of the two fittest models. In the Fashion-MNIST, the ensembles report classification accuracies of 95.11%, and 95.26%, considering only the fittest, or the two fittest models, respec-

tively. The same analysis in the SVHN reports classification accuracies of 96.88%, and 97.02%, considering only the fittest, or the two fittest models, respectively.

The results in MNIST, Fashion-MNIST, and SVHN allow us to state that the models evolved by DENSER generalise, i.e., despite not directly evolved for solving these tasks, the CNNs perform well beyond evolution. The comparison with the state of the art is carried out in Section 4.2.5.

4.2.4.3 Rectangles and CIFAR-100

In all of the above experiments, we have always sought to solve problems with 10 classes. To analyse the scalability of the models generated by DENSER, we test with Rectangles (an artificial problem with 2 classes), and CIFAR-100 (with 100 classes). We test in the two Rectangles variants: with and without background images. The same flow-chart is followed: we take the fittest CNNs and re-train them on the Rectangles variants and CIFAR-100 datasets.

The fittest CNNs excel in the classification of Rectangles: without background, we get an average test accuracy of 100%; with background, we reach a classification accuracy of 99.64%. Data augmentation of the test set does not improve the results. We do not experiment ensembling with the models trained in the Rectangles without background because no improvements are possible. Ensembling the classifiers trained with background slightly increases the classification accuracy to 99.73%.

In CIFAR-100, we report classification accuracies of 73.33%, and 74.94%, not performing and performing data augmentation over the test set, respectively. The ensemble formed only by the training sessions of the fittest model gives an average accuracy of 77.51%; with the two fittest models, the performance increases to 78.75%.

4.2.5 Discussion

To investigate the ability of DENSER to effectively evolve DANNs, we perform two independent sets of experiments: (i) evolutionary search of CNNs for the classification of CIFAR-10; and (ii) analysis of the generalisation, robustness, and scalability of the networks generated by DENSER for the classification of CIFAR-10. To measure these properties, we use the MNIST, Fashion-MNIST, SVHN, Rectangles, and CIFAR-100 datasets.

Evolution works as expected: DENSER promotes the evolution of high performing networks, and at the end of evolution, the average fitness of the population is very close to the average fitness of the best candidate solutions. Besides, the evolutionary method replicates behaviours that are common in hand-designed networks. In particular, when, at the beginning of evolution, the ANNs are trimmed to facilitate the optimisation of the parameters. On the other hand, DENSER generates networks that are novel, and that human-designers would unlikely think of.

The networks that report the best classification accuracy are applied to a wide range of datasets and, for all of them, they report high accuracy values. As such, DENSER is able to evolve ANNs that generalise well. In particular, we analyse the robustness by predicting the class labels of MNIST perturbed instances (rotated and/or added background) using CNNs that are trained in the standard MNIST. At first, it may seem that the networks underperform but, we attribute that to the used data augmentation method, that does not promote the rotation of the instances (the MNIST variant where the performance is lowest). When the CNNs are re-trained with images from all variants, the CNNs attain performances that are close to the ones obtained in standard MNIST. Finally, the scalability of the evolved CNNs is tested in the Rectangles and CIFAR-100 datasets. The networks excel, with the performance in CIFAR-100 surpassing the state-of-the-art results (discussed next).

Table 4.4: Performance of different CNNs on the classification of the datasets used in the experiments conducted with DENSER. The VGG, ResNet results for the MNIST and Fashion-MNIST are from `github.com/zalandoresearch/fashion-mnist`. Automatic approaches are marked with an *.

Approach	Dataset	Accuracy
ResNet [101]	MNIST	97.90%
EvoCNN [238]*		98.82%
MetaQNN [22]*		99.56%
(Simard et al., 2003) [223]		99.60%
(Graham, 2014) [87]		99.68%
VGG [224]		99.68%
DENSER (best network)*		99.70%
VGG [224]	Fashion-MNIST	93.50%
EvoCNN [238]*		94.53%
DENSER (best network)*		94.70%
ResNet [101]		94.90%
DENSER (ensemble)*		95.26%
(Sermanet et al., 2012) [219]	SVHN	95.10%
DENSER (best network)*		96.23%
DENSER (ensemble)*		97.02%
MetaQNN [22]*		97.72%
(Goodfellow et al., 2013) [83]		97.84%
(Loshchilov and Hutter, 2016) [155]*	CIFAR-10	90.70%
VGG [224]		92.26%
CoDeepNEAT [171]*		92.70%
MetaQNN [22]*		93.08%
ResNet [101]		93.39%
(Snoek et al., 2015) [226]*		93.63%
CGP-CNN [236]*		94.02%
DENSER (best network)*		94.13%
DENSER (ensemble)*		95.22%
AE-CNN [239]*		95.3%
(Real et al., 2017) [203]*		95.60%
(Graham, 2014) [87]		96.53%
NASNet-A [280]*		96.59%
AmoebaNet-A [202]*		96.66%
ResNet [101]	CIFAR-100	71.14%
VGG [224]		71.95%
(Snoek et al., 2015) [226]*		72.60%
MetaQNN [22]*		72.86%
(Graham, 2014) [87]		73.61%
DENSER (best network)*		74.94%
(Real et al., 2017) [203]*		77.00%
AE-CNN [239]*		77.60%
DENSER (ensemble)*		78.75%

Table 4.4 enumerates state-of-the-art results in the classification of the datasets considered above. We focus on the comparison of our results with those reported by other CNNs. Some of the state-of-the-art results were obtained by hand-designed networks, instead of using an automatic approach (evolutionary or any other form of automation). As such, the ones that are obtained by automatic CNN designing methods are marked with an asterisk (*). We only consider for comparison with the state of the art the datasets that are most often used. For the MNIST, Fashion-MNIST, SVHN, CIFAR-10, and CIFAR-100, we achieve average test classification accuracies of 99.70%, 94.70%, 96.23%, 94.13%, and 74.94%, respectively. The previous results do not consider the ensembles, which slightly increase the performance. Comparing with the results enumerated on the table, we can observe that the results reported by DENSER are superior to the state of the art in the MNIST, and Fashion-MNIST, and are competitive in the remaining benchmarks.

Focusing the analysis on CIFAR-10 and CIFAR-100, DENSER obtains results that are 2.53%, and 2.66% below AmoebaNet-A, and AE-CNN, respectively, which are the best performing automatic approaches for these benchmarks. We note that in what regards the search space, these approaches are slightly different from DENSER. As discussed in Section 2.4.2, AmoebaNet-A is based on the evolution of blocks that are placed in a fixed architecture (the same happens with NASNet-A), and AE-CNN is based on ResNet and DenseNet blocks, i.e., evolution is biased towards high-performing structures. That is, both works impose constraints to evolution that make the search space smaller, and facilitate the emergence of fit solutions, i.e., the search does not start from scratch. We note that Real et al. use ensembling, and thus their results are to be compared with our ensembling ones. DENSER reports average ensembling test accuracies of 95.22%, and 78.75%, respectively for CIFAR-10, and CIFAR-100. A deeper analysis of their work shows that the search space is considerably smaller than ours, with several constraints on the convolution parameters, and no pooling or fully-connected layers. DENSER searches a larger space, obtaining a similar performance in CIFAR-10. For CIFAR-100, Real et al. re-conduct evolution from scratch; we just take the fittest network found for CIFAR-10, and apply it to CIFAR-100, outperforming Real et al.'s results.

To compare the generated networks with hand-designed ones, we can also look at Table 4.4: all the results not obtained by automatic approaches are the outcome of iterative trial-and-error human design (i.e., all those not marked with *). Concerning only the best-generated model, DENSER outperforms the human-designed CNNs in the MNIST, and CIFAR-100 datasets, and obtains competitive results in the Fashion-MNIST. The results obtained in SVHN and CIFAR-10 are below the ones reported by human-designed models. More precisely, to address CIFAR-10, Graham [87] proposes fractional max-pooling, obtaining an accuracy that is 2.4% above the fittest DENSER model. To the moment, fractional max-pooling is not considered in the search space of DENSER but can be easily added.

Regarding the architectures of the hand-designed models, they tend to be different than what the evolutionary process generates. When the networks are designed by a human, we often find blocks of layers that are replicated several times, e.g. convolution layer followed by pooling [83], or a set of convolution layers with different resolution levels [224]. As it is observable from the model of Figure 4.6 this will very hardly happen when evolving the topology from scratch, without defining what blocks are, i.e., with no prior knowledge about the search space. The same happens when we compare the models designed by other evolutionary systems with those designed by a human practitioner. Unless we specifically define a macrostructure that establishes the notion of block [202, 236, 280], the generated networks will have a novel architecture, different from the vast majority of existing models. From our point-of-view, this is a key advantage of applying EC to search for effective architectures: it enables us to obtain out-of-the-box models that a human designer would probably never imagine. The networks generated by DENSER and

other evolutionary systems are similar. This does not mean that the same sequence of layers is used, but rather that the built models have a very flexible, and unconstrained structure.

4.3 Limitations

The main issue with the automatic search of ANNs resorting to NE is the time required for evaluating the population. The evaluation of each individual implies training, which is time-consuming. DENSER suffers from the same problem. On average, each evolutionary run on the search for CNNs for the CIFAR-10 dataset took 1083 hours to perform 100 generations, i.e., an average of 10.83 hours/generation, or an average of 6.5 minutes/individual (10 epochs). The timings are measured considering a dedicated machine with 4 1080 Ti GPUs, and 64 GB of memory. Each evolutionary run is conducted on a single GPU, i.e., we perform 4 runs in parallel. Another option would be to evolve the weights directly, instead of performing the training of the deep networks. The problem is that evolving thousands or even millions of weights is likely even more time-consuming.

The above experiments focus only on the optimisation of the topology of the network, and by the end of evolution, we take the best-found models and experiment with different learning strategies. The best performances were obtained when, after evolution, the CNNs were trained, for 400 epochs, with a varying learning rate policy. However, considering that, on average, training an individual for 10 epochs takes 6.5 minutes, performing 400 epochs is unfeasible. On the other hand, there is no guarantee that the used learning strategy is adequate for all problems/networks. The problem is that the simultaneous search for the topology and learning strategy increases the search space, and thus it is expected that more generations would be required, increasing the search time.

In addition to the aforementioned limitations, two others can be pointed out. Firstly, the ANNs that are generated throughout evolution are not fully-trained, i.e., they cannot be deployed right-off evolution, and require further training (as performed in the experiments described in the current chapter). Secondly, during evolution, the search is conducted towards just one particular task. Although the networks are robust, generalisable, and scalable, when one requires networks to a very different problem the search has to be re-started from scratch. There is no way to expand the system so that it cumulatively learns new tasks. The upcoming chapters expand DENSER to overcome these limitations.

4.4 Summary

The current chapter describes DENSER: a general-purpose layer and grammar-based NE representation. To represent the individuals, DENSER introduces a novel two-level scheme: the outer level encodes an ordered linear sequence of evolutionary units, and the inner level encodes the parameters of each evolutionary unit. We also design specific genetic operators that focus on the evolution of DANNs based on the two-level representation of the genotype. How the solutions are encoded allows a two-fold gain: (i) the genetic material is encapsulated, which facilitates the application of the genetic operators; and (ii) the grammatical nature of the method makes it easy to evolve solutions to different problems, or different network structures.

The results show the effectiveness of DENSER. We conduct experiments in the evolution of CNNs for CIFAR-10, and the results show that DENSER is currently the evolutionary approach, with no prior knowledge, that generates the best performing networks. In addition, the evolved networks generalise, are robust, and scale. The most remarkable result of this is the performance in CIFAR-100: without further evolution, when the fittest networks in CIFAR-10 are re-trained

in CIFAR-100, they obtain performances that surpass the current state-of-the-art results, with an average classification accuracy of 78.75%.

Despite the ability of DENSER to generate high performing networks, the designed experiments do not address the optimisation of the learning strategy and/or learning parameters. The evaluation of the networks is conducted for 10 epochs, and thus it is hard to evolve effective learning rate schedules. In the next chapters, we investigate and introduce extensions to DENSER that speedup search enabling the learning to be considered during evolution.

Chapter 5

Fast Deep Evolutionary Network Structured Representation

Driven by the ability of DENSER to generate high performing DANNs, we investigate solutions to mitigate some of the aforementioned framework limitations. In particular, Fast-DENSER, as the name suggests, tackles the issue of the time it takes DENSER to search for effective solutions. In addition to speeding up evolution, Fast-DENSER also extends the representation of DENSER to a more flexible scheme that enables the layers of the evolved networks to connect to any of the previous layers. Therefore, the framework allows the emergence of skip connections or even residual networks, and avoids the merge-input parameter. To overcome the need for further training after evolution, Fast-DENSER introduces a new mutation operator that allows the continuous increase of the training time of each individual.

The differences between Fast-DENSER and DENSER are detailed in Section 5.1. The code for Fast-DENSER is available under the Apache 2.0 license at `https://github.com/fillassuncao/fast-denser3`. To facilitate the configuration of the framework, we release docker images (with and without GPU support), which are available at `https://hub.docker.com/r/fillassuncao/f-denser`. The details of the framework implementation and an example of its application to the Fashion-MNIST dataset can be found in Appendix B.

We compare the performance of Fast-DENSER and DENSER on the evolution of CNNs for CIFAR-10 (Section 5.2), and later we apply Fast-DENSER to a problem in the domain of physics, which concerns the the automatic search of CNNs to the gamma/hadron discrimination problem, based on the ground impact patterns (Section 5.3). The results are summarised in Section 5.4.

5.1 Extensions to DENSER

The main goal of Fast-DENSER is to speedup DENSER, i.e., to find solutions in less time but without compromising the performance. To that end, we change how evolution is conducted. In DENSER the evolutionary search is guided by a GA, and in Fast-DENSER we replace the evolutionary engine by a (1+λ)-ES (Section 5.1.1). In addition, we observe that in the canonical implementation of DENSER, in the initial generations, the DANNs are trimmed, and thus, we adapt the initialisation procedure to deepen the networks as necessary (Sections 5.1.3).

Fast-DENSER also adds a new genotypic level to DENSER. The goal of the new genotypic level is to make the representation more flexible, by enabling the layers of the evolved networks to be connected to any of the previous layers. This way, it is easy to promote the emergence of

skip connections or even residual networks (Section 5.1.2). The new genotypic level requires new genetic operators (Section 5.1.4), that also deal with the continuous adaptation of the training time of the networks.

5.1.1 Evolutionary Engine and Evaluation Strategy

In DENSER the evolutionary procedure relies on a standard GA method where a relatively large population is evolved throughout generations. In the experiments described in Chapter 4, for each run, we conducted 100 generations with a population size of 100, i.e., there are a total of 10 000 individuals evaluated per run. Taking into account that all individuals are trained for 10 epochs, this translates into a total of 100 000 epochs.

To speed up evolution, Fast-DENSER replaces the GA by a $(1+\lambda)$-ES. The rationale behind choosing an ES is related to reducing the number of individuals that need to be evaluated in each generation. With a $\lambda = 4$, in Fast-DENSER, we only assess the quality of 5 individuals in each generation, which translates into performing 1/20th of the evaluations. The question lies in understanding the impact that having fewer individuals (e.g., diversity loss) has on the quality of the best-found DANNs.

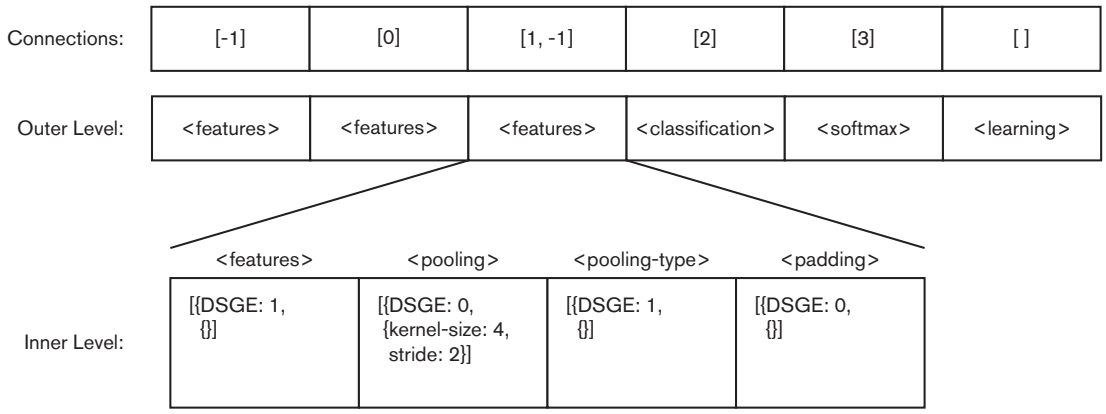
The training of ANNs is stochastic, due to the initial random setting of the weights, and thus different training sessions can obtain different performances. With this in mind, and while in DENSER the elite was not re-evaluate, in Fast-DENSER the best individual that passes from one generation to the next is re-evaluated. With this change, we seek to mitigate the effect of the randomness of the initial weights on the training of the network, and thus generate DANNs that are robust to initial conditions.

So far, in all the designed experiments, to evaluate the performance of the evolved DANNs we train the networks for a limited number of epochs and measure the performance as the classification accuracy on a given task (or any other performance metric). This evaluation policy is designed to avoid very long learning schedules that, as a consequence, are time-consuming. The problem with such a small number of epochs is that it makes it difficult to optimise the learning parameters, and also make unfeasible the generation of networks that require no further training after evolution. On the other hand, it can be a somewhat unfair training stopping method: the training time for two different network topologies and/or learning strategies that are trained for the same number of epochs can be very different. Therefore, we investigate the training of the individuals for a maximum GPU training time and compare this stopping criterion with the previous, i.e., a maximum number of epochs. We also use early stopping, and thus learning can be halted before the maximum GPU training time is elapsed.

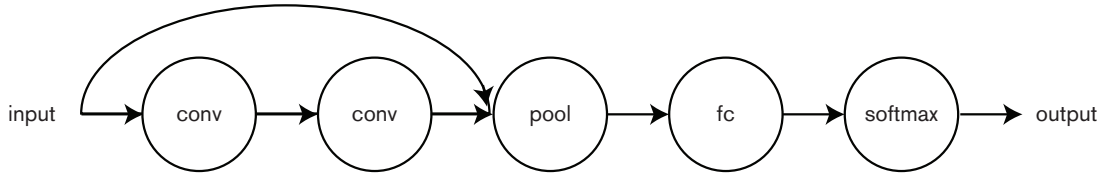
5.1.2 Representation

The individuals in DENSER are encoded as ordered linear sequences of feed-forward layers. Notwithstanding, in the original experiments of Section 4.2.3, the grammar defines a merge-input parameter for the convolution layers. The objective of this parameter is to merge the output of the convolution layer with its input. The merge-input parameter takes part in many of the convolution layers of the best performing networks. The problem is that this operator is layer-specific, and not generalisable to connect to any (or multiple) of the previous layers.

The core of the representation of Fast-DENSER is similar to DENSER: there is an ordered sequence of evolutionary units, and each evolutionary unit in the outer level has a corresponding inner level. The difference lies in the encoding of the connections between layers. Instead of assuming that the connection between two consecutive evolutionary units encoding layers is restricted to a connection from the previous to the next layer, there is a list of connections for



(a) Genotype. The connections define the list of inputs of each layer; -1 is the input of the network, and the remaining are the indexes of the outer level (starting in 0). The connections are restricted to the evolutionary units that encode layers. We only represent the inner level of one of the layers to ease readability. For a complete inner level genotype check Figure 4.1.



(b) Phenotype. We assume that the layers that do not have their inner level specified correspond to convolution, convolution, and fully-connected, respectively.

Figure 5.1: Example of the representation of an individual in Fast-DENSER. The example encodes a CNN, and considers Grammar 4.1.

each layer evolutionary unit, that specifies its inputs. This way, the encoding is close to a directed acyclic graph that is mapped into an ANN. Resembling what is done in graph-based approaches, like CGP [254], we introduce a new parameter: the maximum number of levels back, which sets a bound for the number of previous layers that can be used as input. To avoid disjoint graphs that generate invalid solutions, the layers have to be always connected to the previous layer.

Figure 5.1 depicts an example of the encoding of an individual, using the outer level structure [(features, 1, 10), (classification, 1, 4), (softmax, 1, 1), (learning, 1, 1)], and Grammar 4.1. The decoding procedure is the same as DENSER, but where the connections of the evolutionary units that encode layers are gathered from the connections genotypic level.

5.1.3 Initialisation

The experiments with DENSER on the evolution of CNNs for CIFAR-10 (Section 4.2.3) have demonstrated that when the initial population is generated entirely at random the number of layers is trimmed during the initial generations. This is expected, as generating a high number of hyper-parameters for very deep networks is unlikely to provide high performing networks. Therefore, in Fast-DENSER the initial population is still generated at random, but we impose a low upper bound on the maximum number of layers of the outer level structure. The initial

Algorithm 3 Fast-DENSER parent selection mechanism.

```
1: parent ← select_fittest(population)
2: if parent.train_time > DEFAULT_TIME then
3:     tmp_parent ← select_fittest(population-parent)
4:     retrain(tmp_parent, parent.train_time)
5:
6:     if tmp_parent.fitness > parent.fitness then
7:         return tmp_parent
8:     else
9:         return parent
10: else
11:     return parent
```

individuals have a small number of layers, and evolution proceeds in a constructive way, by deepening the ANNs. It is expected that by constraining the size of the initial individuals, we obtain a speedup because the initial generations do not trim the networks, and shallower networks tend to be faster to train.

5.1.4 Mutations

To promote evolution, we only apply mutations to the individuals. The genotype is similar to DENSER, and thus the mutation operators are the same as the ones described in Section 4.1.2: add, replicate and remove an evolutionary unit; and grammatical, integer and float mutations. In addition, we define three new operators. The first two introduce variations in the connections genotypic representation, and the last in the training of the individuals:

Add connection – picks at random a layer and adds another layer to its inputs. The inputs are restricted by the number of levels back;

Remove connection – removes a connection from a randomly chosen layer. The connection to the previous layer cannot be removed;

Training time – does not alter any of the evolutionary units, nor their parameters. Instead, it increases the training time of the individual (does not affect the remainder of the population). The training time is increased by multiples of the default training time. Indirectly, we are evolving solutions that have to train fast (or better, in the defined default time-window), but that given more time are likely to improve performance.

The new operators enable the evolution of unrestricted networks, and make the maximum GPU training time specific to each individual: in the initial population all individuals are trained for the same amount of time, and as the networks are simple they require less time. As the generations proceed, more complex solutions tend to emerge, that may benefit from longer evaluation cycles. That is, the training time grows as required by the networks. When a mutation that affects the network structure is applied, the training time is reset to the default.

The training time mutation operator makes it possible for individuals within the same population to have different evaluation times. This indirectly implies that the parent selection mechanism has to be changed so that the comparison between individuals in the population is fair. If the fittest individual has been trained for the default training time, the selection is the same as before, i.e., the fittest individual seeds the next generation. Otherwise, when the fittest

Table 5.1: Fast-DENSER experimental parameters.

Category	Parameter	Fast-DENSER	DENSER
Evolutionary Engine	Number of runs	10	
	Number of generations	150	100
	Population size	5	100
	Crossover rate	–	70%
	Mutation rate	–	30%
	Add layer rate	25%	–
	Remove layer rate	25%	–
	Add connection rate	15%	–
	Remove connection rate	15%	–
	DSGE-level rate	15%	–
	Tournament size	–	3
	Elite size	–	1
Dataset	Training set	42500 instances	
	Validation set	7500 instances	
	Test set	10000 instances	
Training	Number of epochs	10	
	Training time	10 min.	–
	Loss	Categorical Cross-entropy	
	Batch size	125	
	Learning rate	0.01	
	Momentum	0.9	
Data Augmentation	Padding	4	
	Random crop	4	
	Horizontal flipping	50%	

individual is trained for longer than the default training time, the fittest individual of those that were trained for the default training time is re-trained, and the fittest among the two seeds the next generation. That is, the variations of the parent are initially evaluated for the default time, and if in the population there is an individual evaluated for longer, the fittest individual is also granted the same time. The parent selection mechanism is clarified in Algorithm 3.

5.2 Evolution of Convolutional Neural Networks for Object Recognition

To compare Fast-DENSER to DENSER, we conduct a wide set of experiments on the evolution of CNNs for CIFAR-10. We divide the experiments in four different sets: (i) we compare Fast-DENSER to DENSER on the optimisation of the topology of CNNs, evaluated for 10 epochs (Section 5.2.2); (ii) we investigate the evaluation of the networks for a maximum granted GPU training time on the optimisation of the topology, and on the simultaneous optimisation of the topology and learning strategy (Section 5.2.3); (iii) we check the ability of Fast-DENSER to create networks that connect to multiple of the previous layers (Section 5.2.4); and finally (iv) we let the training time to grow continuously, i.e., we only use the training time mutation operator in this last experiment (Section 5.2.5). The experimental setup is detailed in Section 5.2.1.

