Evolving Adaptive Neural Network Optimizers for Image Classification

Pedro Carvalho(✉), Nuno Lourenço, and Penousal Machado

CISUC, Department of Informatics Engineering, University of Coimbra, Polo II - Pinhal de Marrocos, 3030 Coimbra, Portugal
{pfcarvalho,naml,penousal}@dei.uc.pt

Abstract. The evolution of hardware has enabled Artificial Neural Networks to become a staple solution to many modern Artificial Intelligence problems such as natural language processing and computer vision. The neural network's effectiveness is highly dependent on the optimizer used during training, which motivated significant research into the design of neural network optimizers. Current research focuses on creating optimizers that perform well across different topologies and network types. While there is evidence that it is desirable to fine-tune optimizer parameters for specific networks, the benefits of designing optimizers specialized for single networks remain mostly unexplored.

In this paper, we propose an evolutionary framework called Adaptive AutoLR (ALR) to evolve adaptive optimizers for specific neural networks in an image classification task. The evolved optimizers are then compared with state-of-the-art, human-made optimizers on two popular image classification problems. The results show that some evolved optimizers perform competitively in both tasks, even achieving the best average test accuracy in one dataset. An analysis of the best evolved optimizer also reveals that it functions differently from human-made approaches. The results suggest ALR can evolve novel, high-quality optimizers motivating further research and applications of the framework.

Keywords: Neuroevolution · Adaptive Optimizers · Structured Grammatical Evolution

1 Introduction

Artificial Neural Networks (ANN) are an essential part of modern Artificial Intelligence (AI) and Machine Learning (ML). These systems are popular as solutions in a variety of different tasks such as computer vision [5,11], and natural language processing [8,12].

ANN's design is loosely inspired by the workings of the biological brain. Like their biological counterpart, ANNs are comprised of several inter-connected units called neurons. Each connection has an associated value called *weight* which determines the strength of the connections between neurons. When using an ANN for a specific task, a suitable set of weights is necessary to solve the problem.

E. Medvet et al. (Eds.): EuroGP 2022, LNCS 13223, pp. 3–18, 2022.
https://doi.org/10.1007/978-3-031-02056-8_1

The process through which these weight are found is called *training*. Proper training is vital for ANN performance, motivating extensive research into how ANNs should be trained [1,6,9,10,14,20]. As a result, several methodologies and hyper-parameters were developed to tune the training process.

One vital hyper-parameter is the *Learning Rate* (LR), a numeric value that scales changes made to the weights during training. The choice of LR value has a profound impact on the effectiveness of training, motivating the researchers to create various solutions (known as optimizers) to optimize the size of the changes made during training. While optimizers vary in their complexity and effectiveness [1,6,10,14], one aspect most optimizers share is their generality. Since training is ubiquitous across most applications of ANNs, optimizers are designed to be effective on a wide variety of problems and ANN architectures. This general approach has led to the creation of optimizers that are effective and easy to apply, but it also raises the question: Can optimizers be pushed further if we specialize them for specific problems?

To answer this question, we must first establish a way to specialize optimizers for a specific problem. It is challenging for humans to understand all the dimensions required for manual specialization because ANNs comprise many interdependent components and parameters. However, it is possible to use a search algorithm to perform this specialization automatically. Evolutionary algorithms (EA) are strong candidates for this task; these heuristic algorithms can navigate complicated problem spaces efficiently through biologically inspired procedures (e.g., crossover, mutation, selection). Using an EA, it is possible to test several different optimizers and combine the best performing ones to achieve progressively better results. The benefits of specialization can then be assessed by comparing the evolved optimizers with standard, human-made optimizers.

