UNIVERSIDADE D
COIMBRA

Pedro Filipe Gomes Ramos de Carvalho

EVOLVING LEARNING RATE SCHEDULERS

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Nuno Lourenço & Penousal Machado and presented to
Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2020

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Evolving Learning Rate Schedulers

Pedro Filipe Gomes Ramos de Carvalho

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Professor Nuno Lourenço & Professor Fernando Penousal Machado and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2020

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Abstract

The choice of a proper learning rate is paramount for good Artificial Neural Network training and performance. Currently, a plethora of state of the art automatic methods exist that make the search for a good learning rate easier, called Learning Rate Optimizers. While these techniques are effective and have yielded good results over the years, they are general solution i.e. they do not take into account the characteristics of a specific network.

As a result, the possible benefits of the optimization of learning rate for specific network topologies remains largely unexplored. Since neural networks are complex systems with many interdependent components it is not possible for humans to infer how an optimizer can be specialized for a certain network topology. Nevertheless, heuristic optimization techniques such as Evolutionary Algorithms can be used to search for custom optimizers that work well for specific network architectures.

In this work we propose AutoLR, a framework that uses Structured Grammatical Evolution to evolve learning rate optimizers. Two versions of this system are implemented for different types of optimizers. Dynamic AutoLR (DLR) is used to evolve static and dynamic learning rate optimizers. The best evolved dynamic optimizer outperforms the established baseline and utilizes some techniques found in the literature. Even though DLR achieved good results the optimizers evolved by this system only take into account the previous learning rate and current training epoch. In order to overcome these limitations we devised a new method called Adaptive AutoLR (ALR). This version of the system evolves adaptive optimizers that have access to more information about training. These optimizers are able to fine tune a different learning rate for each network weight which makes them generally more effective. The most notable evolved adaptive optimizer is able to perform on par with the best state of the art methods, even outperforming them in some scenarios. Furthermore, the system was able to automatically discover a novel optimizer, ADES. To the best of our knowledge, no adaptive optimizers present in the literature are similar to ADES.

Keywords

Artificial Neural Network; Structured Grammatical Evolution; Learning Rate Optimization

This page is intentionally left blank.

Resumo

A escolha de uma boa taxa de aprendizagem é fulcral para um bom treino e performance de Redes Neuronais Artificias. Atualmente, existem vários métodos automáticos que facilitam a busca por uma boa taxa de aprendizagem, e apesar destas técnicas serem eficazes e produzirem bons resultados ao longo dos anos, estas são soluções generalistas e não tem em conta as características de uma rede especifica.

Desta forma, os possíveis benefícios de otimizar a taxa de aprendizagem para uma topologia de rede especifica permanecem inexplorados. Como as redes neuronais artificiais são sistemas complexos com muitos componentes interdependentes, não é possível para um humano inferir como é que um otimizador pode ser especializado para uma certa topologia. Apesar disso, técnicas de otimização heurística como Algoritmos Evolucionários podem ser utilizados para procurar otimizadores personalizados que funcionem bem para uma arquitetura de rede neuronal especifica.

Neste trabalho propomos o AutoLR, um sistema que utiliza Evolução Gramatical Estruturada para evoluir otimizadores de taxas de aprendizagem. Em concreto, são implementadas duas versões deste sistema para dois tipos de otimizadores. O AutoLR Dinâmico (DLR) é utilizado para evoluir otimizadores estáticos e dinâmicos. O melhor otimizador dinâmico evoluído tem melhor performance que o otimizador de controlo estabelecido e utiliza algumas técnicas encontradas na literatura. Apesar de o DLR atingir bons resultados, os otimizadores evoluídos por este sistema só tem em consideração a taxa de aprendizagem anterior e a época de treino atual. De modo a superar estas limitações desenvolvemos um novo método chamado AutoLR Adaptativo (ALR). Esta versão do sistema evolui otimizadores adaptativos que têm acesso a mais informação sobre o treino. Estes otimizadores são capazes de afinar a taxa de aprendizagem para cada peso da rede individualmente, o que os torna mais eficazes. O otimizador adaptativo evoluído mais notável é capaz de competir com os melhores métodos do estado da arte, conseguindo até superá-los em alguns casos. Por último, o sistema foi capaz de descobrir um novo otimizador, ADES. Tanto quanto sabemos não existem otimizadores adaptativos na literatura que sejam semelhantes ao ADES.

Palavras-Chave

Redes Neuronais Artificiais; Evolução Gramatical Estruturada; Optimização da Taxa de Aprendizagem

This page is intentionally left blank.

Acknowledgements

To my advisors, Professor Nuno and Professor Penousal, thank you for your continued guidance and advice. I would not have been able to present a work that I am proud of were it not for your availability and patience. You have given opportunities that I will always be grateful for.

Leonardo, Guilherme, Artur and Tiago, thank you for being with me every day in these trying times. It was your support, company and friendship that kept me happy and motivated during the endless hours spent in quarantine.

Thank you to my parents for always caring for me and being there for me during all of my hardships. It is your unconditional support that kept the self doubt and stress away during the hardest time.

Finally, I want to thank all of the close friends that I have made over the year. There are too many of you to name but know that I have learned much from all of you and I am very grateful. Thank you for making me the man I am today.

Contents

This page is intentionally left blank.

Acronyms

Adaptive AutoLR Implementation of the AutoLR framework for adaptive optimizers. xiii, 35, 39, 41–43

AI Artificial Intelligence. 1, 2

ANN Artificial Neural Network. 1–3, 5–7, 10–12, 18, 24, 26–28, 42, 44

CNN Convolutional Neural Network. xiii, 7, 26–28

DENSER Deep Evolutionary Network Structured Representation. 27, 30, 44, 48, 49

DNN Deep Neural Network. 6, 7

Dynamic AutoLR Implementation of the AutoLR framework for dynamic and static optimizers. 23, 35, 38–40, 42, 43

EA Evolutionary Algorithm. 2, 8–10, 25, 26

GE Grammatical Evolution. 9, 10, 17

LR Learning Rate. 1, 6, 10, 11

SGE Structured Grammatical Evolution. xiii, 10, 17–19, 27–29, 37, 42

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

The study of Artificial Neural Networks (ANN) is a field in modern Artificial Intelligence (AI). These systems have become renowned for their impressive results in many difficult tasks such as computer vision [3, 4], games [5, 6], medicine [7] and natural language processing [8, 9]. ANNs are inspired by the functioning of the human brain. Subsequently they are comprised of small units called neurons (or nodes) that are interconnected. Each of these connections has an internal value known as a **weight**. **Training** is the process through which the correct weights for a given problem are found. Fine tuning an ANN's weights is crucial in order to obtain a consistently useful system and there are several parameters that regulate the training as a result. One such parameter is the **learning rate** (LR).

The learning rate determines the magnitude of the changes that are made to the weights. As a consequence, choosing an adequate learning rate is paramount for effective training and desirable results. If the value utilized is too low the network will be unable to make impactful changes to its weights, making the training process excessively slow. On the other hand, if the LR is too high the system will make radical changes even in response to small mistakes, causing inconsistent and unpredictable results. On top of this, research suggests that the best training results are achieved by adjusting the LR over the course of the training process [10].

The learning rate is such an important piece in an ANN's performance that there is extensive research in the optimization of this one parameter. Modern learning rate optimization solutions (commonly abbreviated to **optimizers**) are varied in their approach, complexity and effectiveness. One aspect all of these methods have in common is their generality. Current learning rate optimization solutions can be applied to any network and problem, given that they go through a typical training process. This versatility comes up naturally since there is a large number of network architectures that are used for many different problems. As a result, the most useful optimizers are the ones that can be used in any context and still perform well. But this does not mean there is no interest pushing the limits of performance in specific problems and networks. In fact, research into ANN frequently focuses on techniques that can be used to obtain the best possible performance in a specific problem domain.

So this raises the question: can we improve our optimizers by specializing them for a certain ANN?

Since ANNs are comprised of many interdependent components and parameters [11] it is not possible for a human to accurately assess how a learning rate optimizer can be specified

for a certain network architecture. As a result, one must rely on an algorithm that will search for the optimizers best suited to the network. There is a class of algorithm known as Evolutionary Algorithms (EA) that are fit for this task. These are heuristic optimization algorithms that utilize procedures inspired by biological evolution (e.g. mutation, natural selection) to find good solutions to difficult problems in an acceptable time frame. EA work by testing a large number of solutions and combining the best performing ones in order to produce progressively better results. In this context, an EA can be used to search for optimizers that perform well on a specific ANN. The network's performance with the evolved optimizer can then be compared to the performance with a state of the art standard optimizer to assess the benefits of specialization.

In this work we explore the possibilities, benefits and limitations of bringing an evolutionary approach to the field of learning rate optimization.

1.1 Motivation

There are two prime motivators behind this work.

First, the novelty of evolving learning rate optimizers creates possible topics for future research. Although the approach used in this work is justified and adequate, other researchers may be able to learn from our results and bring their own ideas to this topic. Additionally, the results of this work may raise other worthwhile questions that can motivate further research question such as: 1. Are the evolved optimizers different from the man made ones? 2. Can we learn something new from the optimizers created through evolution? 3. Do the evolved solutions show us a way to specialize optimizers without going through the evolutionary process?.

The potential for future research is a key motivator behind this work.

Additionally, if the evolved optimizers show a significantly better performance than the current state of the art, the system created may be valuable as a stand alone optimization tool for AI problems. This is another reason to explore this topic, contributions in learning rate optimization are valuable beyond the field due to the importance of the learning rate in neural network performance as a whole.

Our objectives are naturally connected with these motivators. The primary objective of this work is to answer the question: **"is evolving learning rate optimizers for specific neural network architectures a valid means of improving their performance?"**. This topic is worthwhile for several reasons. The learning rate optimizers we have had contact with during our research aim to create optimizers that are useful across many network architectures. This general approach has contributed a lot to the community but the possible benefits of a system that creates optimizers for specific architectures still remain unexplored. In this work we are studying the viability of this approach and how it may contribute to the field of learning rate optimization as a whole.

The use of such an approach also has the possibility of answering an array of secondary questions. Firstly **'are evolved learning rate optimizers similar to the manually designed ones?"**. Learning rate optimizers can take many different shapes, it will be notable if our system is capable of automatically discovering algorithms that are variations of the ones found in the literature. Such a result is interesting because if this approach is able to evolve solutions that are widely accepted as high quality it is possible that these same ideas can be used to find still undiscovered, better methods. This give way to a second, possibly more interesting question: **"do the evolved learning rate optimizers**

only perform well on the network they were evolved in?". Current learning rate optimizers are general, i. e. the optimizer is not affected by the architecture of the network it is used on. It is possible that results suggest that the specific neural network used during evolution plays an important role in the optimizer generated. While this is not useful per se it suggests that the creation of a system that can infer specialized optimizers from an ANN's characteristics is possible. Such a system would be a valuable contribution to the field if it could outperform the current general optimizers consistently.

1.2 Contributions

This works resulted in several contribution, which are summarized below.

1. AutoLR, a framework that can evolve learning rate optimizers for specific neural network architectures. AutoLR allows for the evolution of several types of learning rate optimizers and this work presents two versions of this system.

2. Dynamic AutoLR (DLR) is an implementation of the framework focused on evolving learning rate schedulers. This proposal was accepted as a full paper in GECCO 2020, the premier conference for genetic and evolutionary computation. The full paper is included in Appendix B.

3. Adaptive AutoLR (ALR), is being presented for the first time in this work. This version of the system is able to evolve variations of known state of the art adaptive optimizers as well as new solution of similar efficacy.

4. A new type of adaptive optimizer called Adaptive Evolutionary Squared Optimizer (ADES) is also proposed. This optimizer is based on some notable individuals evolved while using ALR. Results suggest that ADES is competitive with known state of the art optimizers.

The code for all versions of the framework and ADES will be open source at a later date.

1.3 Structure

Chapter 2 of this document provides the necessary **Context** for the reader to understand the rest of the work and its relevance in the contemporary landscape of the field. It will elaborate on the topics of *Evolutionary Algorithms* and *Artificial Neural Networks* as well as present the state of the art approaches to *Learning Rate Optimization*.

Chapter 3 presents **AutoLR**, the framework built to run our experiments. This chapter does a high level description of the system's components and why they are important to our hypothesis.

Chapter 4 and 5 describe the two implemented versions of the AutoLR framework: Dynamic AutoLR and Adaptive AutoLR respectively. Each chapter presents its version's *Grammar*, *Fitness Function* and other important implementation details. The reasoning behind our design decisions are also explained, presenting the benefits and limitations of our choices and contrasting them with some alternatives. Furthermore these chapters present the *Experimental Setup* that was performed with these systems as well as the *Results* achieved.

The last chapter, Chapter 6, relates our results to the hypothesis posed at the start and draws a **Conclusion** of the work as a whole. This chapter also looks into possibilities for *Future Work* that come from this work.

Chapter 2

Context

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a machine learning approach the design of which draws from the Biological Neural Networks seen in nature in order to create a computing system that is able to learn. Figure 2.1 shows a graphical depiction of an ANN.

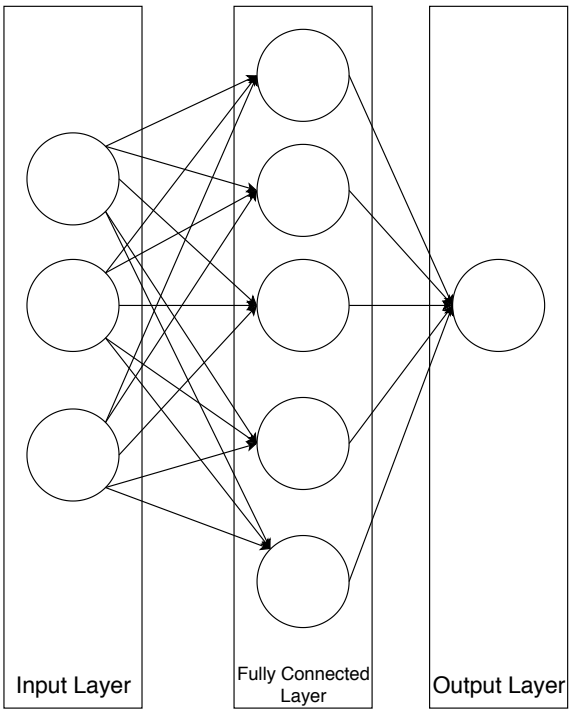


Figure 2.1: Graphical depiction of a Artificial Neural Network.

While these networks can have different architectures (e.g. LSTM [8], ResNet [12]) we will start by focusing on multi-layer feed-forward models (MFF) [13] as they are the most simple [14] and relevant to this work. MFFs are comprised by a set of nodes and edges. The nodes are grouped into separate layers connected sequentially. These layers are flanked by an input and output layer which are responsible for receiving the data that the network will process, and yield the result of the network's calculations respectively. Edges are a directed connection between two nodes from different layers and are the means through which information travels through the network. Each node performs an operation (e.g. a

mathematical function) on the values it receives from the previous layer and sends the new value to all nodes it is connected to in the next layer. Every edge has a weight that scales the value it carries, i.e. the value a node outputs is always adjusted before it is provided to the nodes in the next layer.

While we chose to focus on this type of network it should be noted that the ideas presented here are applicable to any network as long as the back-propagation training algorithm [15] is used. Back-propagation is the most common training algorithm and a centerpiece of this work. In order to understand many of our conclusions and insights throughout the document it is important that the reader has a grasp on this algorithm.

Back-propagation is a class of algorithm that is able to quickly compute the gradient of a function for a pair of input and output. In other words when given an input, a function and expected output, back-propagation algorithms are able to calculate the direction in which the input should be changed in order to approach the expected output (gradient). These algorithms are utilized in ANNs to quickly assess the changes that should be made to the weights after each training example. In this context back-propagation is going to take the network's expected output and determine how each of the network's weight should be changed to approximate this value.

In this case we are calculating the gradient of the loss function. This is possible due to the chain rule. The chain rule is a formula that can calculate the gradient for variables that depend on other variables. The output of an ANN depends on the result of the output layer which in turn depends on the layer before that and so on until the input layer is reached. The chain rule and back-propagation algorithm can be used together to calculate the gradient for each weight in a neural network. The gradient is the key to training since it gives the network a way to improve itself. It must be noted that back-propagation is only responsible for calculating the gradient. The reason learning rate optimization is so important is because training effectiveness varies greatly based on what is done with the gradient once it is calculated.

For an in-depth explanation of back-propagation and chain rule we direct the reader to Chapter 6.5 in [16].

ANNs can be used to solve tasks of many different types of problems such as classification [17] and regression [18]. In this work we will be experimenting on supervise learning classification problems and we will focus on this topic as a result.

In this particular problem the system is tasked with learning a function that can categorize data instances into their respective classes. To achieve this the network is provided a set of example and their classifications.

The ANN is then trained using this example set. Training is an iterative process where the network compares its attempted classifications of a subset of examples with the expected ones and adjusts its weights to get closer to the correct results. There is a function that compares the classification and measures how incorrect the network's output was, this is known as the **loss function**. The size of the changes made to the weights is partially given by the error returned by the loss function (a larger error leads to larger changes). There is also a numerical parameter that determines magnitude of the adjustments made. This parameter is called the **Learning Rate** (LR) and it is one of the main subjects of this work.

Deep Neural Networks (DNN) are a subset of ANN notable for having multiple *hidden layers* between the input and output ones. This added complexity in the network's topology allows it to solve harder problems. While it is important to note that increasing the number

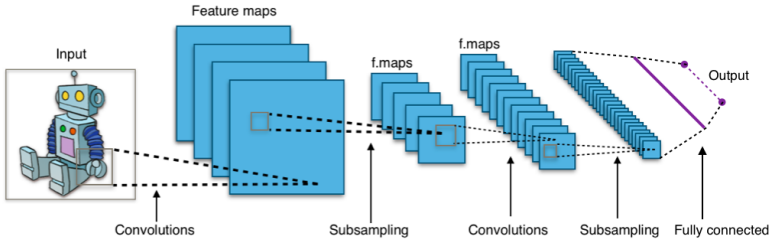


Figure 2.2: Representation of a Convolutional Neural Network [1].

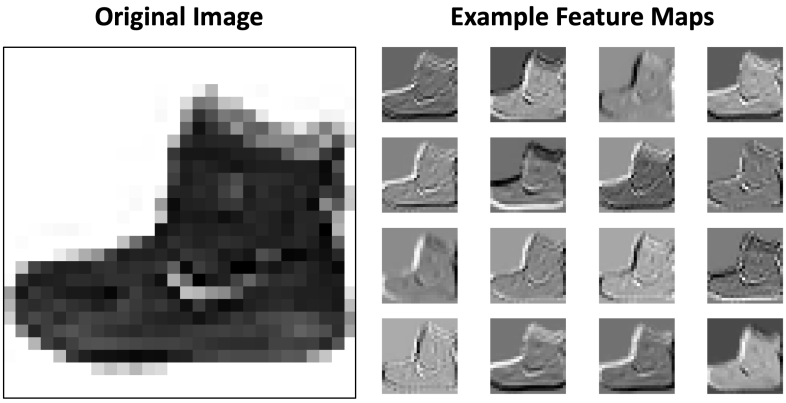


Figure 2.3: Example of feature maps after training [2].

of layers does not solve harder problems per se, there are certain problems that cannot be solved without the added power of a DNN. DNNs can be further specialized for certain tasks by selecting the operations used in each layer's nodes.

A Convolutional Neural Network (CNN) [19] is a type of DNN that is designed to perform better on image related problems. Using image as an input to a neural network comes with certain challenges that make the use of a specific architecture desirable. Pixels in an image only yield valuable visual information when in the context of their neighbours. This means that having a sequence of fully connected layers would result in many useless edges as their weights would be attempting to learn the connection between pixels that are not related (e.g. pixels in separate corners of an image). Ideally, we want our system to be able to find patterns in small regions and connect those patterns into progressively more complex shapes. A representation of this type of architecture can be found in Figure 2.2.

CNNs achieve this through the use of their namesake operation, **convolution**. Convolutional layers break down their input and pass it through a series of filters, each filter corresponds to a pattern (e.g. a corner). This layer outputs a measure of how well each region of the input matches the filter (known as a **feature map**). An example of what these feature maps look like can be seen in Figure 2.3. The information output by a sequence of these layers is clearer than the original input meaning that if we feed it into fully connected layers there will typically be few meaningless weights.

For a more profound explanation of ANN see [20].

2.2 Evolutionary Algorithms

Evolutionary Algorithms [21] are a type of optimization algorithm inspired by the biological evolution. These algorithms attempt to emulate its processes to perform metaheuristic optimization. This means that EA aim to explore possible solutions in an efficient, stochastic way, with no guarantee that they will ever find the best possible solution to the problem at hand. Practically speaking, these implementations are able to find near-optimal solutions with a fraction of the resources necessary for a deterministic approach, given the right conditions. An example of a generic evolutionary algorithm can be found in Algorithm 1.

Algorithm 1: Generic Evolutionary Algorithm Implementation

```
params: population_size, num_generations, crossover_chance, mutation_chance
population = generate_initial_population(population_size);
generation = 0;
while generation < num_generation do
    new_population = [];
    new_individual_counter = 0;
    while new_individual_countar < population_size do
        new_individual = selection_method();
        if random() < crossover_chance then
            parent = selection_method();
            new_individual = crossover(new_individual, parent);
        if random() < mutation_chance then
            new_individual = mutation(new_individual);
        new_individual.fitness = evaluate(new_individual);
        new_individual_counter += 1;
        new_population.append(new_individual);
    generation += 1;
```

The optimization starts with the creation of a population of individuals. This first generation of individuals is typically generated randomly but more sophisticated initialization methods exist that can be beneficial in certain conditions [22].

Each individual of the population has genetic information that can be translated into a solution to the problem at hand. The structure of this information must be chosen by the user and it is called **genetic representation** (in most EAs, representation is separated into genotype and phenotype but this distinction will be temporarily omitted for a clearer explanation). A good choice of representation is characterized by high locality, meaning that the difference in individuals' performance should grow larger as their representations diverge.