<features> ::= <convolution> | <convolution> |<pooling> (1)
| <pooling> | <batch-norm> (2)
<convolution> ::= layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,2,5] (3)
[stride,int,1,1,3] <padding> <activation> <bias> (4)
<batch-norm> ::= layer:batch-norm (5)
<pooling> ::= <pool-type> [kernel-size,int,1,2,5] [stride,int,1,1,3] <padding> (6)
<pool-type> ::= layer:pool-avg | layer:pool-max (7)
<padding> ::= padding:same | padding:valid (8)
<classification> ::= <fully-connected> (9)
<fully-connected> ::= layer:fc <activation> [num-units,int,1,128,2048] <bias> (10)
<activation> ::= act:linear | act:relu | act:sigmoid (11)
<bias> ::= bias:True | bias:False (12)
<softmax> ::= layer:fc act:softmax num-units:10 bias:True (13)

Grammar 5.1: Grammar used by Fast-DENSER for the evolution of the topology.

5.2.1 Experimental Setup

The parameters used for performing the experiments described above are summarised in Table 5.1. The table is organised into sections: (i) evolutionary engine – parameters required by the evolutionary algorithm; (ii) dataset – partitioning of the dataset; (iii) training – training stop conditions, and learning algorithm parameters used for the experiments conducted with a fixed learning policy; and (iv) data augmentation – strategy for augmenting the dataset during training (never evolved in the reported experiments).

Despite a higher number of generations in Fast-DENSER than in DENSER, the number of evaluated individuals is a small fraction. Fast-DENSER performs, in each run, 750 evaluations, and DENSER performs, in each run, a total of 10000 evaluations. To speed up search, Fast-DENSER applies a higher mutation rate, and multiple mutations can be applied to the same individual in one generation. The mutation rates concerning the number of layers are per individual, and the remaining ones are by layer. The DSGE-level mutation rate is the per-gene probability of changing any of the expansion possibilities or terminals.

The dataset is partitioned (in a stratified way) into three disjoint sets. The training set is used to tune the weights of the individuals, and the validation set is used to assign fitness, and to perform early stopping based on the validation loss (in the experiments that use time as stopping criterion). The test set is kept out of evolution. In the experiments where the learning parameters are evolved, the batch size, learning rate, and momentum are not fixed. For the experiments where the training stopping criterion is the maximum GPU time, it is important to mention that we are using 8 GeForce GTX 1080 Ti GPUs. A run only uses one GPU. The experiments are conducted on CIFAR-10 [137].

For the different setups, we use different grammars. To evolve the topology with Fast-DENSER, we use Grammar 5.1. To evolve the topology with DENSER, we use Grammar 4.1. To simultaneously optimise the topology and learning strategy we use Grammar 5.2. The main differences between the grammars used by Fast-DENSER for the optimisation of the topology, or the simultaneous optimisation of the topology and learning strategy lies in the dropout layers, and in the learning production rules. The evaluation for a maximum GPU time enables the use

<features> ::= <convolution> | <convolution> |<pooling> (1)
| <pooling> | <batch-norm> | <dropout> (2)
<convolution> ::= layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,2,5] (3)
[stride,int,1,1,3] <padding> <activation> <bias> (4)
<batch-norm> ::= layer:batch-norm (5)
<pooling> ::= <pool-type> [kernel-size,int,1,2,5] [stride,int,1,1,3] <padding> (6)
<pool-type> ::= layer:pool-avg | layer:pool-max (7)
<padding> ::= padding:same | padding:valid (8)
<classification> ::= <fully-connected> | <dropout> (9)
<fully-connected> ::= layer:fc <activation> [num-units,int,1,128,2048] <bias> (10)
<dropout> ::=layer:dropput [rate,float,1,0,0.7] (11)
<activation> ::= act:linear | act:relu | act:sigmoid (12)
<bias> ::= bias:True | bias:False (13)
<softmax> ::= layer:fc act:softmax num-units:10 bias:True (14)
<learning> ::= <bp> <early-stop> [batch_size,int,1,50,500] epochs:400 (15)
| <rmsprop> <early-stop> [batch_size,int,1,50,500] epochs:400 (16)
| <adam> <early-stop> [batch_size,int,1,50,500] epochs:400 (17)
<bp> ::= learning:gradient-descent [lr,float,1,0.0001,0.1] <nesterov> (18)
[momentum,float,1,0.68,0.99] [decay,float,1,0.000001,0.001] (19)
<nesterov> ::= nesterov:True | nesterov:False (20)
<adam> ::= learning:adam [lr,float,1,0.0001,0.1] [beta1,float,1,0.5,1] (21)
[beta2,float,1,0.5,1] [decay,float,1,0.000001,0.001] (22)
<rmsprop> ::= learning:rmsprop [lr,float,1,0.0001,0.1] (23)
[rho,float,1,0.5,1] [decay,float,1,0.000001,0.001] (24)
<early-stop> ::= [early_stop,int,1,5,20] (25)

Grammar 5.2: Grammar used by Fast-DENSER for the evolution of the topology and learning.

of dropout and facilitates the tuning of the learning strategy, and that is the reason why they are only included in Grammar 5.2. The Fast-DENSER and DENSER grammars are different in what concerns the features and convolution production rules. The convolution layer in DENSER has more properties, namely, batch-normalisation and merge-input. DENSER had to tailor the convolution layer to encompass these extra parameters. We decided to follow a more flexible approach, and thus disregard these parameters. It is possible to mimic the merge-input with the novel feature of Fast-DENSER that enables layers to have more than one input connection.

5.2.2 Fast-DENSER vs. DENSER

To compare the performance of Fast-DENSER and DENSER, we conduct the same experiments of Section 4.2.3, i.e., we promote the search for the topology of CNNs for CIFAR-10, trained with BP with a fixed learning policy for 10 epochs (Table 5.1). The results are summarised in Table 5.2: evolution refers to the average of the best result of each run found throughout evolution; re-trained is the average performance of each of the best CNNs re-trained 5 times;

Table 5.2: Results of the evolution of CNNs with Fast-DENSER and DENSER. The accuracy results evolution and evolution (re-trained) are based on the validation set. The last row reports results on the test set. The value within brackets is the accuracy on the test set using augmented versions of the instances, i.e., each image is augmented 100 times, and the assigned label is the average of the maximum of the average of the confidence values.

	Fast-DENSER	DENSER	p-value
Evolution	84.54%	85.51%	0.308
Evolution (re-trained)	83.21%	83.62%	0.624
Test (no limit)	87.79% (89.28%)	88.19% (89.65%)	0.624 (0.497)

and no limit is the average classification accuracy of the best networks trained without limit on the number of epochs, instead an early stop (without improvement for 12 epochs) based on the validation loss is used. This table structure is kept for the remainder of the experiments in this section. The analysis of the results indicates that the performance of the evolutionary results of DENSER is slightly superior to Fast-DENSER. Notwithstanding, this changes when the networks are re-trained 5 times, demonstrating that the re-evaluation of the elite during evolution generates networks that are more robust. The training for an unlimited number of epochs leads to better performances in both methods.

The performance of the two methods is very similar (there is no statistical difference). On the other hand, the time complexity needed for obtaining the best solutions is very different. The average run-time of each run of Fast-DENSER is 55 hours, while DENSER requires an average of 1083 hours to complete 100 generations, i.e., there is a speedup of approximately 20× when we use Fast-DENSER. Thus, Fast-DENSER can find solutions that are highly competitive with the ones discovered by DENSER, but 20 times faster. Focusing on the structure of the evolved networks, it is noticeable that Fast-DENSER generates CNNs that have fewer layers (12.5) than the ones generated by DENSER (16.8), which may be motivated by starting evolution with networks that have fewer layers. This difference is statistically significant (p-value=0.0375).

To better investigate the comparative performance of Fast-DENSER and DENSER, we perform new experiments with DENSER on two different sets: (i) with the same population size of the original experiments (i.e., 100 individuals), but during less generations (6 generations); and (ii) with the same number of generations of Fast-DENSER (i.e., 150 generations), but with a smaller population (4 individuals). The experiments conducted using Fast-DENSER evaluate in each generation 4 new individuals (λ=4), and thus during 150 generations a total of 600 evaluations are performed. For the new experiments with DENSER, the population size and number of generations were chosen to keep the same number of evaluations of the experiments performed with Fast-DENSER, i.e., 600 evaluations. With the first setup, i.e., 100 individuals for 6 generations, DENSER reports an average classification accuracy of 75.51% on CIFAR-10. In the second setup, 4 individuals evaluated for 150 generations, DENSER reports an average classification accuracy of 72.47%. These results are below the performance of Fast-DENSER, and the differences are statistically significant (p-values of 0.00018, and 0.00058 for the first and second setups, respectively); the effect size is large.

The experiments conducted in the current sub-section have demonstrated that Fast-DENSER can generate ANNs that report a performance very similar to those of the ANNs discovered by DENSER using a substantially superior number of evaluations (no statistical difference). Further, Fast-DENSER generates networks that are similar in performance but in a small fraction of the time, with a speedup of roughly 20× when compared to the time taken by DENSER. However, when given the same number of evaluations, Fast-DENSER reports results that are statistically

Table 5.3: Results of the evolution of the topology (T) and learning strategy (L) parameters of CNNs with Fast-DENSER. The first p-value concerns the statistical result of the comparison between Epochs (T) and Time (T), and the second p-value the results of the comparison between Time (T) and Time (T+L).

	Epochs (T)	Time (T)	p-value	Time (T+L)	p-value
Evolution	84.54%	88.52%	0.000	88.29%	0.187
Evolution (re-trained)	83.21%	87.47%	0.000	87.56%	0.347
Test (no limit)	87.79% (89.28%)	87.65% (88.89%)	0.472 (0.103)	87.76% (88.90%)	0.308 (0.472)

superior to those obtained by DENSER. For these reasons, in the remainder of the current section, the experiments are conducted with Fast-DENSER.

5.2.3 Evaluation Stop Criteria

In the previous section, we have acknowledged that Fast-DENSER can generate results that match the performance of those obtained by DENSER but in a fraction of the time. More precisely, the speedup is roughly 20×. In the current section, we investigate the impact that the evaluation stop condition has on the performance of the generated models. Two different stop conditions are tested in the evolution of the topology of CNNs for CIFAR-10: (i) the training of the individuals for 10 epochs; and (ii) the training for a maximum GPU time of 10 minutes. Last, we apply the stopping criterion that reports the best results to the simultaneous evolution of the topology and learning strategy.

The first two columns of Table 5.3 compare the two stopping criteria on the optimisation of the topology (T): the evaluation based on the number of epochs vs. the evaluation for a maximum GPU training time. A perusal analysis of the results indicates that evaluating the CNNs up to a maximum of 10 minutes generates the best results, and therefore, without further training, the results that come out of evolution are better when evaluating the networks up to a maximum GPU time. The difference is statistically significant. Notwithstanding, we note that the evaluation during 10 epochs corresponds to varying training times, i.e., the networks can take much less or much more than 10 minutes to complete the 10 epochs. Thus, we re-evaluate the best networks generated with 10 epochs for 10 minutes. Each network is evaluated 5 times, and the validation accuracy increases from 83.21% to 85.33% (the test accuracy is not recomputed because it already has no time limit). The opposite, i.e., the re-evaluation of the networks generated with 10 minutes, for 10 epochs, decreases the average validation accuracy from 87.47% to 76.60%. This shows that when evaluating the CNNs considering the number of epochs the outcome of evolution are networks that train in few epochs, compared to networks that are capable to learn during more epochs. The results reported by both methods are competitive and consequently, we conduct the remaining experiments with the time as the stopping criterion because it enables longer learning sessions (in terms of the number of epochs), which make the evolution of learning policies more efficient. When we use time as the stopping criterion an average of 61.2 epochs are performed.

The results of the simultaneous evolution of the topology and learning strategy (T+L) are summarised in the last columns of Table 5.3. In terms of performance, there are no statistical differences between evolving the topology, or simultaneously the topology and learning strategy. The results of the evolution of the topology are as good as the simultaneous evolution of topology and learning because there is a first stage of defining a learning rate that is known to work well. This indicates that it is advantageous to evolve both. Despite the larger search space, given the

Table 5.4: Results of the evolution of topology and learning strategy (T+L) and topology, learning strategy and backward connections (T+L+B) of CNNs with Fast-DENSER.

	Time (T+L)	Time (T+L+B)	p-value
Evolution	88.29%	87.73%	0.271
Evolution (re-trained)	87.56%	86.88%	0.187
Test (no limit)	87.76% (88.90%)	87.17% (88.32%)	0.472 (0.430)

same time, the results are of the same magnitude. Therefore, there is no need to a-priori define learning parameters we are not sure to be the most appropriate ones.

5.2.4 Connection to Previous Layers

To investigate the effect of allowing the layers to have more than one input, we experiment with the number of levels back set to 5, i.e., we focus the simultaneous evolution of the topology, learning, and layer's connectivity (T+L+B). The features layers can connect to the 5 previous layers. We evaluate each individual for a maximum GPU time of 10 minutes. The results are summarised in Table 5.4, and compare the evolution of the topology, and learning strategy, to the optimisation of the topology, learning strategy, and connections.

The results show that the performance when considering the optimisation of the connections is slightly bellow to when the networks only connect to the previous layer. Despite the differences, they are not statistically significant. On the other hand, there is a statistical difference in the number of layers (p-value=0.0375): whilst the average number of layers of the CNNs where a given layer only connects to the previous layer is 14.9, when allowing multiple connections to be established to previous layers the average is 12.2. Therefore, despite the lack of performance gains, the evolved ANNs are shallower. A one-by-one analysis of the fittest networks shows that 4 out of the 10 best networks have at least a layer that receives more than one input connection.

The optimisation of the connections of each layer significantly increases the search space, and therefore we perform 50 more generations with this setup (totalling 200 generations per run). The additional experiments slightly increase the average evolutionary performance to 88.31%, and the test classification accuracy to 87.86% (and 88.92% when data augmentation is applied). The number of the best networks that have at least one of the layers connecting to multiple inputs also increases to 7. If we only consider the average performance of the networks that connect to multiple inputs, the evolutionary performance is 88.68%, and the test accuracy is 88.10% (and 89.25% when using data augmentation).

5.2.5 Generation of Fully-Trained Networks

The above experiments highlight that Fast-DENSER: generates CNNs that report the same performance as the ones obtained by DENSER; that the evaluation of the individuals for a maximum GPU training time enables the effective optimisation of the learning strategy; and, that the simultaneous optimisation of the topology, learning, strategy and connectivity of each layer, at least for CIFAR-10, is not advantageous. However, in all the experiments, the generated networks benefit from further training after evolution. To test the ability of Fast-DENSER to search for ready-to-deploy ANNs, we simultaneously optimise the topology and learning strategy of CNNs. For that purpose, evolution is allowed to continuously grant more training time to networks that may improve with further training. The results are reported in Table 5.5, and compare the performance when allowing the training time to grow, to when the training time is the same throughout evolution. The table presents different data than the above tables because

Table 5.5: Comparison of the results obtained by Fast-DENSER on the evolution of the topology and learning strategy with increasing training time (T+L+I) and with all the networks evaluated for the same maximum GPU training time (T+L). The results report the evolutionary results (evolution), and the test accuracy, and are measured with the generated networks right off evolution (test evolution), and when trained for longer (test no limit). The longer training is not applicable to when the training time is allowed to increase.

	Fast-DENSER (T+L+I)	Fast-DENSER (T+L)	p-value
Evolution (re-trained)	89.44%	87.56%	**0.038**
Test (evolution)	88.73% (89.56%)	86.91% (88.11%)	**0.032** (0.064)
Test (no limit)	n/a	87.76% (88.90%	0.308 (0.430)

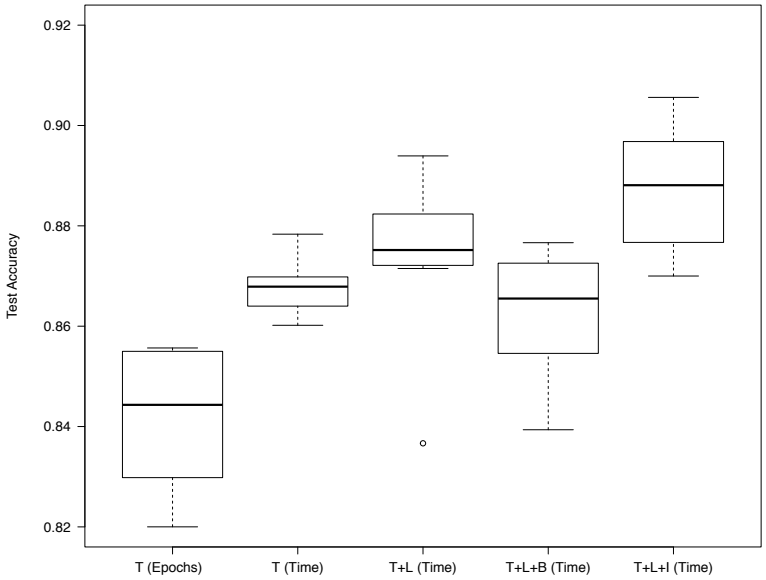
when enabling the training time to increase as needed, retraining the networks is not necessary. Therefore, we compare the results of Fast-DENSER with increasing training time to the previously reported by Fast-DENSER, with the networks re-trained 5 times. For the same reason, we focus on the analysis of the performance of the networks directly on the test set (test evolution).

The analysis of the results makes it clear that allowing the training time to grow continuously generates higher-performing CNNs than when all the individuals are granted the same GPU training time. The results are statistically significant, with large effect sizes. When the networks are trained for the same training time, after evolution, we require longer training sessions. This does not happen when the training time increases during evolution. The last row of Table 5.5 shows the results when the networks are re-trained until convergence (determined by early stopping). Even when re-trained, the results of evolution with the increasing training time are superior to the previous results. However, the difference is not statistically significant.

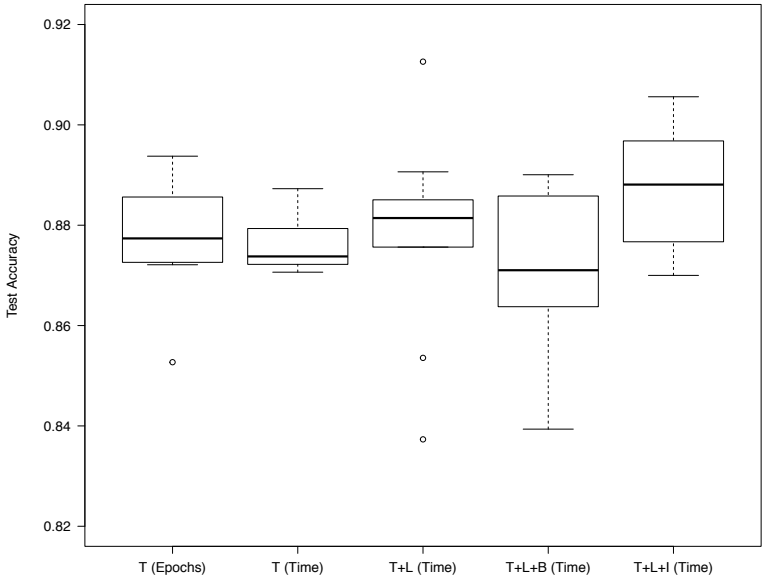
5.2.6 Discussion

Figure 5.2 depicts a box-plot of the test accuracies of all the experiments conducted with Fast-DENSER. The analysis of the results that are generated by evolution (without further training) proves that the performance on the evolution of the topology when the individuals are evaluated for 10 minutes – T (Time) – is superior to when the individuals are evaluated for 10 epochs – T (Epochs) – (see Figure 5.2a). This difference is mitigated when the individuals are further trained until convergence (see Figure 5.2b). In this last scenario, the difference between the two stopping criteria is not statistically significant. Nonetheless, because the evaluation for 10 minutes facilitates the evolution of learning policies, we carried out the remainder of the experiments considering the evaluation for a maximum of 10 minutes. This approach also has the advantage of enabling a fair comparison between individuals by restricting the access to computational resources equally.

The theory of Turner and Miller [254] is supported by the comparison between the evolution of topology, and topology and learning strategy: it is advantageous to evolve both the topology and learning simultaneously. In fact, the best performing CNN in terms of validation accuracy (among all experiments) was generated when evolving both topology and learning – this DANN has an average test accuracy of 91.26% (i.e., an error of 8.74%). The experiment where a layer can establish connections to multiple previous layers reported a performance that is lower than the remainder experiments. However, the comparison is not fair: we are searching in larger search spaces, but evolution is conducted during the same search time (i.e., 150 generations). To test this hypothesis, we perform 50 more generations for the evolution of the topology and learning, and topology, learning, and layer's connectivity. The new results report an average test classification performance of 88.22% (89.33%), and 87.86% (88.92%), respectively. These results



(a) Box-plot of the test accuracies when the networks are trained with the same learning strategy used during evolution, i.e., 10 epochs or 10 minutes, depending on the training stop criterion.



(b) Box-plot of the test accuracies when the networks are trained until convergence, with early stopping. The T+L+I results come directly from evolution, as the training time increases as required.

Figure 5.2: Box-plots of the test accuracies of the experiments conducted with Fast-DENSER for the evolution of the topology (T), topology and learning strategy (T+L), topology, learning strategy, and backward connections (T+L+B), and topology, learning strategy, and training time (T+L+I). The stop condition is within brackets.

are slightly superior to those reported in Table 5.4, and show that longer evolutionary cycles are likely to provide better results.

The comparison between the simultaneous optimisation of the topology and learning, and topology, learning and training time shows that evolving the training time is advantageous. It is demonstrated that Fast-DENSER can effectively generate networks that are ready to be deployed right-off evolution, i.e., there is no need for further training. This helps in the testing of the evolved training policy, as it is used until convergence. When we do not optimise the training time, the learning policies that are generated by Fast-DENSER, despite providing good results, may not be the most adequate ones when applied for longer training cycles. Most importantly, the above results are achieved without a major increase in the time required to search for the networks: from an average of 0.73 hours/generation to an average of 1.13 hours/generation, which is still fairly bellow the average of 10.83 hours/generation of DENSER.

With DENSER, we have reported for CIFAR-10 that the best-found CNN has an average error of 7.30% (with the same learning policy used during evolution), Stanley et. al. al. [171] also report an error of 7.30%, and Suganuma et al. [236] an error of 5.98% (with a learning rate policy different than the used during evolution). The error reported by the fittest CNN discovered by Fast-DENSER is comparable to those of DENSER and Stanley et. al. (8.6% vs. 7.30%). Most importantly, Fast-DENSER generates a competitive result within a fraction of the time (and computational resources) of the other methods. The result of Suganuma et al. is on a different learning rate policy, that still had to be tested and fine-tuned after the end of evolution. Therefore, the conducted experiments show that the results of Fast-DENSER are comparable to those obtained by DENSER, with a significant time gain. It is also proven that Fast-DENSER can effectively evolve the topology and learning strategy of CNNs. To sum up, it is demonstrated that Fast-DENSER can generate ready-to-deploy networks that require no further training nor tuning after the end of the evolutionary search procedure. In fact, the best performing network is generated when simultaneously evolving the topology, and learning strategy, with the training time increasing as required.

5.3 Evolution of Convolutional Neural Networks for the Detection of Gamma-Rays

To better understand the behaviour of Fast-DENSER, we apply it to search for CNNs in a problem of the physics domain. High-energy gamma-rays constitute one of the best probes to investigate extreme phenomena in the Universe, such as gamma-rays arising from fast-rotating neutron stars or supermassive black holes. The detection of this kind of astrophysical radiation, whose energies span from 10 GeV up to 100 TeV, can be done at lower energies by satellite bourne detectors. However, above a few hundred GeV, the flux becomes too small, and only ground-based experiments can measure, indirectly, gamma-rays. These experiments take advantage of the electromagnetic cascade that is produced by the interaction of gamma-rays with Earth's atmosphere to infer the direction and energy of the primary gamma-ray. When the energy of the gamma-ray is sufficiently high, and the detection of the secondary shower particles is done at high altitude, it is possible to survey large portions of the sky and be sensitive to transient phenomena. The observation of high-energy gamma-rays with ground-arrays, although effective, comes with a cost: one has to deal with the huge background of cosmic rays that bombard the Earth continuously. To select gamma-rays out of the hadronic background, one can explore the characteristics of the shower development. In comparison to pure electromagnetic showers, hadron induced showers produce high transverse momentum particles, which lead to the transverse broadening of the shower and the creation of clusters. Experimentally, the above

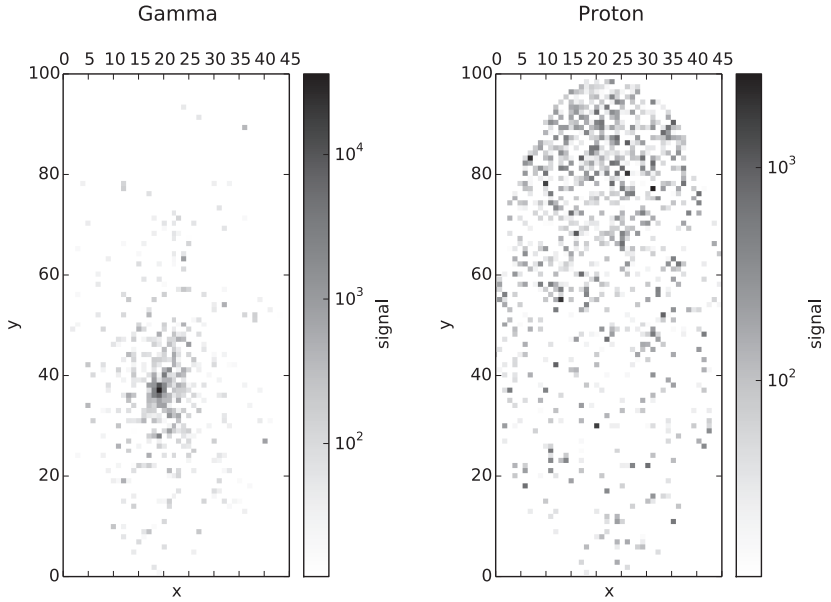


Figure 5.3: Example of the ground impact patterns of gamma (left) and proton (right) radiations.

features can be explored by measuring the steepness and bumpiness of the lateral distribution of particles at the ground with respect to the shower core position or by measuring the relative amount of signal (number of particles) at large distances from the shower core. However, the patterns of the secondary particles at the ground remain to be explored, although some studies have shown that this might have some gamma/hadron discrimination power.