This work uses an evolutionary framework to create optimizers for specific ML problems. The resulting evolved optimizers are benchmarked against state-of-the-art hand-made optimizers. Finally, the applicability of evolved optimizers to different problems is also tested. The results suggest that the evolved optimizers can compete with the human-made optimizers developed over decades of research. Additionally, one of the evolved optimizers, ADES, remains competitive even when applied to tasks that were not addressed during evolution, suggesting EAs may be used to create generally applicable optimizers. Finally, ADES does not function like human-made optimizers; hinting that the evolutionary approach can find creative solutions undiscovered by humans.

The structure of this paper is the following: In Sect. 2 we give historical background on the human-made optimizers created over the years. In Sect. 3 we describe how Adaptive AutoLR, the evolutionary framework, is used to evolve ANN optimizers. The components developed for this work are also presented and discussed. In Sect. 4 we present the experiments performed and discuss the results. The evolutionary parameters used are presented as well as the resulting optimizers. In this we section also compare the evolved optimizers with human-made solutions in performance and ability to generalize. In Sect. 5 we review the work presented in this article and summarizes our contributions.

2 Background

In a typical training procedure, after each training epoch, the system compares the ANN's output with the expected output and calculates the error. Based on this error, back-propagation [19] is used to calculate the changes that should be made to each weight (known as the gradient). The gradient, often scaled by the LR, is frequently use to dictate the direction and size of weight changes.

The original LR optimizer, SGD [1], simply sets the new weight (w_t) to be the difference between the old weight (w_{t-1}) and the product of the learning rate (lr) with the gradient (∇l), shown in Eq. 1.

$$w_t \leftarrow w_{t-1} - \mathrm{lr} * \nabla l(w_{t-1}) \tag{1}$$

Traditionally, a single LR value is used for the entirety of the training. In this case, all the tuning must be done before the training starts. The problem with this approach is that one is often forced to rely on experience and trial-and-error to find an adequate static LR. Research also suggests that different LR values may be preferable at different points in training [20], meaning a single, static LR is rarely ideal.

These limitations led to the creation of dynamic LRs which vary the LR value as training progresses. Dynamic approaches are frequently used [7,22] because they are easy to implement and usually outperform static LRs [20]. However, these approaches are limited because they only change the size of changes based on the training epoch. This is shortcoming motivated the development of the more sophisticated adaptive optimizers.

Adaptive optimizers are variations of SGD that use long-term gradient information to adjust the changes made. In adaptive optimizers, the LR is a static value combined with weight-specific auxiliary variables. While it is possible to utilize gradient information to adjust a single LR value, most adaptive optimizers use different rates for each weight. The result is an ANN optimizer that allows each weight to be updated at a different rate. The most straightforward adaptive optimizer is the momentum optimizer [9], shown in Eq. 2. The auxiliary variable is a momentum term (x_t) that increases the size of adjustments made to weights that keep changing in the same direction. Two constants accompany this term: the learning rate (lr) is responsible for directly scaling the gradient, the momentum constant (mom) dictates how strong the effect of the momentum is.

$$\begin{aligned} x_t &\leftarrow mom * x_{t-1} - lr * \nabla l(w_{t-1}) \\ w_t &\leftarrow w_{t-1} + x_t \end{aligned} \tag{2}$$

A variation of the momentum optimizer, known as Nesterov's momentum [14] is presented in Eq. 3. Nesterov's momentum varies from the original because the gradient is calculated for the weight plus the momentum term. As a result, the optimizer can look-ahead and make corrections to the direction suggested by the momentum. The look-ahead is beneficial because the momentum term is slow to change which may hinder the training process.

$$\begin{aligned} x_t &\leftarrow mom * x_{t-1} - lr * \nabla l(w_{t-1} + mom * x_{t-1}) \\ w_t &\leftarrow w_{t-1} + x_t \end{aligned} \tag{3}$$

RMSprop [6] is an unpublished optimizer that divides the LR by a moving discounted average of the weights' changes. This optimizer will decrease the LR when the weight changes rapidly and increase it when the weight stagnates. This LR annealing simultaneously helps the weights converge and prevents them from stagnating. In Eq. 4, x_t is the moving average term, and ρ is the exponential decay rate used for this same average. The root moving average is then used in w_t to scale the LR and gradient.