An individual's performance is measured by the **fitness function**. This function's design is another important decision that must be made in the creation of an EA. Its role is to take an individual and quantitatively evaluate how well it solves the proposed problem. These fitness values are used for direct comparison so it is important that they are numerical and varied as it is beneficial that fitness values are shared by the smallest number of individuals possible.

Following in the footsteps of biology, the next step is to emulate the natural mechanisms that cause populations to evolve to maximize fitness. In this case the processes we will be replicating are **selection**, **recombination** and **mutation**.

Selection is the process through which the individuals that are allowed to reproduce are chosen. It is desirable that this process is stochastic in nature with some bias towards the fittest individuals. This means that if only selection is used the average fitness of the population should increase until it stagnates. It is important that the selection method used is not too top heavy as this rapidly reduces diversity which will be detrimental to the operators that follow.

It is important to understand that EA utilize a set of probabilities to determine what occurs in the evolutionary process. While it is possible that an individual is selected to pass all of its genetic material (effectively creating a copy of the individual in the next generation) most of the new generation will be comprised of individuals that are an hybrid between two chosen individuals. The process that creates these hybrids (or children) is known as **recombination** or **crossover**. The **crossover probability** is therefore the chance that an individual in a new generation was created by performing a crossover on two individuals of the old generation (or parents) chosen with the selection method in use. The function (also known as an operator) used to perform a crossover must be chosen with the representation in mind as it will be tasked with creating a child using genetic information from both parents. These functions are also stochastic, meaning that performing a crossover on the same couple will not yield the same child every time. The restrictions imposed on the crossover function are also an important decision, it can be beneficial to allow the creation of individuals that are not valid solutions to the problem in the hopes that a subset of the information present in the invalid solution might be useful for the evolutionary process. If this idea is pursued there must be a **penalty** mechanism in the fitness function that can rate these invalid solutions as they cannot be applied to the problem directly. The alternatives to the penalty system are **discarding invalid solutions** (replacing the individual until a valid one is produced) or creating a complementary **repair function** that can fix the invalid solutions.

The last changes made to the population before the generation can be evaluated are made through **mutation**. The reasoning behind mutation is that as evolution progresses it is likely that the population can only be improved further through small changes to their genetic information. Crossover is rarely able to achieve this as most crossover functions are destructive (meaning that the child is unlike either parent) so a new mechanism is necessary to make these small adjustments. We can deduce from this that the function used for mutation takes an individual and makes a small change to its genetic information. If the operator is allowed to create invalid individuals it is necessary that the system is prepared to handle them as was the case with the crossover (with the techniques described above). Finally, mutation is not only applied randomly (using a **mutation probability**) but the process itself should be random, meaning that applying the mutation operator on the same individual should not always produce the same result.

The last mechanism utilized is not inspired directly by biology but it has proved to be a helpful complement to these algorithms. This mechanism is **elitism** and it guarantees that the best n individuals bypass the other mechanisms and are automatically present in the next generation. This can be beneficial as it guarantees that the best genetic material is always passed to the next generation but in some situations it can also speed up stagnation which hurts the end result.

Grammatical Evolution (GE) [23] is a type of EA that has the specific purpose of evolving a program that performs a certain task. Additionally, GE stands out as its representation is split into a series of integers (genotype) that is mapped into executable code through a grammar for evaluation (phenotype). While the use of different structures for genotype and phenotype is common, the level of separation between these specific structures is enough

that it is possible to work the genotype in ways that are not possible in the more common representations of code (such as trees). This means that it is hard to represent a program in a way that can be effectively manipulated by genetic operators and GE is a solution to that problem.

This representation is not without issues, however. Since the genotype is not as intimately connected to the phenotype, this approach suffers from poor **locality**[24, 25]. SGE [26, 27] is a variant of GE that uses an altered genotype representation to address this issue. As opposed to using a single series of integers, SGE uses one such series for each non-terminal rule in the grammar. This address the issue of the traditional GE representation by associating parts of the genetic information with specific grammar rules, connecting the genotype and phenotype without losing the benefits of a numerical representation.

However, these approaches add another layer of decision making, namely in the form of the grammar design [24]. The grammar used for any GE experiment will define what kind of programs the engine is able to create and this has many implicit consequences. The most obvious one is that the grammar provided must be able to generate the program desired. While this seems trivial it must be understood that not knowing the composition of the desired program is one of the main motivations to use this type of system in the first place. This also means, however, that the grammar specificity can be increased as more knowledge of the problem is available, aiding the search process.

This specific type of EA is suited for this work as the functions we are looking to evolve are very specific. This means that our domain knowledge is high, there is a strong understanding of what our desired program is like in advance. As previously mentioned we can use this knowledge to create a grammar that enhances results by narrowing the search space. Another advantage in this approach is that it is not difficult to extend grammars to incorporate new information about the problem. This is valuable in our case as the state of the art approaches to our problem are varied in the information they use. An in-depth explanation of these algorithms can be found in [20].

2.3 Learning Rate Optimization

In the context of this work, **hyperparameters** are the set of parameters that configure an ANN and its training. The **learning rate** (LR) is one such parameter and its role is to scale the changes made to a network's weights during training. Research suggests that hyperparameter optimization is an effective way of improving a system's performance without increasing complexity [28].

Before we discuss existing approaches to learning rate optimization it is import we first understand exactly how the learning rate works and why it is important. The term learning rate is frequently used to describe a value used in several optimization methods. The most common optimizer is stochastic gradient descent (SGD) and most mentions of the learning rate refer to its use in SGD.

As was discussed in Section 2.1, when a neural network is being trained its weights are adjusted in response to the gradient of the loss function. SGD is the most common solution used to make these adjustments. SGD changes the weights through the formula seen in Equation 2.1 where:

- w_{t-1} is the value of the weight before adjustment.
- w_t is the weight value after adjustment.

- lr is the learning rate being used.
- $\nabla l(w_{t-1})$ is the gradient of the loss function for w_{t-1}.

$$w_t = w_{t-1} - \text{lr} * \nabla l(w_{t-1}) \tag{2.1}$$

Equation 2.1: Stochastic Gradient Descent optimizer weight update rule

The learning rate is important in this context because using the gradient directly will typically cause the network to overshoot the desired adjustment. By using a learning rate value between 0 and 1 the network is able to make small, directed adjustments to its weights without over correcting. The learning rate chosen will directly affect the network's final performance. Researchers have as a result developed many systems that attempt to make the most of the learning rate.

The traditional approach is to use a single learning rate for the entire training process. Under these circumstances all optimizing must be done before training proper even starts. Oftentimes the programmer must rely on their expertise in order to *guess* adequate learning rate values. While automatic solutions to this problem exist they are, to the best of our knowledge, either comparable to manual optimization [28] or non-trivial in implementation [29]. Much of the difficulty of finding a convenient solution to this is issue stems from the fact that hyperparameters are inter-dependent [11]. This means that even if an ideal learning rate is found there is no guarantee that this value remains optimal (or even usable) as the other parameters are tweaked.

These nuances make the use of a static learning rate a possible drawback. It is desirable that the method we are using to determine our learning rate is robust enough that performance does not dip with every change to the system. In order to increase flexibility we would ideally have a method to change the LR as training progresses as this means that even if the initial value is not adequate the system has a chance to correct its course. This strategy will be referred to as a **dynamic learning rate**. The most uncomplicated policy for varying the learning rate can be inferred intuitively. It is expected that as training progresses the ANN's performance gradually improves as it gets better at solving the task at hand. If the system is potentially closer to its objective it seems desirable that it does not stray from its course. In order to improve, the network requires progressively finer tuning, this can be achieved with a **decaying learning rate** (meaning that the LR decreases as learning progresses). There are some issues that are frequently encountered during training that make this approach not ideal however. Better performance is rarely and indicator that the network is closer to a perfect solution. There is a concept known as "local optimum" which refers to a solution to an optimization problem that is the best but only within a region smaller that the entire problem space. Using a decaying learning rate leaves the system susceptible to early stagnation in a local optimum. This is not ideal despite the fact that a local optimum can be sufficient for most situations as this approach can lead to early stagnation if applied incorrectly. Despite these limiting factors decaying learning rates can lead to improvement over static ones as seen in [10].

In order to expand on these ideas we need to apply the concepts of **exploration** and **exploitation**. These refer to the two complementary strategies that can be used in heuristic optimization. Exploration is the idea of using a mechanism that helps the algorithm *explore* solutions that do not seem as promising in an attempt to avoid falling into a local optimum. The contrasting technique is exploitation, in this strategy we adjust our approach to make sure the algorithm is able to find the local optimum (once it reaches a promising region).

Finding a proper balance between these two strategies is crucial for further improvement of the dynamic learning rate. Smith et. al. propose the use of a *cyclical learning rate* in [30]. Their approach fluctuates the learning rate between a maximum and a minimum bound. While the system uses no information about whether or not it is stuck by periodically increasing the learning rate it is able to explore the search space more effectively. This technique is consequently less vulnerable to early stagnation than decaying learning rate policies. This method is, to the best of our knowledge, the most efficient use of dynamic learning rates.

2.4 Adaptive Optimizers

Further improvements in training effectiveness can still be achieved by utilizing more sophisticated optimizers.

So far we have been working with a single value learning rate that affects all weights in the same way. Not all weights are created equal however. It can be beneficial for different weights to have their gradient scaled by different factors. Consider the following scenario, an ANN is being trained for 100 generations with a single value learning rate. One specific weight of the network reaches a near optimal value within the first 5 generations, but all of the others are still off the mark. There is no one ideal learning rate in this situation. On one hand, using a small learning rate will benefit the fine tuning of the node that is already performing well. A larger learning rate, on the contrary, will allow the sub optimal weights to find better values. The optimizers we will be discussing are able to adapt changes made to a weight based on how its gradient evolves. Ruder provides a great overview of all gradient descent optimizers in [31].

Notation and nomenclature is not consistent throughout the literature. Since later on in this work we will be comparing the form of these optimizers we establish a notation that will be used to translate all of these algorithms where:

- w_{t-1} is the weight value before being updated and w_t is the weight value after update.
- The weight update function is always the last equation and it takes the form of $w_t = w_{t-1} - ...$
- lr is the learning rate being used.
- $\nabla l(weight)$ is the gradient of the loss function for *weight*. This will commonly appear as $\nabla l(w_{t-1})$ which is the value of the gradient for the weight before update.
- x, y, z are discrete variables that are updated by their own functions along with the weights. The meaning of these variables changes from optimizer to optimizer and will be mentioned in the sections where they appear. When unspecified these variables are initialized at 0.
- Any hyperparameters will appear with the notation used in the original paper whenever possible and they will be explained as well.

2.4.1 Momentum

Momentum optimizers are a variation of SGD proposed by Rumelhart et. al. in [15]. In this optimizer the changes made to the weight are informed not only by the gradient (as

done by SGD) but also by the last update. This means that if the optimizer keeps making changes in the same direction it will build up speed and start making bigger updates. If the opposite happens and the direction of the gradient keeps changing then the updates will make gradually smaller. This method was shown to perform better than standard SGD in [32]. The weight update rule for momentum optimization can be seen in Equation 2.2 where x is the variable that stores the previous update and mom is a value between 0 and 1 used to scale the contribution of the previous change. $x_{t-1} * mom$ is commonly refer to as the "momentum term".

$$\begin{aligned} x_t &= \text{mom} * x_{t-1} - \text{lr} * \nabla l(w_{t-1}) \\ w_t &= w_{t-1} + x_t \end{aligned} \tag{2.2}$$

Equation 2.2: Momentum optimizer weight update rule

In 1983, Nesterov proposed an accelerated method of solving convex programming problems in [33]. While his work did not directly discuss the application of the method to stochastic gradient descent this approach would later be used by Sutskever et. al. in [34] to create what would come to be known as *Nesterov momentum*. Nesterov momentum is different from the standard implementation as the gradient is calculated for $w_{t-1}+mom*x_{t-1}$. This means the gradient is effectively looking ahead to the value suggested by the momentum term. The reason this is beneficial is because the momentum term can be slow to change and cause incorrect changes as a result. With this approach the gradient is able to make corrections in case there are problems with the direction suggested by the momentum term.

$$\begin{aligned} x_t &= \text{mom} * x_{t-1} - \text{lr} * \nabla l(w_{t-1} + mom * x_{t-1}) \\ w_t &= w_{t-1} + x_t \end{aligned} \tag{2.3}$$

Equation 2.3: Nesterov Momentum optimizer weight update rule

2.4.2 Adagrad

Adagrad is an optimizer proposed by Duchi et. al. in [35]. This algorithm explores the idea of gradually reducing the speed at which each weight is updated based on the changes made so far. To contrast with the more simple decay policies presented in the previous section, Adagrad uses the L2 norm of previous gradients as a measure of how much the weight has been changed. This accumulator increases indefinitely as training progresses. The learning rate is then divided by this value, meaning that weights that have suffered more changes will update slower and vice versa. This is desirable for the reasons explained previously. To reiterate, as training progresses it is expected that the weights will be closer to their ideal values and it makes sense to decrease the learning rate as a result. This method is different from a simple decay policy because it is able to measure how much training has progressed for each weight individually.

Seen in Equation 2.4 is the algorithm for this optimizer where x stores the value of the L2 norm for the current iteration and δ is a non negative value used to initialize x.

This approach has some limitations due to its irreversible nature. Since the accumulator is always incremented by the square of the gradient the value of x will never decrease. If a weight has high gradient at the start of training that will slow its progress down throughout the rest of the process and this may be detrimental to optimization.

$$
\begin{aligned}
x_0 &= \delta \\
x_t &= x_{t-1} + \nabla l(w_{t-1})^2 \\
w_t &= w_{t-1} - \frac{lr * \nabla l(w_{t-1})}{\sqrt{x_t} + \epsilon}
\end{aligned}
\tag{2.4}
$$

Equation 2.4: Adagrad optimizer weight update rule

2.4.3 RMSprop

RMSprop is an unpublished [36] variation of Adagrad that replaces the accumulator used to scale the learning rate with a moving discounted average. This change means that the denominator seen in Adagrad no longer grows indefinitely. This retains the benefits of Adagrad whilst removing the irreversible nature of learning rate decay present in that optimizer. Seen in Equation 2.5 is the algorithm, the only changes from Equation 2.4 is that x is now taking on the role of a moving discounted average.

$$
\begin{aligned}
x_t &= \rho x_{t-1} + (1-\rho)\nabla l(w_{t-1})^2 \\
w_t &= w_{t-1} - \frac{lr * \nabla l(w_{t-1})}{\sqrt{x_t} + \epsilon}
\end{aligned}
\tag{2.5}
$$

Equation 2.5: RMSprop optimizer weight update rule

This optimizer can be extended to utilize momentum as well. See Equation 2.6 and Equation 2.7 for the algorithm using standard and Nesterov momentum respectively. A new variable y is created to serve as the momentum term along with a decay constant between 0 and 1 ρ that is used to prevent indefinite growth.

$$
\begin{aligned}
x_t &= \rho x_{t-1} + (1-\rho)\nabla l(w_{t-1})^2 \\
y_t &= mom * y_{t-1} + \frac{\nabla l(w_{t-1})}{\sqrt{x_t + \epsilon}} \\
w_t &= w_{t-1} - lr * y_t
\end{aligned}
\tag{2.6}
$$

Equation 2.6: RMSprop optimizer with momentum weight update rule

2.4.4 Adadelta

Adadelta is another variation of Adagrad that seeks to improve upon it in two ways: avoid the indefinite reduction of the learning rate (due to the accumulator) and avoid the need for manual selection of a general learning rate. Much like RMSprop, Adadelta also changes the accumulator to a moving discounted average. This optimizer also replaces the learning rate with the moving discounted average of all weight updates.

In order to understand why this is a viable approach we must take a step back and discuss SGD as a whole. The reason we utilize a learning rate in SGD is because the gradient changes with every new input. This means that it is not ideal that we follow the gradient blindly. The learning rate is used so the weights change in the direction of the gradient but only by a small amount. This gives the opportunity for the new position to be assessed by the gradient again so corrections can be made. If there was a way to predict when the

$$
\begin{aligned}
x_t &= \rho x_{t-1} + (1-\rho)\nabla l(w_{t-1})^2 \\
y_t &= mom * y_{t-1} + \frac{\nabla l(w_{t-1} + mom * y_{t-1})}{\sqrt{x_t + \epsilon}} \\
w_t &= w_{t-1} - lr * y_t
\end{aligned}
\tag{2.7}
$$

Equation 2.7: RMSprop optimizer with nesterov momentum weight update rule

gradient was going to change, the learning rate would no longer be required. There is a way to predict this called Newton's method. While this has been used for neural network optimization (see [37]) it is not common because the calculations required are too time consuming.

In Adadelta, Zeiler utilizes the moving discounted average of the weight updates to approximate Newton's Method. We recommend that the reader checks the original paper [38] for a deeper understanding of this approach.

In Equation 2.8 we present the algorithm where x is the moving average of the gradient squared, y is the weight update calculation and z is the moving average of the weight updates.

$$
\begin{aligned}
x_t &= \rho x_{t-1} + (1-\rho)\nabla l(w_{t-1})^2 \\
y_t &= -\frac{\sqrt{z_{t-1}}}{\sqrt{x_t} + \epsilon} * \nabla l(w_{t-1}) \\
z_t &= \rho z_{t-1} + (1-\rho) y_t^2 \\
w_t &= w_{t-1} + y_t
\end{aligned}
\tag{2.8}
$$

Equation 2.8: Adadelta optimizer weight update rule

2.4.5 Adam

Adam and its variations are the final optimizers that we will discuss. Adam was proposed by Kingma et. al. in [39] where it is shown to outperform the other optimizers presented on a variety of different problems. Adam is yet another variation of Adagrad. What sets this algorithm apart is that it acknowledges that because the moving averages used are initialized at zero they are have a bias towards this value. The optimizer then employs a new term (z seen in Equation 2.9) to correct this bias. Adam also utilizes $\frac{x_{t-1}}{\sqrt{y_{t-1}}}$ to calculate a range where it believes the gradient can be trusted and uses this information to determine the magnitude of updates made to the weights. β_1 and β_2 are two constants between 0 and 1 (typically very close to 1) that control the exponential decay rate of the moving averages (similar to ρ in the previous optimizers).

In the same paper a variation of Adam called Adamax is also proposed. The difference in this version is that we replace the L2 norm that we have been using since Adagrad with an infinity norm and adjust the bias correction accordingly. This variation is not tested in the original paper but [40] concluded that Adamax did not perform significantly better than Adam.

The algorithms for both optimizers can be seen in Equation 2.9 and Equation 2.10. These are the efficient versions of these algorithms that are typically implemented, the original paper [39] features more readable versions that are easier to understand.

$$
\begin{aligned}
x_t &= \beta_1 x_{t-1} + (1-\beta_1)\nabla l(w_{t-1}) \\
y_t &= \beta_2 y_{t-1} + (1-\beta_2)\nabla l(w_{t-1})^2 \\
z_t &= lr * \frac{\sqrt{1-\beta_2^t}}{(1-\beta_1^t)} \\
w_t &= w_{t-1} - z_t * \frac{x_t}{\sqrt{y_t}+\epsilon}
\end{aligned}
\tag{2.9}
$$

Equation 2.9: Adam optimizer weight update rule

$$
\begin{aligned}
x_t &= \beta_1 x_{t-1} + (1-\beta_1)\nabla l(w_{t-1}) \\
y_t &= max(\beta_2 y_{t-1}, |\nabla l(w_{t-1})|) \\
z_t &= \frac{lr}{(1-\beta_1^t)} \\
w_t &= w_{t-1} - z_t * \frac{x_t}{y_t}
\end{aligned}
\tag{2.10}
$$

Equation 2.10: Adamax optimizer weight update rule

The idea behind Adam's x term is similar to the ones used in momentum optimizers as this term is able to accelerate optimization if the gradient goes in the same direction repeatedly. This led to the creation of Nadam, a modified version of Adam that incorporates Nesterov Momentum. This was proposed by Dozat [40] and the results showed that while it did not always outperform Adam it did so by a significant margin when it happened.

$$
\begin{aligned}
x_t &= \beta_1 x_{t-1} + (1-\beta_1)\nabla l(w_{t-1}) \\
y_t &= \beta_2 y_{t-1} + (1-\beta_2)\nabla l(w_{t-1})^2 \\
z_t &= z_{t-1} * \beta_1 * (1 - 0.5 * 0.96^{0.004*(t+1)}) \\
w_t &= w_{t-1} - lr * (1 - \beta_1 * (1 - 0.5 * 0.96^{0.004*(t)})) * \frac{\frac{\nabla l(w_{t-1})}{z_{t-1}} + z_t * \frac{x_{t-1}}{1-z_t}}{\sqrt{y_t+\epsilon}}
\end{aligned}
\tag{2.11}
$$

Equation 2.11: Nadam optimizer weight update rule

Chapter 3

AutoLR

The system built for this work has two versions. Both versions are built on top of a common framework that we have named AutoLR. This chapter describes this framework including information that is important if the reader wishes to use it. This framework is comprised of two sub systems that are each explained in their respective sections. There is also a third section dedicating to explaining how the system can be used to evolve solutions and suggestions on how to validate them.

3.1 Evolutionary Engine

Since the novelty of our approach lies in the use of an evolutionary algorithm it should come as no surprise that the choice of the engine used for the evolutionary process is critical.

In the context of our work, learning rate policies will be executable computer code. This suggests we should either use Genetic Programming or Grammatical Evolution, since these are the approaches best suited to evolve actual executable code. The deciding factor was the fact optimizers of the same type typically share some structure. This lead us to choose Grammatical Evolution since grammars provide a readable, understandable and easy to change way to insert said structure into our individuals. This choice allows us to impose syntactic restrictions on the optimizers. In concrete, we chose Dynamic Structured Grammatical Evolution as our evolutionary engine due to its good performance in comparison to other GE approaches. Another benefit of this approach is that by using an established engine we do not have to develop specific genetic operators for our use case.