In this section, we apply Fast-DENSER to the real-world problem of exploring the difference in the patterns at the ground, between gamma and proton-induced showers, resorting to the evolution of CNNs, which are appropriate to distinguish spatially-correlated data. The remainder of this section is organised as follows. Section 5.3.1 details the dataset; Section 5.3.2 defines the metric that is used to assess the fitness of the individuals; Section 5.3.3 enumerates the experimental parameters; and Section 5.3.4 and 5.3.5 analyse and discuss the results, respectively.

5.3.1 Dataset

The dataset is composed of gamma and proton (hadron) simulations, generated with CORSIKA [102], as described in [7]. The detectors have been simulated with the Geant4 toolkit [2], and the recorded signals have been used to reconstruct the main shower characteristics (energy, direction, primary) so that the sensitivity of this experiment to gamma-ray sources could be evaluated realistically. The detector unit is composed of small Water-Cherenkov Detectors (WCDs) (which maximise the trigger efficiency), and segmented resistive plate chambers (which have a good time resolution providing in this way a good shower geometry reconstruction). This detector concept was chosen to lower the energy threshold of previous experiments and bridge the energy gap between satellite-bourne and present ground-based experiments.

The main aim of the current experiments is to prove that the analysis of the pattern at the ground can be used to improve current gamma/hadron discrimination techniques. As such, we have opted to use only the information of the WCDs. Moreover, only showers reconstructed with energies between 1 and 1.7 TeV were used. Secondary shower particles that hit the WCD

will produce light that can be recorded by photomultipliers mounted sideways. As such, for each shower event, a WCD station provides the following information: its position (x and y coordinates of the centre of the WCD), and the recorded signal (approximately proportional to the number of particles in it). It is only this information that shall be used to distinguish gamma from hadron induced showers.

The gamma-ray detector is composed of 3m $\times$ 1.5m individual stations that occupy a full circle array with a radius of approximately 80m. Therefore, each event is a matrix with the recorded signal by each of the cells. The dataset is composed of 79856 instances (shower events) of two disjoint classes: gamma or proton. Each instance is a 45 $\times$ 100 matrix. The positions of the matrix where there are no cells (because the grid is circular and the matrix is rectangular) are set to 0. An example of the impact patterns of gamma and proton radiations is depicted in Figure 5.3. The main difference in the ground impact patterns between gamma and proton radiations is that the dispersion of the signal of the gamma radiations tends to be more compact than the dispersion of the signal of the protons.

5.3.2 Fitness Function

To assess the fitness of each individual, we compute the True Positive Rate (TPR) and False Positive Rate (FPR) to build the Receiver Operating Characteristic (ROC) curve. We consider the positive class as the instances classified as protons. The fitness of each individual of the population (ind) is computed as:

$$\text{fitness(ind)} = max\left(\frac{\text{TPR(x)}}{\sqrt{\text{FPR(x)}}}\right),$$

where TPR(x) and FPR(x) are the TPR and FPR of the model at the point x of the FPR threshold, respectively. Since we are maximising, the models assigned with higher fitness values are those with a higher response of TPR for each FPR point, with emphasis to points with low FPR threshold.

The choice of the fitness function is related to the fact that the observation of astrophysical gamma-ray sources relies on the identification of gamma-rays, which are immersed in a huge cosmic ray (hadronic) background. As the background is continuous and isotropic, while gamma-rays are localised in space, an excess of events coming from the gamma-rays sky region should be visible if one acquires during enough time. To state that there is an excess of events, the number of gamma-ray events has to be higher than the fluctuations of the background. As events are considered independent, the fluctuations follow the Poisson distribution, i.e., the square root of the number of events measured. By taking the number of background events much higher than the number of signal events, one can neglect the signal contribution in the square root, which finally leads to the chosen fitness equation.

5.3.3 Experimental Setup

The evolutionary engine parameters are the same as the detailed for the object detection experiments (check Table 5.1, in the previous section). We perform 30 evolutionary runs. No data augmentation is applied, and the dataset is pre-processed by feature-wise centring and standard deviation normalisation.

To search for CNNs, we use the outer level structure [(features, 1, 10), (classification, 1, 10), (softmax, 1, 1), (learning, 1, 1)], and Grammar 5.2. The only change to the grammar is that the batch size can vary between 50 and 300, instead of between 50 and 500.

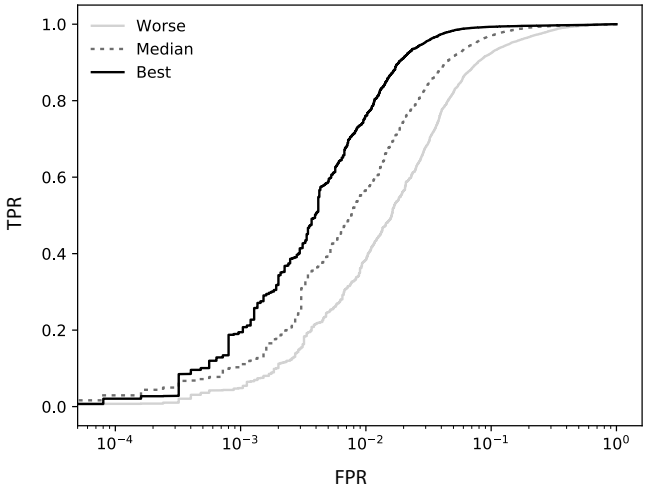


Figure 5.4: ROC curves of the worse, median, and best fittest individuals discovered by Fast-DENSER for the gamma/proton discrimination problem. A log scale is used.

5.3.4 Experimental Results

The analysis of the experimental results focuses on the performance of the evolved networks, measured in the test set. The fitness function described in Section 5.3.2 is strictly related to the ROC curve, and thus in Figure 5.4 we depict the ROC curves of the fittest networks that achieve the worse, median, and best fitness values. The fittest networks are selected according to their fitness value on the validation set.

The curve of the individual with the median fitness value is close to the best individual, suggesting that the results are consistent, i.e., a high performing network is not discovered by chance but is instead an outcome of the evolutionary search of Fast-DENSER. The minimum, average, median, and maximum fitness values are 4.07, 5.78, 5.89, and 8.72, respectively.

Despite the importance of the analysis of the overall results, the ultimate goal is to select a model that is capable of addressing the problem we have at hand, in this case, a CNN which is capable of classifying between gamma and proton radiations. We select the best performing network according to the fitness on the validation set. This way, the choice is not biased. Later, we compare the results based on a different, disjoint, set of instances.

The topology of the best performing network is shown in Figure 5.5. The CNN is composed of 8 hidden-layers: 3 convolutional, 2 average pooling, 2 fully-connected, and 1 dropout. As typical, when the networks are designed by human practitioners, the convolutional layers tend to be followed by pooling layers. On the other hand, the network also shows the previously enumerated aspects that make it novel (check Section 4.2.3), and thus evolution helps to generate novel and out-of-the-box topologies that human-designers would hardly think of. The fittest CNN is trained using the Adam [125] learning algorithm with a learning rate of 0.0001, a beta 1 of 0.86486, a beta 2 of 0.68028; the learning rate decay is 0.00068, and the batch size is 117. We compare the fittest CNN with the performance of other approaches in the next section.

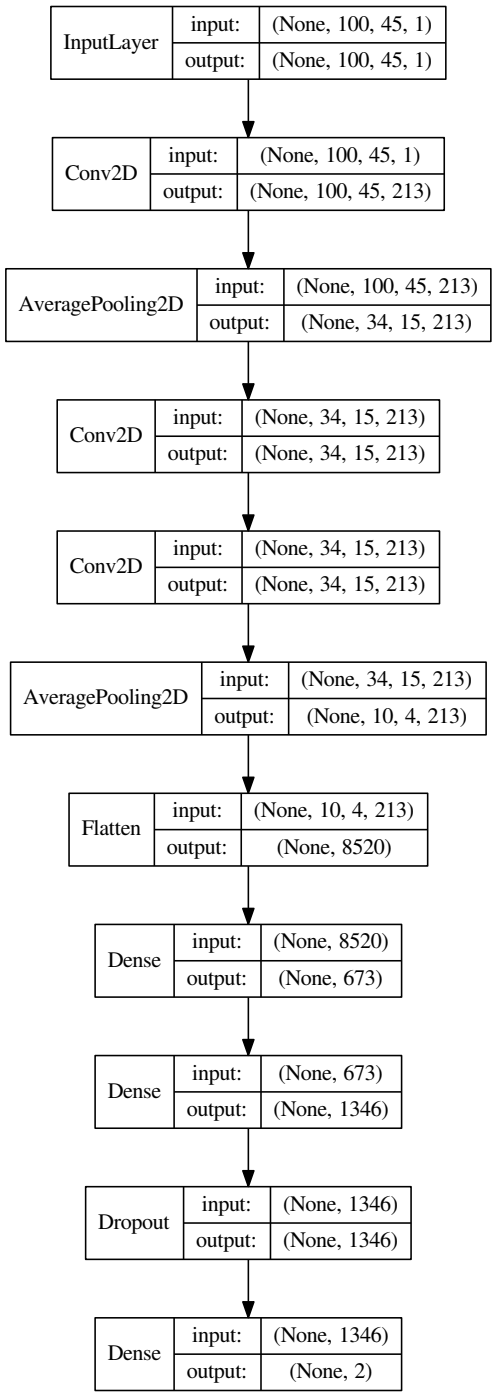


Figure 5.5: Topology of the fittest CNN discovered by Fast-DENSER for the gamma/proton discrimination problem.

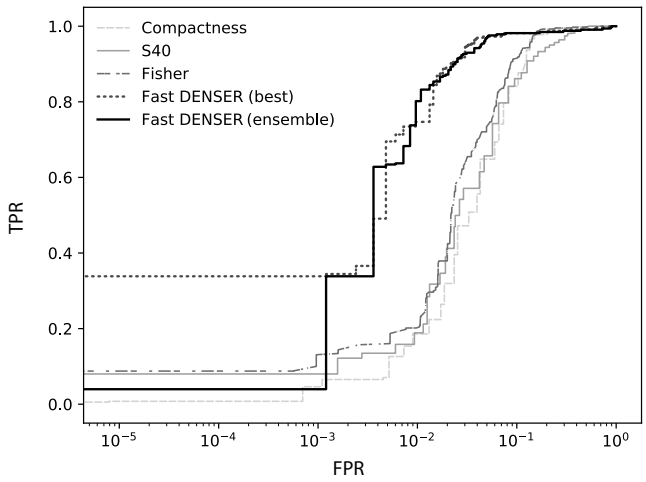


Figure 5.6: Comparison between the CNNs discovered by Fast-DENSER for the gamma/proton discrimination problem (best and ensemble) and other ML methods: Compactness, S40, and Fisher. A log scale is used.

5.3.5 Discussion

Assis et al. [7] demonstrated that this detector concept can perform the usual gamma/hadron discrimination. Two discrimination variables, based only on the WCD information, were built: Compactness and S40. The former explores the information in the shower Lateral Distribution Function (LDF), in particular, the steepness and bumpiness. This is done comparing the shower event LDF to a reference gamma LDF, built from the average of many gamma showers. The variable S40 is used to identify particle clusters away from the shower core. This is achieved by computing, for stations above 40 meters away from the reconstructed shower core, the ratio between the signal of the hottest station and the total signal. Although there is some level of correlation between the two variables, they carry independent information. To further explore the combined discrimination power of Compactness and S40, linear discriminant analysis is used, henceforth referred to as Fisher. It is worth to mention that although the above quantities explore the shower ground pattern, these classical statistics cannot fully extract all the information due to the stochastic nature of the shower, forcing the use of non-parametric cuts.

Figure 5.6 compares the ROC curves of the fittest CNN discovered by Fast-DENSER, with the ROC curves of the classic statistics (Compactness, S40, and Fisher). To better analyse the evolutionary results, we perform statistical tests to confirm whether or not the data follows a random distribution, i.e., to investigate the consistency of the results. In particular, we use the chi-square test. With a significant level of $\alpha = 0.05$, the test reveals that the data does not follow a random distribution, and thus it highlights that the high performances tend to be consistent.

In addition to the fittest network, we also investigate the performance of the ensemble formed by the best networks (one from each run). The generated networks are diverse in topology, and consequently, they may more adequate to some patterns of inputs over others, i.e., while some of the networks can fail to predict a specific instance others can predict it correctly. The ensemble is formed by 16 voters, which are the CNNs that report a performance above the average performance of the 30 evolutionary runs (this choice is based on the validation performance,

and thus not biased). The predicted class is computed based on the maximum of the average confidences. For all methods, we measure the performance on the same partition of the data, and thus the results are comparable. The data is the same as in [7], which is distinct from the one used in the evolutionary experiments, but it is generated from the same source. It consists of 1158 instances: 328 gamma, and 830 protons. The dataset is unbalanced and follows the distribution expected in nature.

The analysis of the plot shows that the CNNs generated by Fast-DENSER surpass the results obtained by the classic statistics. Further, the average fitness of the evolved CNNs, the fitness of the fittest CNN, and the fitness of the ensemble, Compactness, S40, and Fisher are of approximately 7.34, 10.01, 10.45, 3.13, 3.35, and 4.22, respectively. Comparing to the best result of the classic statistics, the average of all the generated CNNs, the fittest CNN, and the ensemble, improve the previous result by a factor of 1.71, 2.37, and 2.48, respectively. Recall that the results of evolution do not follow a random distribution (confirmed by statistics), and thus the comparison to the previous (deterministic) state-of-the-art results can be established based on the average performance of the evolutionary results.

5.4 Summary

This section introduces Fast-DENSER: an extension to DENSER that seeks to speedup DENSER by replacing the evolutionary engine by an ES, initialising search from shallow topologies, and changing the evaluation stopping criteria, i.e., the individuals are evaluated up to a maximum GPU training time. In addition to introducing changes that speedup search, the representation is expanded to allow an unconstrained search of the topologies and layer connections. The new method is also able to generate ready-to-deploy ANNs, i.e., the networks generated by Fast-DENSER require no further training after the end of the evolutionary search. To sum up, Fast-DENSER is a complete general-purpose NE method that, for a specific task, can search for topologies, learning strategies (and any other parameterisations) for networks of unrestricted structure. The generated models can be directly used.

A wide and comprehensive set of experiments are conducted to investigate the comparative performance of Fast-DENSER and DENSER. The results show that Fast-DENSER can generate ANNs that have the same overall performance of those generated by DENSER but in a fraction of the time, with a speedup of approximately 20x. DENSER takes, on average, 1083 hours per run, and Fast-DENSER only 55 hours per run. The results also demonstrate that Fast-DENSER can simultaneously optimise the topology and learning of ANNs, and even the topology, learning, and connectivity of each layer. This way, we do not need to conduct the search in various separate steps (manually find an adequate learning strategy and only then apply NE to the topology), but instead, we can let evolution search for all the required parameters of the network.

To answer the question of whether or not Fast-DENSER is able to generate fully-trained networks, we compare the performance of the ANNs that are obtained with and without applying the mutation operator that incrementally increases the training time. The analysis of the results makes it clear that when we allow the training time to grow continuously, the performance even surpasses the one obtained when the networks are further trained after evolution.

Finally, we apply Fast-DENSER to a real-world problem of the physics domain: the gamma/hadron discrimination problem. The objective is to distinguish between gamma and proton radiations based on their ground impact patterns, and only considering the impact energies at the WCDs. The models created by Fast-DENSER surpass the performance of the previous approaches. The performance increases by a factor of up to 2.48. This improvement in performance translates into investment savings, as a considerably smaller grid of sensors can be used.

The results, show the effectiveness of Fast-DENSER, but also expose one of its weaknesses: the search is always conducted from scratch, and when one requires the search for ANNs for other similar tasks there is no way to incorporate previously acquired knowledge. In the next chapter, we address this problem and incorporate incremental development in Fast-DENSER.

Chapter 6

Incremental Development

One of the main limitations of NE lies in the fact that the majority of the methods only address a specific problem, i.e., the ANNs are evolved for one task, and when there is the need to solve a new problem the entire search procedure is re-started from scratch. Therefore, the methods do not take advantage of any of the information available from addressing previous similar tasks. In addition, NE approaches tend to evolve large populations of individuals that are continuously optimised throughout an usually large number of generations. The evaluation of a single ANN is time-consuming because it often requires the training of the network with a defined (or evolved) learning strategy. Consequently, the search for ANNs resorting to NE tends to be slow. This problem is even more striking when optimising DANNs.

This chapter extends Fast-DENSER to incremental development, i.e., we transfer and re-use knowledge acquired when optimising DANNs to previous problems, and cumulative apply it to learn new classification tasks. To clarify, the proposed incremental development strategy re-uses sets of evolutionary units that were discovered for addressing previous tasks, and thus the approach is incremental in the sense that the optimisation for the secondary tasks does not start from the scratch, but can instead build on top of the previously best-found network topologies. This concept contrasts with the incremental / transfer learning definition in ML, and thus the concepts should not be associated. The results show that the proposed methodology is able to incrementally evolve the network topology for solving different tasks, and given the same search time, the results are statistically superior to those reported by the canonical version of Fast-DENSER. In addition, we show that incremental development aids in the generation of DANNs that generalise to yet unseen problems.

The remainder of the chapter is organised as follows. Section 6.1 introduces the extension of Fast-DENSER to incremental development; Section 6.2 presents the experimental setup and results; and Section 6.3 draws conclusions.

6.1 Incremental Development of Deep Neural Networks

The experiments presented in previous chapters have shown that Fast-DENSER, given the same computational time budget, can obtain results that are superior to those reported by DENSER. The results were achieved without taking advantage of any of the knowledge acquired when solving other problems. In this chapter, we investigate the impact of building the networks incrementally, i.e., we take into account the DANNs that are generated for solving previous related problems, speeding up evolution, and possibly finding better solutions.

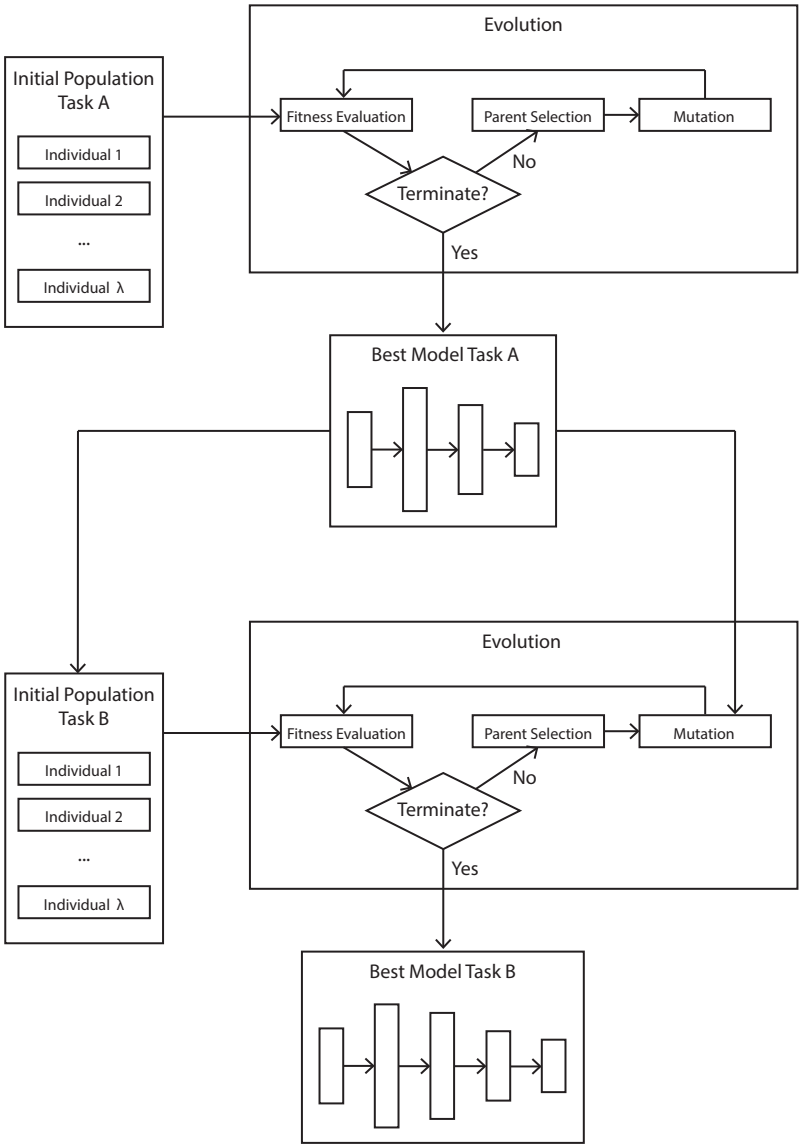


Figure 6.1: Flow-chart of incremental development applied to Fast-DENSER.

Figure 6.1 depicts the flow-chart that illustrates the extension of Fast-DENSER to incremental development. It shows how the method proceeds, to address two different generic tasks A and B. For the first task, task A, the method works similarly to Fast-DENSER: an initial population is randomly created and evolves until the stop criterion is met. The difference occurs when we solve a new task, given that we have information on a prior one. For task B, the creation of the initial population takes into account the best model found for a previous task (in this case task A). During evolution, past knowledge can also be incorporated. This rationale is generalised for more than two tasks, i.e., if we later address a task C, we use the knowledge obtained when addressing tasks A and B; the initial population for task C is formed considering the best solution for task

B (which derives from task A), and at any given point during evolution past knowledge from addressing tasks A and B can be incorporated. Next, we will discuss how the prior knowledge is introduced in the initial population, and during evolution.

The initial population is formed by individuals that are entirely generated at random or that use sets of evolutionary units from past models. The evolutionary units are transferred, taking into account the macrostructure. For example, considering the macrostructure introduced above for CNNs, [(features, 1, 10), (classification, 1, 2), (softmax, 1, 1), (learning, 1, 1)], the initial population can contain individuals that (i) have all the layers comprising the feature extraction, and generate the classification layers at random; (ii) generate at random the feature extraction layers, and copy the layers that perform classification from previous models; (iii) copy only the learning evolutionary unit, and generate all the remaining ones at random; (iv) generate all evolutionary units at random, not using any previous knowledge; or (v) any other possible combination. It is important to mention that this incremental development approach only focuses on the evolutionary units, and consequently the weights are not transferred from previous models. At most, we allow the learning strategy evolutionary unit to be ported.

The models generated for solving each of the previously addressed problems are important during evolution too. The mutations in Fast-DENSER are tailored to manipulate DANNs: they enable the addition, removal, and/or duplication of any evolutionary unit, the perturbation of the integer and/or float values, and the manipulation of the training time. The duplication mutation replicates a given evolutionary unit by reference, and thus, any mutation that later affects this evolutionary unit changes all of its copies. In the incremental development version of Fast-DENSER, the duplication can copy evolutionary units either from the individual or from any of the best models that were generated for solving previous tasks.

The individuals are evaluated only on the new problem. Therefore, up to the moment, this method is incremental in the sense that the DANNs for solving new and unseen problems do not start evolution from scratch, but can instead build on top of the best-generated network topologies for previous problems. Incremental development does not mean that, by the end of evolution, the generated models can solve multiple tasks. However, it is expected that the models that are built considering previous knowledge generalise better than those that are always evolved from a random population. That is, we expect the models created by incremental development to perform well in other tasks when re-trained.

