AutoLR: An Evolutionary Approach to Learning Rate Policies

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

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

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

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 of 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 a 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:
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.

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, 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 harder problems. In the current work we will 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-terinal 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
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 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 utilized 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.

### 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.

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.
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).
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.

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
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.

| Scenario | Policy       | 1st Validation | 2nd Validation | 3rd Validation |
|----------|--------------|----------------|----------------|----------------|
|          | A            | B              | A              | B              |
| 1        | 0.751 ± 0.167 | n/a            | 0.859 ± 0.003  | 0.850 ± 0.004  |
|          | 0.692 ± 0.241 | n/a            | 0.856 ± 0.004  | 0.856 ± 0.004  |
| 2        | n/a          | 0.854 ± 0.009  | 0.848 ± 0.007  | 0.854 ± 0.002  |
|          | n/a          | 0.854 ± 0.009  | 0.888 ± 0.002  | 0.875 ± 0.004  |
| 3        | 0.894 ± 0.004 | 0.891 ± 0.003  | 0.887 ± 0.002  | 0.854 ± 0.009  |
|          | 0.887 ± 0.002 | 0.854 ± 0.009  | 0.875 ± 0.004  | 0.875 ± 0.004  |

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 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.

4.4.2 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
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 to 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 SCE. 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 Ph.D 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/. (In d.). Accessed: 2020-01-19.
[2] [n. d.]. Matlab Training Option Documentation. https://www.mathworks.com/help/deeplearning/ref/trainingoptions.html. (In d.). Accessed: 2020-01-19.
[3] Martin 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, Fernando 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, Pernousal 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.
[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 (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