3.1.1 Structured Grammatical Evolution

We must also go over how SGE [26, 27] works as the grammars used down the line were designed taking into account the inner workings of this system.

As was briefly mentioned in Section 2.2, SGE uses a genotype where each production has a list of integers in contrast with GE's single list.

An important consequence of this change is that a certain value in a production's list is always interpreted as the same thing. This contrasts with GE where a segment of the genotype can have different meanings depending on context. Consequently, an individual's genotype and phenotype are more closely related in SGE. This property is known as locality and it is desirable to have a high locality as this means that making small changes to the

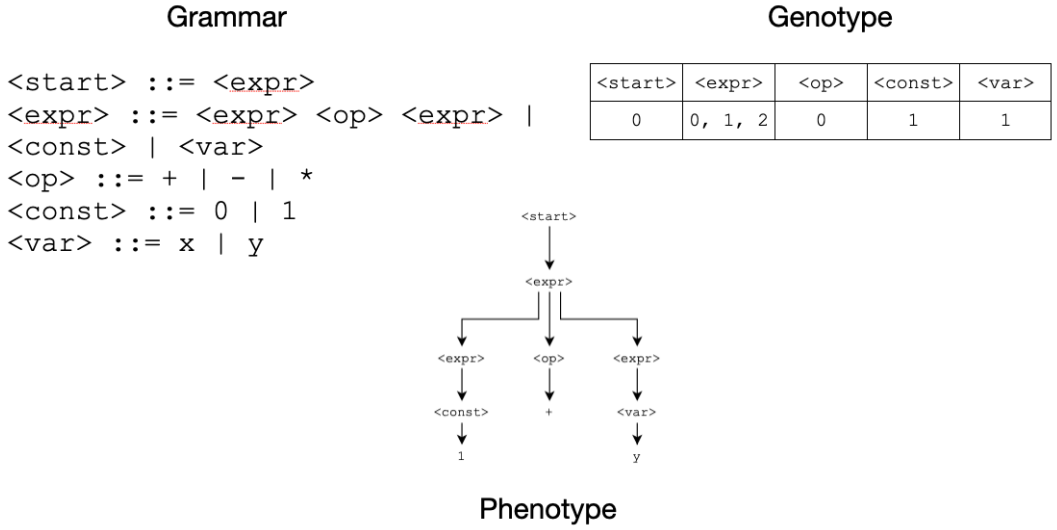


Figure 3.1: Sample mapping of an individual in SGE.

genotype is more likely to only slightly influence the resulting individual. An engine with low locality has a hard time navigating the problem space since slight adjustments to the genotype can lead to massive differences in results making the search process erratic.

It is also important to understand how the genetic operators work with this unique genotype. The crossover operation picks, for each production, a single parent to inherent from. This means that, on a production level, genetic material is preserved. Mutation is then responsible for changes inside the productions, randomly altering values inside a productions list to other possible options.

Finally it is worth noting that SGE is implemented in the Python programming language and open source [41]. Since the framework is built around this engine it follows that the whole system is also built in Python. An additional benefit of this choice is that this language has vast support for ANN handling through the **Tensorflow** [42] library.

3.2 Optimizer Evaluator

An optimizer has no value in a vacuum. In order for us to assess whether a certain optimizer is "good" or "bad" we must provide an environment where it can be used.

As previously stated, the breadth of machine learning problems where learning rate optimizers are useful is massive. Since this framework can feasibly be used in any problem that makes use of an SGD based optimizer it can be applied to all of these problems.

For evolution to progress smoothly, however, a task of high difficulty should be used since it is important that the better optimizers are in an environment where they can showcase

their ability. If the problem chosen is too easy there is a risk the results will be overly homogeneous, meaning that all optimizers are able to achieve indistinguishably good results.

3.3 Evolution

Up until this point we have described the framework we will be using to evolve our optimizers.

3.3.1 Grammar

Since we are using SGE as our engine, the first challenge when setting up any evolutionary run is designing the grammar. There is no objectively correct grammar for an individual problem. When creating a grammar we must weigh the benefits and limitations of our decisions based on our own criteria. As a result it is important to discuss the criteria used when designing the grammars used for our experiments:

- Reproducibility - Since we will be comparing our solutions with standard optimizers it is important that our grammar can create individuals that can reproduce at least an approximation of the optimizers they will be competing with. This also goes in line with one of our objectives: "Are evolved learning rate optimizers similar in shape to the manually designed ones?" by giving the algorithm the ability to reproduce man-made approaches we can gather insights on whether automatic evolution favors the same ideas as researchers. This also makes the benchmarking phase more fair since evolved optimizers have access to roughly the same tools as standard ones.
- Dimension - More specifically it is important to be mindful of the dimension of the search space. As the number of possible individuals the grammar can create increases so does the time necessary for the evolutionary algorithm to find interesting solutions. Theoretically this should not be much of a concern but practically speaking this work (and most others) must meet certain deadlines with access to limited resources and the grammars used must accommodate this.
- Efficiency - Leaning on the fact that optimizers are functions with a specific task, we are interested in creating grammars that, more often than not, generate functioning optimizers. This means we are looking to make the grammars that make the most out of the resources we have e.g. if we can afford to use 100 constants their values should all be conceivably useful be for an optimizer.

While the weight of each of these criteria vary from work to work these are the most important aspects to take into account during grammar design.

3.3.2 Fitness

A crucial part of every evolutionary algorithm is the fitness function. This is the function that takes an individual solution and quantifies their performance. This is where the *optimizer evaluator* previously described is used. It is recommended that not all of the data available is used in the fitness function. If the evolutionary phase has access to all available data then it becomes impossible to rate the optimizers generated based on their

accuracy on data they had no contact with (which is the most important metric to measure the success of an optimizer.).

Image datasets usually provide a set of test instances that should be used to make the final assessment of the network's accuracy. In AutoLR it is desirable that these test instances are not inserted into the evolutionary process so they can later be used to judge the optimizers.

The exact neural network architecture and setup used to assess the quality of the optimizers change based on the needs of the user. In our case the architecture and fitness function will be specified in the chapter dedicated to each version.

3.4 Benchmark

In the context of our work we must be able to draw comparisons between our evolved optimizers and standard ones. While the common user of the framework may not be interested in this specific situation it is recommended that the evolved optimizers are validated in some way.

While the first step in our experiments is always to produce some capable evolved optimizers through the process of evolution. We must then pick the best solutions produced and benchmark them against the standard optimizers in an environment that is fair for both of them.

While an individual's fitness during training is a good representation of its quality as an optimizer there is an even more accurate measurement that can be used for our final results.

Because the evolutionary process demands the fitness to be calculated many times this function typically works on only a subset of the whole data available. This means it is possible to create an enhanced version of the fitness function that takes longer to run but gives us a more accurate assessment of the optimizer quality.

This function is called the *benchmark* function and it is the same across all versions of AutoLR.

Algorithm 2: Simplified version of the benchmark function used to determine the average performance of an optimizer

params: network, learning_rate_optimizer, training_data, validation_data, test_data

```
1 trained_network ← train(network, learning_rate_optimizer, training_data,
   validation_data);
2 fitness_score ← get_test _accuracy(trained_network, test_data);
3 return fitness_score;
```

This function is different from any of the fitness functions seen later on because the final accuracy of the network is measured in Fashion-MNIST test data. The training process also makes use of all 60000 available training instances, under these circumstances the optimizers are able to display their peak performance under ideal and fair conditions.

After training is concluded the network's final accuracy is calculated on the aforementioned 10000 test instances. Figure 3.2 shows the difference in how the data is split for evolution and benchmark.

Naturally we are interested in doing a statistical analysis of our results, consequently both

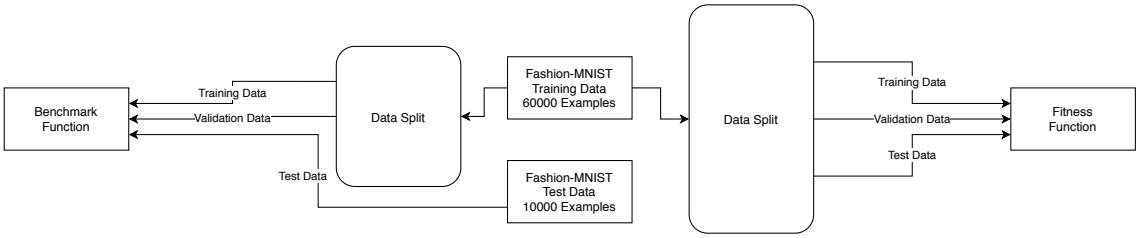


Figure 3.2: Difference in data used for evolution (fitness function) and benchmark (benchmark function).

evolved and standard optimizers will be benchmark-ed several times and they will be judged based on their average accuracy and standard deviation.

Chapter 4

Dynamic AutoLR

Dynamic AutoLR (Dynamic AutoLR) is an evolutionary system created to evolve dynamic and static learning optimizers (also refer to as *policies* or *schedulers*). The system is by definition limited since schedulers are not the most sophisticated approach to the problem.

In the Keras [43] library, LearningRateScheduler is the name of an object class that is able to produce both static and dynamic learning rate optimizers. Learning rate schedulers are functions called periodically during training that update the learning rate value. In other words, we are evolving the initial learning rate and the ensuing variation function. These functions' inputs are limited to the learning rate of the previous epoch and the number of the epoch. Since these inputs do not provide much information about how the training process is going these functions are typically quite simple.

Nevertheless, Dynamic AutoLR is useful as it allows us to explore the hypothesis without diving into the more complex problem space of adaptive optimizers. This chapter walks the reader through the components that comprise this system and explains the decisions behind their design. This chapter also describes the experiments performed and their results.

4.1 Grammar

The grammar defines the search space of the learning rate schedulers. We are evolving if-else systems that yield a new learning rate based on the current learning rate and epoch. The grammar used is shown in Figure 4.1.

The individuals created by this grammar will typically resolve into a sequence of chained if-else conditions (created by the *logic_ expr* production) that once evaluated will yield a learning rate (provided by the terminals in *lr_ const*). This means that the majority of policies created by the system are dynamic in nature. A notable exception to this is that the system can resolve the initial *expr* production into a *lr_ const*, creating a static learning rate policy.

An *if_ func* is a simple function that does the same as a regular if-then-else construct. Since the code for this system was written in Python this function was created so all individuals could be described in a single line that can be read easily by the user. The code for this function is shown in Algorithm 3.

The conditions used by *if_ func* are generated by *logic_ expr*. This production will com-

pare one of the input variables (learning_rate, epoch) with the corresponding constants (lr_const and ep_const, respectively) using one of several logical operators from *logic_op*. *logic_op* includes all logical operators with the exception of equality (==) and inequality (!=). These operators are too specific and broad respectively, meaning that they have little impact in the output of any individual that includes them. This makes them unable to contribute meaningfully to the evolutionary process and they were removed as a result.

The constants chosen for *lr_const* and *ep_const* are both 100 values with evenly spaced between the minimum and maximum value for each of the variables. It should be noted that these production rules have been abridged in the figure. Only a few of the lowest and highest possible values are shown so that the range is accurately portrayed whilst keeping the figure brief. Determining the limits of the interval for the epoch constants was simple. Our training starts in epoch 1 and ends in epoch 100, since we are also using 100 values for our constant, we used every possible epoch value (every natural number from 1 to 100) for *ep_const*. *lr_const* values are more complicated as there is an infinite number of valid learning rates. We keep the values of the learning rate bounded between 0.001 and the 0.1 as all values in this range are suitable for training.

This grammar is capable of creating a large variety of individuals despite its simplicity. While it is not possible for our trees to exactly recreate the dynamic solution functions mentioned in Section 2.3 they can reproduce approximated versions that exhibit similar behaviour.

Algorithm 3: Template of the code used to implement the if_func routine.

```
params: condition, state1, state2
if condition then
    return state1;
else
    return state2;
```

<expr> ::= if_func(<logic_expr>, <expr>, <expr>)
| <lr_const>
<logic_expr> ::= learning_rate <logic_op> <lr_const>
| epoch <logic_op> <ep_const>
<logic_op> ::= < | ≤ | > | ≥
<lr_const> ::= 0.0001 | 0.00110909 | 0.00211818 | 0.00312727 | 0.00413636
...
0.09596364 | 0.09697273 | 0.09798182 | 0.09899091 | 0.1
<ep_const> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
...
91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99| 100

Figure 4.1: CFG for the optimisation of learning rate schedulers.

Figure 4.2 depicts an example of a learning rate scheduler. In this case the ANN will train using a learning rate of 0.1 for the first 10 epochs as this is when the condition *epoch* < *10* is met. This learning rate will be used until 10th epoch is reached, at which point the

learning rate scheduler will automatically decrease the learning rate to 0.05. Following the same rationale, after the 50th epoch the learning rate to use is 0.01.

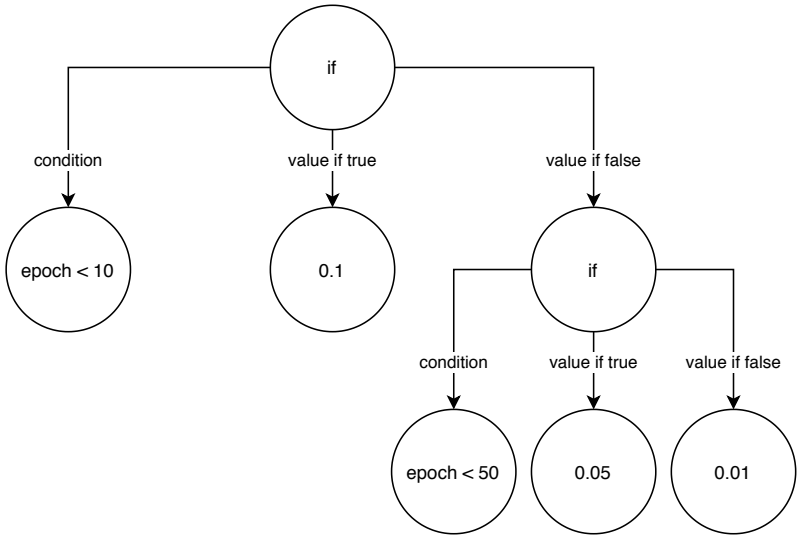


Figure 4.2: Example of a learning rate scheduler.

4.2 Dataset

When attempting to solve a machine learning problem we must concern ourselves with key choices such as neural network design and dataset collection/selection. By opting to tackle a common task, however, we are able to to make use of established architectures and datasets. This is beneficial since we can skip some of the setup that comes with our hypothesis, allowing us to focus time and resources on developing a solution.

The most fitting problem found for this work was *image classification*. Image classification problems are challenging, which is a necessary attribute for a smooth evolution. There is also an abundance of past research into this subject, with many networks [44, 45] and datasets [46, 47] readily available.

Within the space of image classification there are several datasets to choose from. The dataset used in this work is **Fashion-MNIST** [47] as we found it to be an accessible classification problem for our approach. This dataset has the right difficulty to allow evolution to occur smoothly. Classifying Fashion-MNIST instances is hard enough that the best individuals are able to set themselves apart whilst being easy enough that genetic material with some potential does not get mixed in with random search. Instances of Fashion-MNIST are $28X28$ grayscale images. These images can be processed quickly due to their small size and networks are able to train quickly on this data as a result. This dataset is composed by 70000 instances and corresponding classifications, 60000 training and 10000 for testing respectively.

4.3 Fitness

As the main hypothesis implies, we are looking to evolve learning rate optimizers. In this case we are narrowing the scope to static and dynamic optimizers. This means that we will be using the EA on a population of learning rate policies. Additionally, our hypothesis

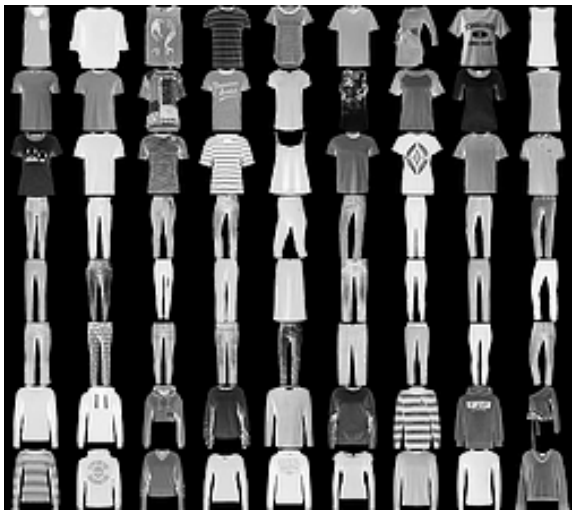


Figure 4.3: Example images from the Fashion-MNIST dataset.

demands that the fitness of an individual must be some measure of the CNN's performance when trained using that specific solution. This is necessary since if the evolutionary process is not guided properly its results will not address the question we posed.

We designed the function seen in Algorithm 4 in order to best assess the policy's performance.

Algorithm 4: Simplified version of the fitness function used to evaluate a learning rate policy

params: network, learning_rate_policy, training_data, validation_data, test_data

```
1 trained_network ← train(network, learning_rate_policy, training_data,
    validation_data);
2 fitness_score ← get_test _accuracy(trained_network, test_data);
3 return fitness_score;
```

To elaborate on the listing above, our fitness function will use 4 components:

- **network** - The ANN we are evolving for, this network is the same throughout the entire evolutionary run.
- **learning_rate_policy** - The solution we want to evaluate, this is the function the EA is evolving. This implies that this function will be used on multiple policies (one at a time).
- **training/validation/test_data** - This is the data of the problem the ANN will be attempting to solve. As the name implies, training data is used for training. Validation data is a set of data that is used to assess the quality of training as it progresses. Test data is a separate set of examples that are used to evaluate the network's performance once training is complete.

Moreover, this function has two phases: In line 1 of Algorithm 4 the network is trained using the evolved policy. Notice that this is the only use of the learning rate policy in the fitness function, the policy only comes into direct contact with the training and validation data.

We could at this point retrieve the best performance the network achieved during training. We do not take this approach as it is not the most accurate measure of an ANN's real effectiveness. The objective of the training is that the network learns a set of weights that solve the proposed problem. The data used for training is only a sample of all possible inputs. As training progresses a network might become too attached to the training data i.e. the network might overfit. Overfitting means that the network has learned everything about the training data. This is not desirable since training data will often have some noise (i.e. information that is not important to solve the task) that will be detrimental when dealing with inputs not included in training. Consequently, we measure the effectiveness of training by how well the network performs on a second set of data that it has not come into contact with. We call this second set the test data. This can be seen in line 2 of Algorithm 4.

Every policy will be evaluated using the same network and training data meaning that learning rate is the only varying component between individuals. Since all other hyperparameters are fixed we consider the result of evaluating the trained network's performance on the test data to be an adequate measure of the policy's fitness as it is directly related to the policy itself.

The actual evaluation of an individual is done by mapping their genotype into a proper learning rate policy through SGE. This policy is then wrapped in a Keras Learning Rate Scheduler object and attached to the training process. The 60000 training examples are split into 3 groups: training, validation, and testing. The training set directly affects the network's weights, meaning that the learning process is done in response to the network's error on this dataset. The validation data is used as a control mechanism *during training*. Once the network has completed a training epoch it is **evaluated** in the validation set. This is a good method for detecting overfitting since if the network's accuracy improves in training but dips in validation that typically means it is becoming too attached to the training data. The weights chosen at the end of training are the ones that achieved the best performance on the validation set.

Finally, the testing data is used after training is complete to obtain a fitness value. The fact we are restoring the weights that performed best on the validation set implicitly biases the network towards the validation set which is why we are required to use this third dataset.

4.4 Network Architecture

The network architecture we used was automatically generated using DENSER [48] – a grammar-based NeuroEvolution approach. The CNN optimised by DENSER was evaluated using a fixed learning rate strategy, and thus it is likely that better learning policies exist. The architecture was generated for the CIFAR-10 dataset using a fixed learning rate of 0.01, where the individuals were trained for 10 epochs. The details of how the network was created are important as they might inform our conclusions later on. The specific topology of the network is described in Figure 4.4.

4.5 Experimental Setup

As was specified in Chapter 3 we divide the experimental setup into two parts: the evolutionary search (Section 4.5.1); and the longer assessment done after the end of evolution (Section 4.5.2).

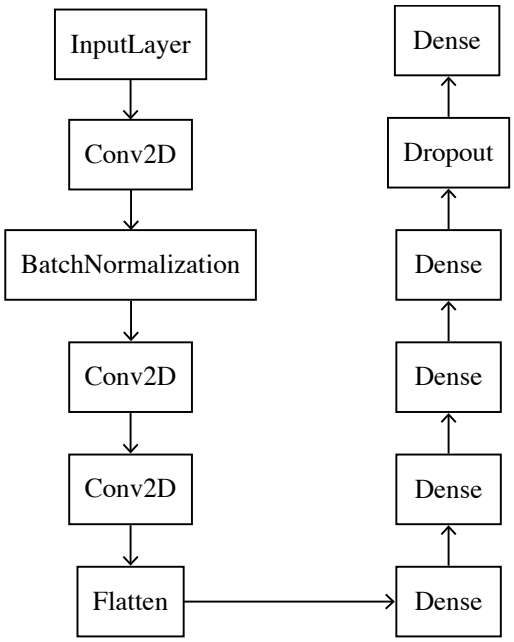


Figure 4.4: Topology of the used CNN.

4.5.1 Evolution

The experimental parameters are summarised in Table 4.1. They are organized into five sections:

- SGE Parameters – parameters of the evolutionary engine.
- Dataset Parameters – number of instances of each of the data partitions.
- Early Stop – the stop condition used to halt the training of the ANN.
- Training Data Augmentation – real time data augmentation parameters.
- Network Training Parameters – parameters used when training the ANN.

The experimental parameters are picked with according to some considerations. Since evolutionary algorithms are very demanding in terms of computation resources, it is paramount that the parameters used allow us to perform meaningful evolutionary runs that can be completed in an acceptable time-frame. Additionally, the fitness function operates on a fraction of the dataset, as training utilizing all 60000 training examples with data augmentation is too time consuming.