6.2 Experimentation

To compare the incremental and non-incremental implementations of Fast-DENSER, we consider four computer vision datasets: MNIST, SVHN, Fashion-MNIST, and CIFAR-10. In particular, we conduct experiments for the following setups: (i) MNIST; (ii) SVHN; (iii) Fashion-MNIST; (iv) CIFAR-10; (v) MNIST $\rightarrow$ SVHN; (vi) MNIST $\rightarrow$ SVHN $\rightarrow$ Fashion-MNIST; (vii) MNIST $\rightarrow$ SVHN $\rightarrow$ CIFAR-10. The symbol $\rightarrow$ denotes the incremental build of the model from one task to the next. The setups are chosen according to the relatedness and expected difficulty of the tasks: the MNIST and SVHN datasets are composed of digits, and then transferred to two different domains, Fashion-MNIST, and CIFAR-10. The parameters required for the conducted experiments are detailed in Section 6.2.1. The experimental results are divided into three sections. First, in Section 6.2.2, we analyse the evolutionary performance when evolving DANNs for MNIST, SVHN, Fashion-MNIST, and CIFAR-10 with and without incremental development. Second, in Section 6.2.3, we investigate the incremental development of the topologies. Third, in Section 6.2.4, we analyse the generalisation ability of the different models. The experimental results are discussed in Section 6.2.5.

Table 6.1: Average performance of the DANNs optimised by Fast-DENSER with and without incremental development. The results are averages of 10 independent runs. Bold marks the highest average performance value.

Dataset	Evolutionary Accuracy	Test Accuracy
MNIST	98.86 ± 0.465	98.80 ± 0.298
SVHN	93.28 ± 0.863	93.31 ± 0.955
MNIST → SVHN	**94.01 ± 0.891**	**94.04 ± 0.887**
Fashion	92.42 ± 1.224	91.41 ± 1.049
MNIST → SVHN → Fashion	**93.92 ± 0.930**	**92.96 ± 0.742**
CIFAR-10	87.18 ± 1.242	86.19 ± 1.672
MNIST → SVHN → CIFAR-10	**89.06 ± 1.488**	**88.19 ± 1.669**

6.2.1 Experimental Setup

The parameterisation of the Fast-DENSER evolutionary engine is the same as in previous experiments (Table 5.1). The number of generations is different for each of the datasets. In particular, we perform 20, 30, 50, and 100 generations for the MNIST, Fashion-MNIST, SVHN, and CIFAR-10 datasets, respectively. The number of generations for each dataset was set empirically based on previous experiments, and according to how challenging each problem is expected to be.

The datasets have different shapes: MNIST and Fashion-MNIST are 28×28×1, and SVHN and CIFAR-10 are 32×32×3. To facilitate the application of the optimised DANNs to all datasets, we reshape the MNIST and Fashion-MNIST to 32×32×3. The image width and height are resized using the nearest neighbour method, and to pass from one to three channels we replicate the single-channel three times. The same data augmentation strategy is applied to all datasets: padding, random cropping, horizontal flipping, and re-scaling to $[0, 1]$. We do not subtract the mean image nor normalise.

The networks are trained for an initial maximum GPU time of 10 minutes, and thus it is important to mention that we perform each evolutionary run in a GeForce GTX 1080 Ti GPU. For the experiments conducted in this chapter we use Grammar 5.2, and the macrostructure: [(features, 1, 30), (classification, 1, 10), (softmax, 1, 1), (learning, 1, 1)].

6.2.2 Experimental Results: Incremental Development

To start, we compare the performance of the DANNs generated by Fast-DENSER with and without incremental development. The results are summarised in Table 6.1. We report the evolutionary accuracy (i.e., fitness), and the test accuracy (i.e., the accuracy of the models on an unseen partition of the datasets). The results are averages of 10 independent runs. The first conclusion is that, given the same computational time (number of generations), the results reported by the incremental development are always superior to when evolution starts from scratch. The performance of MNIST → SVHN is superior to the performance of SVHN, the performance of MNIST → SVHN → Fashion is superior to the performance of Fashion, and the performance of MNIST → SVHN → CIFAR-10 is superior to the performance of CIFAR-10.

To better assess the differences between the fittest DANNs generated with and without incremental development, we use statistical tests. To check if the samples follow a Normal Distribution, we use the Kolmogorov-Smirnov and Shapiro-Wilk tests, with $\alpha = 0.05$. The tests reveal that the data does not follow a Normal distribution, and thus, the non-parametric Mann-Whitney U test ($\alpha = 0.05$) is used to perform the comparisons between the setups. The statistical tests show that the results of MNIST → SVHN → Fashion, and MNIST → SVHN → CIFAR-10

Table 6.2: Accuracy of the best performing DANN optimised by Fast-DENSER with and without incremental development. Bold marks the highest performance value.

Dataset	Evolutionary Accuracy	Test Accuracy
MNIST	99.46	99.12
SVHN	94.20	93.88
MNIST → SVHN	**94.80**	**94.14**
Fashion	93.91	92.92
MNIST → SVHN → Fashion	**94.80**	**93.92**
CIFAR-10	88.74	88.14
MNIST → SVHN → CIFAR-10	**91.06**	**89.79**

are statistically superior (in evolution and test) to Fashion (evolutionary p-value=0.00736, test p-value=0.00278), and CIFAR-10 (evolutionary p-value=0.00804, test p-value=0.01552), respectively. The effect size is large for all the statistically significant comparisons ($r > 0.5$). The difference between MNIST → SVHN and SVHN is not statistically significant (evolutionary p-value=0.05876, test p-value=0.0536). With only 20 generations the MNIST setup is the one that attains the highest average accuracy results. This indicates that MNIST is an easy-to-solve problem, and consequently, no much knowledge is acquired from addressing it. This is a well-known fact: a simple fully-connected network can attain good performances in this dataset.

The above results show that incremental development, given the same number of generations, designs DANNs that outperform those generated without incremental development. On the other hand, what if we only conduct evolution for a smaller amount of generations so that the cumulative number of generations is inferior to when evolution starts from scratch? The cumulative number of generations is the sum of the number of generations of each incremental step. For example, for the MNIST → SVHN, the cumulative number of generations is 70 (20 when evolving for MNIST + 50 when evolving for SHVN from MNIST). Therefore, in this scenario, and to match the cumulative number of generations to the number of generations provided to the target task when evolution starts from scratch, we consider 30 generations for the MNIST → SVHN (i.e., 50-20) and also 30 generations to the MNIST → SVHN → CIFAR-10 (i.e., 100-50-20) setups. For the MNIST → SVHN → Fashion setup, we evaluate the performance on the initial population since the cumulative number of generations of the MNIST → SVHN setup is already superior to the number of generations provided when tackling the generation of CNNs for SVHN from scratch. The average evolutionary performance of the fittest networks of each run slightly decreases to 93.69 ± 0.912, 92.91 ± 1.150, and 87.13 ± 2.225, respectively for the MNIST → SVHN, MNIST → SVHN → Fashion, and MNIST → SVHN → CIFAR-10 setups. With these results, there is no statistical difference for any of the setups, i.e., with incremental development, given a cumulative search time that equals the search time from scratch, we are able to generate DANNs that report the same performance as those optimised without incremental development for more generations. In other words, the use of previous knowledge speeds up evolution.

In addition to analysing the average performance, we also look at the performance of the fittest DANN. This is important considering that in a real-world scenario, by the end of evolution, what interests the user is the best performing model, which is the one to be potentially deployed. To avoid an unbiased choice of the best model, the decision is taken only with regard to the evolutionary performance. The results are reported in Table 6.2, and once again show that the best performances are obtained by incremental development. The most striking result is the one of CIFAR-10, where the difference introduced by incremental development is the highest.

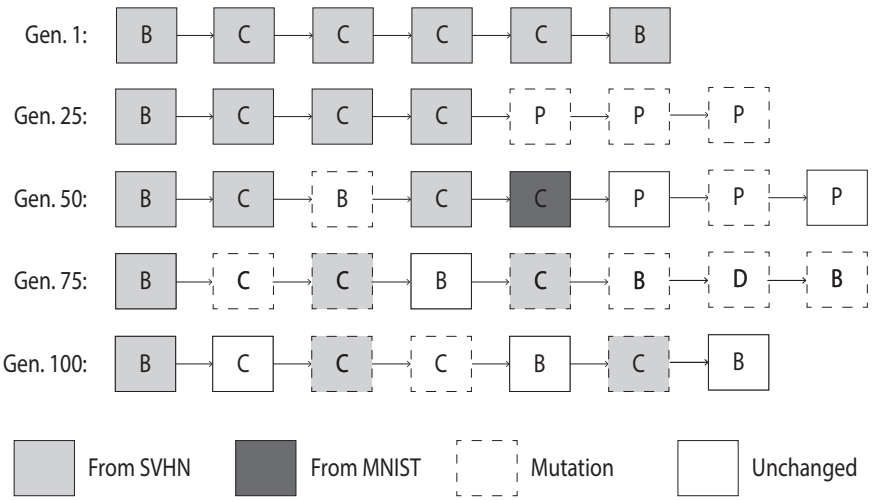


Figure 6.2: Overview of the evolution on the incremental development setup MNIST → SVHN → CIFAR-10. We provide a snapshot of the feature-layers of the best individual on the 1st, 25th, 50th, 75th, and 100th generations. To enhance readability, we focus on the feature extraction layers: Convolution (C), Pooling (P), Batch-Normalisation (B), and Dropout (D).

6.2.3 Experimental Results: Topology Analysis

To analyse how evolution affects the architecture of the generated CNNs, we inspect the topology of the best networks as evolution proceeds. Figure 6.2 shows the evolution of the structure of the networks on the setup MNIST → SVHN → CIFAR-10. For the purpose of the example, we select the setup where more generations were performed. To avoid a biased selection over the worst or best result, we focus on the run that generates the DANN with the median fitness value. The objective of the figure is to illustrate the exploration of knowledge incorporation, and thus the parameters of the layers are omitted.

The figure makes it evident that the amount of layers that come from previously addressed tasks diminishes as evolution proceeds. That is the expected behaviour: in the initial generation the fittest DANN re-uses all layers from the best network generated to address the SVHN. Throughout generations, these layers are adapted to tackle CIFAR-10 (e.g., convolutional in generation 75). During evolution, new layers are also randomly created (e.g., batch-normalisation in generation 50), and others removed (e.g., dropout in generation 100). Similarly to the non-incremental approach, new random layers can be added. Additionally, in the incremental development strategy, we can also add layers that come from the previously solved tasks (e.g., convolutional layer that is transferred from the MNIST in generation 50).

The snapshots show that incremental development can generate better results based on the re-use of evolutionary units that aid in solving previous problems. The evolutionary units are not only incorporated in the generation of the initial population, but also during evolution. We inspect the evolutionary results of other setups, and the conclusions are in line with the reported.

6.2.4 Experimental Results: Generalisation of the Models

To study the generalisation ability of the generated models, we measure their performance on all the considered datasets. For example, we take the best-performing CNNs for the MNIST

Table 6.3: Performance of the DANNs evolved by Fast-DENSER (with and without incremental development) when applied to other datasets. The results are averages of 10 independent runs, where each DANN is trained 5 times. The setups are the rows of the table, and the datasets are columns of the table.

	MNIST	SVHN	Fashion	CIFAR-10
MNIST	98.80±0.298	71.31±29.60	90.17±1.842	63.63±23.29
SVHN	96.87±5.426	93.31±0.955	91.60±1.289	78.49±7.899
MNIST→SVHN	98.93±0.266	94.04±0.887	91.83±1.312	82.58±2.414
Fashion	92.73±16.75	89.16±3.551	91.41±1.049	77.32±4.893
MNIST→SVHN→Fashion	98.89±0.273	**92.48±2.167**	92.96±0.742	**83.47±2.294**
CIFAR-10	99.06±0.039	90.18±9.282	92.91±0.479	86.19±1.672
MNIST→SVHN→CIFAR-10	**99.11±0.071**	90.08±5.924	**93.16±0.333**	88.19±1.669

dataset and apply them to the SVHN, Fashion, and CIFAR-10 datasets. No further evolutionary optimisation is performed. The networks are re-trained on the target datasets with the same topology and learning strategy that is optimised for the source task. Table 6.3 summarises the test results for all the setups. The values in bold mark the best generalisation performance, i.e., for each dataset (column) we mark the setup (row) that attains the best performance but where the networks were not trained on that dataset, e.g., for the SVHN dataset (second column), except for the setups that target this dataset (SVHN, and MNIST→SVHH), the setup that attains the best performance in SVHN is MNIST→SVHH→Fashion.

The analysis of the results shows that incremental development always generates better results, even for tasks that have not been addressed previously. To better understand the differences, we perform statistical analysis, and compare the performances reported by the non-incremental and incremental approaches. Therefore, we compare the SVHN and MNIST→SVHN setups on the MNIST, Fashion, and CIFAR-10 datasets, and we do similarly with the remaining pairs: Fashion vs. MNIST→SVHN→Fashion, and CIFAR-10 vs. MNIST→SVHN→CIFAR-10. The same conditions of the above statistical comparison are applied. The statistical tests reveal that there are only significant differences between the Fashion and MNIST→SVHN→Fashion setups, with p-values of 0.02574, and 0.01732, respectively for the SVHN and CIFAR-10 datasets (the effect size is large). The direct comparison to the dataset used for evolution (in this case Fashion) was performed above and revealed a statistical significance in favour of incremental development for the setups that include two incremental development steps.

If we order the datasets by difficulty, given by the non-incremental test performance on each dataset, we have MNIST, SVHN, Fashion, and CIFAR-10, where the leftmost is the easiest one, and the rightmost is the most challenging one. From these results, we hypothesise that superior generalisation performances are obtained by incremental development, when passing from more simple to more challenging datasets. That is the reason why the difference in the CIFAR-10 vs. MNIST→SVHN→CIFAR-10 setups is too small to be statistically significant: CIFAR-10 is more challenging to solve than the remaining datasets, and therefore, as already noticed in Chapter 4, the DANNs generated for addressing CIFAR-10 tend to be able to solve other easier problems efficiently. The remarkable aspect of incremental development is when a DANN optimised for Fashion is able to get better results on CIFAR-10, compared to when the DANNs for Fashion are not evolved incrementally.

6.2.5 Discussion

The results of this section compare in terms of performance, topology, and generalisation ability the search conducted by non-incremental and incremental development. The evolutionary results show that, given the same search time, the DANNs obtained by incremental development statistically outperform the non-incremental counterparts. Additionally, the incremental strategy speeds up evolution and, given the same cumulative search time, reports results that match the non-incremental performances.

The speedup in evolution is facilitated by the topological warm-start of incremental development, and the possibility to incorporate knowledge from previous tasks by mutation as generations proceed. We show an example of this by representing several snapshots of the best individual of an evolutionary run across generations. In particular, in the first generation of the selected run of the MNIST→SVHN→CIFAR-10 setup, the best individual replicates all the layers from the MNIST→SVHN setup, which are continuously modified and adapted to CIFAR-10. During evolution, the parameters of the layers that are copied from the previous setup are changed, new layers (random, and from previous setups) are added, and others removed. This is the behaviour we expect of incremental development.

Finally, we analyse the generalisation ability of the generated DANNs. We do not perform further evolution, i.e., the same topology is re-trained on the remaining target datasets with the learning strategy of the source task. The results show that, on average, the incremental development results are superior to the non-incremental counterpart. Moreover, the differences among results are statistically significant when the generated DANNs are applied to a more difficult task. This indicates that incremental development helps in generating increasingly more complex networks that can tackle increasingly more challenging tasks, and that there are no major differences when performing the opposite, i.e., when transferring knowledge from difficult to simpler tasks.

6.3 Summary

Motivated by the difficulty and time required to design DANNs, we investigate how to incorporate past knowledge to aid evolution. In particular, we extend Fast-DENSER to take advantage of the evolutionary units acquired when optimising DANNs for previous tasks. This novel topology incremental developmental approach enables the incorporation of knowledge from any of the previously addressed tasks in any stage of evolution: both during the generation of the initial population, and by the application of mutations, as the generations proceed.

The results show that incremental development improves the search performed by Fast-DENSER enabling it to obtain statistically superior results. Additionally, incremental development speeds up evolution, and is able to obtain the same results as non-incremental evolution given the same cumulative search time, i.e., fewer generations are required to conduct the search in the target dataset. Furthermore, the DANNs obtained by the end of evolution generalise better when we use incremental development in the search: the networks designed for easy problems perform better in more challenging and yet unseen problems.

Chapter 7

Conclusions and Future Work

The research hypothesis of this Thesis is that GGP methods can be applied to develop a general-purpose, flexible, and efficient framework for the automatic optimisation of ANNs that can be built incrementally. To test the research hypothesis, we formulate four research questions:

RQ1: Which GGP method is effective in NE?

RQ2: Which representation scheme is adequate to a flexible encoding of ANNs?

RQ3: How can the ANNs be evaluated and evolved efficiently?

RQ4: Can the ANNs be developed incrementally?

The answer to these questions culminated in the main contribution of the Thesis: the Fast-DENSER framework, which was proposed, developed and tested.

The Thesis starts with the introduction of ML concepts and approaches. We focus on ANNs due to the difficulty to define the topology and the hyper-parameters. Several AutoML methods try to overcome this challenge. However, our interest is on EC approaches since they are flexible and scalable. The field that applies EC to the optimisation of ANNs is known as NE. The NE approaches are often divided depending on the target of the optimisation. Instead, we group them in small-scale, and large-scale methods. One of the main limitations of NE is that the approaches search for solutions to a single task, and to address a new one the search is re-started from scratch. To avoid this, we investigate transfer, multi-task, and incremental learning strategies. The survey of these themes is carried out in Chapter 2 and enables the identification of the concepts that are important to develop a general-purpose, flexible, and efficient NE framework to incrementally build DANNs.

To answer RQ1, in Chapter 3, we compare different GGP methods. In particular, we compare GE to SGE on the optimisation of the topology and connection weights of ANNs. The results show that, given the same search time, the performance of SGE is statistically superior to the performance of GE. There is however one main limitation: SGE cannot evolve multi-layered ANNs. The grammars are fixed, and thus it is impossible to keep track of the placement of the neurons in the network. To mitigate this issue, we propose DSGE – an extension to SGE that only encodes the genes that are required in the genotype to phenotype mapping; the genotype is expanded incrementally as more genes are needed. DSGE statistically outperforms SGE on the evolution of single-layered ANNs. To optimise multi-layered ANNs, in addition to DSGE, we introduce dynamic production rules. The dynamic production rules facilitate the modification and creation of new grammatical expansion possibilities in run-time, and thus the grammar

becomes a property of the individual. The results show that DSGE can generate multi-layered ANNs and that the performance of the multi-layered is superior to the performance of the single-layered ANNs.

The grammar-based methods are also applied to the optimisation of learning weight policies. We compare GE, DSGE, tree-based GP, and NEAT on the evolution of CPPFs. The results show that tree-based GP and NEAT statistically outperform GE and DSGE. There are no significant differences between tree-based GP and NEAT, and DSGE tends to report results superior to GE. These conclusions were expected because of the known limitation of GGP methods to directly optimise real-values. Therefore, the answer to RQ1 is that, from the explored approaches, DSGE is the most adequate GGP for NE. It is effective in optimising multi-layered ANNs and is superior to the other considered GGP methods on the search for learning policies.

In an attempt to optimise the topology of DANNs, we evolve unconstraint AEs. The AEs are not restricted to a funnel-structure. Instead, the AEs can rather have any layer sequencing and are represented at the layer-level. The goal of evolution is to create compressed representations of the raw input. Thus, the AEs are evaluated based on the classification performance at the chokepoint. For the same reason, the AEs are trained (using BP) to reconstruct the mean signal of each class. This methodology was able to highly compress the input data without sacrificing the classification performance.

The experiments performed in Chapter 3 enable us to acknowledge that: (i) DSGE can effectively optimise the learning and connection weights of small-scale networks; (ii) GGP methods are ineffective at optimising real-values; and (iii) a layer-based representation is adequate to the optimisation of DANNs. These concepts were important to the definition of a flexible representation for evolving DANNs. This representation is central to DENSER (Chapter 4).

DENSER is our first proposal of a NE representation to optimise DANNs. The representation is based on DSGE, and the genotype of the individuals is divided into two genotypic levels. The outer level encodes the macrostructure of the network, i.e., the sequence of evolutionary units (e.g., layers, learning strategy). The inner level encodes the hyper-parameters of each evolutionary unit. There is a one-to-one mapping between the outer and the inner levels. The real-values are encoded directly to overcome the difficulty of DSGE in tuning them. According to Baldominos, Saez, and Isasi [24], DENSER is categorised as a "settlement and mature" NE representation. This representation scheme addresses RQ2. The training of the ANNs is limited to 10 epochs to avoid very time-consuming network evaluations. We test DENSER in the evolution of CNNs for the classification of the CIFAR-10 dataset. The generated CNNs are highly effective and competitive with the state of the art. In particular, the best-found model reports an accuracy of 94.13%. To investigate the generalisation, robustness, and scalability of the best-evolved CNNs, we apply them to the MNIST, Fashion-MNIST, SVHN, Rectangles, and CIFAR-100 datasets. The results show that the DENSER's models perform well beyond evolution. We highlight the performance on the CIFAR-100 dataset. Without further evolution, we report an accuracy of 74.94%.

Despite the high quality results, we can still point out some limitations to DENSER. First, even when restricting the evaluation of each ANN to 10 epochs, the search for high-performing models is time-consuming. Second, DENSER generates networks that are not ready-to-deploy, i.e., after the end of the evolutionary search, further and longer training is required to achieve state of the art performance. Third, the search re-starts from scratch when a second task is addressed. To solve the first two limitations, we introduce Fast-DENSER (Chapter 5). Fast-DENSER is an extension to DENSER that adds a third genotypic level to facilitate the encoding of the connectivity between layers; more specifically, this third level encodes, for each layer, its inputs. Therefore, the third genotypic level makes the representation more flexible, e.g., it enables the creation of skip connections. To speed up search, the evolutionary engine is replaced

by a (1+λ)-ES. Furthermore, the stop criterion is modified to a maximum GPU training time. The training time is set per individual and can grow during evolution. The objective is to grant more time to more complex networks. By the end of evolution, the ANNs are fully-trained. We compare Fast-DENSER to DENSER on the optimisation of CNNs to CIFAR-10. The results show that Fast-DENSER can attain results that are competitive to those reported by DENSER, but in a small fraction of the time. The speedup is roughly 20×, solving RQ3. We also conduct experiments in a real-world problem: the evolution of CNNs for gamma-ray detection based on the ground impact patterns. Once again, the results show that Fast-DENSER is effective, with the created networks outperforming state-of-the-art results.

To address the third limitation of DENSER, and simultaneously answer RQ4, we propose incremental development and adapt it to Fast-DENSER (Chapter 6). The main objective of incremental development is to identify parts of the networks of the source tasks that can aid and speed up evolution in a target task. The methodology works by warm-starting evolution. Notwithstanding, blocks from past networks can be incorporated (by mutation) in any evolutionary step. We compare the performance of the incremental and non-incremental versions of Fast-DENSER applied to the evolution of CNNs for CIFAR-10. In terms of performance, incremental development helps to attain high results in fewer generations. The analysis of the evolution of the structure of the networks shows that incremental development works by transferring evolutionary units in multiple stages of the search.

The work conducted throughout this Thesis shows that Fast-DENSER is an efficient framework that can generate high-performing models incrementally. The representation of the individuals is flexible and enables the optimisation of the structure (including the connectivity of the layers), learning strategy, and theoretically, any other network hyper-parameters. The method is tested in several image classification benchmarks (MNIST, Fashion-MNIST, SVHN and CIFAR-10), and in a real-world problem from the physics domain. The results are competitive with the state of the art. Most importantly, the search procedure shows a good trade-off between performance and search-time. The solutions are generated in an acceptable amount of time and are obtained resorting to limited computational resources.

A final word goes to the dissemination of the results obtained throughout this Thresis, which resulted in two national [15, 16] and eight international [8, 9, 11, 12, 18, 14, 19, 13] peer-reviewed conference papers. In addition to disseminating in conferences, we have also published three journal articles [20, 10, 17], and one book chapter [158]. The results of this research work were distinguished three times, as previously mentioned.

In addition to the work strictly related to the core of this Thesis, I have supervised a Master's Dissertation on the field of AutoML, which resulted in the publication of one of the above mentioned international conference papers [19], and still related to AutoML I co-authored a paper on the automatic optimisation of learning rate schedulers using DSGE. In addition, I participated in a project focused on the analysis of the retail data of SONAE - the largest Portuguese retail chain [197], and conducted side-research on the application of EC to the optimisation of edge bundling parameters [196], and on the design of a framework powered by EC to aid the design of fashion items [157].