$$\begin{aligned} x_t &\leftarrow \rho x_{t-1} + (1-\rho)\nabla l(w_{t-1})^2 \\ w_t &\leftarrow w_{t-1} - \frac{lr * \nabla l(w_{t-1})}{\sqrt{x_t} + \epsilon} \end{aligned} \tag{4}$$

The final optimizer we will be discussing is Adam [10]. Adam is similar to RMSprop, but it attempts to correct the bias of starting the moving average at 0 using a new term (z_t). Adam also calculates a range where it expects the gradient to remain consistent ($\frac{x_{t-1}}{\sqrt{y_{t-1}}}$). In Eq. 5 x_t and y_t are both moving averages; β_1 and β_2 are exponential decay rates for the averages (similar to ρ in Eq. 4).

$$\begin{aligned} x_t &\leftarrow \beta_1 x_{t-1} + (1-\beta_1)\nabla l(w_{t-1}) \\ y_t &\leftarrow \beta_2 y_{t-1} + (1-\beta_2)\nabla l(w_{t-1})^2 \\ z_t &\leftarrow lr * \frac{\sqrt{1-\beta_2^t}}{(1-\beta_1^t)} \\ w_t &\leftarrow w_{t-1} - z_t * \frac{x_t}{\sqrt{y_t} + \epsilon} \end{aligned} \tag{5}$$

3 Adaptive AutoLR

AutoLR is an open-source [16] framework developed to evolve ANN optimizers. This framework has previously been used to evolve dynamic LR policies [2]. In this work, we propose **Adaptive AutoLR** (ALR), an implementation of the framework capable of evolving the more complex adaptive ANN optimizers. This framework is used to create optimizers specialized for specific tasks to assess the benefits of optimizer specialization and the potential of evolved optimizers.

For this work, ALR is used to create, evaluate, and improve optimizers during the **evolutionary phase**. A separate **benchmark phase** is performed, where the evolved optimizers are fairly compared with the human-made solutions. In the following sections, we will describe the grammar used to determine the structure of the optimizers and the fitness function utilized to quantify the quality of the evolved solutions.

3.1 Grammar

Adaptive optimizers are comprised of a few functions that calculate a set of auxiliary variables and adjust the weights of the ANN. This definition is expansive, creating a complex problem space that demands many evaluations during evolution. The grammar must account for the problem's difficulty, enabling diversity while promoting a smooth evolutionary process.

A consequence of the adaptive optimizer's broad definition is that the majority of possible solutions cannot train the ANN. It is possible to counteract this issue through a restrictive grammar that limits the types of functions that can be evolved. However, we are interested in promoting novel optimizers as much as possible and will avoid such restrictions as a result. The complete grammar used for ALR cannot be included due to space restrictions, but an abridged version is presented in Fig. 1 (full version is available in [15]).

```
<start> ::= x_func, y_func, z_func, weight_func =
            <x_expr>, <y_expr>, <z_expr>, <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>)
           | 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> ::= <x_const> | x | grad | grad
<x_const> ::= 4.53978687e-05 | ... | 9.99954602e-01
...
<weight_expr> ::= <weight_func> | <weight_terminal>
<weight_func> ::= negative(<weight_expr>) |...
<weight_terminal> ::= <weight_const> | x | y | z
<weight_const> ::= 4.53978687e-05 | ... | 9.99954602e-01
```

Fig. 1. CFG for the evolution of ANN optimizers.