The training parameters also take the computational costs into account. Ideally, evolution would be performed on 100 training epochs with no early stop as we want to optimize the network's performance as much as possible. Instead, two sets of experiments were performed: 1. using 100 epochs and an early stop mechanism; 2. using 20 epochs with no early stop.

We started by reducing the computational cost through the implementation of an early stop mechanism. Notwithstanding, we were concerned that the evolutionary process would exploit this mechanism, which motivated the 20 epochs experiment, where no early stop is used and the cost is instead reduced by reducing the training epochs.

SGE Parameter	**Value**
Number of runs	10
Number of generations	50
Number of individuals	5
Tournament size	5
Mutation rate	0.15
Dataset Parameter	**Value**
Training set	7000 instances from the training
Validation set	1500 instances from the training
Test set	1500 instances from the training
Training Data Augmentation	**Value**
Feature-wise Center	True
Feature-wise Std. Deviation	True
Rotation Range	20
Width Shift Range	0.2
Height Shift Range	0.2
Horizontal Flip	True
Early Stop	**Value**
Patience	3
Metric	Validation Loss
Condition	Stop if Validation Loss does not improve in 3 consecutive epochs
Network Training Parameter	**Value**
Batch Size	1000
Epochs	100 / 20
Metrics	Accuracy

Table 4.1: Experimental parameters.

4.5.2 Benchmark

After evolution is completed further testing must be performed to assess the quality of the generated policies. Since the test set used during evolution is always the same it is possible for evolution to make the scheduler biased towards it. This is similar to the problem that motivated the three way data split used during evolution. The benchmarking function is similar to the fitness function, the major difference is in the data used (see Table 4.2). During the evolutionary process we were forced to use only a subset of the training data as using all 60000 examples was too time consuming.

Dataset Parameter	**Value**
Train set	52500 instances from training data
Validation set	7500 instances from training data
Test set	10000 instances form test data

Table 4.2: Data distribution of the Fashion-MNIST dataset used in benchmarking.

In our benchmark we use all training instances (splitting them into train and validation) to train the network using the policy we want to evaluate. This network is subsequently tested using the entirety of the test data (that was not used in any capacity previously) to obtain an unbiased **test accuracy**. We will also be tracking each policy's best **validation**

accuracy through this process as these give us additional insights into how well the learned weights are able to generalize.

We also needed to decide on a policy to serve as a baseline. We chose to use a *static learning rate policy of 0.01*. Not only is this the policy used to evolve the topology itself (in DENSER [48]), it is also the most common out-of-the-box policy [43, 49].

We have three testing scenarios:

- The **first scenario** is the same as the first evolutionary scenario, training is done for **100 epochs with the early stop mechanism**.
- The **second scenario** also trains for **100 epochs, but the early stop mechanism is disabled**.
- The **third scenario** trains for only **20 epochs, with no early stop**.

The best policy from the early stop evolution (henceforth referred to as policy A) will be tested in the first and second scenarios. We considered testing it in the third scenario as well but observing the validation accuracy during training hinted that this would not be worthwhile (this policy was performing poorly in epoch 20 for most observed trials). These two scenarios will allow us to analyse how well a policy evolved using early stop performs into a scenario without early stop and how that compares to our baseline

The policy resulting from the no early stop evolution (policy B, henceforth) will be tested in all three scenarios.

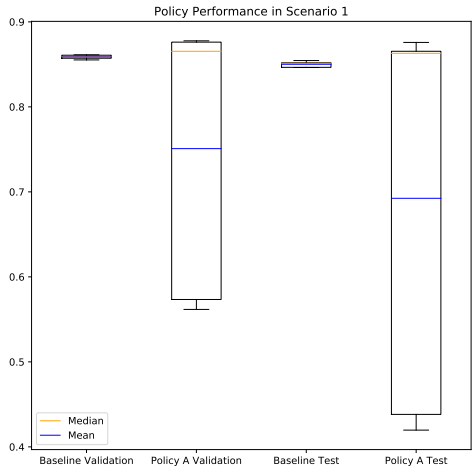
Each policy is tested in each scenario five times and the results presented are the average and standard deviation of those trials.

4.6 Experimental Results

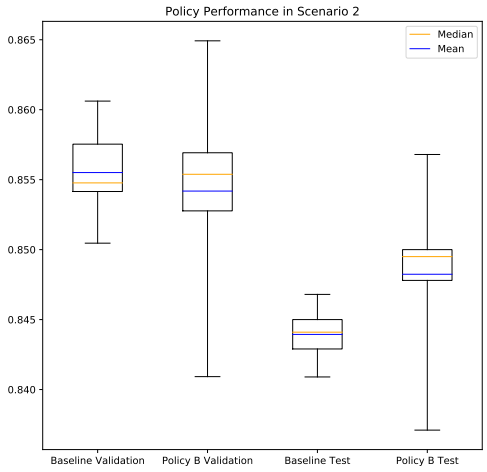
The results of our experimentation can be seen in Figure 4.5 where Figures 4.5a, 4.5b and 4.5c show a box plot for each of the testing scenarios. The table presented in Table 4.3 summarizes the results of our experimentation, showing the actual average and standard deviation of the accuracy of a given policy in a specific scenario over five runs. As detailed in Section 4.5.2, each run trains the network using the chosen policy and subsequently tests its accuracy on the 10000 test instances.

Scenario		**Policy**		
		A	**B**	**Baseline**
1	**Validation**	75.09 ± 16.785%		**85.87 ± 0.25%**
	Test	69.24 ± 24.07%		**85.01 ± 0.36%**
2	**Validation**		85.42 ± 0.87%	**85.55 ± 0.38%**
	Test		**84.82 ± 0.71%**	84.39 ± 0.22%
3	**Validation**	**89.38 ± 0.39%**	89.13 ± 0.33%	88.85 ± 0.22%
	Test	**88.68 ± 0.22%**	86.58 ± 1.47%	87.47 ± 0.37%

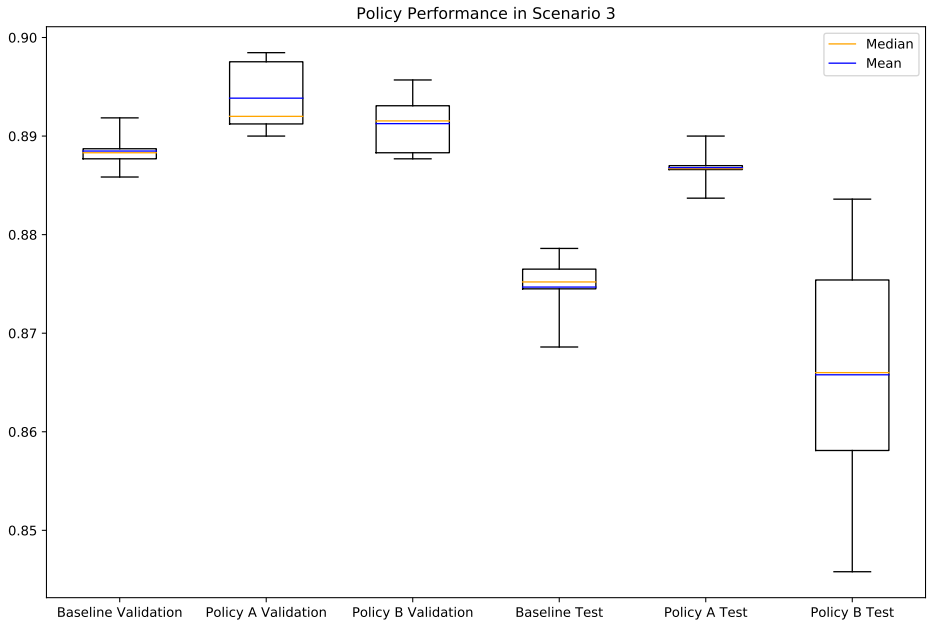
Table 4.3: Accuracy of the evolved policies (A & B) on their evolutionary environment (1 & 2 respectively) and scenario 3 (representative of an actual use case), compared with the baseline policy.



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Figure 4.5: Box plot showcasing the results of the test scenarios. Whiskers show the minimum and maximum result. Box size shows the lower and upper quartile. Mean and median of each population are marked inside each box.

4.6.1 Scenario 1

Scenario 1 yields results that are not intuitive given the circumstances. Training in this scenario can be halted by an early stop mechanism. Since policy A was evolved using this same kind of training it is to be expected that it would perform well in these conditions. However, the results show the opposite. Policy A, in fact, performs far worse than the baseline when early stop is in use. Analysing individual results showed that this policy

will occasionally trigger the early stop in the first few epochs (this can be observed in the large standard deviation associate with these trials).

There are several interpretations for the implications this has on the validity of the evolutionary process. On the one hand it can be argued that this demonstrates an issue with the evolutionary process since the policy is not a consistent solution to the problem it is supposed to solve. While it is a fact that the policy is an inconsistent solution we do not believe this implies any problems with the evolution. The fact that this policy can, on occasion, yield the best performance implies the genetic information of this individual is useful for the evolutionary process.

4.6.2 Scenario 2

Scenario 2's results are more in line with our expectations. We can observe that, albeit only marginally, policy B shows better test accuracy than the baseline when trained under the parameters it was evolved for. It is noteworthy that policy B does not have superior accuracy in validation. This suggests that the **evolved policy is outperforming the baseline in its ability to generalize** when moved to a different set of data.

4.6.3 Scenario 3

Scenario 3 was designed to test which policy is able to get the most out of the architecture of the network and it gave the most important set of results. The results show that, under these conditions, **the best accuracy this network achieved was obtained using an evolved policy for training**. On average, policy A performs better than the baseline in the test set by 1.2% and it obtains these good results more consistently. Another interesting result is that policy B (that was previously outstanding because of its ability to generalize) suffers the biggest dip in performance from validation to test in this scenario. Ideally, both evolved policies would outperform the baseline. There are, however, some possibly limiting factors. Namely, it is possible that the shorter training duration used in scenario 2 discourages the evolution of policies that translate well into scenario 3. This topic is discussed further in Section 4.6.4 as we analyse policy B's shape.

4.6.4 Shape

As discussed in our objectives (Section 1.1), we are interested in analysing the shapes that our evolved policies take. Namely, in this section we will be analysing the shape of the previously discussed Policies A and B. These policies can be observed in Figures 4.6 and 4.7. These figures show how the learning rate evolved over time as well as a vertical line that signals the epoch where the training using this policy stopped.

Observing the shape of policy A (seen in Figure 4.6) led to some interesting insights. Initially, it seemed that this policy only had the best performance during evolution because its shape cheats the early stop mechanism. We suspected that by frequently using a high learning rate it might be possible to create false improvements that trick the system. To elaborate, it is feasible for a policy to routinely worsen and subsequently improve its performance *on purpose* in order to pass the early stop check. We can, in fact, observe that this policy is able to train for a long time despite the early stop as the vertical line shows. As a comparison, the baseline policy typically triggers the early stop between epochs 20-30, which means that policy A is able to train for twice as long.

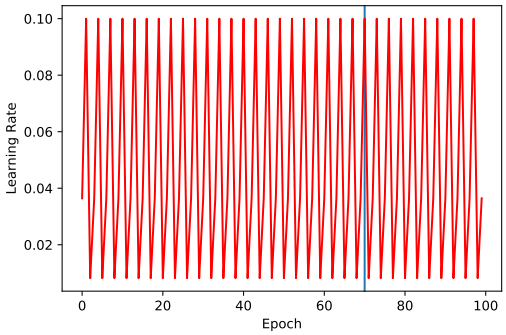


Figure 4.6: Policy A

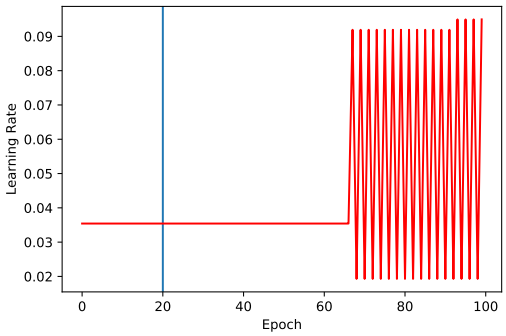


Figure 4.7: Policy B

Policy B (seen in Figure 4.7) took on a very different shape. Despite the erratic behaviour shown past epoch 60, this policy is effectively a static learning rate as its training always ended in epoch 20 (as a reminder, no early stop was used in the evolution of the policy). While this initially seems disappointing (finding an adequate constant is not something that requires such a complex system), it is important to understand that, due to the reduced training duration, there is a possibility that the benefits of using a dynamic policy in this context are negligible, stifling probability that they show up in evolution. The idea that the evolution of dynamic methods is suppressed under these circumstances is further supported by the fact that all twelve of the best policies during the evolution of policy B were constants. In this context the twelve best policies we are referring to is the set of policies that were, at some point during evolution, the best policy in all runs.

We have, up until this point, observed two types of evolved functions shapes (within the individuals that perform well). The first type is constants, these comprise the majority of the search space so their presence is expected. The second type can be observed in policy A, we refer to these as *oscillator policies*. We believe that these policies are approximations of the policies used in [30]. While it would be disingenuous to claim that we are evolving cyclical policies, it seems possible that the evolved oscillator policies are effective for the same reasons as the cyclical ones. It is notable that while many known dynamic policies are decaying policies we have not observed any evolved policies with a similar shape performing well.

Chapter 5

Adaptive AutoLR

Adaptive learning rate policies do not follow the basic outline we find in static and dynamic approaches. In the previous chapter we were able to take advantage of the LearningRateScheduler class found in Keras. For this version of the system we will have to create a customized version of the Optimizer class, we will call this new class CustomOptimizer. Adaptive AutoLR (Adaptive AutoLR) comes with additional technical challenges due to the vast difference in complexity between these two components.

In Dynamic AutoLR we were able to reduce the search space by creating a grammar that produces policies that were approximations of the standard solutions. The same is not possible in Adaptive AutoLR, Adaptive approaches make use of the gradient of the loss function to create many different variables with all sorts of meanings. Since these variables perform many different roles it is also much harder to anticipate the values they will take. For these reasons we decided that creating a grammar that produced approximations was not feasible for this problem. This means that Adaptive AutoLR's search space is much larger and harder to navigate than Dynamic AutoLR's. The main consequence of this is that it is that expectable that Adaptive AutoLR will need to produce many more individuals during the evolutionary process to achieve a good accuracy. It is therefore needed to put increased attention into optimizing this version of the system as a result.

5.1 CustomOptimizer

LearningRateScheduler was able to take Python code and execute it directly. This made the process of designing our grammar straightforward. In CustomOptimizer we are mostly concerned with implementing a single function from its super class, *_resource_apply_dense*. This is the function called by the training algorithm to update the network's weights and it has two mandatory arguments:

- *grad*, a Tensor that holds the loss function gradient for the weight being updated.
- *var*, a Tensor that holds the variable corresponding to the weight being updated.

Both of these are supplied automatically by Keras. This is quite limiting as it means we are not able to pass our own variables into this function. We chose to adhere to Keras' established architecture for Optimizers since this library is extremely optimized for performance. Instead of changing the inner workings of the training algorithm we opted to pass variables into the function through attributes in our CustomOptimizer class.

This function is expected to return an object of the type Operation. The most efficient way we found to create these objects is using Tensorflow training_ops [50] wrapper that allows us to run these operations on the lowest level possible. This wrapper comes with some limitations as it is designed to only execute the adaptive methods found in the literature meaning that our evolved policies must conform to this.

Our approach to this problem was searching for a training operation that was general enough that it could be used to replicate several adaptive methods. Through this approach we found that the majority of adaptive methods can be reproduced through repeated uses of the operation wrapper for gradient descent.

This wrapper is called *resource_apply_gradient_descent* and takes the arguments *var, alpha, delta* and performs the following operation on them:

$var_t = var_{t-1} - (alpha * delta)$

While this operation is simple, we are able to reproduce a variety of adaptive methods by calling it several times.

It must be noted that in order for this approach to work the CustomOptimizer must save the values of the auxiliary variables (x_t, y_t z_t). For the reasons explained above we will not be tampering with the *resource_apply_dense* function's inputs. This brings some challenges since we must keep in mind that these variables have a different value for each weight in the neural network. We chose to address this using dictionaries, one for each auxiliary variable, that use the name of the weight as a key for the value of the corresponding auxiliary variable. This is possible because the weight gets passed into the *resource_apply_dense* function in the form of the parameter *var*. *var* is an object of the Tensorflow Variable class and it has a unique name we can use as a key in our dictionaries. This allows us to make the connection between the function's inputs and our own auxiliary variables.

Pseudocode for the CustomOptimizer implementation of *resource_apply_dense* can be found in Algorithm 5, the auxiliary variables are called x, y and z and are stored in dictionaries named after (e.g *x_dict*). Each of them as a respective function that calculates it (*x_func, y_func* and *z_func*). The fourth function is the **weight update function** referred to as *weight_func* in our code. Both the dictionaries and the functions are attributes of the CustomOptimizer class, making them accessible inside *resource_apply_dense*.

Algorithm 5: Simplified version of the CustomOptimizer's implementation of the _resource_apply_dense method

params: grad, var

```
1 x_var ← x_dict[var.name];
2 y_var ← y_dict[var.name];
3 z_var ← z_dict[var.name];
4 resource_apply_gradient_descent(var=x_var, alpha=1.0, delta=_func(x_var,
   grad));
5 resource_apply_gradient_descent(var=y_var, alpha=1.0, delta=y_func(x_var,
   y_var, grad));
6 resource_apply_gradient_descent(var=z_var, alpha=1.0, delta=z_func(x_var,
   y_var, z_var, grad));
7 resource_apply_gradient_descent(var=var, alpha=1.0, delta=weight_func(x_var,
   y_var, z_var, grad));
```

The inputs of each of the auxiliary functions are also worth discussing. It is necessary that

weight_func has access to all the auxiliary variables (the whole purpose of these variables is to allow more complex changes to the weight) but the choice of inputs for the other three functions is not as clear cut. In most adaptive methods the auxiliary methods only require access to themselves and the gradient. Examining Equation 2.5 shows an adaptive approach where one of the variables is calculated using the updated value of another. We are interested in allowing our system to explore as many of the optimizers found in the literature as possible so we must understand how to meet the requirements demanded by RMSProp.

Knowing that auxiliary variables can depend on each other we must establish some hierarchy. This allows us to determine the order of execution of the functions as well as their inputs. We decided that the order of execution would be *x_func* followed by *y_func* followed by *z_func* and finally *weight_func*. We found an instance of an adaptive approach that calculated variables after making changes to the weight ([38]). The benefits of allowing the system to evolve these types of approaches were weighed against the increased search space it pushed onto the evolutionary process and decided not to implement it. We found no approaches where the third auxiliary variable required access to the others as the third variable is rarely necessary. Nevertheless we decided the function of the third variable should have access to the other two as this is consistent with the rest of the architecture and gives the system room to discover new approaches.

5.2 Grammar

We have established that our individuals will be comprised of four mathematical functions with varying inputs. This represents a massive problem space that is hard to navigate since only a small subset of all mathematical functions will be useful as optimizers. In this section we will explain how our grammar was designed to allow for the creation all sorts of functions whilst promoting the structures and patterns found in optimizers. The grammar used can be seen in Appendix A. It must be noted that the grammar presented is abridged. All *const* productions show only the first and last 4 values. *y_func, z_func* and *weight_func* are all abbreviated as they employ the same operations as *x_func*.

The grammar employed for this task has a lot of redundancy by design. In fact, each auxiliary function has five productions (*expr*, *update*, *func*, *terminal* and *const*) and these productions are almost identical across all functions. The reason we chose to keep them separate is due to the way crossover works in SGE (detailed in Section 3.1. While all auxiliary functions have the same role, to allow the *weight_func* to adapt to training as it is happening, they do not all accomplish this task in the same way. The most clear reason for this is that they have access to different information.

If these functions are going to be optimizing towards different things we consider it would be detrimental for them to share genetic material. By removing redundancy from the grammar, combining all the *const* productions for example, we are opening up the possibility of constants that were useful in the context of one function being transferred to another. This can negatively impact the evolutionary process and we would not be fully taking advantage of the engine's ability to separate the genotype into cohesive segments.

The mathematical operations (seen in the *func* productions) were all chosen for their presence in one or more standard optimizers. The most noteworthy operations are those that would seem redundant. Multiply & negative, power & square; the reason we need to keep these separate (as opposed to reproducing one using the other) is due to our constants. The constant values we chose (seen in the *const* productions) are fine tuned

to the problem at hand and do not include some of the more common constants such as -1, 0 and 1. Following a similar approach to the one used in Dynamic AutoLR, the *const* productions have 100 different values. The key difference is that these values follow a sigmoid function as opposed to a linear one. We decided to go with this distribution after analysing the constants typically found in standard adaptive optimizers. While in the previous version of the system the constants played a role as the learning rate, in adaptive methods constants are frequently used as decay rates. These decay rates have different values from learning rates, being either very close to 0 or to 1, leading to the use of the sigmoid function. This is not to say that adaptive methods do not use learning rates. Learning rate constants are still present in these methods but, once again, they tend to take values close to 0 that would not be available if we were to use a linear distribution in our constants.

The components described so far showcase how the grammar was built in order to reproduce standard adaptive approaches. The cost of reproducibility is that we are faced with a very complex search space that will be hard to navigate in a timely fashion. To address this we developed the remaining productions in a way that will facilitate the discovery of key components of the typical optimizer.

Looking at our terminal productions it is clear that all auxiliary terminals have been biased towards using the gradient. If the individual does not make use of the gradient it is not an proper optimizer. The gradient yields all the information related to training. Without it the individual is blindly making arbitrary changes to the weights that should never result in any improvements to network performance. We increase the likelihood of the gradient being chosen over other terminals since we know all feasible solutions will include it in some way. Each terminal production is setup so the chance of the gradient being picked is the same as all other terminals combined.