We answered the four initially formulated research questions. Notwithstanding, during the performed experiments new questions arise. These constitute future work. In Chapter 5 we extend DENSER to the optimisation of the connectivity of the layers. The results seem to indicate that the evolved networks may benefit from this characteristic. To confirm this trend, experiments in more complex datasets are necessary. The experimentation with more complex datasets is challenging, as it is likely to slow the evaluation. Therefore, it is of importance to investigate surrogate models that can estimate the performance of a given network without training it. Last, the incremental development method that we propose does not overcome

catastrophic forgetting in a strict sense. That is, the models that are re-used from previous tasks are not changed, and thus they can still be applied to address those tasks. However, we are not creating a single model that can solve multiple tasks. To this end, we need a unified approach that transfers the topology and the weights. The evolution of modular networks can be a solution to this question. An advantage is that it is fairly easy to adapt Fast-DENSER to evolve modular architectures. The only modification that has to be performed is to change the production rules to recursive ones. For example to evolve modular networks, taking into account Grammar 5.2, we just need to add the production rule <features-rec> ::= <features-rec> <features> | <features>.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *OSDI*. USENIX Association, 2016, pp. 265–283.

[2] S. Agostinelli et al. "GEANT4: A Simulation toolkit". In: *Nucl. Instrum. Meth.* A506 (2003), pp. 250–303. DOI: 10.1016/S0168-9002(03)01368-8.

[3] Arbab Masood Ahmad and Gul Muhammad Khan. "Bio-signal Processing Using Cartesian Genetic Programming Evolved Artificial Neural Network (CGPANN)". In: *FIT*. IEEE Computer Society, 2012, pp. 261–268. DOI: 10.1109/FIT.2012.54.

[4] Fardin Ahmadizar, Khabat Soltanian, Fardin AkhlaghianTab, and Ioannis Tsoulos. "Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm". In: *Engineering Applications of Artificial Intelligence* 39 (2015), pp. 1–13. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2014.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0952197614002759.

[5] Samuel Alvernaz and Julian Togelius. "Autoencoder-augmented neuroevolution for visual doom playing". In: *CIG*. IEEE, 2017, pp. 1–8. DOI: 10.1109/CIG.2017.8080408.

[6] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. "An evolutionary algorithm that constructs recurrent neural networks". In: *IEEE Trans. Neural Networks* 5.1 (1994), pp. 54–65. DOI: 10.1109/72.265960.

[7] P. Assis et al. "Design and expected performance of a novel hybrid detector for very-high-energy gamma-ray astrophysics". In: *Astroparticle Physics* 99 (2018), pp. 34–42. ISSN: 0927-6505. DOI: https://doi.org/10.1016/j.astropartphys.2018.02.004. URL: http://www.sciencedirect.com/science/article/pii/S0927650518300586.

[8] Filipe Assunção, N. Lourenço, P. Machado, and B. Ribeiro. "Automatic Generation of Neural Networks with Structured Grammatical Evolution". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. June 2017, pp. 1557–1564. DOI: 10.1109/CEC.2017.7969488.

[9] Filipe Assunção, N. Lourenço, P. Machado, and B. Ribeiro. "Towards the Evolution of Multi-layered Neural Networks: A Dynamic Structured Grammatical Evolution Approach". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 393–400. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071286. URL: http://doi.acm.org/10.1145/3071178.3071286.

[10] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. "DENSER: Deep Evolutionary Network Structured Representation". In: *Genetic Programming and Evolvable Machines* 20.1 (Mar. 2019), pp. 5–35. ISSN: 1573-7632. DOI: 10.1007/s10710-018-9339-y. URL: https://doi.org/10.1007/s10710-018-9339-y.

[11] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. "Evolving the Topology of Large Scale Deep Neural Networks". In: *Genetic Programming*. Ed. by Mauro Castelli, Lukas Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez. Cham: Springer International Publishing, 2018, pp. 19–34. ISBN: 978-3-319-77553-1. DOI: 10.1007/978-3-319-77553-1_2.

[12] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. "Fast DENSER: Efficient Deep NeuroEvolution". In: *Genetic Programming*. Ed. by Lukas Sekanina, Ting Hu, Nuno Lourenço, Hendrik Richter, and Pablo García-Sánchez. Cham: Springer International Publishing, 2019, pp. 197–212. ISBN: 978-3-030-16670-0. DOI: 10.1007/978-3-030-16670-0_13.

[13] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. "Using GP Is NEAT: Evolving Compositional Pattern Production Functions". In: *Genetic Programming*. Ed. by Mauro Castelli, Lukas Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez. Cham: Springer International Publishing, 2018, pp. 3–18. ISBN: 978-3-319-77553-1. DOI: 10.1007/978-3-319-77553-1_1.

[14] Filipe Assunção, Nuno Lourenço, Bernardete Ribeiro, and Penousal Machado. "Evolution of Scikit-Learn Pipelines with Dynamic Structured Grammatical Evolution". In: *International Conference on the Applications of Evolutionary Computation*. Cham: Springer International Publishing, 2020.

[15] Filipe Assunção, Nuno Lourenço, Bernardete Ribeiro, and Penousal Machado. "Evolutionary Machine Learning: An Essay on Benchmarking". In: *Proceedings of the 23rd Portuguese Conference on Pattern Recognition (RecPad)*. 2017.

[16] Filipe Assunção, Nuno Lourenço, Bernardete Ribeiro, and Penousal Machado. "Evolutionary Machine Learning: An Essay on Experimental Design". In: *Proceedings of the 23rd Portuguese Conference on Pattern Recognition (RecPad)*. 2017.

[17] Filipe Assunção, Nuno Lourenço, Bernardete Ribeiro, and Penousal Machado. "Fast-DENSER: Fast Deep Evolutionary Network Structured Representation". In: *SoftwareX* 14 (2021), p. 100694. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2021.100694.

[18] Filipe Assunção, Nuno Lourenço, Bernardete Ribeiro, and Penousal Machado. "Incremental Evolution and Development of Deep Artificial Neural Networks". In: *European Conference on Genetic Programming (EuroGP)*. Cham: Springer International Publishing, 2020.

[19] Filipe Assuncao, D. Sereno, N. Lourenco, P. Machado, and B. Ribeiro. "Automatic Evolution of AutoEncoders for Compressed Representations". In: *2018 IEEE World Congress on Computational Ingelligence (WCCI)*. July 2018, pp. 1–8. DOI: 10.1109/CEC.2018.8477874.

[20] Filipe Assunção et al. "Automatic Design of Artificial Neural Networks for Gamma-Ray Detection". In: *IEEE Access* 7 (2019), pp. 110531–110540. DOI: 10.1109/ACCESS.2019.2933947.

[21] Farooq Azam. "Biologically inspired modular neural networks". PhD thesis. Virginia Tech, Blacksburg, VA, USA, 2001. URL: http://hdl.handle.net/10919/27998.

[22] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. "Designing Neural Network Architectures using Reinforcement Learning". In: *ICLR (Poster)*. OpenReview.net, 2017.

[23] Alejandro Baldominos, Yago Saez, and Pedro Isasi. "Evolutionary convolutional neural networks: An application to handwriting recognition". In: *Neurocomputing* 283 (2018), pp. 38–52. DOI: 10.1016/j.neucom.2017.12.049. URL: https://doi.org/10.1016/j.neucom.2017.12.049.

[24] Alejandro Baldominos, Yago Saez, and Pedro Isasi. "On the automated, evolutionary design of neural networks: past, present, and future". In: *Neural Computing and Applications* (Mar. 2019). ISSN: 1433-3058. DOI: 10.1007/s00521-019-04160-6. URL: https://doi.org/10.1007/s00521-019-04160-6.

[25] J Mark Baldwin. "A new factor in evolution". In: *The american naturalist* 30.354 (1896), pp. 441–451.

[26] S. Bengio, Y. Bengio, and J. Cloutier. "Use of genetic programming for the search of a new learning rule for neural networks". In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. June 1994, 324–327 vol.1. DOI: 10.1109/ICEC.1994.349932.

[27] Yoshua Bengio and Olivier Delalleau. "On the Expressive Power of Deep Architectures". In: *ALT*. Vol. 6925. Lecture Notes in Computer Science. Springer, 2011, pp. 18–36. DOI: 10.1007/978-3-642-24412-4_3.

[28] Thomas G van den Berg and Shimon Whiteson. "Critical factors in the performance of HyperNEAT". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM. 2013, pp. 759–766. DOI: 10.1145/2463372.2463460.

[29] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-Parameter Optimization". In: *NIPS*. 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.

[30] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305.

[31] James Bergstra, Daniel Yamins, and David D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 115–123.

[32] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies – A comprehensive introduction". In: *Natural Computing* 1.1 (2002), pp. 3–52. ISSN: 1572-9796. DOI: 10.1023/A:1015059928466. URL: http://dx.doi.org/10.1023/A:1015059928466.

[33] Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. "Applications of convolutional neural networks". In: *International Journal of Computer Science and Information Technologies* 7.5 (2016), pp. 2206–2215.

[34] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN: 9780387310732.

[35] Michael GB Blum, Maria Antonieta Nunes, Dennis Prangle, Scott A Sisson, et al. "A comparative review of dimension reduction methods in approximate Bayesian computation". In: *Statistical Science* 28.2 (2013), pp. 189–208.

[36] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer Science & Business Media, 2007.

[37] Bobby D. Bryant and Risto Miikkulainen. "Acquiring Visibly Intelligent Behavior with Example-Guided Neuroevolution". In: *AAAI*. AAAI Press, 2007, pp. 801–808. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-127.php.

[38] Zdeněk Buk, Jan Koutník, and Miroslav Šnorek. "NEAT in HyperNEAT substituted with genetic programming". In: *Adaptive and Natural Computing Algorithms* (2009), pp. 243–252. DOI: 10.1007/978-3-642-04921-7_25.

[39] L. Cardamone, D. Loiacono, and P. L. Lanzi. "Learning to Drive in the Open Racing Car Simulator Using Online Neuroevolution". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.3 (Sept. 2010), pp. 176–190. ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2010.2052102.

[40] Rich Caruana. "Multitask Learning: A Knowledge-based Source of Inductive Bias". In: *Proceedings of the Tenth International Conference on International Conference on Machine Learning*. ICML'93. Amherst, MA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 41–48. ISBN: 1-55860-307-7. DOI: 10.1016/b978-1-55860-307-3.50012-5.

[41] Xin Chen et al. "Gene expression patterns in human liver cancers". In: *Molecular biology of the cell* 13.6 (2002), pp. 1929–1939.

[42] François Chollet et al. *Keras*. https://github.com/keras-team/keras. 2015.

[43] Dondapati Chowdary et al. "Prognostic gene expression signatures can be measured in tissues collected in RNAlater preservative". In: *The journal of molecular diagnostics* 8.1 (2006), pp. 31–39.

[44] Zheng Chunhong and Jiao Licheng. "Automatic parameters selection for SVM based on GA". In: *Intelligent Control and Automation, 2004. WCICA 2004. Fifth World Congress on*. Vol. 2. IEEE. 2004, pp. 1869–1872.

[45] Dan Claudiu Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Transfer learning for Latin and Chinese characters with Deep Neural Networks". In: *IJCNN*. IEEE, 2012, pp. 1–6. DOI: 10.1109/IJCNN.2012.6252544.

[46] Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. "The evolutionary origins of modularity". In: *Proceedings of the Royal Society of London B: Biological Sciences* 280.1755 (2013). ISSN: 0962-8452. DOI: 10.1098/rspb.2012.2863. URL: rspb.royalsocietypublishing.org/content/280/1755/20122863.

[47] Ronan Collobert and Jason Weston. "A unified architecture for natural language processing: deep neural networks with multitask learning". In: *ICML*. Vol. 307. ACM International Conference Proceeding Series. ACM, 2008, pp. 160–167. DOI: 10.1145/1390156.1390177.

[48] Paulo Cortez. "Data Mining with Neural Networks and Support Vector Machines Using the R/rminer Tool". In: *ICDM*. Vol. 6171. Lecture Notes in Computer Science. Springer, 2010, pp. 572–583. DOI: 10.1007/978-3-642-14400-4_44.

[49] Victor Costa, Nuno Lourenço, João Correia, and Penousal Machado. "COEGAN: evaluating the coevolution effect in generative adversarial networks". In: *GECCO*. ACM, 2019, pp. 374–382. DOI: 10.1145/3321707.3321746.

[50] Charles Darwin. *On the Origin of Species*. New York: D. Appleton and Company, 1859, p. 470.

[51] Omid E. David and Iddo Greental. "Genetic algorithms for evolving deep neural networks". In: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM. 2014, pp. 1451–1452. DOI: 10.1145/2598394.2602287.

[52] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II". In: *Parallel Problem Solving from Nature PPSN VI*. Ed. by Marc Schoenauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 849–858. ISBN: 978-3-540-45356-7.

[53] Travis Desell. "Large scale evolution of convolutional neural networks using volunteer computing". In: *GECCO (Companion)*. ACM, 2017, pp. 127–128. DOI: 10.1145/3067695.3076002.

[54] Keith L. Downing. "The baldwin effect in developing neural networks". In: *GECCO*. ACM, 2010, pp. 555–562. DOI: 10.1145/1830483.1830588.

[55] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[56] Kaibo Duan and S. Sathiya Keerthi. "Which Is the Best Multiclass SVM Method? An Empirical Study". In: *Multiple Classifier Systems*. Vol. 3541. Lecture Notes in Computer Science. Springer, 2005, pp. 278–285. DOI: 10.1007/11494683_28.

[57] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12 (July 2011), pp. 2121–2159. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2021068.

[58] Eclipse Deeplearning4j Development Team. *Deeplearning4j: Open-source distributed deep learning for the JVM*. https://deeplearning4j.org. 2016.

[59] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing, Second Edition*. Natural Computing Series. Springer, 2015. DOI: 10.1007/978-3-662-44874-8.

[60] Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. "Neural Modularity Helps Organisms Evolve to Learn New Skills without Forgetting Old Skills". In: *PLoS Computational Biology* 11.4 (2015). DOI: 10.1371/journal.pcbi.1004128. URL: https://doi.org/10.1371/journal.pcbi.1004128.

[61] Abdelrahman Elsaid et al. "Neuro-Evolutionary Transfer Learning through Structural Adaptation". In: *EvoApplications*. Vol. 10784. Lecture Notes in Computer Science. Springer, 2020.

[62] Chrisantha Fernando et al. "Convolution by Evolution: Differentiable Pattern Producing Networks". In: *GECCO*. ACM, 2016, pp. 109–116. DOI: 10.1145/2908812.2908890.

[63] Chrisantha Fernando et al. "Meta-learning by the Baldwin effect". In: *GECCO (Companion)*. ACM, 2018, pp. 109–110. DOI: 10.1145/3205651.3205763.

[64] Chrisantha Fernando et al. "Pathnet: Evolution channels gradient descent in super neural networks". In: *arXiv preprint arXiv:1701.08734* (2017).

[65] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *NIPS*. 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.

[66] Andy Field. *Discovering statistics using SPSS:(and sex, drugs and rock'n'roll)*. Vol. 497. 2009, 2000. ISBN: 978-1-84787-906-6.

[67] Dario Floreano, Peter Dürr, and Claudio Mattiussi. "Neuroevolution: from architectures to learning". In: *Evolutionary Intelligence* 1.1 (Mar. 2008), pp. 47–62. ISSN: 1864-5917. DOI: 10.1007/s12065-007-0002-4. URL: https://doi.org/10.1007/s12065-007-0002-4.

[68] David B Fogel. "Using evolutionary programming to create neural networks that are capable of playing tic-tac-toe". In: *International Conference on Neural Networks*. IEEE Press San Francisco, CA. 1993, pp. 875–880.

[69] David B. Fogel. "Evolutionary programming: an introduction and some current directions". In: *Statistics and Computing* 4.2 (1994), pp. 113–129. ISSN: 1573-1375. DOI: 10.1007/BF00175356. URL: http://dx.doi.org/10.1007/BF00175356.

[70] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. "DEAP: evolutionary algorithms made easy". In: *J. Mach. Learn. Res.* 13 (2012), pp. 2171–2175. URL: http://dl.acm.org/citation.cfm?id=2503311.

[71] David Foster. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019.

[72] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. "WEKA - A Machine Learning Workbench for Data Mining". In: *The Data Mining and Knowledge Discovery Handbook*. Springer, 2005, pp. 1305–1314. DOI: 10.1007/978-0-387-09823-4_66.

[73] Robert M. French. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3.4 (1999), pp. 128–135. ISSN: 1364-6613. DOI: https://doi.org/10.1016/S1364-6613(99)01294-2. URL: http://www.sciencedirect.com/science/article/pii/S1364661399012942.

[74] Frauke Friedrichs and Christian Igel. "Evolutionary tuning of multiple SVM parameters". In: *Neurocomputing* 64 (2005), pp. 107–117. DOI: 10.1016/j.neucom.2004.11.022.

[75] Limin Fu and Enzo Medico. "FLAME, a novel fuzzy clustering method for the analysis of DNA microarray data". In: *BMC bioinformatics* 8.1 (2007), p. 1. DOI: 10.1186/1471-2105-8-3.

[76] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[77] Edgar Galván and Peter Mooney. "Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges". In: *CoRR* abs/2006.05415 (2020).

[78] Stuart Geman, Elie Bienenstock, and René Doursat. "Neural networks and the bias/variance dilemma". In: *Neural computation* 4.1 (1992), pp. 1–58. DOI: 10.1162/neco.1992.4.1.1.

[79] John Gideon, Soheil Khorram, Zakaria Aldeneh, Dimitrios Dimitriadis, and Emily Mower Provost. "Progressive Neural Networks for Transfer Learning in Emotion Recognition". In: *INTERSPEECH*. ISCA, 2017, pp. 1098–1102. URL: http://www.isca-speech.org/archive/Interspeech%5C_2017/abstracts/1637.html.

[80] Faustino J. Gomez and Risto Miikkulainen. "Incremental Evolution of Complex General Behavior". In: *Adaptive Behaviour* 5.3-4 (1997), pp. 317–342. DOI: 10.1177/105971239700500305.

[81] Faustino J. Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. "Accelerated Neural Evolution through Cooperatively Coevolved Synapses". In: *Journal of Machine Learning Research* 9 (2008), pp. 937–965.

[82] Faustino J Gomez and Risto Miikkulainen. "Solving non-Markovian control tasks with neuroevolution". In: *IJCAI*. Vol. 99. 1999, pp. 1356–1361.

[83] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay D. Shet. "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks". In: *ICLR*. 2014.

[84] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: *CoRR* abs/1406.2661 (2014). arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

[85] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[86] R. Paul Gorman and Terrence J. Sejnowski. "Analysis of hidden units in a layered network trained to classify sonar targets". In: *Neural networks* 1.1 (1988), pp. 75–89. DOI: 10.1016/0893-6080(88)90023-8.

[87] Benjamin Graham. "Fractional max-pooling". In: *arXiv preprint arXiv:1412.6071* (2014).

[88] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. "Evolving Neural Turing Machines for Reward-based Learning". In: *GECCO*. ACM, 2016, pp. 117–124. DOI: 10.1145/2908812.2908930.

[89] Marek Grochowski and Norbert Jankowski. "Comparison of Instance Selection Algorithms II. Results and Comments". In: *Artificial Intelligence and Soft Computing-ICAISC 2004* (2004), pp. 580–585. DOI: 10.1007/978-3-540-24844-6_87.

[90] Frederic Gruau et al. "Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm". In: (1994).

[91] Frédéric Gruau, Darrell Whitley, and Larry Pyeatt. "A Comparison Between Cellular Encoding and Direct Encoding for Genetic Neural Networks". In: *Proceedings of the 1st Annual Conference on Genetic Programming*. Stanford, California: MIT Press, 1996, pp. 81–89. ISBN: 0-262-61127-9. URL: http://dl.acm.org/citation.cfm?id=1595536.1595547.

[92] Frédéric Gruau and L. Darrell Whitley. "Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect". In: *Evolutionary Computation* 1.3 (1993), pp. 213–233. DOI: 10.1162/evco.1993.1.3.213.

[93] Ivan Gruber, Miroslav Hlaváč, Miloš Železný, and Alexey Karpov. "Facing Face Recognition with ResNet: Round One". In: *Interactive Collaborative Robotics*. Ed. by Andrey Ronzhin, Gerhard Rigoll, and Roman Meshcheryakov. Cham: Springer International Publishing, 2017, pp. 67–74. ISBN: 978-3-319-66471-2. DOI: 10.1007/978-3-319-66471-2_8.

[94] Yunhui Guo et al. "SpotTune: Transfer Learning Through Adaptive Fine-Tuning". In: *CVPR*. Computer Vision Foundation / IEEE, 2019, pp. 4805–4814. DOI: 10.1109/CVPR.2019.00494.

[95] Abhishek Gupta, Yew-Soon Ong, and Liang Feng. "Multifactorial Evolution: Toward Evolutionary Multitasking". In: *IEEE Trans. Evolutionary Computation* 20.3 (2016), pp. 343–357. DOI: 10.1109/TEVC.2015.2458037. URL: https://doi.org/10.1109/TEVC.2015.2458037.

[96] Isabelle Guyon et al. "A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention". In: *Proceedings of the 2016 Workshop on Automatic Machine Learning, AutoML 2016, co-located with 33rd International Conference on Machine Learning (ICML 2016), New York City, NY, USA, June 24, 2016*. Vol. 64. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 21–30.

[97] Isabelle Guyon et al. "Design of the 2015 ChaLearn AutoML challenge". In: *IJCNN*. IEEE, 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280767.

[98] John H. Holland. "Genetic Algorithms". In: *Scientific american* 267.1 (1992), pp. 44–50.

[99] Nikolaus Hansen and Andreas Ostermeier. "Completely Derandomized Self-Adaptation in Evolution Strategies". In: *Evolutionary Computation* 9.2 (2001), pp. 159–195. DOI: 10.1162/106365601750190398.

[100] Matthew J. Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. "A Neuroevolution Approach to General Atari Game Playing". In: *IEEE Trans. Comput. Intellig. and AI in Games* 6.4 (2014), pp. 355–366. DOI: 10.1109/TCIAIG.2013.2294713.

[101] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *CVPR*. IEEE Computer Society, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[102] D. Heck, G. Schatz, T. Thouw, J. Knapp, and J. N. Capdevielle. "CORSIKA: A Monte Carlo code to simulate extensive air showers". In: 1998.

[103] Geoffrey E. Hinton and Steven J. Nowlan. "How Learning Can Guide Evolution". In: *Complex Systems* 1.3 (1987).

[104] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Computation* 18.7 (2006), pp. 1527–1554. DOI: 10.1162/neco.2006.18.7.1527.

[105] Geoffrey E. Hinton and Richard S. Zemel. "Autoencoders, Minimum Description Length and Helmholtz Free Energy". In: *NIPS*. Morgan Kaufmann, 1993, pp. 3–10. URL: http://papers.nips.cc/paper/798-autoencoders-minimum-description-length-and-helmholtz-free-energy.

[106] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507.

[107] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[108] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks". In: *CVPR*. IEEE Computer Society, 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243.

[109] Navdeep Jaitly, Patrick Nguyen, Andrew W. Senior, and Vincent Vanhoucke. "Application of Pretrained Deep Neural Networks to Large Vocabulary Speech Recognition". In: *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*. 2012, pp. 2578–2581. URL: http://www.isca-speech.org/archive/interspeech%5C_2012/i12%5C_2578.html.

[110] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical Reparameterization with Gumbel-Softmax". In: *ICLR (Poster)*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=rkE3y85ee.

[111] Norbert Jankowski and Marek Grochowski. "Comparison of Instances Seletion Algorithms I. Algorithms Survey". In: *ICAISC*. Springer. 2004, pp. 598–603. DOI: 10.1007/978-3-540-24844-6_90.

[112] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *ACM Multimedia*. ACM, 2014, pp. 675–678. DOI: 10.1145/2647868.2654889.

[113] Álvaro Barbero Jiménez, Jorge López Lázaro, and José R. Dorronsoro. "Finding optimal model parameters by deterministic and annealed focused grid search". In: *Neurocomputing* 72.13-15 (2009), pp. 2824–2832. DOI: 10.1016/j.neucom.2008.09.024.

[114] James D. Kelly Jr. and Lawrence Davis. "A Hybrid Genetic Algorithm for Classification". In: *IJCAI*. Morgan Kaufmann, 1991, pp. 645–650.

[115] Jae-Yoon Jung and James A. Reggia. "Evolutionary Design of Neural Network Architectures Using a Descriptive Encoding Language". In: *IEEE Trans. Evolutionary Computation* 10.6 (2006), pp. 676–688. DOI: 10.1109/TEVC.2006.872346.

[116] Lukasz Kaiser et al. "One Model To Learn Them All". In: *CoRR* abs/1706.05137 (2017). arXiv: 1706.05137. URL: http://arxiv.org/abs/1706.05137.

[117] Maarten Keijzer, Michael O'Neill, Conor Ryan, and Mike Cattolico. "Grammatical Evolution Rules: The Mod and the Bucket Rule". In: *EuroGP*. Vol. 2278. Lecture Notes in Computer Science. Springer, 2002, pp. 123–130.