Individuals in ALR are made up of 4 functions, named: x_func, y_func, z_func and $weight\_func$. Functions x through z work as the auxiliary functions found in human-made adaptive optimizers; these functions have an associated result stored between epochs (e.g., x_t). By default, the previous iteration result is included in the function, as shown in Eq. 6, but this behavior can be unlearned

using the grammar provided. These stored values are a staple of adaptive optimizers as they are essential to implement mechanisms such as momentum.

$$\begin{aligned} x_t &\leftarrow x_{t-1} - ... \\ y_t &\leftarrow y_{t-1} - ... \\ z_t &\leftarrow z_{t-1} - ... \\ w_t &\leftarrow w_{t-1} - ... \end{aligned} \tag{6}$$

When the training algorithm calls the optimizer, the individual utilizes its functions to calculate the new weight values. The auxiliary functions are called first; the role of these functions is to calculate and store relevant information based on the gradient changes. These functions are executed sequentially, starting with x and ending with z. The order is essential because each auxiliary function has access to the result of those that precede it. After all auxiliary functions have been executed, the *weight* function is called with access to all the results. The result of the weight update function, $weight\_func$, is then used as the weight for the next epoch.

There are some aspects of the grammar design that must be discussed. It should be noted that several productions in the grammar used are identical, but they are not combined in order to keep the genotype of each function isolated. The operations and constants were chosen for their presence in human-made adaptive optimizers. The grammar also includes some bias to facilitate evolution. The weight function is not allowed to use the gradient; this encourages the use of auxiliary functions. Auxiliary functions' terminals are biased in favor of the gradient, so it is picked more often. Additionally, the *expr* productions are biased to facilitate the removal of the function's previous iteration from the calculations.

3.2 Fitness Function

ALR is usable in any ML application that employs gradient-based training. In this work, we focus on applying ALR to image classification as there is a vast backlog of research on the topic that provides proven models and datasets. Specifically, we chose Fashion-MNIST as it is a good balance between an easy dataset (e.g., regular MNIST) and a harder one (e.g., CIFAR-10). The ANN used in ALR can be found in [18]. This ANN is compatible with the Fashion-MNIST dataset and trains quickly as it has a small number of weights.

The objective of ALR is to create solutions that maximize the accuracy of the ANN's predictions. As a result, the fitness function (shown in Algorithm 1) will utilize the evolved optimizer to train an ANN and use its accuracy after training and fitness.

However, additional measures are implemented to ensure the fitness value accurately measures the solution's actual quality. Specifically, the data used by the fitness function is split into three sets. The **evolutionary training set** is used to train the ANN; this is the only data that interacts with the optimizers directly. The **evolutionary validation set** is used to calculate validation metrics to track training progress. An early stop mechanism also monitors the validation loss, aborting the training when the validation loss stagnates. The early

Algorithm 1: Simplified version of the fitness function used to evaluate optimizers in ALR

```
params: network, optimizer, evolutionary_training_data,
        evolutionary_validation_data, fitness_assignment_data,
        evaluation_number
minimum_acceptable_accuracy ← 0.8;
fitness ← 1.0;
evaluation_count ← 0;
while evaluation_count < evaluation_number do
    trained_network ← train(network, optimizer, evolutionary_training_data,
    evolutionary_validation_data);
    evaluation_accuracy ← get_accuracy(trained_network,
    fitness_assignment_data);
    if evaluation_accuracy < fitness then
        fitness ← evaluation_accuracy;
    if evaluation_accuracy < minimum_acceptable_accuracy then
        return fitness_score;
    evaluation_count ++;
return fitness;
```

stop mechanism helps prevent over-fitting and saves computational resources. After training is complete, the ANN is used to classify the third set of data, the **fitness assignment set**. We consider that the accuracy of the ANN in this final dataset is an accurate measure of the optimizer's fitness. We refer to this process of training and calculating the accuracy of the ANN as an **evaluation**. It is worth noting that there are other desirable optimizer features that this fitness function does not account for, such as convergence speed and hyper-parameter sensitivity.