It must also be noted that the *weight_terminal* production does not have access to the gradient. This was not the case initially but early experimentation suggested that the benefits of using the auxiliary functions are hard for the evolutionary process to find. What this means is that when we allowed *weight_func* to use the gradient directly the system would consistently stagnate with a population of individuals that did not use the auxiliary functions, thus creating static learning rate optimizers. Our response to this issue was to force the individuals to use the auxiliary functions to gain access to the gradient by removing it from *weight_terminal*. This incentivizes the evolutionary process to explore solutions that rely on the auxiliary functions. It is also noteworthy that the methods presented at the start of the chapter are still reproducible without using the gradient in the weight update function. This can be accomplished by moving the gradient into an auxiliary variable and removing the adaptive component of the function. This can be seen in Equation 5.1 where a new variable (y_t) is used to accommodate this change.

$$\begin{aligned} x_t &= x_{t-1} - \nabla l(w_{t-1})^2 \\ y_t &= y_{t-1} - (y_{t-1} - \nabla l(w_{t-1})) \\ w_t &= w_{t-1} - y_t * \frac{\alpha}{\sqrt{-x_t} + \epsilon} \end{aligned} \tag{5.1}$$

Equation 5.1: Gradient descent wrapper applied to Adagrad with the gradient removed from the weight update function

Finally we must discuss the importance of the *expr* and *update* productions as separate entities. As was discussed in the beginning of the chapter all variables are updated using the following formula: $var_t = var_{t-1} - (var\_func)$

Initially we believed this would not impact our grammar in any way. This formula naturally leads our functions to exhibit adaptive behaviour as they are forced to use values from previous iterations to obtain new ones. This might seem desirable but it becomes problematic when combined with our decision to remove the gradient from the *weight_term* production. Under these circumstances the only way for a solution to utilize the gradient is through a competent adaptive auxiliary function. The problem with this is that while developing competent adaptive auxiliary functions is our end goal, we cannot expect the evolutionary process to land on such a function if there is no progression. To elaborate, only a small subset of all auxiliary functions that utilize the gradient are adequate optimizers. Figure 5.1 illustrates this point well as we can see that even the most simple standard adaptive optimizer is quite a complex individual in the context of this framework. This

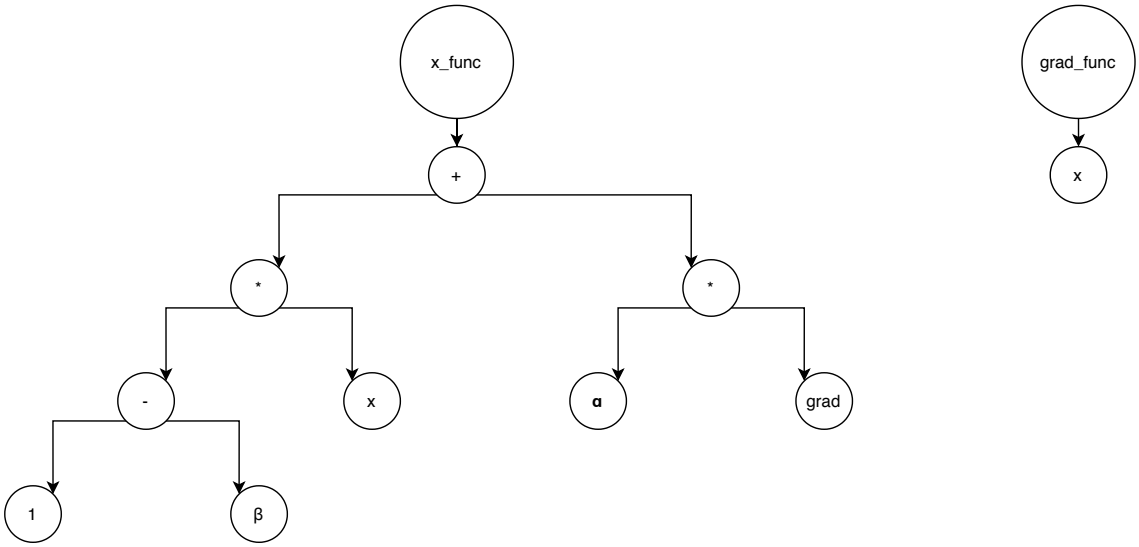


Figure 5.1: Phenotype of a momentum optimizer in Adaptive AutoLR, *y_func* and *z_func* are omitted as they are not used.

problem is addressed by the approach taken in the *expr* productions. The purpose of this approach is to make the auxiliary functions assume one of two "modes". This rule can resolve directly into an update, which in turn will resolve into a mathematical function, leading to a high chance the function will exhibit adaptive behaviour. Alternatively this same rule can be resolved by preceding the mathematical function with a sum with the variable's current value, which inhibits the adaptive aspect of the function as was observed in the y variable in Equation 5.1.

5.3 Fitness

The fitness function for Adaptive AutoLR adheres to many of the principles discussed in Section 4.3 and is quite similar as a result. The large search space used this version of the system leads to a wide range of individuals that are poor optimizers. This happened to some extent in Dynamic AutoLR, but since the solutions created in that version were forced to always return a learning rate from a range of appropriate values, even the worst solutions could be passable.

Many individuals generated by Adaptive AutoLR are not even real optimizers and cannot help in the training process whatsoever. These individuals will naturally be abandoned by the evolutionary process quite quickly but there is a second class of individual that has the

capacity to ruin evolution if left unchecked.

We are talking specifically about individuals that have two silent auxiliary function, one function that returns the gradient and a weight update function that applies it directly. An example can be found in Equation 5.2.

$$\begin{aligned} x_t &= x_{t-1} - (x_{t-1} - \nabla l(w_{t-1})) \\ &= \nabla l(w_{t-1}) \\ y_t &= ... \\ z_t &= ... \\ w_t &= w_{t-1} - x_t \end{aligned} \tag{5.2}$$

Equation 5.2: Example of an individual that can be harmful to the evolutionary process.

There are a few reason that this type of individual can be harmful. The procedure this solution uses is equivalent to using a static learning rate of 1. As was previously discussed using a high learning rate such as this one leads to widely inconsistent results. Also this type of solution is very easy for the evolutionary algorithm to find since the same optimization can be produced by many different genotypes. Combining these two characteristics we get a class of individual that is able to, sporadically, achieve decent results and is very common in our evolutionary process. Early trials done using a similar experimental setup and fitness as the ones used in Dynamic AutoLR were dominated by this type of individual and stagnated quickly as a result.

Besides the superficial issue posed by these individuals (that being that their dominance inhibits the capacity of the system to find adaptive solutions) there is a second, perhaps more pressing problem with these results. An optimizer that can only occasionally produce good results is not a useful optimizer. We consider the value of the optimizer is in its ability to consistently produce good results. If our fitness function's definition of good does not match this requirement it is not fit for this purpose.

The solution was to trial each individual several times on each call of the fitness function. The fitness value returned is the worst test accuracy the individual achieved across all trials. This approach heavily penalizes inconsistency and demands that optimizers are consistently good to achieve a good fitness. Additionally, all individuals are reevaluated even if no changes to their genotype, has been made further preventing unreliable optimizers from exploiting the system. This is much more in line with our own perception of an individual's quality.

In order to further speed up the evolutionary process an additional stop mechanism was introduced. Since optimizers are now being judged based on their worst performance we decided that a minimum acceptable score could be established. If an optimizer's accuracy is below the minimum established score even once then all subsequent trials are cancelled and that accuracy is used as fitness. We consider that as long as the minimum established score is lenient enough this will not harm the evolutionary process. The value used in this version is 0.8 as this was the accuracy that the harmful individuals struggled to overcome.

The function designed to assess performance can be seen in Algorithm 6.

In order to accommodate this new fitness function the way data is split is also different from the approach seen in Dynamic AutoLR. The dataset used is Fashion-MNIST, this choice was explained in Section 4.2. The same concerns are also present. Fashion-MNIST's training examples must be split into training, validation and test. The key difference is

Algorithm 6: Simplified version of the fitness function used to evaluate optimizers in Adaptive AutoLR

```
params: network, learning_rate_optimizer, training_data_groups, validation_data,
        test_data, trial_number,
minimum_acceptable_score ← 0.8;
fitness_score ← 1.0;
trial_count ← 0;
while trial_count < trial_number do
    trained_network ← train(network, learning_rate_optimizer,
     training_data_groups[trial_count], validation_data);
    trial_test_score ← get_test_accuracy(trained_network, test_data);
    if trial_test_score < fitness_score then
        fitness_score ← trial_test_score;
    if trial_test_score < minimum_acceptable_score then
        return fitness_score;
    trial_count ++;
return fitness_score;
```

that now we will separate the training data into several groups, one for each trial. This guarantees the network goes through a unique training process in each trial increasing the effectiveness of this approach. It is important, however, that despite this split all the trials are of similar difficulty. This is why only the training data is different for each trial. The network is evaluated on the same set of test data in all trials. Additionally, we made sure the classes are balanced across all the training groups guaranteeing all trials get access to a representative set of training data. The validation data is also shared by all the trials so that the early stop condition is the same for all of them in case we decide to use it.

5.4 Limitations

While the system presented is able to reproduce the majority of adaptive optimizers there are some limitations to the solution presented. Nesterov momentum is the most notable technique this system is unable to reproduce. As a reminder, Nesterov momentum works by calculating the gradient for the weight modified by the momentum factor.

The CustomOptimizer class only gets access to the gradient for the current weights of the network. We found no way to calculate the modified gradient required by Nesterov momentum without comprising the rest of the system's structure, so this limitation is acknowledged and accepted.

Some adaptive optimizers (e.g. Adagrad) present the option of initializing their auxiliary variables with custom values. This is not possible in Adaptive AutoLR. All auxiliary variables in the system are initialized as 0. It should be noted that all the auxiliary variables from standard optimizers presented include 0 in their range of acceptable values.

It should also be addressed that 0 and 1 are constants that are present in many optimizers but not available in our grammar. These values can be obtained by performing operations on other operations, so we do not consider this a limitation.

5.5 Network Architecture

The network architecture used for Adaptive AutoLR can be found in [51]. This architecture was created for the MNIST dataset but is directly applicable to Fashion-MNIST since the instances in both of these datasets share the same size. During preliminary testing it was found that the network used in Dynamic AutoLR was quite slow to train, due to its many weights and complex architecture. Since efficiently using our resources is very important, for this system we opted to switch to a simpler architecture that could be trained quicker.

5.6 Experimental Setup

5.6.1 Evolution

SGE Parameter	**Value**
Number of runs	5
Number of generations	1500
Number of individuals	20
Tournament size	5
Mutation rate	0.15
Dataset Parameter	**Value**
Training set	53000 instances from the training 10600 instances per trial
Validation set	3500 instances from the training
Test set	3500 instances from the training
Early Stop	**Value**
Patience	5
Metric	Validation Loss
Condition	Stop if Validation Loss does not improve in 5 consecutive epochs
Network Training Parameter	**Value**
Batch Size	1000
Epochs	100
Metrics	Accuracy

Table 5.1: Experimental parameters.

The experimental parameters are summarised in Table 5.1. They are organized into four sections:

- SGE Parameters – parameters of the evolutionary engine.
- Dataset Parameters – number of instances of each of the data partitions.
- Early Stop – the stop condition used to halt the training of the ANN.
- Network Training Parameters – parameters used when training the ANN.

In order for the evolutionary algorithm to effectively navigate the large search space created by our grammar we found that 1500 generations were necessary. This contrasts greatly

with the previous version of the system where we were able find interesting individuals with only 50 generations.

We also found that the evolutionary strategy previously used was not adequate for this new problem space. Evolutionary strategies increase the selection pressure on the population making it hard for mediocre individuals to survive. Trials done with the evolutionary strategy approach were quickly dominated by the harmful individuals seen in Equation 5.2. To counteract this, the population size (i.e. number of individuals) was increased to 20 whilst maintaining the tournament size of 5. The tournament size is an evolutionary parameter that determines how many individuals are considered by the selection method. The selection method picks the best among the individuals considered. Consequently, the closer the tournament size is to the total population size, the harder it is for sub par individuals to reproduce.

These two factors naturally led to a big increase in evaluations. Dynamic AutoLR experiments evaluated at most 250 individuals per run. Adaptive AutoLR always calls the fitness function 30000 times in every run. This might seem unfair to Dynamic AutoLR but we must remind ourselves that since Adaptive AutoLR deals with a much larger search space, this difference is necessary. Since we foresaw this increase in evaluations we have already done most of the optimization necessary through the design of our fitness function and grammar. The final change made for the sake of optimization was the removal of the data augmentation used in Dynamic AutoLR since this mechanism was found to significantly slow down training.

5.6.2 Benchmark

One of the benefits of adaptive methods is that most of them are not very sensitive to initial conditions. This property means it is much easier to establish baseline optimizers than it was in Dynamic AutoLR.

In Dynamic AutoLR our options for baseline policies were limited because static and dynamic optimizers are sensitive to parameterization [30]. This meant it is disingenuous to claim a standard optimizer was performing up to its potential without going through hyperparameter optimization. This is why we settled on a policy we knew worked well for the architecture being used.

Since adaptive optimizers are, by design, more resilient to initial values of hyperparameters, we consider all of them could feasibly be used as a baseline for our benchmarks. In light of this, we opted to use Adam as our baseline as we consider it to be the most consistent optimizer (backed up by the results seen in [31]).

In order to design the scenarios where the optimizers will be benchmarked we must review what properties we are interested in assessing.

First and foremost we wish to compare the performance of evolved optimizers with standard ones. While this can be assessed by testing only the evolved optimizers and Adam, we believe that by including other optimizers we can have a unique insight into the place of the evolved optimizers among mainstream solutions. The optimizers for this scenario were chosen as "representatives" of a certain approach in the field:

- Nesterov Momentum - Momentum-based optimizers.
- RMSprop - Use of discounted moving averages as a scaling factor for the learning rate.

- Adam - Bias correction.

The other hypothesis we wish to test is "Do the evolved learning rate optimizers only perform well on the network they were evolved in?". In order to answer that question another scenario was created where our optimizers and Adam are utilized to solve Fashion-MNIST in the previously used DENSER architecture.

To add more context to the previous scenario, we also decided to test the optimizers on a different problem. By performing this test we can understand whether our optimizers are specializing in the architecture, the problem or neither. The dataset chosen for this scenario was CIFAR10.

CIFAR10 [46] is a common problem addressed by state of the art ANN approaches. The images in this dataset are $32X32$ with 3 color channels. Comparatively speaking, this dataset is harder than Fashion-MNIST due to the increased complexity of the instances. This dataset is comprised of 50000 training examples and 10000 test examples. It should be noted that although the DENSER network, mentioned thus far, was originally evolved for this dataset we opted not to use it in this scenario in favor of a more conventional architecture found in the Keras libraries examples [52].

These benchmark scenarios are summarized in Table 5.2.

Scenario	Network Architecture	Dataset	Optimizers
1	Keras-MNIST	Fashion-MNIST	Evolved
			Adam
			RMSprop
			Nesterov
2	DENSER	Fashion-MNIST	Evolved
			Adam
3	Keras-CIFAR10	CIFAR10	Evolved
			Adam

Table 5.2: Benchmark scenarios for Adaptive AutoLR.

Each scenario is tested with each optimizer five times and the results presented in the next section are the average and standard deviation across the five tests. The data distribution used in Fashion-MNIST and CIFAR10 can be seen in Tables 5.3 and 5.4 respectively.

Dataset Parameter	Value
Train set	53000 instances from training data
Validation set	7000 instances from training data
Test set	10000 instances form test data

Table 5.3: Data distribution of the Fashion-MNIST dataset used in benchmarking.

Dataset Parameter	Value
Train set	43000 instances from training data
Validation set	7000 instances from training data
Test set	10000 instances form test data

Table 5.4: Data distribution of the CIFAR10 dataset used in benchmarking.

5.7 Experimental Results

5.7.1 Evolved Optimizers

Most of our evolutionary runs followed a similar progression. In the first few generations individuals belonging to the class seen in Equation 5.2 would take over the population. This is always followed by a period where the population's fitness is very erratic. Due to the measures taken to prevent the dominance of these individuals they are unable to stabilize their fitness values. This can be observed starting from generation 200 in Figure 5.2. Eventually in all observed runs the recombination of the harmful individuals led to more interesting individuals. In Figure 5.2 this can be seen starting from generation 950 where the average fitness becomes much more stable. We observed the **evolution of known standard optimizers** such as simple SGD (using evolved learning rate constants), traditional momentum and even moving average momentum (similar to the one seen in Adam).

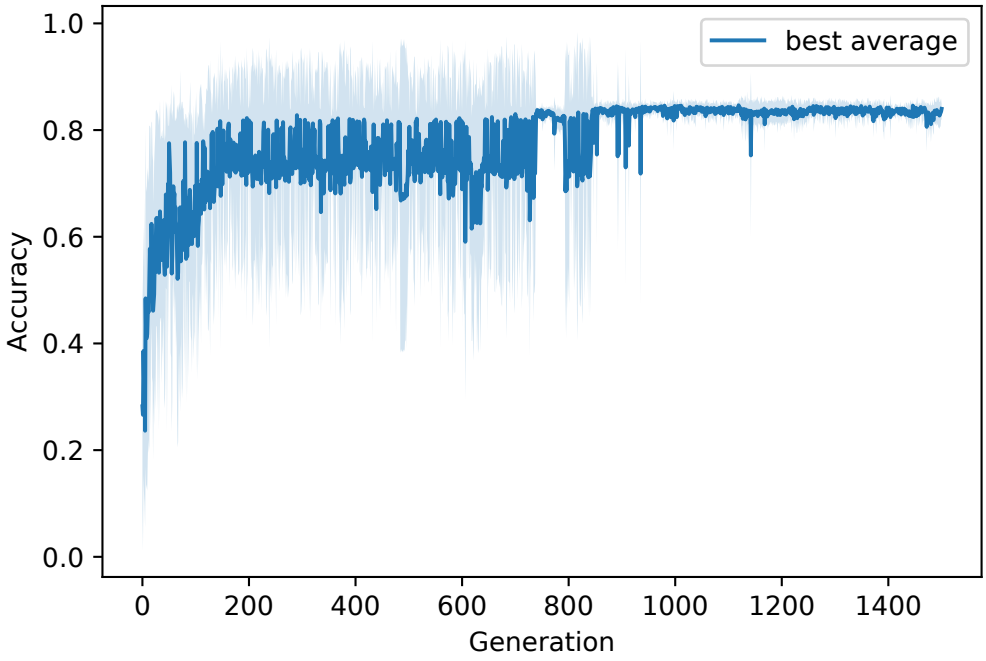


Figure 5.2: Mean best fitness of all experiments over generation.

Another notable type of new optimizer also frequently appeared during evolution. These **evolved optimizers did not follow any structure observed in the state of the art**. The common aspect among them is they run a moving average of a degree 2 polynomial function. While there is a precedent for the use of polynomial functions in learning rate optimization [53, 54, 55] these approaches are not adaptive and make use of much higher degrees. This type of optimizer is, as far as we know, previously undiscovered and thus the best optimizer found in this class was selected for benchmarking. We have named this class of optimizer the Adaptive Evolutionary Squared optimizers (**ADES**). The phenotype of the individual selected can be seen in Equation 5.3. A developed, more readable version of the same individual is also presented in Equation 5.4. These optimizers are hard to understand from a intuitive standpoint (as is typical with solutions resulting from grammatical evolution). While analysis of the properties of this optimizer from a mathematical

standpoint should be possible such a task is outside the scope of this work.

$$\begin{aligned} x_t &= x_{t-1} - (x_{t-1} - (\nabla l(w_{t-1}) * (x_{t-1} + 0,0007674))) \\ y_t &= y_{t-1} - 0,08922 * (y_{t-1} + ((y_{t-1} + 0.999) * (y_{t-1} + \nabla l(w_{t-1})))) \\ z_t &= z_{t-1} - (z_{t-1} + \nabla l(w_{t-1})) \\ w_t &= w_{t-1} + y_t \end{aligned} \tag{5.3}$$

Equation 5.3: ADES optimizer phenotype

$$\begin{aligned} y_t &= (1 - 0,08922)y_{t-1} \\ &\quad - (0,08922 * y_{t-1}^2 + 0,0891 y_{t-1} \nabla l(w_{t-1}) + 0,0891 \nabla l(w_{t-1})) \\ w_t &= w_{t-1} + y_t \end{aligned} \tag{5.4}$$

Equation 5.4: ADES optimizer simplified

However, the best observed optimizer was not a ADES optimizer. The best resulting optimizer from the evolution only appeared in one evolutionary run. This optimizer is also distinct as it opts to ignore the value of the gradient with the exception of the sign. We call this the **Sign Optimizer** after this unique feature. The phenotype of the individual selected can be seen in Equation 5.5. A developed, more readable version of the same individual is also presented in Equation 5.6. Practically speaking this optimizer is actually quite simple. It simply makes very small adjustments in the direction of the gradient regardless of its magnitude. Finally it must be noted that this optimizer is **not** adaptive i.e. this optimizer uses the same learning rate for all weights in the network.

$$\begin{aligned} x_t &= x_{t-1} - (x_{t-1} + x_{t-1})^{x_{t-1} + \nabla l(w_{t-1})} \\ y_t &= y_{t-1} - (y_{t-1} + \nabla l(w_{t-1})) \\ z_t &= z_{t-1} - z_{t-1} \\ w_t &= w_{t-1} + 0.248^2 * \frac{\sqrt{0.0002}}{(\frac{y_t}{\sqrt{(-y_t)^2}})} \end{aligned} \tag{5.5}$$

Equation 5.5: Sign Optimizer phenotype

$$w_t = w_{t-1} - 0,0009 * sign(\nabla l(w_{t-1})) \tag{5.6}$$

Equation 5.6: Sign Optimizer simplified

Since this was the best performing optimizer in all evolutionary runs it was also selected for benchmarking.