[118] Stephen Kelly and Malcolm I. Heywood. "Multi-task Learning in Atari Video Games with Emergent Tangled Program Graphs". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 195–202. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071303.

[119] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. "Measuring Catastrophic Forgetting in Neural Networks". In: *AAAI*. AAAI Press, 2018, pp. 3390–3398. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16410.

[120] Maryam Mahsal Khan, Gul Muhammad Khan, and Julian F. Miller. "Evolution of neural networks using Cartesian Genetic Programming". In: *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586547.

[121] Maryam Mahsal Khan, Gul Muhammad Khan, and Julian Francis Miller. "Evolution of Optimal ANNs for Non-Linear Control Problems using Cartesian Genetic Programming". In: *IC-AI*. CSREA Press, 2010, pp. 339–346.

[122] Dmitrii Khritonenko, Vladimir Stanovov, and Eugene Semenkin. "Applying an instance selection method to an evolutionary neural classifier design". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 173. 1. IOP Publishing. 2017, p. 012007.

[123] Heung Bum Kim, Sung Hoon Jung, Tag Gon Kim, and Kyu Ho Park. "Fast learning method for back-propagation neural network by evolutionary adaptation of learning rates". In: *Neurocomputing* 11.1 (1996), pp. 101–106. DOI: 10.1016/0925-2312(96)00009-4.

[124] S. Kim, T. Hori, and S. Watanabe. "Joint CTC-attention based end-to-end speech recognition using multi-task learning". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2017, pp. 4835–4839. DOI: 10.1109/ICASSP.2017.7953075.

[125] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6980.

[126] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *ICLR*. 2014.

[127] Hiroaki Kitano. "Designing Neural Networks Using Genetic Algorithms with Graph Generation System". In: *Complex Systems* 4.4 (1990). URL: http://www.complex-systems.com/abstracts/v04%5C_i04%5C_a06.html.

[128] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *ECML*. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293.

[129] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-Sklearn". In: *Automated Machine Learning*. The Springer Series on Challenges in Machine Learning. Springer, 2019, pp. 97–111. DOI: 10.1007/978-3-030-05318-5_5.

[130] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn". In: *ICML workshop on AutoML*. 2014.

[131] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA". In: *Journal of Machine Learning Research* 18 (2017), 25:1–25:5.

[132] Jan Koutník, Giuseppe Cuccu, J"urgen Schmidhuber, and Faustino J. Gomez. "Evolving large-scale neural networks for vision-based reinforcement learning". In: *GECCO*. ACM, 2013, pp. 1061–1068. DOI: 10.1145/2463372.2463509.

[133] Jan Koutník, Giuseppe Cuccu, J"urgen Schmidhuber, and Faustino J. Gomez. "Evolving large-scale neural networks for vision-based TORCS". In: *FDG*. Society for the Advancement of the Science of Digital Games, 2013, pp. 206–212. URL: http://www.fdg2013.org/program/papers/paper27%5C_koutnik%5C_etal.pdf.

[134] Jan Koutník, Faustino J. Gomez, and J"urgen Schmidhuber. "Evolving neural networks in compressed weight space". In: *GECCO*. ACM, 2010, pp. 619–626. DOI: 10.1145/1830483.1830596.

[135] Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM. 2014, pp. 541–548. DOI: 10.1145/2576768.2598358.

[136] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, 1992.

[137] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. 2009.

[138] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[139] Kim W. C. Ku, Man-Wai Mak, and Wan-Chi Siu. "A study of the Lamarckian evolution of recurrent neural networks". In: *IEEE Trans. Evolutionary Computation* 4.1 (2000), pp. 31–42. DOI: 10.1109/4235.843493.

[140] Steven Künzel and Silja Meyer-Nieberg. "Evolving Artificial Neural Networks for Multi-objective Tasks". In: *EvoApplications*. Vol. 10784. Lecture Notes in Computer Science. Springer, 2018, pp. 671–686. DOI: 10.1007/978-3-319-77538-8_45.

[141] Jean Baptiste Lamarck and Hugh Elliot. "Zoological philosophy: an exposition with regard to the natural history of animals". In: (1985).

[142] Sean Lander and Yi Shang. "EvoAE–A New Evolutionary Method for Training Autoencoders for Deep Learning Networks". In: *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. Vol. 2. IEEE. 2015, pp. 790–795. DOI: 10.1109/COMPSAC.2015.63.

[143] Hugo Larochelle, Dumitru Erhan, Aaron C. Courville, James Bergstra, and Yoshua Bengio. "An empirical evaluation of deep architectures on problems with many factors of variation". In: *ICML*. Vol. 227. ACM International Conference Proceeding Series. ACM, 2007, pp. 473–480.

[144] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521 (May 2015), pp. 436–444. DOI: 10.1038/nature14539.

[145] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[146] Jason Liang, Elliot Meyerson, and Risto Miikkulainen. "Evolutionary Architecture Search for Deep Multitask Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '18. Kyoto, Japan: ACM, 2018, pp. 466–473. ISBN: 978-1-4503-5618-3. DOI: 10.1145/3205455.3205489. URL: http://doi.acm.org/10.1145/3205455.3205489.

[147] Jason Liang et al. "Evolutionary Neural AutoML for Deep Learning". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. Prague, Czech Republic: ACM, 2019, pp. 401–409. ISBN: 978-1-4503-6111-8. DOI: 10.1145/3321707.3321721. URL: http://doi.acm.org/10.1145/3321707.3321721.

[148] Ricardo H. R. Lima and Aurora T. R. Pozo. "Evolving convolutional neural networks through grammatical evolution". In: *GECCO (Companion)*. ACM, 2019, pp. 179–180. DOI: 10.1145/3319619.3322058.

[149] Chenxi Liu et al. "Progressive Neural Architecture Search". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss. Cham: Springer International Publishing, 2018, pp. 19–35. ISBN: 978-3-030-01246-5. DOI: 10.1007/978-3-030-01246-5_2.

[150] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. "Hierarchical Representations for Efficient Architecture Search". In: *ICLR (Poster)*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=BJQRKzbA-.

[151] Vincenzo Lomonaco and Davide Maltoni. "CORe50: a New Dataset and Benchmark for Continuous Object Recognition". In: *CoRL*. Vol. 78. Proceedings of Machine Learning Research. PMLR, 2017, pp. 17–26. URL: http://proceedings.mlr.press/v78/lomonaco17a.html.

[152] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and Philip S. Yu. "Learning Multiple Tasks with Multilinear Relationship Networks". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1593–1602. ISBN: 978-1-5108-6096-4. URL: http://dl.acm.org/citation.cfm?id=3294771.3294923.

[153] Pablo Ribalta Lorenzo and Jakub Nalepa. "Memetic evolution of deep neural networks". In: *GECCO*. ACM, 2018, pp. 505–512. DOI: 10.1145/3205455.3205631.

[154] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sánchez Ramos, and José Ranilla Pastor. "Particle swarm optimization for hyper-parameter selection in deep neural networks". In: *GECCO*. ACM, 2017, pp. 481–488. DOI: 10.1145/3071178.3071208.

[155] Ilya Loshchilov and Frank Hutter. "CMA-ES for Hyperparameter Optimization of Deep Neural Networks". In: *arXiv preprint arXiv:1604.07269* (2016).

[156] Nuno António Marques Lourenço. "Enhancing Grammar-Based Approaches for the Automatic Design of Algorithms". PhD thesis. University of Coimbra, 2016. URL: http://hdl.handle.net/10316/29450.

[157] Nuno Lourenço, Filipe Assunção, Catarina Maçãs, and Penousal Machado. "EvoFashion: Customising Fashion Through Evolution". In: *Computational Intelligence in Music, Sound, Art and Design - 6th International Conference, EvoMUSART 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*. Ed. by João Correia, Vic Ciesielski, and Antonios Liapis. Vol. 10198. Lecture Notes in Computer Science. 2017, pp. 176–189. DOI: 10.1007/978-3-319-55750-2_12. URL: https://doi.org/10.1007/978-3-319-55750-2%5C_12.

[158] Nuno Lourenço, Filipe Assunção, Francisco B. Pereira, Ernesto Costa, and Penousal Machado. "Structured Grammatical Evolution: A Dynamic Approach". In: *Handbook of Grammatical Evolution*. Ed. by Conor Ryan, Michael O'Neill, and JJ Collins. Cham: Springer International Publishing, 2018, pp. 137–161. ISBN: 978-3-319-78717-6. DOI: 10.1007/978-3-319-78717-6_6. URL: https://doi.org/10.1007/978-3-319-78717-6_6.

[159] Nuno Lourenço, Francisco B. Pereira, and Ernesto Costa. "Unveiling the properties of structured grammatical evolution". In: *Genetic Programming and Evolvable Machines* (2016), pp. 1–39. ISSN: 1573-7632. DOI: 10.1007/s10710-015-9262-4. URL: http://dx.doi.org/10.1007/s10710-015-9262-4.

[160] Zhichao Lu et al. "NSGA-Net: neural architecture search using multi-objective genetic algorithm". In: *GECCO*. ACM, 2019, pp. 419–427. DOI: 10.1145/3321707.3321729.

[161] Davide Maltoni and Vincenzo Lomonaco. "Continuous learning in single-incremental-task scenarios". In: *Neural Networks* 116 (2019), pp. 56–73. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.03.010. URL: http://www.sciencedirect.com/science/article/pii/S0893608019300838.

[162] M. Mandischer. "Evolving recurrent neural networks with non-binary encoding". In: *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*. Vol. 2. Nov. 1995, 584–589 vol.2. DOI: 10.1109/ICEC.1995.487449.

[163] Martin Mandischer. "Representation and Evolution of Neural Networks". In: *Artificial Neural Nets and Genetic Algorithms*. Ed. by Rudolf F. Albrecht, Colin R. Reeves, and Nigel C. Steele. Vienna: Springer Vienna, 1993, pp. 643–649. ISBN: 978-3-7091-7533-0. DOI: 10.1007/978-3-7091-7533-0_93.

[164] Vittorio Maniezzo. "Genetic evolution of the topology and weight distribution of neural networks". In: *IEEE Trans. Neural Networks* 5.1 (1994), pp. 39–53. DOI: 10.1109/72.265959.

[165] Timmy Manning and Paul Walsh. "Improving the Performance of CGPANN for Breast Cancer Diagnosis Using Crossover and Radial Basis Functions". In: *EvoBIO*. Vol. 7833. Lecture Notes in Computer Science. Springer, 2013, pp. 165–176. DOI: 10.1007/978-3-642-37189-9_15.

[166] Xiao-Jiao Mao, Chunhua Shen, and Yu-Bin Yang. "Image Restoration Using Very Deep Convolutional Encoder-Decoder Networks with Symmetric Skip Connections". In: *NIPS*. 2016, pp. 2802–2810. URL: http://papers.nips.cc/paper/6172-image-restoration-using-very-deep-convolutional-encoder-decoder-networks-with-symmetric-skip-connections.

[167] James McDermott. "Why Is Auto-Encoding Difficult for Genetic Programming?" In: *EuroGP*. Vol. 11451. Lecture Notes in Computer Science. Springer, 2019, pp. 131–145.

[168] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. "Grammar-based Genetic Programming: a survey". In: *Genetic Programming and Evolvable Machines* 11.3 (Sept. 2010), pp. 365–396. ISSN: 1573-7632. DOI: 10.1007/s10710-010-9109-y. URL: https://doi.org/10.1007/s10710-010-9109-y.

[169] Martial Mermillod, Aurélia Bugaiska, and Patrick Bonin. "The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects". In: *Frontiers in Psychology* 4 (2013), p. 504. ISSN: 1664-1078. DOI: 10.3389/fpsyg.2013.00504. URL: https://www.frontiersin.org/article/10.3389/fpsyg.2013.00504.

[170] Elliot Meyerson and Risto Miikkulainen. "Beyond Shared Hierarchies: Deep Multitask Learning through Soft Layer Ordering". In: *ICLR (Poster)*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=BkXmYfbAZ.

[171] Risto Miikkulainen et al. "Evolving Deep Neural Networks". In: *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Ed. by Robert Kozma, Cesare Alippi, Yoonsuck Choe, and Francesco Carlo Morabito. Academic Press, 2019, pp. 293–312. ISBN: 978-0-12-815480-9. DOI: https://doi.org/10.1016/B978-0-12-815480-9.00015-3. URL: http://www.sciencedirect.com/science/article/pii/B9780128154809000153.

[172] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. "Designing Neural Networks using Genetic Algorithms". In: *ICGA*. Morgan Kaufmann, 1989, pp. 379–384.

[173] Julian F. Miller. *Cartesian Genetic Programming*. Springer-Verlag Berlin Heidelberg, 2011.

[174] Julian F. Miller and Peter Thomson. "Cartesian Genetic Programming". In: *Genetic Programming*. Ed. by Riccardo Poli et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 121–132. ISBN: 978-3-540-46239-2.

[175] Jae H. Min and Young-Chan Lee. "Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters". In: *Expert Syst. Appl.* 28.4 (2005), pp. 603–614. DOI: 10.1016/j.eswa.2004.12.008.

[176] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. "Cross-Stitch Networks for Multi-Task Learning". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016. DOI: 10.1109/CVPR.2016.433.

[177] Tom Michael Mitchell. *The Discipline of Machine Learning*. Vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[178] Jonas Mockus. "On Bayesian Methods for Seeking the Extremum and their Application". In: *IFIP Congress*. 1977, pp. 195–200.

[179] David J. Montana and Lawrence Davis. "Training Feedforward Neural Networks Using Genetic Algorithms". In: *IJCAI*. Morgan Kaufmann, 1989, pp. 762–767.

[180] David E. Moriarty and Risto Miikkulainen. "Efficient Reinforcement Learning through Symbiotic Evolution". In: *Machine Learning* 22.1-3 (1996), pp. 11–32. DOI: 10.1023/A:1018004120707.

[181] David E. Moriarty and Risto Miikkulainen. "Forming Neural Networks Through Efficient and Adaptive Coevolution". In: *Evolutionary Computation* 5.4 (1997), pp. 373–399. DOI: 10.1162/evco.1997.5.4.373.

[182] David E Moriarty and Risto Miikkulainen. "Learning sequential decision tasks through symbiotic evolution of neural networks". In: *Advances in the Evolutionary Synthesis of Intelligent Agents* (2001), p. 367.

[183] Gregory Morse and Kenneth O. Stanley. "Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks". In: *GECCO*. ACM, 2016, pp. 477–484. DOI: 10.1145/2908812.2908916.

[184] Brandon Muller, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. "Transfer Learning: A Building Block Selection Mechanism in Genetic Programming for Symbolic Regression". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. Prague, Czech Republic: ACM, 2019, pp. 350–351. ISBN: 978-1-4503-6748-6. DOI: 10.1145/3319619.3322072. URL: http://doi.acm.org/10.1145/3319619.3322072.

[185] Yuval Netzer et al. "Reading digits in natural images with unsupervised feature learning". In: *NIPS workshop on deep learning and unsupervised feature learning*. Vol. 2011. 2. 2011, p. 5.

[186] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Vol. 4. Springer Science & Business Media, 2012.

[187] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. "Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science". In: *GECCO*. ACM, 2016, pp. 485–492. DOI: 10.1145/2908812.2908918.

[188] Alexander Ororbia, AbdElRahman ElSaid, and Travis Desell. "Investigating Recurrent Neural Network Memory Structures Using Neuro-Evolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 446–455. ISBN: 9781450361118. DOI: 10.1145/3321707.3321795. URL: https://doi.org/10.1145/3321707.3321795.

[189] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (Oct. 2010), pp. 1345–1359. ISSN: 2326-3865. DOI: 10.1109/TKDE.2009.191.

[190] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. "Continual lifelong learning with neural networks: A review". In: *Neural Networks* 113 (2019), pp. 54–71. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.01.012. URL: http://www.sciencedirect.com/science/article/pii/S0893608019300231.

[191] German I. Parisi, Jun Tani, Cornelius Weber, and Stefan Wermter. "Lifelong Learning of Spatiotemporal Representations With Dual-Memory Recurrent Self-Organization". In: *Frontiers in Neurorobotics* 12 (2018), p. 78. ISSN: 1662-5218. DOI: 10.3389/fnbot.2018.00078. URL: https://www.frontiersin.org/article/10.3389/fnbot.2018.00078.

[192] Matt Parker and Bobby D. Bryant. "Lamarckian neuroevolution for visual control in the Quake II environment". In: *IEEE Congress on Evolutionary Computation*. IEEE, 2009, pp. 2630–2637. DOI: 10.1109/CEC.2009.4983272.

[193] José Parra, Leonardo Trujillo, and Patricia Melin. "Hybrid back-propagation training with evolutionary strategies". In: *Soft Computing* 18.8 (Aug. 2014), pp. 1603–1614. ISSN: 1433-7479. DOI: 10.1007/s00500-013-1166-8. URL: https://doi.org/10.1007/s00500-013-1166-8.

[194] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[195] Riccardo Poli. "Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming". In: *ICGA*. Morgan Kaufmann, 1997, pp. 346–353.

[196] Evgheni Polisciuc, Filipe Assunção, and Penousal Machado. "Towards Partially Automatic Search of Edge Bundling Parameters". In: *Computational Intelligence in Music, Sound, Art and Design - 7th International Conference, EvoMUSART 2018, Parma, Italy, April 4-6, 2018, Proceedings*. Ed. by Antonios Liapis, Juan Jesús Romero Cardalda, and Anikó Ekárt. Vol. 10783. Lecture Notes in Computer Science. Springer, 2018, pp. 223–238. DOI: 10.1007/978-3-319-77583-8_15. URL: https://doi.org/10.1007/978-3-319-77583-8%5C_15.

[197] Evgheni Polisciuc, Catarina Maçãs, Filipe Assunção, and Penousal Machado. "Hexagonal Gridded Maps and Information Layers: A Novel Approach for the Exploration and Analysis of Retail Data". In: SA '16. Macau: Association for Computing Machinery, 2016. ISBN: 9781450345477. DOI: 10.1145/3002151.3002160. URL: https://doi.org/10.1145/3002151.3002160.

[198] João Carlos Figueira Pujol and Riccardo Poli. "Evolving the Topology and the Weights of Neural Networks Using a Dual Representation". In: *Appl. Intell.* 8.1 (1998), pp. 73–84. DOI: 10.1023/A:1008272615525.

[199] Amr Radi and Riccardo Poli. "Discovering Efficient Learning Rules for Feedforward Neural Networks Using Genetic Programming". In: *Recent Advances in Intelligent Paradigms and Applications*. Ed. by Ajith Abraham, Lakhmi C. Jain, and Janusz Kacprzyk. Heidelberg: Physica-Verlag HD, 2003, pp. 133–159. ISBN: 978-3-7908-1770-6. DOI: 10.1007/978-3-7908-1770-6_7. URL: https://doi.org/10.1007/978-3-7908-1770-6_7.

[200] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. "Automated Machine Learning with Monte-Carlo Tree Search". In: *IJCAI*. ijcai.org, 2019, pp. 3296–3303.

[201] Aditya Rawal and Risto Miikkulainen. "Evolving Deep LSTM-based Memory Networks using an Information Maximization Objective". In: *GECCO*. ACM, 2016, pp. 501–508. DOI: 10.1145/2908812.2908941.

[202] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. "Regularized Evolution for Image Classifier Architecture Search". In: *AAAI*. AAAI Press, 2019, pp. 4780–4789. DOI: 10.1609/aaai.v33i01.33014780.

[203] Esteban Real et al. "Large-Scale Evolution of Image Classifiers". In: *ICML*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 2902–2911.

[204] Joseph Reisinger, Kenneth O. Stanley, and Risto Miikkulainen. "Evolving Reusable Neural Modules". In: *Genetic and Evolutionary Computation – GECCO 2004*. Ed. by Kalyanmoy Deb. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 69–81. ISBN: 978-3-540-24855-2. DOI: 10.1007/978-3-540-24855-2_7.

[205] Mark Bishop Ring. "Continual learning in reinforcement environments". PhD thesis. University of Texas at Austin Austin, Texas 78712, 1995. ISBN: 978-3-486-23603-3. URL: http://d-nb.info/945690320.

[206] Sebastian Risi and Julian Togelius. "Neuroevolution in Games: State of the Art and Open Challenges". In: *IEEE Trans. Comput. Intellig. and AI in Games* 9.1 (2017), pp. 25–41. DOI: 10.1109/TCIAIG.2015.2494596.

[207] Miguel Rocha, Paulo Cortez, and José Neves. "Evolution of neural networks for classification and regression". In: *Neurocomputing* 70.16-18 (2007), pp. 2809–2816. DOI: 10.1016/j.neucom.2006.05.023.

[208] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological review* 65.6 (1958), p. 386.

[209] David E. Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. "Backpropagation". In: ed. by Yves Chauvin and David E. Rumelhart. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1995. Chap. Backpropagation: The Basic Theory, pp. 1–34. ISBN: 0-8058-1259-8. URL: http://dl.acm.org/citation.cfm?id=201784.201785.

[210] Andrei A. Rusu et al. "Progressive Neural Networks". In: *CoRR* abs/1606.04671 (2016). arXiv: 1606.04671. URL: http://arxiv.org/abs/1606.04671.

[211] Conor Ryan, JJ Collins, and Michael O'Neill. "Grammatical evolution: Evolving programs for an arbitrary language". In: *Genetic Programming: First European Workshop, EuroGP'98 Paris, France, April 14–15, 1998 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 83–96. ISBN: 978-3-540-69758-9. DOI: 10.1007/BFb0055930. URL: http://dx.doi.org/10.1007/BFb0055930.

[212] Conor Ryan, Michael O'Neill, and J. J. Collins, eds. *Handbook of Grammatical Evolution*. Springer, 2018. ISBN: 978-3-319-78716-9. DOI: 10.1007/978-3-319-78717-6. URL: https://doi.org/10.1007/978-3-319-78717-6.

[213] Alex GC de Sá, Walter José GS Pinto, Luiz Otavio VB Oliveira, and Gisele L Pappa. "RECIPE: a grammar-based framework for automatically evolving classification pipelines". In: *European Conference on Genetic Programming*. Springer. 2017, pp. 246–261. DOI: 10.1007/978-3-319-55696-3_16.

[214] J. D. Schaffer, D. Whitley, and L. J. Eshelman. "Combinations of genetic algorithms and neural networks: a survey of the state of the art". In: *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. June 1992, pp. 1–37. DOI: 10.1109/COGANN.1992.273950.

[215] Jacob Schrum and Risto Miikkulainen. "Constructing Complex NPC Behavior via Multi-Objective Neuroevolution". In: *AIIDE*. The AAAI Press, 2008. URL: http://www.aaai.org/Library/AIIDE/2008/aiide08-018.php.

[216] E. O. Scott and K. A. De Jong. "Automating Knowledge Transfer with Multi-Task Optimization". In: *2019 IEEE Congress on Evolutionary Computation (CEC)*. June 2019, pp. 2252–2259. DOI: 10.1109/CEC.2019.8790224.

[217] A. Sedki, D. Ouazar, and E. El Mazoudi. "Evolving neural network using real coded genetic algorithm for daily rainfall–runoff forecasting". In: *Expert Systems with Applications* 36.3, Part 1 (2009), pp. 4523–4527. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2008.05.024. URL: http://www.sciencedirect.com/science/article/pii/S095741740800225X.

[218] Michael L. Seltzer and Jasha Droppo. "Multi-task learning in deep neural networks for improved phoneme recognition". In: *ICASSP*. IEEE, 2013, pp. 6965–6969. DOI: 10.1109/ICASSP.2013.6639012.

[219] Pierre Sermanet, Soumith Chintala, and Yann LeCun. "Convolutional neural networks applied to house numbers digit classification". In: *Pattern Recognition (ICPR), 2012 21st International Conference on*. IEEE. 2012, pp. 3288–3291.

[220] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[221] Vincent G. Sigillito, Simon P. Wing, Larrie V. Hutton, and Kile B. Baker. "Classification of radar returns from the ionosphere using neural networks". In: *Johns Hopkins APL Technical Digest* 10.3 (1989), pp. 262–266.

[222] Tim Silhan, Stefan Oehmcke, and Oliver Kramer. "Evolution of Stacked Autoencoders". In: *CEC*. IEEE, 2019, pp. 823–830. DOI: 10.1109/CEC.2019.8790182.

[223] Patrice Y Simard, David Steinkraus, John C Platt, et al. "Best practices for convolutional neural networks applied to visual document analysis". In: *ICDAR*. Vol. 3. 2003, pp. 958–962. DOI: 10.1109/ICDAR.2003.1227801.

[224] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR*. 2015.

[225] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *NIPS*. 2012, pp. 2960–2968. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.

[226] Jasper Snoek et al. "Scalable Bayesian optimization using deep neural networks". In: *International Conference on Machine Learning*. 2015, pp. 2171–2180.

[227] Anders Søgaard and Yoav Goldberg. "Deep multi-task learning with low level tasks supervised at lower layers". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 231–235. DOI: 10.18653/v1/P16-2038. URL: https://www.aclweb.org/anthology/P16-2038.