We found that some solutions were inconsistent, producing very different fitness values when repeating the same evaluation. Consequently, we consider that multiple evaluations should be used to calculate the fitness. Specifically, the optimizers are trained and evaluated up to five times. While five evaluations is insufficient to perform any statistical analysis, we found that it was enough to nurture the evolution of stable solution. The evolutionary training data is split among the evaluations, forcing the solutions to train using different data each time. Since each evaluation is computationally expensive and it is desirable to minimize the number of evaluations. As a result, we define a *minimum acceptable accuracy*. If the accuracy achieved in the fitness assignment set is below this threshold, the optimizer is not considered a viable solution, and the rest of the evaluations are canceled. This mechanism significantly reduces the resources used to evaluate low-quality solutions. We consolidate all the results into a single fitness value using the *worst accuracy across all evaluations* as this further incentivizes the system to produce consistently good solutions.

4 Experimental Study

This section documents the experiments performed to validate ALR. In Sect. 4.1 we detail the configuration used for the evolutionary process, going over the parameters used to configure ALR and train the ANN. In Sect. 4.2 we present and analyze the results of evolution. The typical progress of an evolutionary run is discussed, and the most notable evolved optimizers are showcased. In order to properly compare the quality of the evolved optimizers to human-made solutions, additional experiments are performed to benchmark their quality; this procedure is documented in Sect. 4.3. Benchmarks are performed on two different problems. In Sect. 4.4 we present and discuss the performance of the optimizers in the problem used in evolution, Fashion-MNIST. In Sect. 4.5 we conduct the same analysis in a different image classification task, CIFAR-10.

4.1 Evolutionary Runs

ALR has a set of evolutionary parameters that must be configured for experimentation. The parameters used in our experiments are presented in Table 1. The search space posed in this problem is vast; as a result, we found it adequate to use a high number of generations and a small population. This combination of parameters is likely to stagnate the population, so a large tournament size is used to reduce selective pressure.

Fashion-MNIST is the dataset used to evolve the optimizers, and it is comprised of training (refer to as Fashion-MNIST-Training) and test (Fashion-MNIST-Test) data. We will only use Fashion-MNIST-Training (60000 examples) in the evolutionary runs, splitting it into the evolutionary training set, evolutionary validation set, and fitness assignment set with 53000, 3500, and 3500 examples, respectively. The 53000 evolutionary training examples are split evenly among the evaluations (resulting in 10600 training examples per evaluation). The Fashion-MNIST-Test is deliberately excluded from the evolutionary process; it is essential to reserve a set of data that the evolved solutions never interact with to draw fair comparisons with human-made optimizers later. Additionally, the early stop mechanism interrupts training when the validation loss does not improve for 5 consecutive epochs (controlled by the *Patience* parameter). Each evaluation trains the ANN for a maximum of 100 epochs using a batch size of 1000. We empirically found that using these parameters with a human-made optimizer was sufficient to train competent networks.

4.2 Evolutionary Results

Figure 2 shows the averages of the best solution and average population quality across all runs throughout the evolutionary process. The typical behavior of the runs can be described as follows. In an early stage, the population is dominated by individuals that utilize the gradient directly to adjust the weights, without an LR or any type of adaptive components. While these individuals can train ANNs adequately occasionally, they fail to replicate their success across different

Table 1. Experimental parameters.

SGE Parameters	**Value**
Number of runs	9
Number of generations	1500
Number of individuals	20
Tournament size	5
Crossover rate	0.90
Mutation rate	0.15
Dataset Parameters	**Value**
Dataset	Fashion-MNIST-Training (60000 instances)
Evolutionary Training set	53000 instances, 10600 instances per evaluation
Evolutionary Validation set	3500 instances
Fitness set	3500 instances
Early Stop	**Value**
Patience	5
Metric	Validation Loss
Condition	Stop if Validation Loss does not improve in 5 consecutive epochs
ANN Training Parameters	**Value**
Batch Size	1000
Epochs	100
Metrics	Accuracy

evaluations. Nevertheless, these individuals play a vital role in the evolutionary process as they identify the importance of including the gradient. In most runs, this genetic material is used in more robust optimizers that can consistently train competent ANNs leading to an increase in solution quality.