5.7.2 Scenario 1

Scenario 1 was devised to see how the two evolved optimizers, ADES and Sign, compare in terms of performance with the standard approaches. To assess this, the network and dataset used in evolution were maintained. In this environment the evolved optimizers should be at their best while the standard adaptive optimizers are in their typical use case (being used out-of-the-box with no parameter optimization). Three metrics were used to evaluate the optimizers: *Validation Accuracy*, *Test Accuracy* and *Generalization*

Rate. While the accuracy metrics are self explanatory *Generalization Rate* requires further explanation. *Generalization Rate* is the quotient of the *Test Accuracy* over the *Validation Accuracy*. This is used as a measure of how well the optimizer is able to generalize (i.e. avoid overfitting). The results for scenario 1 are presented in Table 5.5 and Figure 5.3.

Optimizers	Validation Accuracy	Test Accuracy	Generalization Rate
ADES	92.35 ± 0.25%	92.33 ± 0.14%	**99.99%**
Sign	90.22 ± 0.60%	89.94 ± 0.91%	99.69%
Adam	**93.13 ± 0.23%**	**92.70 ± 0.14%**	99.54%
RMSprop	92.90 ± 0.15%	92.68 ± 0.11%	99.77%
Nesterov	91.85 ± 0.25%	91.23 ± 0.19%	99.32%

Table 5.5: Benchmark results of evolved (ADES, Sign) and standard (Adam, RMSprop, Nesterov) optimizers in scenario 1.

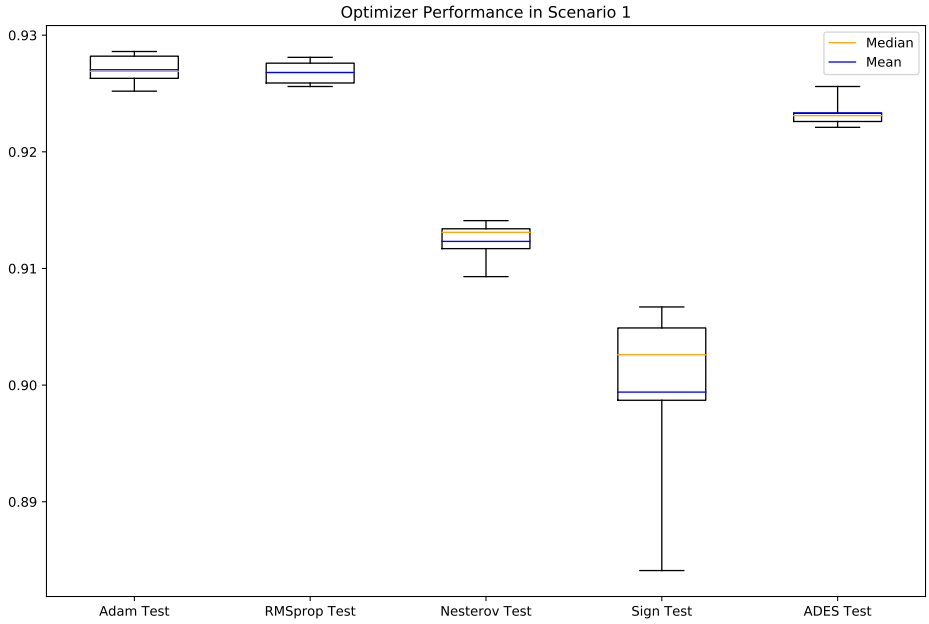


Figure 5.3: Box plot showcasing the results of scenario 1 scenarios. Whiskers show the minimum and maximum result. Box size shows the lower and upper quartile. Mean and median of each population are marked inside each box. Only test accuracy is shown.

The evolved ADES optimizer shows a comparable test accuracy with RMSprop and Adam. It also has the best registered Generalization Rate if only by a small margin. From a historical perspective, these results suggest that the ADES optimizer's performance is somewhere between the momentum-based methods and the discounted average ones.

The Sign optimizer was not able to benefit much from the additional resources provided during benchmarking. It is not easy to understand why this optimizer was favored by the evolutionary process over others. A possible explanation is that several mechanisms put in place to punish the inconsistent individuals ended favouring this overly safe solution. The most likely culprit it the early stop mechanism. While the mechanism's patience was intentionally increased to avoid this sort of situation, it is possible that as long as this mechanism is present, sub-optimal individuals will thrive during evolution. Unfortunately

disabling this mechanism increases the time needed for evolution beyond our available resources ,so experimenting without it was not possible in this work.

In conclusion, this scenario show us that ADES optimizers are able to achieve similar results to standard adaptive optimizers, even surpassing the Nesterov optimizer by a significant margin.

5.7.3 Scenario 2

Scenario 2 showcases the evolved optimizers' capacity to perform in networks different from the one they were evolved in. As a reminder, this scenario utilized the Fashion-MNIST dataset and the neural network from DENSER. The metrics used are the same as in scenario 1. A notable difference is the box plot showing the results of scenario 2 (Figure 5.4) which does not include the ADES optimizer. This optimizer under performed in this scenario using its values would ruin the scale making comparison between more meaningful results impossible. The reason behind this under performance is still being scrutinized.

Optimizers	Validation Accuracy	Test Accuracy	Generalization Rate
ADES	10.01 ± 0.36%	10.00 ± 0.00%	**100.04%**
Sign	88.04 ± 1.12%	87.71 ± 0.66%	99.64%
Adam	**91.59 ± 0.44%**	**91.09 ± 0.26%**	99.46%

Table 5.6: Benchmark results of evolved (ADES, Sign) and standard (Adam, RMSprop, Nesterov) optimizers in scenario 2.

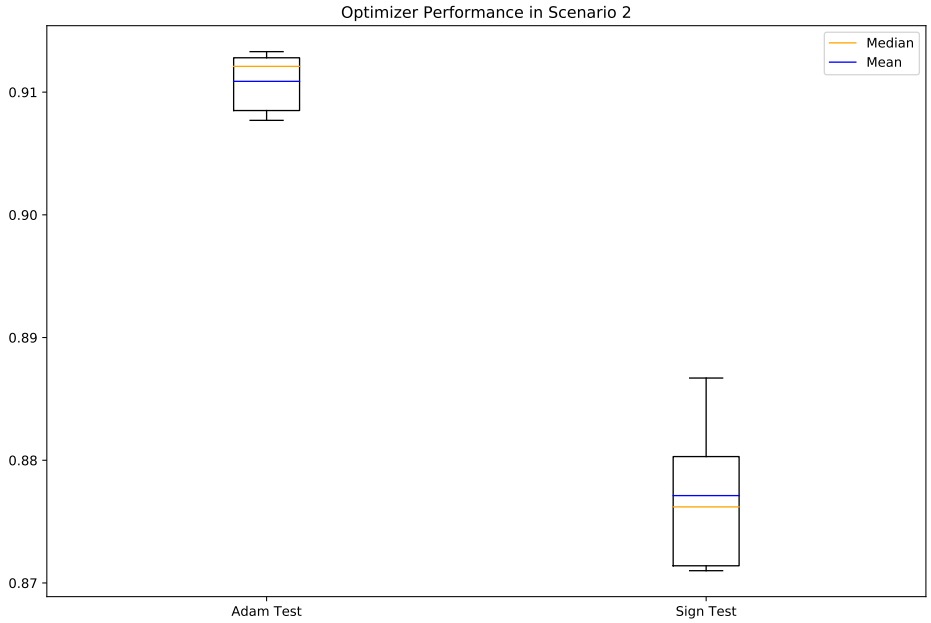


Figure 5.4: Box plot showcasing the results of scenario 2 scenarios. Whiskers show the minimum and maximum result. Box size shows the lower and upper quartile. Mean and median of each population are marked inside each box. Only test accuracy is shown.

The most interesting result in this scenario is that the ADES optimizer was simply unable to train this network architecture. This can be observed in Table 5.6. While it is expectable that the performance of the optimizer decreased when changed to a new network this result suggests that the ADES approach is incompatible with some component of the DENSER network.

Some experiments were performed to try to discern what caused this incompatibility. In these experiments the ADES optimizer was used in several network architectures that included the more niche aspects of the DENSER topology but we were unable to reproduce the incompatibility. This is further address in next chapter in Section 6.1.

This incompatibility is the reason why the benchmarks in scenario 3 were performed with a more standard architecture as opposed to the DENSER one.

The results of Adam and Sign's performance are in line with expectations. In this case the isolated accuracy achieved is not particularly interesting. It it more insightful to analyse how much the evolved optimizer's performance suffers from being moved to a different network. This value in isolation does not gives us the full picture, as it is impossible to tell whether the loss in accuracy is due to the network or the optimizer. Let us instead look at the loss in performance suffered by Sign in comparison to Adam, shown in Table 5.7.

While this measure only works for direct comparisons it is useful to contrast two optimizers' ability to work in different network architectures, since the only varying factor between scenario 1 and 2 is the architecture used.

Optimizers	**Validation Accuracy Difference**	**Test Accuracy Difference**
Sign	-2.18%	-2.23%
Adam	**-1.55%**	**-1.62%**

Table 5.7: Difference in accuracy between optimizer accuracy in scenario 1 and 2. Accuracy difference = average accuracy in scenario 1 - average accuracy in scenario 2. Showcases the optimizer's ability to work in different network architectures.

In this case the results suggest that Adam is not only the superior optimizer but is also more robust to changes in architecture as is expected of a general method.

To elaborate, it is expectable that Adam performs better than Sign due to the results from scenario 1. What the results in Table 5.7 show is that the this gap in performance gets even bigger when the Sign optimizer is moved outside of its native architecture. While the increase is not great (Adam only becomes better by 0.6%) it still suggests that the Sign optimizer is at least partially specialized in the network used in scenario 1.

This is expectable once we consider how the Sign optimizer works. As was discussed in Section 5.7.1, the Sign optimizer is not adaptive. In fact, this optimizer works in a very peculiar way as it discards the actual values of the gradient in favor of using only the sign. The size of weight adjustments made by this optimizer is constant and it is surprising that this optimizer performs as well as it does. Our assessment is that this constant weight adjustment value is intimately related to the problem and network being used and the further removed the optimizer is from these conditions the worse it will perform.

5.7.4 Scenario 3

Scenario 3 yielded the most surprising results. In this scenario the optimizers are moved to a different network and problem. This should further our understanding of how effective the

evolved optimizers are as general out-of-the-box optimizers. The results for this scenario are shown in Table 5.8. Figure 5.5 shows these same results in a box plot. Once again one optimizer has been omitted as its values damage the scale. In this case it is the Sign optimizer that is not present.

Optimizers	Validation Accuracy	Test Accuracy	Generalization Rate
ADES	**80.06 ± 0.74%**	**79.60 ± 0.43%**	**99.44%**
Sign	62.89 ± 1.42%	62.11 ± 2.18%	98.75%
Adam	79.37 ± 0.85%	78.70 ± 0.29%	99.17%

Table 5.8: Benchmark results of evolved (ADES, Sign) and Adam optimizers in scenario 3.

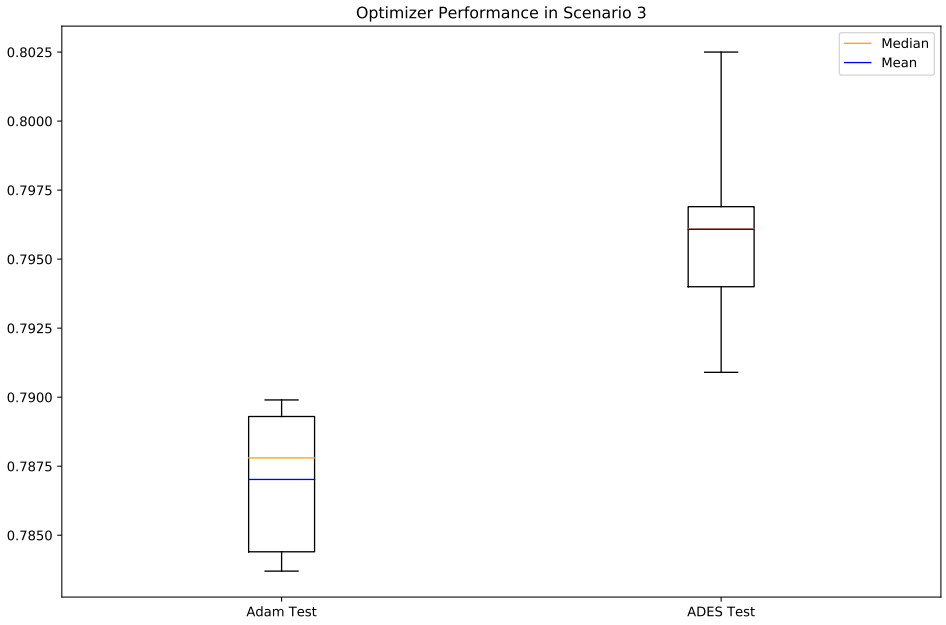


Figure 5.5: Box plot showcasing the results of scenario 3 scenarios. Whiskers show the minimum and maximum result. Box size shows the lower and upper quartile. Mean and median of each population are marked inside each box. Only test accuracy is shown.

There are several interesting insights we can take from these results. Perhaps the most interesting result is that the evolved ADES optimizer is the best performing optimizer in this scenario 3. This suggest that a good general optimizer might have been discovered in the process of evolving optimizers for a specific neural network. While the margin is not large it must still be acknowledged that an evolved optimizer is out performing a state of the art standard approach outside of the network it was evolved for. This result is not per se enough to make substantiated claims about the performance of ADES optimizers but it does raise two other ideas.

First, the results suggest that ADES optimizers may be useful as a standalone contribution, separated from AutoLR. Naturally, further research is necessary (specially considering the results seen in scenario 2), but the fact that this optimizer can be competitive with Adam, outside of its evolutionary environment, is very interesting. It is important to understand that the postulated advantage of an evolved optimizer is that it can learn something about

the network architecture or problem. The fact an evolved optimizer outperformed the state of the art when we removed this supposed advantage is what makes this such an interesting result.

This result also adds value to the use of a system such as AutoLR. Even if upon further inquiry the ADES optimizer is shown to be mediocre, the analysis of this optimizer and why it works may provide useful insights for the development of better standard optimizers in the future. This is specially true in the field of adaptive learning rate optimizers, where we frequently see ideas from weaker methods being retooled to push the state of the art further.

Returning to the original purpose of this scenario, let us also judge the optimizers based on the accuracy difference described in the previous section. Table 5.9 shows the accuracy differences for each of the optimizers when moved from scenario 1 to scenario 3.

Optimizers	**Validation Accuracy Difference**	**Test Accuracy Difference**
ADES	**-12.29%**	**-12.73%**
Sign	-27.33%	-27.83%
Adam	-13.77%	-14.00%

Table 5.9: Difference in accuracy between optimizer accuracy in scenario 1 and 3. Accuracy difference = average accuracy in scenario 1 - average accuracy in scenario 3. Showcases the optimizer's ability to work in different network architectures.

Since the ADES optimizer defied expectations with its performance, when moved to a different network, this is reflected in this metric as well. The ADES optimizer was able to only lose on average approximately 13% of its test accuracy when moved to a different network and much harder dataset. This is about 1% better than Adam which should not be taken lightly when we consider the test set has 10000 examples.

Our assumptions on the nature of the Sign optimizer are also confirmed by these results. In the previous section we stated the Sign optimizer's performance should deteriorate when it is used in contexts that are very different from the one it was evolved in. We can see that this optimizer suffers a massive lost in accuracy in this scenario. This is in line with our proposition, since this scenario has nothing in common with the original evolutionary one (different data, different architecture).

Chapter 6

Conclusion

In this work we wanted to understand whether or not evolving learning rate optimizers for specific neural network architectures could improve their performance. In order to test this we created AutoLR, a framework that evolves learning rate optimizers for specific neural networks. Furthermore two specific versions of this framework were created that focus on specific types of optimizers.

The first, Dynamic AutoLR was created to evolve static and dynamic optimizers, commonly known as policies. The system was then used to evolve two policies. When compared with a widely used and adequate baseline policy both evolved policies were able to improve upon on it in certain scenarios. Most notably, the network used in this version of the system achieved its best recorded performance using an evolved policy. This supports our hypothesis that this evolutionary approach can, in fact, be used to further optimize neural network performance. It must also be mentioned that the best evolved policy resembled a man-made policy seen in [30]. This suggests that the system might implicitly be able to discover the ideas behind the traditional approach. DLR was accepted as a full paper in GECCO2020 [56], the premier conference for genetic and evolutionary computation. The paper is included in B.

A second version of the framework called Adaptive AutoLR was then created to evolve adaptive learning rate optimizers. Since these optimizers are much more complex this system had an increased focus on making the evolutionary process as efficient as possible. Throughout the evolutionary process, this system naturally evolved a few of the more simple standard adaptive optimizers. Once the evolutionary process concluded, we found two optimizers with unique characteristics unseen in the state of the art. These two optimizers were then benchmarked against state of the art optimizers. One of the evolved optimizers, named ADES exhibits very interesting results. A general algorithm for this class of optimizer can be seen in Equation 6.1.

$$\begin{aligned} y_t &= (1-\beta_1)y_{t-1} - \beta_1(y_{t-1}+\beta_2)(y_{t-1}+\nabla l(w_{t-1})) \\ w_t &= w_{t-1} + y_t \end{aligned} \tag{6.1}$$

Equation 6.1: General ADES optimizer

In our best observed individual we noticed that $\beta_2 \approx 1$. The algorithm can be further simplified if we assume $\beta_2 = 1$ as seen in Equation 6.2.

This optimizer is adaptive and unlike anything seen in optimizers presented in the literature. Despite being unique, its performance was comparable with the standard optimizers.

$$
\begin{aligned}
y_t &= (1-\beta)y_{t-1} - \beta(y_{t-1}^2 + \nabla l(w_{t-1}) * (y_{t-1}+1)) \\
w_t &= w_{t-1} + y_t
\end{aligned}
\tag{6.2}
$$

Equation 6.2: General ADES optimizer simplified

Even more notably, when moved outside of the network and dataset it was evolved in the optimizer outperformed the standard baseline adaptive optimizer used.

While further testing is required to properly assess the quality of this new class of optimizer this outcome is promising as it suggests that the optimizer evolved is useful not only in its specific setting but as a general optimization tool as well.

In conclusion, the tool developed was able to accomplish its task effectively and the results support our initial hypothesis.

6.1 Future Work

As was mentioned in the beginning of this document, this work addresses a topic that can be tackled in many different ways. Looking at the course of this work we believe several new opportunities for research were created.

For Dynamic AutoLR there are two extensions we feel could yield interesting results.

The grammar used in this work for this version of the system was quite simple and produced only approximations. In Adaptive AutoLR, a more complex grammar was used and this led to unique and interesting individuals. Taking a similar approach in DLR could also result in new and interesting policies.

DLR was also only used in conjunction with SGD. We found no prior work where schedulers were used to optimize parameters in adaptive methods. Evolving policies for other hyper parameters in more complex optimizers also has the potential to yield interesting results and insights.

For ALR it is possible that evolution could still find other, better individuals. The evolutionary process naturally has diminishing returns as it progresses but we believe that, since the evolved optimizers still only use a small part of the complexity allowed by the grammar, there is potential for even more interesting results.

There are also some possibilities when it comes to expanding the grammar. In this work we created a very low level grammar. Our approach leaves room to insert more high level terminals into the grammar, such as the moving averages found in most adaptive methods. It is possible that through this approach the evolutionary process discovers new and better individuals. It should be noted, however, that this approach should be more resource intensive since we will be calculating a lot of high level information that might never be used.

Finally there is a lot of analysis work that needs to be done on ADES optimizers to understand why they work well and the extent of their usefulness. This was unfortunately outside the scope of this work but it is important to dive deeper into these optimizers to understand how much they can contribute to the field on their own.

References

[1] Aphex34, "typical cnn architecture," December 2015.

[2] M. West, "Example feature maps generated by a trained cnn," June 2020. Pending author approval.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[4] C. Couprie, L. Najman, and Y. Lecun, "Learning Hierarchical Features for Scene Labeling," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1915–1929, 2013.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019.

[7] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature Medicine*, vol. 25, no. 1, pp. 24–29, 2019.

[8] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.

[9] M. M. Lopez and J. Kalita, "Deep learning applied to NLP," *CoRR*, vol. abs/1703.03091, 2017.

[10] A. Senior, G. Heigold, M. Ranzato, and K. Yang, "An empirical study of learning rates in deep neural networks for speech recognition," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2013.

[11] T. M. Breuel, "The effects of hyperparameters on sgd training of neural networks," 2015.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "ResNet," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016.

[13] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and Intelligent Laboratory Systems*, vol. 39, no. 1, pp. 43–62, 1997.

[14] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] C. Blum and K. Socha, "Training feed-forward neural networks with ant colony optimization: an application to pattern classification," in *Fifth International Conference on Hybrid Intelligent Systems (HIS05)*, p. 6 pp., 2005.

[18] T. yau Kwok and D.-Y. Yeung, "Constructive feedforward neural networks for regression problems: A survey," 1995.

[19] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, 1980.

[20] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd ed., 2015.

[21] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Natural Computing Series, Springer, 2003.

[22] B. Kazimipour, X. Li, and A. K. Qin, "A review of population initialization techniques for evolutionary algorithms," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*, pp. 2585–2592, IEEE, 2014.

[23] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, 2001.

[24] M. Keijzer, M. O'Neill, C. Ryan, and M. Cattolico, "Grammatical evolution rules: The mod and the bucket rule," in *Genetic Programming, 5th European Conference, EuroGP 2002, Kinsale, Ireland, April 3-5, 2002, Proceedings*, pp. 123–130, 2002.

[25] A. Thorhauer and F. Rothlauf, "On the locality of standard search operators in grammatical evolution," in *Parallel Problem Solving from Nature - PPSN XIII - 13th International Conference, Ljubljana, Slovenia, September 13-17, 2014. Proceedings*, pp. 465–475, 2014.

[26] N. Lourenço, F. B. Pereira, and E. Costa, "Unveiling the properties of structured grammatical evolution," *Genetic Programming and Evolvable Machines*, vol. 17, pp. 251–289, Sep 2016.

[27] N. Lourenço, F. Assunção, F. B. Pereira, E. Costa, and P. Machado, "Structured grammatical evolution: a dynamic approach," in *Handbook of Grammatical Evolution*, pp. 137–161, Springer, 2018.