[228] K. Soltanian, F. A. Tab, F. A. Zar, and I. Tsoulos. "Artificial neural networks generation using grammatical evolution". In: *2013 21st Iranian Conference on Electrical Engineering (ICEE)*. May 2013, pp. 1–5. DOI: 10.1109/IranianCEE.2013.6599788.

[229] Andrea Soltoggio. "Short-term plasticity as cause–effect hypothesis testing in distal reward learning". In: *Biological Cybernetics* 109.1 (Feb. 2015), pp. 75–94. ISSN: 1432-0770. DOI: 10.1007/s00422-014-0628-0.

[230] Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1 (Jan. 2019), pp. 24–35. ISSN: 2522-5839. DOI: 10.1038/s42256-018-0006-z. URL: https://doi.org/10.1038/s42256-018-0006-z.

[231] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks". In: *Artificial Life* 15.2 (2009), pp. 185–212. DOI: 10.1162/artl.2009.15.2.15202.

[232] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[233] Vladimir Stanovov, Eugene Semenkin, and Olga Semenkina. "Instance Selection Approach for Self-Configuring Hybrid Fuzzy Evolutionary Algorithm for Imbalanced Datasets". In: *ICSI (1)*. Vol. 9140. Lecture Notes in Computer Science. Springer, 2015, pp. 451–459. DOI: 10.1007/978-3-319-20466-6_47.

[234] W. Nick Street, William H. Wolberg, and Olvi L. Mangasarian. "Nuclear feature extraction for breast tumor diagnosis". In: *IS&T/SPIE's Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics. 1993, pp. 861–870.

[235] Felipe Petroski Such et al. "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning". In: *CoRR* abs/1712.06567 (2017). arXiv: 1712.06567. URL: http://arxiv.org/abs/1712.06567.

[236] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. "A Genetic Programming Approach to Designing Convolutional Neural Network Architectures". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 497–504. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071229. URL: http://doi.acm.org/10.1145/3071178.3071229.

[237] Jing Sun, Cheng Wan, Jun Cheng, Fengli Yu, and Jiang Liu. "Retinal Image Quality Classification Using Fine-Tuned CNN". In: *Fetal, Infant and Ophthalmic Medical Image Analysis*. Ed. by M. Jorge Cardoso et al. Cham: Springer International Publishing, 2017, pp. 126–133. ISBN: 978-3-319-67561-9. DOI: 10.1007/978-3-319-67561-9_14.

[238] Y. Sun, B. Xue, M. Zhang, and G. G. Yen. "Evolving Deep Convolutional Neural Networks for Image Classification". In: *IEEE Transactions on Evolutionary Computation* (2019), pp. 1–1. ISSN: 1941-0026. DOI: 10.1109/TEVC.2019.2916183.

[239] Yanan Sun, Bing Xue, and Mengjie Zhang. "Automatically Evolving CNN Architectures Based on Blocks". In: *CoRR* abs/1810.11875 (2018).

[240] R. Sutton. "Two problems with back propagation and other steepest descent learning procedures for networks". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986* (1986), pp. 823–832. URL: https://ci.nii.ac.jp/naid/10012746305/en/.

[241] Christian Szegedy et al. "Intriguing properties of neural networks". In: *ICLR (Poster)*. 2014.

[242] Matthew E. Taylor and Peter Stone. "An Introduction to Intertask Transfer for Reinforcement Learning". In: *AI Magazine* 32.1 (Mar. 2011), p. 15. DOI: 10.1609/aimag.v32i1.2329. URL: https://www.aaai.org/ojs/index.php/aimagazine/article/view/2329.

[243] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. "Transfer via Inter-task Mappings in Policy Search Reinforcement Learning". In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '07. Honolulu, Hawaii: ACM, 2007, 37:1–37:8. ISBN: 978-81-904262-7-5. DOI: 10.1145/1329125.1329170. URL: http://doi.acm.org/10.1145/1329125.1329170.

[244] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[245] Ann Thorhauer and Franz Rothlauf. "On the Locality of Standard Search Operators in Grammatical Evolution". In: *PPSN*. Vol. 8672. Lecture Notes in Computer Science. Springer, 2014, pp. 465–475.

[246] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms". In: *KDD*. ACM, 2013, pp. 847–855. DOI: 10.1145/2487575.2487629.

[247] Sebastian Thrun. "Is Learning The n-th Thing Any Easier Than Learning The First?" In: *NIPS*. MIT Press, 1995, pp. 640–646. URL: http://papers.nips.cc/paper/1034-is-learning-the-n-th-thing-any-easier-than-learning-the-first.

[248] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2 (2012), pp. 26–31.

[249] Sreenivas Sremath Tirumala. "Implementation of Evolutionary Algorithms for Deep Architectures". In: *Proceedings of the 2nd International Workshop on Artificial Intelligence and Cognition (AIC)*. 2014, pp. 164–171.

[250] Schaul Tom et al. "PyBrain". In: *Journal of Machine Learning Research* 11 (2010), pp. 743–746.

[251] Ioannis Tsoulos, Dimitris Gavrilis, and Euripidis Glavas. "Neural network construction and training using grammatical evolution". In: *Neurocomputing* 72.1 (2008). Machine Learning for Signal Processing (MLSP 2006) / Life System Modelling, Simulation, and Bio-inspired Computing (LSMS 2007), pp. 269–277. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2008.01.017. URL: http://www.sciencedirect.com/science/article/pii/S0925231208000830.

[252] A. M. Turing. "Computing Machinery and Intelligence". In: *Mind* 59.236 (1950), pp. 433–460. ISSN: 00264423, 14602113. URL: http://www.jstor.org/stable/2251299.

[253] Andrew James Turner and Julian Francis Miller. "Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks". In: *GECCO*. ACM, 2013, pp. 1005–1012. DOI: 10.1145/2463372.2463484.

[254] Andrew James Turner and Julian Francis Miller. "The Importance of Topology Evolution in NeuroEvolution: A Case Study Using Cartesian Genetic Programming of Artificial Neural Networks". In: *SGAI Conf.* Springer, 2013, pp. 213–226. DOI: 10.1007/978-3-319-02621-3_15.

[255] Pedro Ángel Castillo Valdivieso et al. "Lamarckian Evolution and the Baldwin Effect in Evolutionary Neural Networks". In: *CoRR* abs/cs/0603004 (2006).

[256] Phillip Verbancsics and Josh Harguess. "Image classification using generative neuro evolution for deep learning". In: *2015 IEEE Winter Conference on Applications of Computer Vision, WACV 2015, Waikoloa, HI, USA, January 5-9, 2015*. IEEE. 2015, pp. 488–493. DOI: 10.1109/WACV.2015.71.

[257] Phillip Verbancsics and Kenneth O. Stanley. "Evolving Static Representations for Task Transfer". In: *J. Mach. Learn. Res.* 11 (2010), pp. 1737–1769. URL: http://portal.acm.org/citation.cfm?id=1859909.

[258] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion". In: *J. Mach. Learn. Res.* 11 (2010), pp. 3371–3408. URL: http://portal.acm.org/citation.cfm?id=1953039.

[259] Steven Walczak and Narciso Cerpa. "Heuristic principles for the design of artificial neural networks". In: *Information and software technology* 41.2 (1999), pp. 107–117. DOI: 10.1016/S0950-5849(98)00116-5.

[260] Cen Wan, Alex A Freitas, and João Pedro De Magalhães. "Predicting the pro-longevity or anti-longevity effect of model organism genes with new hierarchical feature selection methods". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 12.2 (2015), pp. 262–275. DOI: 10.1109/TCBB.2014.2355218.

[261] Lin Wang, Yi Zeng, and Tao Chen. "Back propagation neural network with adaptive differential evolution algorithm for time series forecasting". In: *Expert Syst. Appl.* 42.2 (2015), pp. 855–863. DOI: 10.1016/j.eswa.2014.08.018.

[262] Darrell Whitley. "Applying genetic algorithms to neural network learning". In: *Proceedings of the Seventh Conference (AISB89) on Artificial Intelligence and Simulation of Behaviour*. Morgan Kaufmann Publishers Inc. 1989, pp. 137–144.

[263] L. Darrell Whitley, Timothy Starkweather, and Christopher Bogart. "Genetic algorithms and neural networks: optimizing connections and connectivity". In: *Parallel Computing* 14.3 (1990), pp. 347–361. DOI: 10.1016/0167-8191(90)90086-O.

[264] Alexis P. Wieland. "Evolving Controls for Unstable Systems". In: *Connectionist Models*. Ed. by David S. Touretzky, Jeffrey L. Elman, Terrence J. Sejnowski, and Geoffrey E. Hinton. Morgan Kaufmann, 1991, pp. 91–102. ISBN: 978-1-4832-1448-1. DOI: https://doi.org/10.1016/B978-1-4832-1448-1.50015-9. URL: http://www.sciencedirect.com/science/article/pii/B9781483214481500159.

[265] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal Component Analysis". In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

[266] Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. "Transfer Learning with Neural AutoML". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 8356–8365. URL: http://papers.nips.cc/paper/8056-transfer-learning-with-neural-automl.pdf.

[267] Z. Wu, C. Valentini-Botinhao, O. Watts, and S. King. "Deep neural networks employing Multi-Task Learning and stacked bottleneck features for speech synthesis". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Apr. 2015, pp. 4460–4464. DOI: 10.1109/ICASSP.2015.7178814.

[268] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[269] Liqiang Xiao, Honglun Zhang, Wenqing Chen, Yongkun Wang, and Yaohui Jin. "Learning What to Share: Leaky Multi-Task Network for Text Classification". In: *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 2055–2065. URL: https://www.aclweb.org/anthology/C18-1175.

[270] Lingxi Xie and Alan L. Yuille. "Genetic CNN". In: *ICCV*. IEEE Computer Society, 2017, pp. 1388–1397. DOI: 10.1109/ICCV.2017.154.

[271] Xin Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447.

[272] Xin Yao and Yong Liu. "A new evolutionary system for evolving artificial neural networks". In: *IEEE Trans. Neural Networks* 8.3 (1997), pp. 694–713. DOI: 10.1109/72.572107.

[273] B. Yegnanarayana. *Artificial Neural Networks*. Prentice-Hall of India Private Limited, New Delhi, 2004.

[274] Ozal Yildirim, Ulas Baran Baloglu, Ru-San Tan, Edward J. Ciaccio, and U. Rajendra Acharya. "A new approach for arrhythmia classification using deep coded features and LSTM networks". In: *Computer Methods and Programs in Biomedicine* 176 (2019), pp. 121–133. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2019.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0169260718314329.

[275] Jaehong Yoon, Eunho Yang, Jungtae Lee, and Sung Ju Hwang. "Lifelong Learning with Dynamically Expandable Networks". In: *Sixth International Conference on Learning Representations*. ICLR. 2018. URL: https://openreview.net/forum?id=Sk7KsfW0-.

[276] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. "How transferable are features in deep neural networks?" In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 3320–3328. URL: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf.

[277] Steven R. Young, Derek C. Rose, Thomas P. Karnowski, Seung-Hwan Lim, and Robert M. Patton. "Optimizing deep learning hyper-parameters through an evolutionary algorithm". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. ACM, 2015, 4:1–4:5. DOI: 10.1145/2834892.2834896.

[278] Chunkai Zhang, Huihe Shao, and Yu Li. "Particle swarm optimisation for evolving artificial neural network". In: *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: "Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions", Sheraton Music City Hotel, Nashville, Tennessee, USA, 8-11 October 2000*. Oct. 2000, pp. 2487–2490. DOI: 10.1109/ICSMC.2000.884366.

[279] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang. "Facial Landmark Detection by Deep Multi-task Learning". In: *ECCV (6)*. Vol. 8694. Lecture Notes in Computer Science. Springer, 2014, pp. 94–108. DOI: 10.1007/978-3-319-10599-4_7.

[280] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. "Learning Transferable Architectures for Scalable Image Recognition". In: *CVPR*. IEEE Computer Society, 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907.

Appendix A

AutoML-DSGE: Evolution of Scikit-Learn Pipelines

The goal of this appendix is to introduce a new framework, which we call AutoML-DSGE, that adapts DSGE to the evolution of classification pipelines. Our motivation is to investigate whether or not DSGE, in addition to optimising ANNs, can optimise ML in the broad sense. In particular, we optimise Scikit-Learn [194] pipelines. Next, we define pipelines (Section A.1), the used grammar (Section A.2), and detail the evolution of pipelines using DSGE (Section A.3). The experimental results are analysed and discussed in Section A.4. The code for AutoML-DSGE is released as open-source software and can be found in the GitHub repository `https://github.com/fillassuncao/automl-dsge`.

A.1 Pipelines

In the field of ML, a classification pipeline is defined as an ordered set of operations that are performed to the data instances to accurately separate them in the multiple classes of the dataset. The operations in the pipeline can be divided into 3 disjoint sets: (i) data pre-processing; (ii) feature design and selection; and (iii) classification. Table A.1 enumerates the methods that are considered to form the pipelines in the current work. Recall that we focus on classification pipelines, and thus only classification algorithms are taken into account. Nonetheless, the extension of the approach to regression algorithms is straight-forward. We will optimise Scikit-Learn pipelines, and thus the methods in the table are Scikit-Learn implementations. Further details can be found in `https://scikit-learn.org/stable/user_guide.html`.

A.2 Grammar

The grammar used by AutoML-DSGE describes the search space of the Scikit-Learn classification pipelines. The grammar is shown in Figure A.1. The production rules are only partially shown because of space constraints: the grammar is comprised of 89 production rules that encode the different pipeline methods and their parameterisation. The complete grammars can be found in `https://github.com/fillassuncao/automl-dsge/tree/master/sge/grammars`. There is a separate grammar for each dataset because of specific dataset parameters, e.g., number of features. The used grammars are adapted from the grammars used by RECIPE, which is the method we compare AutoML-DSGE to.

Table A.1: Scikit-Learn classes that are allowed to be part of the AutoML-DSGE pipelines.

Pre-processing	Feature manipulation	Classification
Imputer	VarianceThreshold	ExtraTreeClassifier
Normalizer	SelectPercentile	DecisionTreeClassifier
MinMaxScaler	SelectFpr	GaussianNB
MaxAbsScaler	SelectFwe	BernouliNB
RobustScaler	SelectFdr	MultinominalNB
StandardScaler	RFE	SVC
	REFCV	NuSVC
	SelectFromModel	KNeighborsClassifier
	IncrementalPCA	RadiusNeighborsClassifier
	PCA	NearestCentroid.
	FastICA	LDA
	GaussianRandomProjection	QDA
	SparseRandomProjection	LogisticRegression
	RBFSampler	LogisticRegressionCV
	Nystroem	PassiveAggressiveClassifier
	FeatureAgglomeration	Perceptron
	PolynomialFeatures	Ridge
		RidgeCV
		AdaBoostClassifier
		GradientBoostingClassifier
		RandomForestClassifier
		ExtraTreesClassifier

The axiom of the grammar is the pipeline non-terminal symbol, and consequently, the pipeline can be formed by pre-processing and classification methods (line 1), or just by the classification method (line 2). The current version of AutoML-DSGE does not consider ensembles. The extension of AutoML-DSGE to enable the optimisation of ensembles could be easily introduced by adding a recursive production rule to build pipelines with more than one classifier algorithm, each a voter of the ensemble. The pre-processing methods manipulate the dataset and features (lines 3-6), and the classification methods cover a wide range of ML approaches, amongst which, are clustering, SVMs, trees, or ANNs (lines 11-18). In more detail, the pipeline methods are encoded as follows: the pre-processing and classification methods are preceded respectively by the preprocessing and classifier tags, that are placed before the method name (e.g., classifier:radius_neighbors, line 17). The method name must match the name of the function that is used in the mapping from the phenotype to the Scikit-Learn interpretable code (see Section 3.2.1). The same rationale is applied to the method parameters, where the parameter name precedes the parameter value. The parameters can be of three types: (i) closed choice, e.g., the weights parameter, in line 20, that can assume the values uniform or distance; (ii) random integer, e.g., the leaf size parameter (line 22); or (iii) random float, e.g., the radius parameter (line 19).

The search space of AutoML-DSGE, i.e., the number of possible combinations of the grammar is greater than 9.39×10^{17}. The continuous parameters can generate an infinite number of possibilities, and thus are not considered in the search space size. The parameters related to the number of features are also not taken into account because they are problem-dependent.

```
<pipeline> ::= <preprocessing> <algorithm>                                  (1)
             | <algorithm>                                                  (2)
<preprocessing> ::= <imputation> | <bounding> | <dimensionality> |          (3)
             | <binarizer> | <imputation> <bounding>                       (4)
             | <imputation> <binarizer>                                     (5)
             | ...                                                          (6)
<imputation> ::= preprocessing:imputer <strategy_imp>                       (7)
<strategy_imp> ::= strategy:mean | strategy:median | strategy:most_frequent (8)
             ...                                                            (9)
             ...                                                           (10)
<algorithm> ::= <strong> | <weak> | <tree_ensemble>                        (11)
             ...                                                           (12)
             ...                                                           (13)
<weak> ::= <nearest> | <discriminant> | ...                                (14)
             ...                                                           (15)
             ...                                                           (16)
<nearest> ::= classifier:radius_neighbors <radius> <weights>               (17)
             <k_algorithm> <leaf_size> <p> <d_metric>                      (18)
<radius> ::= radius:RANDFLOAT(1.0,30.0)                                    (19)
<weights> ::= weights:uniform | weights:distance                           (20)
<k_algorithm> ::= algorithm:auto | algorithm:brute| ...                    (21)
<leaf_size> ::= leaf_size:RANDINT(5,100)                                   (22)
             ...                                                           (23)
             ...                                                           (24)
```

Grammar A.1: Grammar used by AutoML-DSGE for optimising Scikit-Learn pipelines.

A.3 Evolution of Pipelines

The pipelines are evolved using DSGE, and therefore, a population of individuals is evolved continuously throughout a given number of generations, until a stop criterion is met. Each individual encodes a different pipeline. The core of the representation of the individuals in AutoML-DSGE is similar to the representation scheme used in DSGE, with one main difference related to the need to directly keep real values in the genotype. Otherwise, they would have to be encoded by production rules, such as:

```
<randfloat> ::= <signal> <rec-number> .<rec-number>
   <signal> ::= - | +
<rec-number> ::= <number> | <number> <rec-number>
   <number> ::= 0 | 1 | 2 | 3 | 4
                5 | 6 | 7 | 8 | 9
```

Table A.2: Description of the datasets used in the AutoML-DSGE experiments.

Dataset	#Inst.	#Feat.	Feat. types	#Classes	Missing
Breast Cancer	699	9	Integer	2	Yes
Car Evaluation	1728	5	Categorical	4	No
Caenorhabditis Elegans	478	765	Binary	2	No
Chen-2002	179	85	Real	2	No
Chowdary-2006	104	182	Real	2	No
Credit-G	1000	20	Real / Categorical	2	No
Drosophila Melanogaster	119	182	Real	2	No
DNA-No-PPI-T11	135	104	Real / Categorical	2	Yes
Glass	214	9	Real	7	No
Wine Quality-Red	1599	11	Real	10	No

The encoding of real values using production rules has two main disadvantages. On the one hand, it enlarges the search space. On the other hand, there is no easy way to control the limits (minimum and maximum) of the generated real values. If the search space encompasses two or more real values, with different ranges, there would be the need for different production rules, one for each real value range. Because of the aforementioned, we encode the integers and floats directly, as real-values. When expanding the grammar, when we reach a terminal symbol that is either RANDINT or RANDFLOAT, we store a tuple in the genotype. The tuple has the format (rand-type, rand-min, rand-max, rand-value), where rand-type can assume integer or float, the rand-min and rand-max are the lower and the upper limits of the range, and the rand-value is the randomly generated value of the type rand-type, and within the [rand-min, rand-max] range. The tuple is necessary for performing the mutation, i.e. when a mutation is applied to an individual, and it is required to generate a new random value for a specific parameter, we must know its type and allowed range.

DSGE is a grammar-based approach, and thus the genotype is completely separate from the phenotype. The phenotype does not directly represent a trainable pipeline. Consequently, for assessing the fitness of the individuals, we have to perform two sequential steps: (i) map the genotype to the phenotype; and (ii) map the phenotype to Scikit-Learn interpretable model. To map the genotype to the phenotype, the decoding procedure of DSGE is adapted: the only difference lies in the decoding of the real-values, where the value in the last position of the tuple is read. The phenotype of AutoML-DSGE is readable, despite not being Scikit-Learn executable code. The readability of the phenotype is facilitated by the fact that each parameter has the parameter name associated with the value; an example of a phenotype is "classifier:random_forest criterion:gini max_depth:None n_estimators:50 min_weight_fraction_leaf:0.01 ...".

To map the phenotype to a Scikit-Learn interpretable pipeline, we have to traverse the phenotype linearly from left to right and, for each pre-processing or classifier method, create the corresponding Scikit-Learn object. Therefore, for each method in the grammar, we have to build a function that creates the Scikit-Learn object. The function receives all the parameters that are encoded in the grammar and outputs the Scikit-Learn object. Whenever the grammar is extended to include more methods, we have to create the corresponding functions.

To evaluate the evolved pipelines, we use cross-validation (with 3 folds). In the current paper, the fitness is the average of the cross-validation performances. The metric used to evaluate the performance is the F-measure. We chose this metric because some of the datasets where we will be conducting the experiments are highly unbalanced.

The goal of AutoML-DSGE is to generate (automatically) effective Scikit-Learn classification

Table A.3: Experimental parameters used in the AutoML-DSGE experiments.

Parameter	Value
Number of runs	30
Number of generations	100
Population size	100
Mutation rate	10%
Crossover rate	90%
Elitism	5 individuals
Tournament size	2
Max. pipeline training time	5 minutes
Max. #generations without improvement	5

pipelines that non-expert ML users can deploy in their problems and domains. With this in mind, similarly to other approaches, we limit the train time of each pipeline to a maximum CPU time, which in the conducted experiments is set to five minutes. For the same reason, evolution is halted when there is no improvement for five generations.

A.4 Experimentation

To investigate the ability of AutoML-DSGE to generate effective classification Scikit-Learn pipelines, we apply it to the classification of 10 datasets, which are described in Section A.4.1. The experimental setup is detailed in Section A.4.2, and the analysis of the evolutionary results, and comparison to the pipelines generated by RECIPE is carried out in Section A.4.3.

A.4.1 Datasets

To enable a fair comparison between AutoML-DSGE and RECIPE, we conduct the experiments on the same datasets used by RECIPE: 10 datasets – 5 from the UCI ML repository [55], and 5 from bio-informatics [41, 43, 260]. A summary of the dataset characteristics is shown in Table A.2. The table provides information on the number of instances (#Inst.), number of features (#Feat.), type of features (Feat. types), number of classes (#Classes), and if there are or not missing values in the dataset (Missing).

A.4.2 Experimental Setup

The parameters required to perform the experiments contained in this appendix are described in Table A.3. The parameters are the same for AutoML-DSGE and RECIPE. The maximum CPU training time is measured in minutes, and thus it is important to mention that the experiments are performed in a dedicated server with an Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz, and 32 GB of RAM.

The code used for AutoML-DSGE, and RECIPE can be found, respectively, in the GitHub repositories `github.com/laic-ufmg/Recipe/`, and `github.com/fillassuncao/automl-dsge`. The code of RECIPE was modified to include the evolution stop criteria based on a maximum number of generations without improvement, which despite described in the framework's paper [213], is not included in the current code version.

To enable the comparison of results, we apply the same dataset partitioning scheme used in RECIPE. The datasets are split using a 10-fold cross-validation strategy, and thus, as we perform

Table A.4: AutoML-DSGE, and RECIPE comparative performance. The results are averages of 30 independent runs.

Dataset	AutoML-DSGE	RECIPE	p-value
Breast Cancer	**0.9568 ± 0.0296**	0.9311 ± 0.0798	**0.0264** (++)
Car Evaluation	**0.9964 ± 0.0068**	0.9962 ± 0.0079	0.9761
Caenorhabditis Elegans	**0.6140 ± 0.0644**	0.6049 ± 0.0681	0.7948
Chen-2002	**0.9451 ± 0.0413**	0.9292 ± 0.0618	0.3371
Chowdary-2006	**0.9970 ± 0.0163**	0.9812 ± 0.0514	0.0679
Credit-G	**0.7400 ± 0.0370**	0.7075 ± 0.0359	**0.0008** (+++)
Drosophila Melanogaster	**0.6679 ± 0.1001**	0.6353 ± 0.1518	0.2585
DNA-No-PPI-T11	**0.7114 ± 0.1194**	0.7021 ± 0.0761	0.9681
Glass	**0.7628 ± 0.1095**	0.7325 ± 0.1021	0.0524
Wine Quality-Red	**0.6600 ± 0.0387**	0.6430 ± 0.0422	**0.0257** (++)

30 evolutionary runs, each fold is kept as the test set three times, and the remaining used for training the pipelines. During each run, the test set is kept aside from evolution, and the training set is used to train the pipelines with cross-validation (3 folds). By the end of evolution, the best pipeline is trained using all the training data and applied to the test set. The evolution is conducted using Grammar A.1.