The best and most robust evolved optimizers employ simple, familiar mechanisms like a static LR or a simple momentum term. However, two evolved optimizers stood out as worthy of a focused study and further experimentation. Since evolved optimizers have a considerable amount of unused genetic material that hurts readability (e.g., complex auxiliary functions that are not used in calculating the weights), we will be presenting simplified versions of the optimizers to improve clarity.

The first notable individual was the best performing optimizer across all runs; a simplified version of this optimizer is shown in Eq. 7; in the instance produced in evolution $lr = 0.0009$. This optimizer is unusual in a few ways. The gradient is never used directly, only its sign. As a result, this optimizer always changes the weights by a fixed amount. We named this optimizer the *Sign Optimizer*. The size of the changes proposed in the evolved instance of this optimizer is small, likely leading to successive small changes that steadily improve the ANN until a local optimum is reached. This strategy allows for a lengthy training procedure even with the early stop mechanism. Early stop only

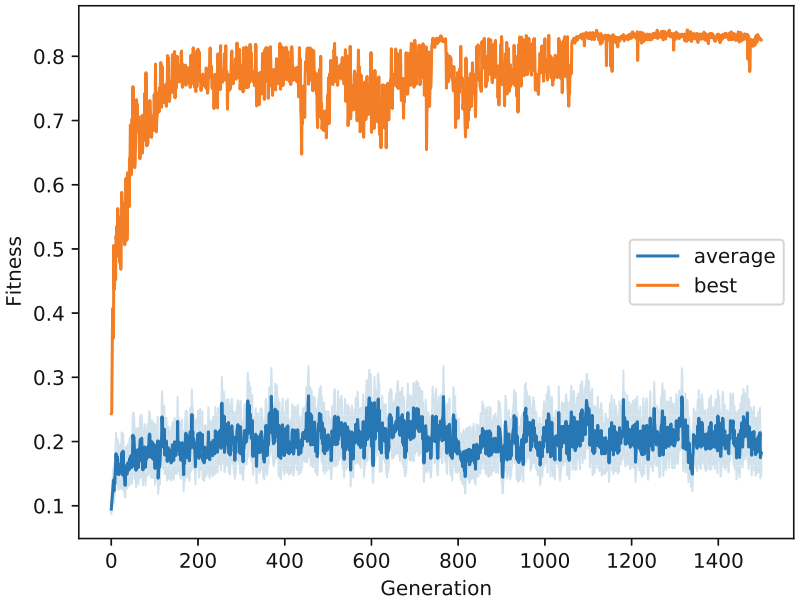

Fig. 2. Progression of average fitness and best fitness throughout evolution. Plot shows the average and standard deviation across all runs.

interrupts training if the validation loss does not improve. The magnitude of improvements is not considered, favoring optimizers that improve the ANN by a small margin many times. Additionally, the use of the sign operation adjusts the direction of the gradient before it is applied to the weights. As far as we know, this strategy is not used in any human-made optimizers, and it is challenging to understand its implications completely. While changing the direction of the gradient seems undesirable, this approach may make the optimizer more resistant to the common vanishing/exploding gradient problems that can occur during training. The Sign optimizer does not exhibit any adaptive components; since adaptive optimizers were the main object of this system, we also selected the best optimizer with adaptive features for benchmark.

$$w_t = w_{t-1} - lr * sign(\nabla l(w_{t-1})) \tag{7}$$

The best adaptive evolved optimizer is presented in Eq. 8. As far as we know, this individual is a novel adaptive optimizer. Specifically, this solution's unique aspect is the presence of a squared auxiliary variable that was not found in human-made approaches. This optimizer is named Adaptive Evolutionary Squared (ADES) after its defining characteristic; in the instance produced in evolution $\beta_1 = 0.08922, \beta_2 = 0.0891$. ADES is considerably more complex than the Sign optimizer, so assessing how and why it functions is challenging. Since ADES does not operate similarly to human-made optimizers, we cannot relate its operations to familiar components. Nevertheless, we empirically observed that

this optimizer employs a momentum-like mechanism that increases the magnitude of changes when moving in the same direction. However, we consider that thoroughly dissecting ADES and understanding the role of all its components is outside the scope of this work as it requires significant study outside the field of evolutionary computation.