[28] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011*, pp. 1–9, 2011.

[29] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[30] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472, IEEE, 2017.

[31] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[32] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, 1988.

[33] Y. NESTEROV, "A method for unconstrained convex minimization problem with the rate of convergence o(1/kˆ2)," *Doklady AN USSR*, vol. 269, pp. 543–547, 1983.

[34] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," *30th International Conference on Machine Learning, ICML 2013*, no. PART 3, pp. 2176–2184, 2013.

[35] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *COLT 2010 - The 23rd Conference on Learning Theory*, vol. 12, pp. 257–269, 2010.

[36] K. S. Geoffrey Hinton, Nitish Srivastava, "Overview of mini-batch gradient descent." University Lecture, 2015.

[37] T. Schaul, S. Zhang, and Y. LeCun, "No more pesky learning rates," *30th International Conference on Machine Learning, ICML 2013*, no. PART 2, pp. 1380–1388, 2013.

[38] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[39] D. P. Kingma and J. 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* (Y. Bengio and Y. LeCun, eds.), 2015.

[40] T. Dozat, "Incorporating Nesterov Momentum into Adam," *ICLR Workshop*, no. 1, pp. 2013–2016, 2016.

[41] N. Lourenço, "sge3: An implementation of dynamic structured grammatical evolution in python 3."

[42] "Tensorflow website." `https://www.tensorflow.org/`.

[43] "Keras optimizers documentation." `https://keras.io/optimizers/`.

[44] D. Lu and Q. Weng, "A survey of image classification methods and techniques for improving classification performance," *International Journal of Remote Sensing*, vol. 28, no. 5, pp. 823–870, 2007.

[45] I. Kanellopoulos and G. G. Wilkinson, "Strategies and best practice for neural network image classification," *International Journal of Remote Sensing*, vol. 18, pp. 711–725, mar 1997.

[46] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *University of Toronto*, may 2012.

[47] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

[48] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "Denser: deep evolutionary network structured representation," *Genetic Programming and Evolvable Machines*, Sep 2018.

[49] "Matlab training option documentation." `https://www.mathworks.com/help/deeplearning/ref/trainingoptions.html`.

[50] "Tensorflow training-ops documentation." `https://www.tensorflow.org/api_docs/cc/group/training-ops`.

[51] F. Chollet *et al.*, "Keras mnist architecture." `https://keras.io/examples/mnist_cnn/`, 2015.

[52] F. Chollet *et al.*, "Keras cifar10 architecture." `https://keras.io/examples/cifar10_cnn_tfaugment2d/`, 2015.

[53] L. C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018.

[54] W. Liu, A. Rabinovich, and A. C. Berg, "Parsenet: Looking wider to see better," *CoRR*, vol. abs/1506.04579, 2015.

[55] P. Mishra and K. Sarawadekar, "Polynomial Learning Rate Policy with Warm Restart for Deep Neural Network," *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, vol. 2019-October, pp. 2087–2092, 2019.

[56] P. Carvalho, N. Lourenço, F. Assunção, and P. Machado, "Autolr: An evolutionary approach to learning rate policies," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, (New York, NY, USA), p. 672–680, Association for Computing Machinery, 2020.

Appendices

Appendix A

ALR Grammar

```
<start> ::= x_func, y_func, z_func, weight_func =
            lambda shape, x, grad: <x_expr>,
            lambda shape, x, y, grad: <y_expr>,
            lambda shape, x, y, z, grad: <z_expr>,
            lambda shape, x, y, z, grad: <weight_expr>,
  <x_expr> ::= add(x, <x_update>) | <x_update>
<x_update> ::= <x_func> | <x_terminal>
  <x_func> ::= negative(<x_expr>)
            | subtract(<x_expr>, <x_expr>)
            | add(<x_expr>, <x_expr>)
            | multiply(<x_expr>, <x_expr>)
            | pow(<x_expr>, <x_expr>)
            | square(<x_expr>)
            | divide_no_nan(<x_expr>, <x_expr>)
            | add(<x_expr>, <x_expr>) | sqrt(<x_expr>)
<x_terminal> ::= tf.constant(<x_const>, shape=shape, dtype=tf.float32) | x
            | grad | grad
  <x_const> ::= 4.53978687e-05 | 5.55606489e-05 | 6.79983174e-05 | 8.32200197e-05
            ...
            | 9.99916780e-01 | 9.99932002e-01 | 9.99944439e-01 | 9.99954602e-01
  <y_expr> ::= add(y, <y_update>) | <y_update>
<y_update> ::= <y_func> | <y_terminal>
  <y_func> ::= negative(<y_expr>) |...| sqrt(<y_expr>)
<y_terminal> ::= tf.constant(<y_const>, shape=shape, dtype=tf.float32) | x | y
            | grad | grad | grad
  <y_const> ::= 4.53978687e-05 | 5.55606489e-05 | 6.79983174e-05 | 8.32200197e-05
            ...
            | 9.99916780e-01 | 9.99932002e-01 | 9.99944439e-01 | 9.99954602e-01
  <z_expr> ::= add(z, <z_update>)| z
<z_update> ::= <z_func> | <z_terminal>
  <z_func> ::= negative(<z_expr>) |...| sqrt(<z_expr>)
<z_terminal> ::= tf.constant(<z_const>, shape=shape, dtype=tf.float32) | x | y
            | z | grad | grad | grad | grad
  <z_const> ::= 4.53978687e-05 | 5.55606489e-05 | 6.79983174e-05 | 8.32200197e-05
            ...
            | 9.99916780e-01 | 9.99932002e-01 | 9.99944439e-01 | 9.99954602e-01
<weight_expr> ::= <weight_func> | <weight_terminal>
<weight_func> ::= negative(<weight_expr>) |...| sqrt(<weight_expr>)
<weight_terminal> ::= tf.constant(<weight_const>, shape=shape, dtype=tf.float32) | x | y | z
<weight_const> ::= 4.53978687e-05 | 5.55606489e-05 | 6.79983174e-05 | 8.32200197e-05
            ...
            | 9.99916780e-01 | 9.99932002e-01 | 9.99944439e-01 | 9.99954602e-01
```

Equation A.1: CFG for the optimisation of learning rate schedulers.

Appendix B

Gecco Paper

AutoLR: An Evolutionary Approach to Learning Rate Policies

Pedro Carvalho
University of Coimbra, CISUC, DEI
Coimbra, Portugal
pfcarvalho@student.dei.uc.pt

Nuno Lourenço
University of Coimbra, CISUC, DEI
Coimbra, Portugal
naml@dei.uc.pt

Filipe Assunção
University of Coimbra, CISUC, DEI
University of Lisbon
Coimbra, Portugal
fga@dei.uc.pt

Penousal Machado
University of Coimbra, CISUC, DEI
Coimbra, Portugal
machado@dei.uc.pt

ABSTRACT

The choice of a proper learning rate is paramount for good Artificial Neural Network training and performance. In the past, one had to rely on experience and trial-and-error to find an adequate learning rate. Presently, a plethora of state of the art automatic methods exist that make the search for a good learning rate easier. While these techniques are effective and have yielded good results over the years, they are general solutions. This means the optimization of learning rate for specific network topologies remains largely unexplored. This work presents AutoLR, a framework that evolves Learning Rate Schedulers for a specific Neural Network Architecture using Structured Grammatical Evolution. The system was used to evolve learning rate policies that were compared with a commonly used baseline value for learning rate. Results show that training performed using certain evolved policies is more efficient than the established baseline and suggest that this approach is a viable means of improving a neural network's performance.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; Supervised learning; **Neural networks**;

KEYWORDS

Learning Rate Schedulers, Structured Grammatical Evolution

ACM Reference Format:
Pedro Carvalho, Nuno Lourenço, Filipe Assunção, and Penousal Machado. 2020. AutoLR: An Evolutionary Approach to Learning Rate Policies. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390158

1 INTRODUCTION

The study of Artificial Neural Networks (ANNs) is a field in modern Artificial Intelligence (AI). These networks' defining characteristic is that they are able to learn how to perform a certain task when provided with an appropriate architecture, data and resources. The networks have a set of internal parameters known as **weights** and **training** is the process through which they are modified so that the network is able to solve a given problem. Fine-tuning the weights of ANNs is crucial in order to obtain a consistently useful system. There are several parameters that regulate training, one the most important parameters is the **learning rate**. In fact, and according to [11, p. 424], if we only have the chance to modify one hyperparameter, the focus should be on the learning rate.

The learning rate determines the magnitude of the changes that are made to the weights. Consequently, the choice of an adequate learning rate is paramount for effective training. When the value of the learning rate is too small the network will be unable to make impactful changes to its weights, making the training slow. On the other hand, if the learning rate is too high the system will make radical changes even in response to small mistakes, causing inconsistent and unpredictable behaviour. On top of this, research suggests that the best training results are achieved by adjusting the learning rate over the course of the training process [22]. One way to make these adjustments during training is by updating the learning rate as training progresses. The functions responsible for these adjustments are known as **learning rate policies**. There is subset of these functions known as learning rate schedulers, i.e., functions that are periodically called during training and return a new learning rate based on multiple training characteristics, such as the current learning rate or the number of performed iterations.

The main objective of this work is to devise an approach that is able to evolve learning rate policies for specific neural network architectures, in order to improve its performance. In concrete, we developed AutoLR, a system that allows us to study the viability of this approach and how it may contribute to the field of learning rate optimization as a whole. Learning rate policies can take many different shapes [23], and therefore it will be notable if our system is capable of automatically discovering functions that are variations of the ones found in the literature. Such a result is interesting because if this approach is able to evolve solutions that are widely accepted it is possible that these same ideas can be used to find still undiscovered, better methods. We are also interested in inspecting the evolved schedulers, and comparing them with human-designed schedulers to obtain meaningful insights. The contributions of this paper are:

GECCO '20, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7128-5/20/07…$15.00
https://doi.org/10.1145/3377930.3390158

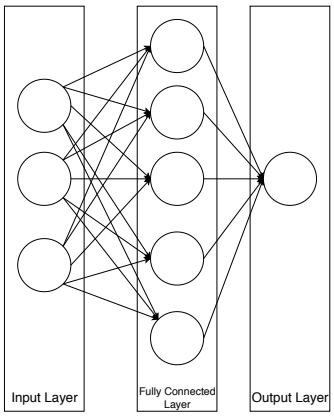


Figure 1: Example of an Artificial Neural Network.

- The proposal of AutoLR, a framework based on SGE that performs automatic optimisation for learning rate schedulers.
- The design, test and analysis of experiments that validate the use evolutionary algorithms to optimise learning rate schedulers.
- We show that the evolved policies are competitive and have characteristics that allow then to thrive in the problems at hand.

The remainder of the paper is organised as follows. Section 2 introduces the background concepts and surveys the key-works related to the optimisation of learning rate schedulers. Section 3 describes AutoLR, the methodology proposed for the evolution of learning rate schedulers. Section 4 details the experimental setup and discusses the experimental results. Finally, Section 5 summarises the main conclusions of the paper and addresses future work.

2 RELATED WORK

This section provides the necessary context for the reader to understand the rest of the paper. Section 2.1 introduces Artificial Neural Networks; Section 2.2 details Structured Grammatical Evolution; and Section 2.3 surveys works related to learning rate optimization and learning rate schedulers.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a machine learning approach that draws inspiration in the biological neural networks seen in nature in order to create a computing system that is able to learn. These systems are comprised by a set of nodes (known as neurons) and edges (known as synaptic weights). An example of the general structure of an ANN is depicted in Figure 1.

Although these networks can have different architectures (e.g., LSTM [13], ResNet [12]) we will, without loss of generality, focus on feed-forward ANNs. In these models the nodes are grouped into separate layers connected sequentially. These layers are flanked by an input and output layer which are responsible for receiving the data that the network will process, and yield the result of the network's calculations, respectively. Edges are directional connections between two nodes from different layers and are the means

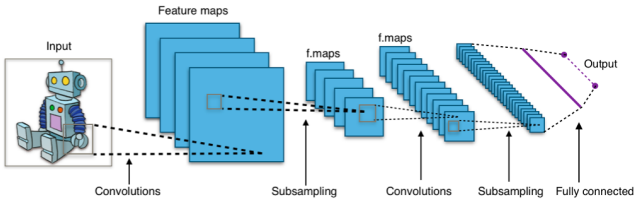


Figure 2: Example of the architecture of a Convolutional Neural Network.

through which information travels through the network. Each node performs an operation (i.e., a mathematical function) on the values it receives from the previous layer and sends the new value to all nodes it is connected to in the next layer. Every edge has a weight that scales the value it carries, i.e., the value that a node outputs is always adjusted before it is provided to the nodes in the next layer.

These networks can be used to solve tasks of many different types. The ideas that will be presented in this work can be widely applied to different types of ANNs. Without loss of generality we will focus in the optimisation of a learning rate scheduler for a supervised learning classification problem. In supervised learning the system is tasked with learning a function that can separate data instances into their respective classes. To achieve this the network is provided with a set of labeled instances.

The training of ANNs is an iterative process where the network compares its attempted classifications of a subset of examples with the expected ones and adjusts its weights to get closer to the correct results. There is a function – known as loss function – that compares the classification and measures how incorrect the network's output was. The size of the changes made to the weights is partially given by the error returned by the loss function (a larger error leads to larger changes). Another parameter, the **Learning Rate** (learning rate), determines the magnitude of the adjustments that are made to the weights. The learning rate is the main subject of this paper. For more details on ANNs refer to [9].

Deep Neural Networks (DNNs) are a subset of ANNs notable for being able to perform representation learning, and consequently the networks are able to automatically extract the features required to solve the problem. This is often associated to the need for deeper architectures, i.e., a greater number of hidden-layers. This allows the networks to possibly solve solve harder problems. In the current work we focus on Convolutional Neural Networks (CNNs) [10], a DNN topology that is known to work well on spatially-related data (e.g., image). An example of the architecture of CNNs is shown in Figure 2. Two layer types are commonly used in CNNs: convolutional and pooling layers. More details can be found in [17]

2.2 Structured Grammatical Evolution

In the current work we will perform the optimisation of learning rate schedulers using SGE. SGE is a variant of GE [19] that uses an altered genotype representation to address the main limitations of GE: low locality and high redundancy. In GE the genotype is encoded as a single list of integers, where each integer encodes a grammatical expansion possibility. Contrary, in SGE there is a separate list for each non-terminal symbol; this avoids the need for

the modulo operation when performing the genotype to phenotype mapping.

These approaches add another layer of decision making however, namely in the form of the grammar design [15]. The grammar used for any GE experiment will define what kind of programs the engine is able to create and this has many implicit consequences. The most obvious one is that the provided grammar must encompass solutions that can solve the problem at hand. While this seems trivial it must be understood that not knowing the composition of the desired program is one of the main motivations to use this type of system in the first place. This also means, however, that the grammar specificity can be increased as more knowledge of the problem is available, aiding the search process.

This specific type of EA is suited for this work because the functions we are looking to evolve are very specific. This means that our domain knowledge is high, and there is a strong understanding of what our desired program is like. As previously mentioned we can use this knowledge to create a grammar that enhances results by narrowing the search space. An in-depth explanation of these algorithms can be found in [9, 21].

2.3 Learning Rate Optimization

In the context of this work, **hyper-parameters** are the set of parameters that configure an ANN and its training strategy. The **learning rate** is one of such parameters and its role is to scale the changes made to the network weights during training. Research suggests that hyper-parameter optimization is effective in improving the system's performance without adding complexity [5].

2.3.1 Static Learning Rate. The traditional approach is to use a single learning rate for the entire training process [20]. Under these circumstances all optimization must be done before training starts. Oftentimes the programmer must rely on expertise and intuition in order to *guess* adequate learning rate values. While automatic solutions to this problem exist they are, to the best of our knowledge, either comparable to manual optimization [5] or non-trivial in implementation [6]. Much of the difficulty of finding a convenient solution to this issue stems from the fact that hyper-parameters are inter-dependent [7]. This means that even when an ideal learning rate is found there is no guarantee that this value remains optimal (or even usable) as the other parameters are tweaked.

2.3.2 Dynamic Learning Rate. The reasons stated in the previous section make the use of a static learning rate a possible drawback. It is desirable that the method we are using to determine our learning rate is robust enough that performance does not dip with every change to the system. In order to increase flexibility we would ideally have a method to change the learning rate as training progresses, i.e., even if the initial value is not adequate the system has a chance to correct its course. This strategy will be referred to as a **dynamic learning rate**. The most uncomplicated policy for varying the learning rate can be inferred intuitively. It is expected that as training progresses the ANN's performance gradually improves as it gets better at solving the task at hand. If the system is potentially closer to its objective it seems desirable that it does not stray from its course. This is to say that, in order to improve, the network requires progressively finer tuning; this can be achieved with a **decaying learning rate** (meaning that the learning rate decreases as learning progresses). There are some issues that are frequently encountered during training that make this approach not ideal however. Better performance is rarely an indicator that the network is closer to a perfect solution. Using a decaying learning rate leaves the system susceptible to early stagnation in a local optimum. This is not ideal despite the fact that a local optimum is sufficient for most situations as this approach can lead to early stagnation if applied incorrectly. Despite these limiting factors decaying learning rates can lead to improvement over static ones as seen in [22].

In order to expand on these ideas we need to apply the concepts of **exploration** and **exploitation**. These refer to the two complementary strategies that can be used in heuristic optimization. Exploration is the idea of using a mechanism that helps the algorithm *explore* solutions that do not seem as promising in an attempt to avoid falling into a local optimum. The contrasting technique is exploitation, in this strategy we adjust our approach to make sure the algorithm is able to find the local optimum (once it reaches a promising region). Finding a proper balance between these two strategies is crucial for further improvement of the dynamic learning rate. *Smith et. al.* propose the use of a *cyclic learning rate* in [23]. Their approach fluctuates the learning rate between a maximum and a minimum bound. While the system uses no information about whether or not it is stuck by periodically increasing the learning rate it is able to explore the search space more effectively. This technique is consequently less vulnerable to early stagnation than decaying learning rate policies. This method is, to the best of our knowledge, the most efficient use of dynamic learning rates.

2.3.3 Adaptive Learning Rate. Further improvements in this area can still be achieved if the system responsible for assigning the learning rate has access to information throughout training. This means that we will now study algorithms that can acknowledge when training is stagnating as it is happening. From this point onward we refer to these methods as **adaptive learning rates**.

These techniques unlock one more option of optimization. So far we have been working with a single value learning rate but with this extra information it is desirable to use a vector of values instead. Consider the following scenario, an ANN is being trained for 100 generations with a single value adaptive learning rate. One specific weight of the network reaches a near optimal value within the first 5 generations, but all of the others are still off the mark. An adaptive learning rate recognizes this and has to decide what is the ideal learning rate value for the next generation. On the one hand, using a small learning rate will benefit the fine tuning of the node that is already performing well. A larger learning rate, on the contrary, will allow the sub optimal weights to find better values. Using vectors of learning rates allows the system to have a learning rate value for each weight, making the most out of these nuanced situations [14]. Several algorithms [8, 16, 24] have been built on this theoretical foundation and these systems are the best learning rate policies we know of.

3 AUTOLR: EVOLUTION OF LEARNING RATE SCHEDULERS

AutoLR is a framework created to apply evolutionary algorithms to learning rate policy optimization. While SGE is used to handle the evolutionary processes, the system's novelty comes from using the algorithm to explore new possibilities in the learning rate policy

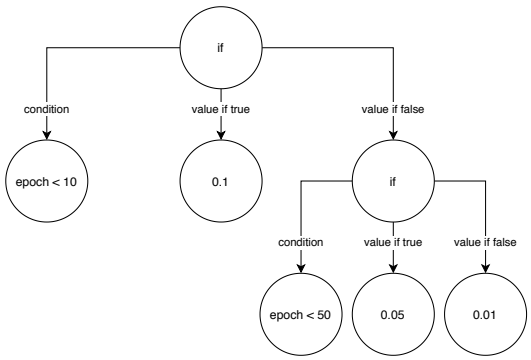


Figure 3: Example of a learning rate scheduler.

search space. This is achieved through the design of a grammar that is able to effectively navigate part of this space and a fitness function that can accurately measure each policy's quality.

3.1 Evolved Policies

The scope of this work is limited to evolving learning rate schedulers. We define learning rate schedulers as it is done in the Keras[1] library. Learning rate schedulers are functions that are called periodically during training (each epoch, in this case) and update the learning rate value. In other words, we are evolving the initial learning rate and the ensuing variation function. These functions' inputs are comprised of the learning rate of the previous epoch and the number of performed epochs. This function returns a single learning rate for all dimensions. Using the terminology established so far, this means **the evolved policies can be either a static or dynamic learning rate solution**. It is important to define the range of our solutions as this establishes what conventional techniques we should be kept in mind during analysis.

Figure 3 depicts an example of a learning rate scheduler. In this case the ANN will train using a learning rate of 0.1 for the first 10 epochs as this is when the condition *epoch* $<$ *10* is met. This learning rate will be used until the 10th epoch is reached, at which point the learning rate scheduler will automatically decrease the learning rate to 0.05. Following the same rationale, after the 50th epoch the learning rate to use is 0.01. The search space that we consider is detail on the next sub-section.

3.2 Grammar

The grammar (Figure 4) defines the search space of the learning rate schedulers. The individuals created by this grammar will typically resolve into a sequence of chained if-else conditions (created by the *logic_expr* production) that once evaluated yield a learning rate (provided by the terminals in *lr_const*). This means that the system is creating dynamic learning rate policies most of the time. A notable exception to this is that the system can resolve the initial *expr* production into a *lr_const*, creating a static learning rate policy.