To establish the pair-wise comparison of the results, and check whether or not the differences between AutoML-DSGE and RECIPE are statistically significant, we use the Wilcoxon Signed-Rank test, with a significance level of 5%. Further, for the statistically significant differences, we compute the effect size.

A.4.3 Experimental Results

To compare the pipelines generated by AutoML-DSGE and RECIPE, we conduct evolution for the same datasets, and using equivalent grammatical formulations, i.e., the search space is the same for both frameworks. The test performance (f-measure) for each dataset is presented in Table A.4. The results are averages of 30 independent runs. A value of f-measure marked in bold indicates the approach that reports the highest average performance. In addition, the table also reports the p-values for the pair-wise comparisons between the two approaches, and bold p-values indicate statistically significant differences. The effect-size is denoted in brackets after the p-value, with +, ++, and +++ denoting small ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$), and large ($r \geq 0.5$) effect sizes, respectively.

The analysis of the results indicates that AutoML-DSGE reports results that are always superior to those obtained by RECIPE. In addition to the higher average, the standard deviation is lower in the AutoML-DSGE results in 7 out of 10 datasets, i.e., for the considered datasets, AutoML-DSGE generates higher results more consistently than RECIPE. These differences are statistically significant in 3 datasets (Breast Cancer, Credit-G, and Wine Quality-Red). The effect size is medium twice and high once. AutoML-DSGE is never worse than RECIPE.

The results of Table A.4 report the average performance of the 30 evolutionary runs, for each dataset. Nonetheless, as we are optimising ML methods, we investigate the generalisation ability of the obtained pipelines. To this end, we compute the average difference between the evolutionary and test performances for the 10 datasets. Except for the Chowdary-2006 and Car datasets, the average difference between the evolutionary and test performance is lower in AutoML-DSGE than in RECIPE. Considering all datasets, the average difference between the evolutionary and test set performance is approximately 0.0328 in AutoML-DSGE and 0.0589 in

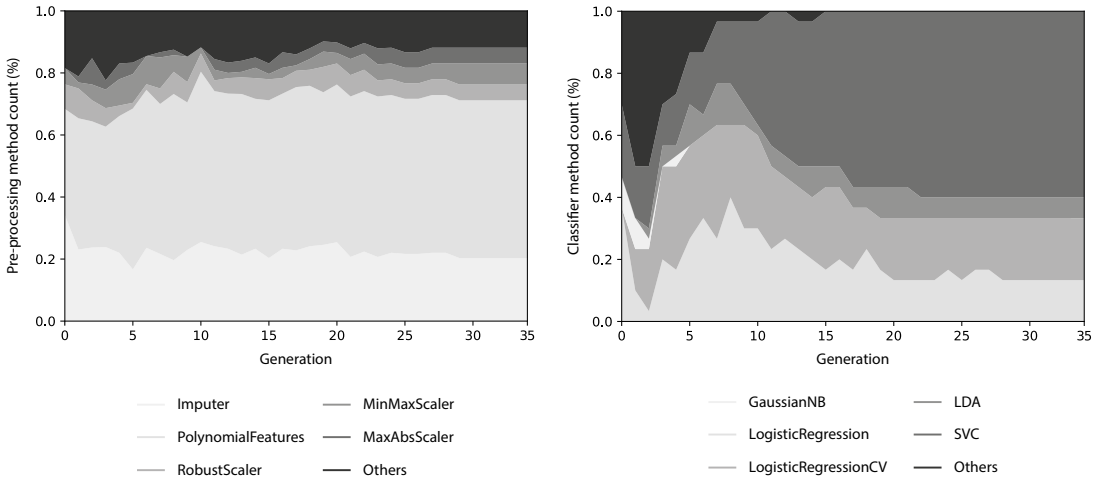


Figure A.1: Stacked area charts of the AutoML-DSGE evolution of the pre-processing (left) and classification (right) methods in the Car dataset. The results reflect the percentage of the best pipelines that use each of the methods.

RECIPE. This proves that the tendency to overfit is lower in AutoML-DSGE, as it reports, more often than RECIPE, evolutionary performances that are closer to the test ones.

To analyse the structure of the pipelines evolved by AutoML-DSGE, we inspect the methods that compose them. We focus on the Car dataset, as it is the dataset where, on average, more generations are performed. Figure A.1 shows the evolution of the pre-processing and classification methods of the best individuals as generations proceed. The results show the evolution of the percentage of the runs that use each of the pre-processing and classification methods. Recall that the different evolutionary runs can differ in the number of performed generations, and therefore, to avoid a misleading representation of the evolution of the methods that compose the pipelines, we consider that all runs have the same number of generations. That is, we consider that all runs evolve for the same number of generations as the longer run (in this case 35 generations). For the evolutionary runs that perform fewer generations, we keep the last generation (which is the best-found solution) for the remainder of the generations. The results show that, for the Car dataset, the pre-processing methods distribution does not change as evolution proceeds. On the other hand, a different behaviour is noticeable on the classifier methods, that converge to the SVC, and LogisticRegression (or LogisticRegressionCV) method. The evolution also shows that evolution is focused on the methods that are more effective for that specific dataset. Otherwise, the used methods would be more diverse, and the percentage of the Others would be higher. In particular, we plot, in Figure A.2, the best pipeline found for classifying the Car dataset. We also inspect the evolutionary patterns in the remaining datasets and acknowledge similar conclusions. It is, however, important to point out that, for the different datasets, evolution focuses on different pre-processing and classification methods.

Ultimately, AutoML-DSGE generates no invalid pipelines. After investigating the pipelines that were assigned with the worse possible fitness, we conclude that their training is halted because they are unable to train in the maximum granted CPU time of five minutes, or because they run out of memory.

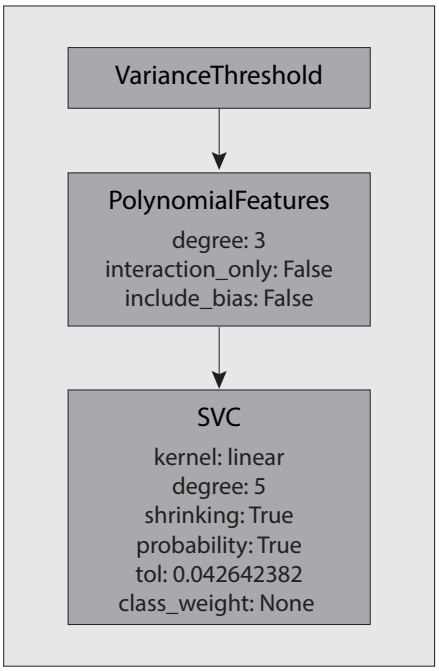


Figure A.2: Best pipeline generated by AutoML-DSGE for classifying the Car dataset. Each box represents a pipeline method and its parameterisation.

A.5 Summary

Before the deployment of a ML model, there are a number of choices that have to be made. There is the need to pre-process the dataset, design, extract and select features, and decide which ML model is the most adequate. On top of that, all these sequential choices are correlated, meaning that one affects multiple others. The choices that have to be made require both domain-specific and ML expertise. In an effort to facilitate the widespread use of ML models, we introduce a novel AutoML framework: AutoML-DSGE.

AutoML-DSGE is a grammar-based AutoML approach, and thus the search space is defined in a human-readable CFG. This key-point of the framework enables the easy adaptation of AutoML-DSGE to tackle different problems using a wide set of methods. Further, it eases the introduction of a-priori knowledge in the search and tuning of the pipelines. The current version of the framework focuses on the optimisation of Scikit-Learn classification pipelines. The code is released as open-source software and can be found in the GitHub repository: `https://github.com/fillassuncao/automl-dsge`.

We compare the performance AutoML-DSGE to RECIPE, which to the best of our knowledge is the only grammar-based AutoML framework. The methods are compared in 10 datasets from different domains. The results show that the pipelines generated by AutoML-DSGE surpass in performance the ones obtained by RECIPE; the average performances of AutoML-DSGE are always superior to RECIPE, and are statistically superior in 3 datasets (with medium and large effect sizes). Moreover, AutoML-DSGE is less prone to overfitting than RECIPE.

Appendix B

Fast-DENSER Software Release

The combination of ANNs and EC makes NE results and code hard to reproduce. Therefore, we decided to make Fast-DENSER code freely available under the Apache License 2.0. The code can be found in `https://github.com/fillassuncao/fast-denser3`. In addition, to avoid the difficulty of having to install dependencies and configure the server, we released docker images, that have been uploaded to docker hub: `https://hub.docker.com/r/fillassuncao/f-denser`. The main objective and functionality of this tool is to automate the search for ANNs of arbitrary depth. It can be used either in classification or regression problems. The general structure of the framework is described in Section B.1. An example of its use in a classification problem is detailed in Section B.2. The extension of the framework, i.e., the addition of new layers and/or learning algorithms is explained in Section B.3. The metadata of the last release of Fast-DENSER is summarised in Table B.3.

B.1 Software Architecture

Fast-DENSER is based on EC, and thus a set of individuals (population) is evolved throughout a defined number of generations. The individuals encode ANNs and need to be mapped into interpretable models to assess their quality (fitness). To promote evolution, from one generation to the next, the population is mutated. In particular, in Fast-DENSER, the evolutionary engine is a (1+λ)-ES: the next generation is formed by the best individual (elite) and λ mutations of it.

To map the individuals to interpretable models, we resort to Keras [42] with Tensforflow [1] background. The framework requires the definition of two major inputs: (i) the network structure that establishes the allowed sequence of evolutionary units; and (ii) the grammar that defines the search space, i.e., layers and parameters. These two components are further detailed in the upcoming sub-sections. There are additional parameters that are enumerated in the framework GitHub page, which can be found at `https://github.com/fillassuncao/fast-denser3`. The evolutionary cycle and its interaction with the inputs, and the mapping from the individuals into trainable DANNs for quality assessment are shown in Figure B.1.

The output of the framework is a fully-trained DANN, tailored to the considered problem. The network is made available as a Keras/Tensorflow model, but there is also a file specifying the structure and all network parameters (including weights) so that the user can later deploy the model in any other framework of his/her choice. Intermediate files are also generated throughout generations, in particular, there is a file for each generation; this file, among other properties, reports the phenotype (i.e., the actual network), fitness value, number of trainable parameters, and training time of each of the individuals of the population.

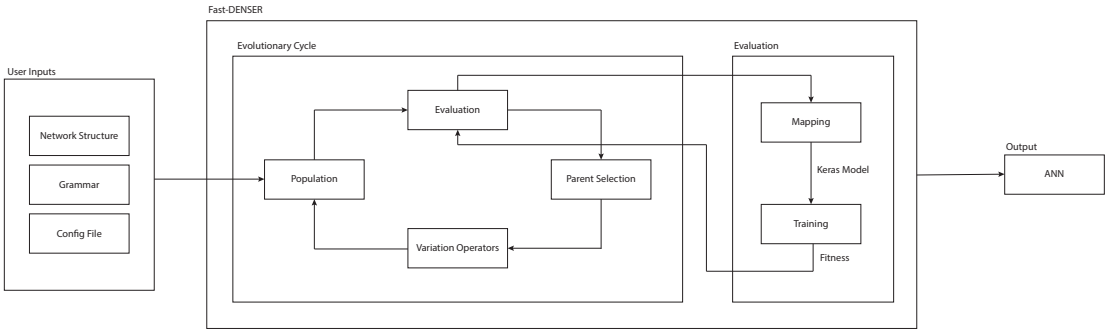


Figure B.1: Architecture of the Fast-DENSER framework.

B.1.1 Network Structure

The network structure is divided into 3 parts: (i) the hidden-layers; (ii) the output; and (iii) the macroblocks. The hidden-layers set the sequence of layers that the framework can use to build DANNs, and is defined by the user as an ordered list of evolutionary units, where each position stores the grammar non-terminal symbols and the minimum and the maximum number of evolutionary units of that type. The output sets the rule that is used to form the output layer. Finally, the macroblocks consider the overall settings of the network, e.g., the learning strategy or the data pre-processing or augmentation policies. Together, the network structure and grammar define the search space. An example of the hidden-layers, output, and macroblocks is respectively, [(features, 1, 10), (classification, 1, 10)]; softmax; and [learning]. The non-terminals require a one-to-one mapping to the grammar (discussed next). That is, the formed networks can have between 1 and 10 feature-related layers, and between 1 and 10 classification-related layers, that are followed by a softmax layer. The learning strategy is also evolved.

B.1.2 Grammar

The grammar is defined in the Backus-Naur Form (BNF) and encodes the hyper-parameters of each of the evolutionary units. A partial example of the grammar is depicted in Figure B.1. The full grammar can be found in `github.com/fillassuncao/fast-denser3/blob/master/example/cnn.grammar`. The non-terminals of the network structure are used as starting symbols. The hyper-parameters can be either integer, float, or closed choice. The integers and floats are parameterised using the block [parameter_name, type, num_values, min_value, max_value] (e.g., num-filters, line 4); the closed choice parameters are parameterised using the grammatical expansions (e.g., padding, line 7).

The evolutionary units encoding layers must start by layer:layer_type (e.g., layer:fc, line 12); the evolutionary units encoding the learning strategy must start by learning:learning_algorithm (e.g., learning:gradient-descent, line 16). Currently, we support the following layers: convolutional (conv), max-pooling (max-pool), average-pooling (avg-pool), fully-connected (fc), dropout (drop), and batch-normalisation (batch-norm). The following learning algorithms are supported: gradient-descent, rmsprop, and adam. Nonetheless, to extend the framework to other layer types and/or learning algorithms, we just need to add the mapping code to the utils.py file / assemble_network and/or assemble_optimiser functions, respectively. An example of the extension to new layers is shown in Section B.3. Table B.1 enumerates the parameters that are tuned for each of the considered layer types and learning algorithms.

```
<features> ::= <convolution> | <convolution>                          (1)
             | <pooling> | <pooling>                                   (2)
             | <dropout> | <batch-norm>                                (3)
<convolution> ::= layer:conv [num-filters,int,1,32,256]                (4)
             [filter-shape,int,1,2,5] [stride,int,1,1,3]               (5)
             <padding> <activation> <bias>                             (6)
<padding> ::= padding:same | padding:valid                             (7)
<classification> ::= <fully-connected> | <dropout>                     (8)
<fully-connected> ::= layer:fc <activation>                            (9)
             [num-units,int,1,128,2048 <bias>                          (10)
<bias> ::= bias:True | bias:False                                      (11)
<softmax> ::= layer:fc act:softmax num-units:2 bias:True               (12)
<learning> ::= <bp> <stop> [batch_size,int,1,50,300]                   (13)
             | <rmsprop> <stop> [batch_size,int,1,50,300]              (14)
             | <adam> <stop> [batch_size,int,1,50,300]                 (15)
<bp> ::= learning:gradient-descent [lr,float,1,0.0001,0.1]             (16)
             [momentum,float,1,0.68,0.99]                              (17)
             [decay,float,1,0.000001,0.001] <nesterov>                 (18)
<stop> ::= [early_stop,int,1,5,20]                                     (19)
```

Grammar B.1: Example of a grammar for encoding CNNs.

B.1.3 Configuration File

The configuration file is a JavaScript Object Notation (JSON) file that keeps the hyper-parameters concerned with the evolutionary engine, network structure, and training. The required parameters and their interpretation are enumerated in Table B.2. The network structure is related to Section B.1.1.

B.2 Convolutional Neural Networks for the Fashion-MNIST

To illustrate the functionalities of the Fast-DENSER framework, we will address the optimisation of the topology and learning strategy of CNNs for the Fashion-MNIST [268]: a dataset composed by grayscale fashion items of 10 independent classes. To promote the evolution of CNNs for the Fashion-MNIST dataset we use Grammar B.1, the outer level structure [(features, 1, 30), (classification), 1, 10)], with the softmax to encode the output. The learning strategy is evolved taking into account the learning production rule of the grammar.

First of all, we need to download the code from the GitHub repository and configure the environment. The requirements of the framework are detailed in Table B.3. Alternatively, we can use the docker image. To initialise the search based on the code, we execute:

```
python -m f_denser.engine -d fashion-mnist
          -c config.json -g cnn.grammar
```

Table B.1: Overview of the hyper-parameters required by each of the evolutionary units of Fast-DENSER: layers (top), and learning (bottom).

Layer Type	**Hyper-parameters**
Convolutional	Number of filters (num-filters), shape of the filters (filter-shape), stride, padding, activation function (act), bias
Pooling	Kernel size (kernel-size), stride, padding
Fully-connected	Number of units (num-units), activation function (act), bias
Dropout	Rate
Batch-normalisation	–

Learning Algorithm	**Hyper-parameters**
Gradient-descent [209]	Learning rate (lr), momentum, lr decay (decay), nesterov, batch size (batch_size), number of epochs (epochs), early stopping (early_stop)
Adam [125]	Learning rate (lr), beta1, beta2, lr decay (decay), batch size (batch_size), number of epochs (epochs), early stopping (early_stop)
RMSProp [248]	Learning rate (lr), rho, lr decay (decay), batch size (batch_size), number of epochs (epochs), early stopping (early_stop)

where -d, -c, and -g respectively set the dataset, and the paths to the configuration, and grammar files. There is another optional input parameter, -r, that sets which run we want to perform (defaults to 0). To perform the experiment using the docker image, we execute:

```
docker run -it -v $PWD/f-denser-files:/f-denser/experiments
           -w /f-denser fillassuncao/f-denser:cpu
           python f_denser.py -d fashion-mnist
                              -c example/config.json
                              -g example/cnn.grammar
```

where -it, -v, and -w are docker specific parameters that respectively set interactive mode, the shared file system, and the container work directory.

The intermediate files generated throughout evolution report the statistics of each generation and are stored, by default, in a folder called experiments (set in the config.json file). There is a sub-folder for each run. Each file keeps the information about the unique identifier of the individual, phenotype, fitness value, metrics (e.g., training and validation loss, and accuracy), number of trainable parameters, number of performed training epochs, maximum allowed training time, and performed training time. The files are formatted in JSON. The best individual found so far is stored in the best.h5 file – a model that can be loaded to Keras using the following code (also in Python):

```
from keras.models import load_model

model = load_model('best.h5')
```

which loads the topology and weights of the best-generated model. New instances can be labeled using the predict method, i.e.:

Table B.2: Example of the configuration file of the hyper-parameters required by Fast-DENSER: evolutionary (top), network (middle) and training (bottom).

Evolutionary Parameter	**Interpretation**
random_seeds	List of seeds for setting the initial random library random seeds. Used to enable reproducibility.
numpy_seeds	List of seeds for setting the initial numpy library random seeds. Used to enable reproducibility.
num_generations	Maximum number of generations of the evolutionary algorithm. Halts evolution when the maximum number of generations are performed.
lambda	Number of offspring to generate in each generation.
max_epochs	Maximum number of epochs to perform (by all trainings). Evolution is halted when the current number of epochs surpasses this value.
save_path	Path where the experiment files are saved. Following the structure save_path/run/.
add_layer	Add a new randomly generated layer mutation rate.
reuse_layer	Likelihood of reusing an existent layer.
remove_layer	Remove a layer mutation rate.
add_connection	Add a new random connection mutation rate.
remove_connection	Remove a connection mutation rate.
dsge_layer	Mutation rate of changing the DSGE genotype, i.e., the grammatical expansion possibilities.
macro_layer	Likelihood of altering the grammatical derivations of the macro structure of the network.
train_longer	Probability to increase the training time of a given individual.

Network Parameter	**Interpretation**
network_structure	Structure of the hidden layers of the network [[non-terminal, min, max], ...].
output	Output layer of the networks.
macro_structure	Macro blocks of the networks, e.g., learning, data preprocessing or data augmentation.
network_structure_init	Allowed number of layers on initialisation.
levels_back	Maximum number of levels back the layers can establish connections to.

Training Parameter	**Interpretation**
datagen	Data augmentation generator for the training data. Must be interpretable by Keras.
datagen_test	Data augmentation generator for the validation and test data. Must be interpretable by Keras.
default_train_time	Maximum default training time for each network (in seconds).
fitness_metric	Fitness assignment evaluation metric.

```
import numpy as np

label_confidences = model.predict(instance)
label = np.argmax(label_confidences)
```

B.3 Extension of the Framework

The framework allows extension by adding new layers or learning policies (Section B.3.1), or by defining new fitness metrics (Section B.3.2).

B.3.1 Add Layers and Learning Algorithms

To add new layers (or simply change the mandatory parameters), one needs to add (or adapt) the mapping from the phenotype to the Keras interpretable model. This is easily accomplished by appending new code to the utils.py file, in the "assemble_network" function of the Evaluator class (line 227). The code should be placed between the "#Create layers – ADD NEW LAYERS HERE", and "#END ADD NEW LAYERS" comments. To change the parameters of an existing layer, we just need to modify the call to the Keras layer constructor. To add new layers, we must add a new Keras layer constructor and pass parameters to it. For example, to add a Depthwise Separable 2D Convolution, we write the following code:

```
from keras.layers import SeparableConv2D

elif layer_type == 'sep-conv':
    sconv = SeparableConv2D(
                filters=int(layer_params['num-filters'][0]),
                kernel_size=int(layer_params['kernel-size'][0]),
                strides=int(layer_params['stride'][0]),
                padding=layer_params['padding'][0],
                dilation_rate=int(layer_params['dilation-rate'][0]),
                activation=layer_params['act'][0],
                use_bias=eval(layer_params['bias'][0])
            )
    layers.append(sconv)
```

To enable the use of the above layer in evolution, we need to add a new production rule to the grammar: "<separable-conv> ::= layer:sep-conv [num-filters,int,1,32,256] [kernel-size,int,1,2,5] [stride,int,1,1,3] <padding> [dilation-rate,int,1,1,3] <activation-function> <bias>".

The rationale applied to the addition of new learning algorithms is the same, with the difference that the code should be added to the "assemble_optimiser" function (line 405 of the utils.py file). For example, to enable the framework to generate ANNs trained with the Adagrad [57] learning algorithm, we would write the following code:

```
from keras.optimizers import Adagrad

elif learning['learning'] == 'adagrad':
    return Adagrad(lr=float(learning['lr']),
                   decay=float(learning['decay']))
```

The following production rule would be added to the grammar: "<adagrad> ::= learning:adagrad [lr,float,1,0.0001,0.1] [decay,float,1,0.000001,0.001]".

Table B.3: Fast-DENSER code metadata.

Nr.	Code Metadata	Metada Value
C1	Current code version	v2.1.0
C2	Permanent link to code/repository used for this code version	`https://github.com/fillassuncao/fast-denser3`
C3	Legal Code License	Apache License, 2.0
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Python3.7
C6	Compilation requirements, operating environments & dependencies	CUDA $\geq$ 10, CuDNN $\geq$ 7, tensorflow (with GPU support), keras, scipy, sklearn, jsmin, Pillow
C7	If available Link to developer documentation/manual	`https://github.com/fillassuncao/fast-denser3`
C8	Support e-mail for questions	fga@dei.uc.pt

B.3.2 Add Fitness Metrics

The creation of new fitness functions is similar to the instantiation of new evolutionary units. We need to create the necessary code and add it to the fitness_metrics.py file. Currently, it supports the accuracy and Mean Squared Error (MSE). For example, to add the RMSE, we add the following code:

```
def rmse(y_true, y_pred):
    from math import sqrt
    return sqrt(mse(y_true, y_pred))
```

After adding the RMSE function, we can set the fitness_metric parameter of the configuration file to rmse.

B.4 Impact

Fast-DENSER is a framework that promotes the automatic generation of DANNs and thus avoids the user the burden of having to manually optimise a network to solve a specific problem. It enables non-expert users to consider ANNs in their domains and aids expert users in tuning their networks, possibly obtaining solutions that they would usually not think of.

The framework is easy to use. The parameters are defined in a human-readable format, and the output is a fully-trained DANN, that can be deployed right-off evolution. This is an advantage compared to the majority of other NE frameworks. Previous approaches tend to evaluate the candidate solutions for a limited amount of time, and thus require further training by the end of the evolutionary search. This is a barrier to non-expert users. In addition, according to Baldominos et al. [24], Fast-DENSER uses an interesting representation scheme and is categorised as a settlement approach, i.e., a stable work that has proved to effectively evolve DANNs.

Despite considering only a set of layers and learning algorithms, the framework can be easily extended. The core is kept the same independently of the considered evolutionary units. The user just needs to add the code to map between the grammar and the Keras model. This is a simple parsing routine, which is similar to the mappings that are already in the code.

The framework has been widely tested and debugged and has led to the generation of DANNs to numerous object recognition benchmarks (Chapter 5). Further, in the physics domain, it has helped to find models that improve by a factor of 2 the gamma/hadron detection based on the ground impact patterns.

Current Code Version

The details on the last code version and requirements are enumerated in Table B.3.