$$\begin{aligned} y_t &= y_{t-1} - (\beta_1 * y_{t-1}^2 + \beta_2 * (y_{t-1} * \nabla l(w_{t-1})) + \beta_2 * \nabla l(w_{t-1})) \\ w_t &= w_{t-1} + y_t \end{aligned} \tag{8}$$

4.3 Benchmark

The evolved optimizers performed well during evolution, but this may not be representative of their actual quality. During evolution, optimizers were evaluated using a limited set of data and an early stop mechanism. Furthermore, training was restricted to 100 epochs at most, possibly limiting the quality of ANNs created. In order to determine the actual quality of evolved solutions, it is essential to benchmark them with human-made optimizers.

Benchmarks use three sets of data with distinct roles: training, validation, and test. Training data, as the name suggests, is used to train the ANN. The validation data is used to monitor the ANN's ability to generalize as training progresses. The test data is used to make a final assessment of the ANN's performance. Furthermore, every benchmark is comprised of two phases. In the first phase (**tuning phase**), Bayesian optimization [13] tunes the hyper-parameters of all optimizers. While Bayesian optimization is often used to search for optimizer and network parameters [21], we believe it remains an adequate solution when applied in this smaller scope. Bayesian optimization tunes all parameters between 0 and 1. Specifically, the algorithm performs 100 function evaluations with 10 restarts and each optimizer's default parameters are used as a probe to help guide the search. During the tuning phase, ANNs are trained using the selected optimizer and parameters during 100 epochs. The Bayesian optimization procedure is guided by the best validation accuracy obtained during training. In the second phase (**trial phase**), the best set of hyper-parameters (i.e., the values that achieved the highest validation accuracy) found for each optimizer through the tuning process are used to train the ANN. In this phase, training is performed for 1000 epochs. After training, the best weights (i.e., the weights that achieved the highest validation accuracy during training) are used to test the ANN on the test data for a final accuracy assessment. Trials are repeated 30 times, and all results presented show the average and standard deviation of these trials. No early stop is used at any point during benchmarks.

The evolved optimizers are compared with three human-made optimizers previously presented: Nesterov's momentum, RMSprop and Adam on two different benchmarks. The first benchmark compares the evolved optimizers with the human-made adaptive optimizers in Fashion-MNIST (the task used in evolution). The second benchmark performs the comparison on CIFAR-10, a different dataset and network architecture (Keras-CIFAR [3], available in [17])), but the task is still image classification.

4.4 Fashion-MNIST

In this benchmark, we test the optimizers in the network architecture (Keras-MNIST [4], available in [18]) and dataset (Fashion-MNIST) used in evolution. Additional measures are necessary to ensure fair comparisons in this environment, since evolved optimizers have an unfair advantage if evaluated on the data used during evolution. As previously mentioned, the Fashion-MNIST-Test was deliberately excluded from evolution to enable just comparisons in this phase. Consequently, it is possible to make fair comparisons between evolved and human-made optimizers as long as the Fashion-MNIST-Test data is used to evaluate the final accuracy. As a result, Fashion-MNIST-Training is used for training (53000 instances) and validation (7000 instances), and Fashion-MNIST-Test is only used for the final accuracy assessment (10000 instances).

The results of the tuning phase (best hyper-parameter values) and the trial phase (validation and test accuracy) are presented in Table 2. All optimizers performed similarly in this benchmark. The exception is the Sign optimizer that performed about 1.5% worse than its peers. We believe the odd way the Sign optimizer changes weights is particularly effective at avoiding the early stop mechanism used during evolution. While the Sign optimizer thrives in the evolutionary system's specific evaluation conditions, when moved into a more traditional training environment, it cannot compete with the other optimizers. Despite weaker results, the Sign optimizer exhibits the smallest accuracy drop between the validation and test sets.