An *if_func* is a simple function that does the same as a regular if-then-else construct. Since the code for this system was written in Python this function was created so all individuals could be described in a single line that can be read easily by the user. The code for this function is shown in Algorithm 1.

```
<expr> ::= if_func(<logic_expr>, <expr>, <expr>)
         | <lr_const>
<logic_expr> ::= learning_rate <logic_op> <lr_const>
         | epoch <logic_op> <ep_const>
<logic_op> ::= < | ≤ | > | ≥
<lr_const> ::= 0.0001 | 0.00110909 | 0.00211818 | 0.00312727 |
         ...
         0.09596364 | 0.09697273 | 0.09798182 | 0.09899091 |
         0.1
<ep_const> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
         ...
         91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99| 100
```

Figure 4: Grammar used for the optimisation of learning rate schedulers.

The conditions used by *if_func* are generated by *logic_expr*. This production will compare one of the input variables (learning_rate, epoch) with the corresponding constants (lr_const and ep_const, respectively) using one of several logical operators from *logic_op*. *logic_op* includes all logical operators with the exception of equality (==) and inequality (!=). Conditions using these operators are too specific since they only return a different value for a single constant. This means that, in the vast majority of situations, conditions using these operators do not change the policy's behaviour. This makes them unable to contribute meaningfully to the evolutionary process.

The constants chosen for *lr_const* and *ep_const* are 100 evenly spaced values between the minimum and maximum value for each of the variables. It should be noted that these production rules have been abridged in the figure. Only a few of the lowest and highest possible values are shown so that the range is accurately portrayed whilst keeping the figure brief. Our training starts in epoch 1 and ends in epoch 100, since we are also using 100 values for our constant we used every possible epoch value (every natural number from 1 to 100) for *ep_const*. *lr_const* values are more complicated as there is an infinite number of valid learning rates. We keep the values of the learning rate bounded between 0.001 and the 0.1 as all values in this range are suitable for training.

This grammar is capable of creating a large variety of individuals despite its simplicity. While it is not possible for our trees to exactly recreate the dynamic solution functions mentioned in Section 2.3 they can reproduce approximated versions that exhibit similar behaviour.

3.3 Fitness Function

As the main hypothesis implies we are looking to evolve learning rate policies. This means that we will be using an EA on a population of learning rate policies. Additionally, our hypothesis demands that an individual's fitness must be some measure of the network's performance when trained using that specific solution. This is necessary since if the evolutionary process is not successful, its results will not address the question we posed.

Algorithm 1: Template of the code used to implement the if_func routine.

```
params: condition, state1, state2
if condition then
    return state1;
else
    return state2;
```

We decided that the best way to assess a policy's performance was through the function seen in Algorithm 2. That is, we train the network and assess its performance using the accuracy metric.

Algorithm 2: Simplified version of the fitness function used to evaluate a learning rate policy

```
params: network, learning_rate_policy, training_data,
        test_data
trained_network ← train(network, learning_rate_policy,
  training_data);
fitness_score ← get_test _accuracy(trained_network,
  test_data);
return fitness_score;
```

To elaborate on the algorithm above, our fitness function will use 4 components

- **network** - The ANN we are optimizing the learning rate scheduler for. This network is the same throughout the entire evolutionary run.
- **learning_rate_policy** - An evolved learning rate scheduler that we want to evaluate.
- **training/test_data** - This is the data of the problem the ANN will be attempting to solve. As the name implies, training data is used for training. Test data is a separate set of examples that are used to evaluate the network's performance once training is complete. In the actual fitness function the training data is further split into training and validation (see 4.3) but this distinction will temporarily be omitted for explanation's sake.

The evaluation function has two phases. First, the network must be trained, this is where the policy we are evaluating will affect the process. The **train** function returns the network provided with its weights changed through the training process. We could at this point also retrieve the best performance the network achieved during training. We do not take this approach as it is not the most accurate measure of an ANN's real effectiveness. The objective of the training is that the network learns a set of weights that solve the proposed problem. The data used for training is only a sample of all possible inputs. As training progresses a network becomes gradually too attached to the training data, this is known as **overfitting**. Overfitting means that the network is to constraint to the training data, and does not represent the general learning problem. This happens since data will often have some noise (i.e. information that is not important to solve the task). It is not desirable for the network to learn to produce solutions based on this noise as that will hurt its performance when dealing with inputs not included in training. Consequently, we measure the effectiveness of training by how well the network performs on a second set of data that it has not come into contact with. We call this second set the test data.

Every policy will be evaluated using the same network and training data meaning that the learning rate scheduler is the only varying component between individuals. Since all other hyper-parameters are fixed, and the used datasets are balanced, we consider the result of evaluating the trained network's **accuracy** on the test data to be an adequate measure of the policy's fitness.

In the context of our work, learning rate policies are executable computer code. We will be using the Python language specifically as it has vast support for ANN handling through the **Tensorflow** [3] library. An EA is also needed for our system, we chose to use GE-based evolutionary engine as it gives us a flexible and readable means of defining the problem space in the form of grammars. In particular, we chose SGE [18] for its Python implementation and superior results over regular GE. Our hypothesis also demands a mindful choice of network architecture. Since we are looking for optimization in specific scenarios, we want to avoid generic architectures. We therefore decided to use a CNN model evolved specifically for image classification obtained from Deep Evolutionary Network Structured Representation (DENSER) [4].

4 EXPERIMENTATION

The objective of this work is to promote the automatic optimisation of learning rate schedulers for a fixed-topology network. Section 4.1 introduces the topology of the used network; Section 4.2 details the dataset; Section 4.3 describes the experimental setup; and Section 4.4 analyses and discusses the experimental results.

4.1 Network Architecture

The network architecture we used was automatically generated using DENSER [4] – a grammar-based NeuroEvolution approach. The CNN optimised by DENSER was evaluated using a fixed learning rate strategy, and thus it is likely that better learning policies exist. The architecture was generated for the CIFAR-10 dataset using a fixed learning rate of 0.01, where the individuals were trained for 10 epochs. The details of how the network was created are important as they might inform our conclusions later on. The specific topology of the network is described in Figure 5.

4.2 Dataset

We opted to use the Fashion-MNIST instead of the network's native CIFAR-10 as it is a dataset where the training is faster. This dataset is composed by 70000 instances: 60000 for training and 10000 for testing. Each instance is 28×28 grayscale image, which contrasts with CIFAR-10's 32×32 RGB images. We will be scaling our images into 32×32 RGB as they would not fit the network's input layer otherwise. This scaling was performed using the nearest neighbour method, and to pass from one to three channels we replicate the single channel three times.

4.3 Experimental Setup

We divide the experimental setup into two parts: the parameters used for the evolutionary search (Section 4.3.1); and for a longer training after the end of evolution (Section 4.3.2).

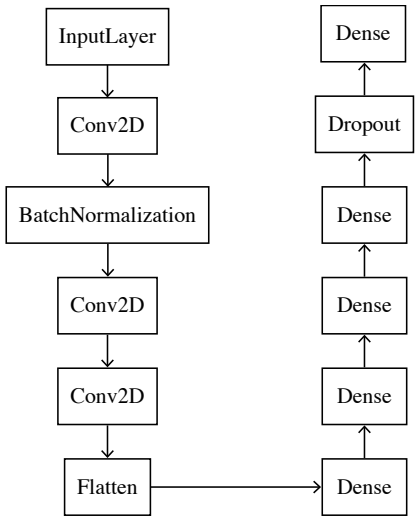


Figure 5: Topology of the used CNN.

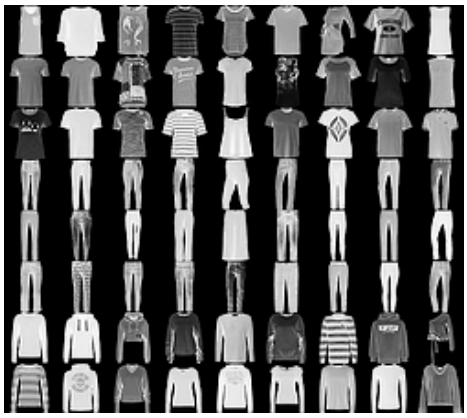


Figure 6: Example images from the Fashion-MNIST dataset.

4.3.1 Evolution. The experimental parameters are summarised in Table 1. They are organized into five sections:

- SGE Parameters – parameters of the evolutionary engine.
- Dataset Parameters – number of instances of each of the data partitions.
- Early Stop – the stop condition used to halt the training of the ANN.
- Training Data Augmentation – real time data augmentation parameters.
- Network Training Parameters – parameters used when training the ANN.

Our experimental parameters were picked with some considerations. Since evolutionary algorithms are very demanding in terms of computation resources it was paramount that the parameters used allowed us to perform meaningful evolutionary runs that could be completed in an acceptable time-frame. This motivated the selection of parameters that effectively reproduce an evolutionary strategy. Additionally, the fitness function operates on a fraction of the dataset as training utilizing all 60000 training examples was too time consuming. We also picked the training parameters accordingly. Ideally, we would perform evolution on 100 training epochs with no early stop as we are trying to optimize the network's performance as much as possible. Instead, we performed two sets of experiments: (i) using 100 epochs and an early stop mechanism; (ii) using 20 epochs with no early stop. We started by reducing the computational cost through the implementation of an early stop mechanism. Notwithstanding, we were concerned that the evolutionary process would exploit this mechanism, which motivated the 20 epochs experience, where no early stop is used and the cost is instead reduced by reducing the training epochs.

SGE Parameter	**Value**
Number of runs	10
Number of generations	50
Number of individuals	5
Mutation rate	0.15
Dataset Parameter	**Value**
Training set	7000 instances from the training
Validation set	1500 instances from the training
Test set	1500 instances from the training
Training Data Augmentation	**Value**
Feature-wise Center	True
Feature-wise Std. Deviation	True
Rotation Range	20
Width Shift Range	0.2
Height Shift Range	0.2
Horizontal Flip	True
Early Stop	**Value**
Patience	3
Metric	Validation Loss
Condition	Stop if Validation Loss does not improve in 3 consecutive epochs
Network Training Parameter	**Value**
Batch Size	1000
Epochs	100 / 20
Metrics	Accuracy

Table 1: Experimental parameters.

Dataset Parameter	**Value**
Train set	52500 instances from training data
Validation set	7500 instances from training data
Test set	10000 instances from test data

Table 2: Dataset information for parameters.

4.3.2 Testing. After the evolutionary process is complete we need to properly assess the quality of the generated policies. The testing routine is the same as our fitness function, differing only in the data used (seen in Table 2).

In our testing routine we use all training instances (splitting them into training and validation) to train the network using the policy

Scenario		Policy		
		A	B	Baseline
1	Validation	0.751 ± 0.167	n/a	**0.859 ± 0.003**
	Test	0.692 ± 0.241	n/a	**0.850 ± 0.004**
2	Validation	n/a	0.854 ± 0.009	**0.856 ± 0.004**
	Test	n/a	**0.848 ± 0.007**	0.844 ± 0.002
3	Validation	**0.894 ± 0.004**	0.891 ± 0.003	0.888 ± 0.002
	Test	**0.887 ± 0.002**	0.854 ± 0.009	0.875 ± 0.004

Table 3: Accuracy of the evolved policies (A & B) on their evolutionary environment (1 & 2 respectively) and scenario 3 (representative of an actual use case), compared with the baseline policy.

we want to evaluate. This network is subsequently tested using all test data (that was not used previously) to obtain an unbiased **test accuracy**. We will also be tracking each policy's best **validation accuracy** to have additional insight into how well the learned weights are able to generalize. Finally, we need to decide on a policy to serve as a baseline. We chose to use a *static learning rate policy of 0.01* for three reasons. The fact that the network was evolved using this learning rate (as was explained in Section 4.1). This assures us that this is an adequate learning rate for this network making it a good benchmark for our evolved policies. Additionally, this particular constant is the most common policy in often used Deep Learning frameworks [1, 2]. We believe that benchmarking against such a widely used policy is a proper way to test our hypothesis. Finally, we had to use a baseline that had similar information to our evolved methods. The adaptive techniques referred to in Section 2.3, for example, use the gradient of the loss function to make more precise adjustments to a per-dimension learning rate. The fact that these methods have access to additional information means they are not suitable as benchmarks.

We have three testing scenarios:

- The **first scenario (1)** is the same as the first evolutionary scenario, i.e., training is done for **100 epochs with the early stop mechanism**.
- The **second scenario (2)** trains for only **20 epochs, with no early stop**.
- The **third scenario (3)** trains for **100 epochs, but the early stop mechanism is disabled**.

The first and second scenarios exist primarily so we can see how the evolved policies compare with the baseline in the conditions they were evolved in. Scenario 3 yields the most important results as its conditions represent the typical use case of a neural network.

In order to make discussion clearer the evolved polices will be referred to as policy A (for the best policy evolved with the early stop mechanism) and policy B (for the best policy evolved with no early stop). These evolved polices were tested in their evolutionary environments (scenario 1 and 2 for policies A and B respectively) and in scenario 3. The baseline policy was tested in all 3 scenarios.

4.4 Experimental Results

The table presented in 3 summarizes the results of our experimentation, showing the average and standard deviation of the accuracy of a given policy in a specific scenario over five runs. As detailed in Section 4.3.2, each run trains the network using the chosen policy and subsequently tests its accuracy on the 10000 test instances.

4.4.1 Scenario 1. yields results that are not intuitive given the circumstances. Training in this scenario can be halted by an early stop mechanism. Since policy A was evolved using this same kind of training it is to be expected that it would perform well in these conditions. However, the results show the opposite. Policy A, in fact, performs far worse then the baseline when early stop is in use. Analysing individual results showed that this policy will occasionally trigger the early stop in the first few epochs (this can be observed in the large standard deviation associate with these trials). There are several interpretations for the implications this has on the validity of the evolutionary process. On the one hand it can be argued that this demonstrates an issue with the evolutionary process since the policy is not a consistent solution to the problem it is supposed to solve. While it is a fact that the policy is an inconsistent solution we do not believe this implies any problems with the evolution. The fact that this policy can, on occasion, yield the best performance implies the genetic information of this individual is useful for the evolutionary process.

4.4.2 Scenario 2. results are more in line with our expectations. We can observe that, albeit only marginally, policy B shows better test accuracy than the baseline when trained under the parameters it was evolved for. It is noteworthy that policy B does not have superior accuracy in validation. This suggests that the **evolved policy is outperforming the baseline in its ability to generalize** when moved to a different set of data.

4.4.3 Scenario 3. was designed to test which policy is able to get the most out of this network's architecture and it gave the most important set of results. The results show that, under these conditions, **the best accuracy this network achieved was obtained using an evolved policy for training**. On average, policy A performs better than the baseline in the test set by 1.2% and it obtains these good results more consistently. Another interesting result is that policy B (that was previously outstanding because of its ability to generalize) suffers the biggest dip in performance from validation to test in this scenario. Ideally, both evolved policies would outperform the baseline. There are, however, some possibly limiting factors. Namely, it is possible that the shorter training duration used in scenario 2 discourages the evolution of policies that translate well into scenario 3. This topic is discussed further in 4.4.4 as we analyse policy B's shape.

4.4.4 Shape. As discussed in 1, we are interested in analysing the shapes that our evolved policies take. Namely, in this section we will be analysing the shape of the previously discussed Policies A and B. These policies can be observed in Figures 7 and 8. These figures show how the learning rate evolved over time as well as a vertical line that signals the epoch where the training using this policy stopped.

Observing the shape of policy A (seen in Figure 7) led to some interesting insights. Initially, it seemed that this policy only had the best performance during evolution because its shape cheat the early stop mechanism. We suspected that by frequently using a high learning rate it might be possible to create false improvements that trick the system. To elaborate, it is feasible for a policy to routinely worsen and subsequently improve its performance *on purpose* in

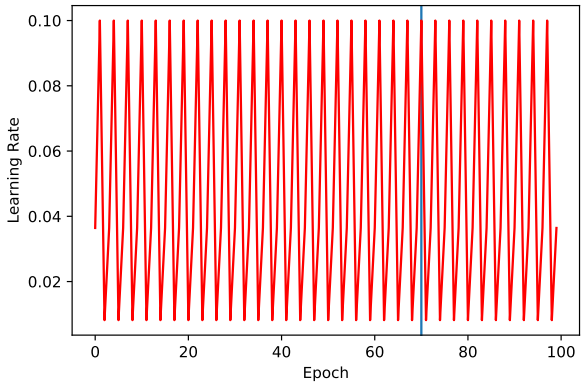


Figure 7: Policy A

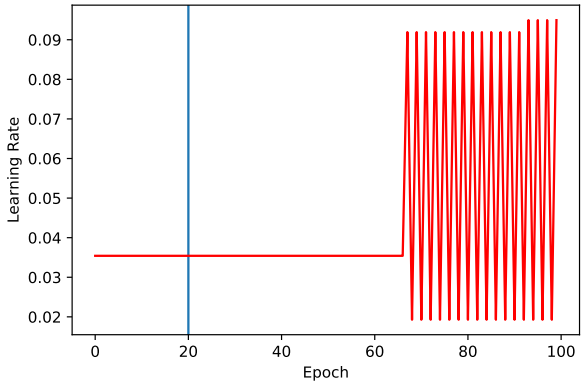


Figure 8: Policy B

order to pass the early stop check. We can, in fact, observe that this policy is able to train for a long time despite the early stop as the vertical line shows. As a comparison, the baseline policy typically triggers the early stop between epochs 20-30, which means that policy A is able to train for twice as long.

Policy B took on a very different shape. Despite the erratic behaviour shown past epoch 60, this policy is effectively a static learning rate as its training always ended in epoch 20 (as a reminder, no early stop was used in the evolution of the policy). While this initially seems disappointing (finding an adequate constant is not something that requires such a complex system), it is important to understand that, due to the reduced training duration, there is a possibility that the benefits of using a dynamic policy in this context are negligible, stifling probability that they show up in evolution. The idea that the evolution of dynamic methods is suppressed under these circumstances is further supported by the fact that all twelve of the best policies during the evolution of policy B were constants. In this context the twelve best policies we are referring to is the set of policies that were, at some point during evolution, the best policy in all runs.

We have, up until this point, observed two types of evolved functions shapes (within the individuals that perform well). The first type is constants, these comprise the majority of the search space so their presence is expected. The second type can be observed in policy A, we refer to these as *oscillator policies*. We believe that these policies are approximations of the policies used in [23]. While it would be disingenuous to claim that we are evolving cyclical policies, it seems feasible that the evolved oscillator policies are effective for the same reasons as the cyclical ones. It is notable that while many known dynamic policies are decaying policies we have not observed any well performing evolved policies with a similar shape.

5 CONCLUSIONS AND FUTURE WORK

In this work we posed the question of whether or not evolving learning rate policies was a viable way of improving a network architecture's performance. To test this, we designed and developed AutoLR. a framework that optimizes learning rate policies using SGE. Furthermore, this framework was then utilized to create two evolved policies. These evolved policies were tested and compared with a widely used baseline policy. Both of the policies evolved were able to improve on the established baseline in some capacity. Not only that, the network's best recorded performance was achieved with an evolved policy, suggesting that **evolving learning rate policies for a specific architecture did in fact improve the network performance**. Additionally, some of the evolved policies resemble man-made policies seen in [23], suggesting that the system might have implicitly discovered the ideas that make such policies effective. In the future we would like to expand the range of policies that can be evolved to enable meaningful comparisons with a wider array of state of the art methods.

6 ACKNOWLEDGEMENTS

This work is funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020.
Filipe Assunção is partially funded by: Fundação para a Ciência e Tecnologia (FCT), Portugal, under the PhD grant SFRH/BD/114865/2016. We also thank the NVIDIA Corporation for the hardware granted to this research.

REFERENCES

[1] [n. d.]. Keras Optimizers Documentation. https://keras.io/optimizers/. ([n. d.]). Accessed: 2020-01-19.
[2] [n. d.]. Matlab Training Option Documentation. https://www.mathworks.com/help/deeplearning/ref/trainingoptions.html. ([n. d.]). Accessed: 2020-01-19.
[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.
[4] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. 2018. DENSER: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines* (27 Sep 2018).
[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing*

Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011 (2011), 1–9.

[6] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13 (2012), 281–305.

[7] Thomas M. Breuel. 2015. The Effects of Hyperparameters on SGD Training of Neural Networks. (2015). arXiv:1508.02788 http://arxiv.org/abs/1508.02788

[8] John Duchi, Elad Hazan, and Yoram Singer. 2010. Adaptive subgradient methods for online learning and stochastic optimization. *COLT 2010 - The 23rd Conference on Learning Theory* 12 (2010), 257–269.

[9] A. E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing* (2nd ed.). Springer Publishing Company, Incorporated.

[10] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* (1980). https://doi.org/10.1007/BF00344251

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. ResNet. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2016). https://doi.org/10.1109/CVPR.2016.90 arXiv:arXiv:1512.03385v1

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* (1997). https://doi.org/10.1162/neco.1997.9.8.1735

[14] Robert A. Jacobs. 1988. Increased rates of convergence through learning rate adaptation. *Neural Networks* (1988). https://doi.org/10.1016/0893-6080(88)90003-2

[15] Maarten Keijzer, Michael O'Neill, Conor Ryan, and Mike Cattolico. 2002. Grammatical Evolution Rules: The Mod and the Bucket Rule. In *Genetic Programming, 5th European Conference, EuroGP 2002, Kinsale, Ireland, April 3-5, 2002, Proceedings*. 123–130. https://doi.org/10.1007/3-540-45984-7_12

[16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (2014), 1–15. arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[18] Nuno Lourenço, Francisco B. Pereira, and Ernesto Costa. 2016. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines* 17, 3 (01 Sep 2016), 251–289.

[19] Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation* (2001). https://doi.org/10.1109/4235.942529

[20] Russell Reed and Robert J MarksII. 1999. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press.

[21] Conor Ryan, Michael O'Neill, and JJ Collins. 2018. *Handbook of Grammatical Evolution*. Springer.

[22] Andrew Senior, Georg Heigold, Marc'Aurelio Ranzato, and Ke Yang. 2013. An empirical study of learning rates in deep neural networks for speech recognition. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. https://doi.org/10.1109/ICASSP.2013.6638963

[23] Leslie N Smith. 2017. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 464–472.

[24] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. (2012). arXiv:1212.5701 http://arxiv.org/abs/1212.5701