Table 2. Trial results of all optimizers in Fashion-MNIST. The parameters tuned for each optimizer, as well as the best values found through Bayesian optimization are also presented.

Optimizer	Parameter			Validation Accuracy	Test Accuracy
ADES	beta_1	beta_2		93.53 ± 0.11%	92.87 ± 0.16%
	0.87621	0.76132			
Sign	lr			92.08 ± 0.13%	91.29 ± 0.25%
	0.0009				
Adam	lr	beta_1	beta_2	93.46 ± 0.10%	92.69 ± 0.20%
	0.00127	0.07355	0.78162		
RMSprop	lr	rho		93.61 ± 0.08%	92.80 ± 0.17%
	0.00097	0.85779			
Nesterov	lr	momentum		93.41 ± 0.14%	92.82 ± 0.15%
	0.09999	0.86715			

Notably, ADES has the best test accuracy in Fashion-MNIST, suggesting ALR succeed in its objective of specialization. While it is expected that the evolved optimizer would perform well in its native task, it is still remarkable

that this automatically generated solution can empirically outperform human-made optimizers. It must be noted that the human-made optimizers are the culmination of many years of research into this subject and ADES competes with these methods despite being automatically generated. Nevertheless, it must be acknowledged that the differences in performance between the four best optimizers are minimal. Performing a Mann-Whitney U test for the null hypothesis comparing the two best solutions: "ADES and Nesterov are equal" with a significance level of 0.05, we are unable to reject the null hypothesis ($p = 0.267$).

4.5 CIFAR-10

This benchmark was designed to test the ability of the evolved optimizers to generalize to a different problem within the same domain. The dataset chosen was CIFAR-10, a common problem used to evaluate image classification approaches. CIFAR-10 also has training (CIFAR-10-Training) and test (CIFAR-10-Test) sets, similar to Fashion-MNIST. Following the procedure outlined in the previous section, CIFAR-10-Training is used for training (43000 instances) and validation (7000 instances), while CIFAR-10-Test (10000 instances) is used to make the final test accuracy assessment. The architecture used was the Keras CIFAR-10 architecture [3] (available in [17]).

The best parameter values found using Bayesian optimization, and the trial phase results are presented in Table 3. Once again, despite its weak performance overall, the Sign optimizer is the most resistant to the dataset change, even slightly improving its performance when moved to the test set. While this may seem unusual, consider that the validation accuracy is only used to select the best weights for testing. Additionally, note that the Sign optimizer does not strictly follow the direction of the gradient when adjusting weights, possibly making Sign resistant to overfitting.

However, the most notable result in this benchmark is that ADES remains one of the best solutions, outperforming Adam and RMSProp in both validation and test accuracy. While the difference in accuracy is not massive, it is essential to acknowledge that an evolved optimized can compete with state-of-the-art solutions even outside its native task. Considering that the supposed advantage of an evolved optimizer is that it is fine-tuned for the task it is evolved in, this result is remarkable.

Additionally, the fact that ADES can be successfully used in other image classification tasks motivates research into other applications of the optimizer. What other datasets, architectures and ML problems can ADES succeed in? Further research is necessary to fully understand the contribution of ADES to ML.

However, the success of ADES in this benchmark also highlights the value of ALR. A vital characteristic of ADES it is competitive with human-made optimizers using a novel way of changing weights. Even if future work reveals that ADES is not widely applicable, understanding its unique features and why it succeeds in specific tasks may provide helpful insights for creating better human-made optimizers. Specially because adaptive optimizers historically retool or

Table 3. Trial results of all optimizers in CIFAR-10. The parameters tuned for each optimizer, as well as the best values found through Bayesian optimization are also presented.

Optimizer	Parameter			Validation Accuracy	Test Accuracy
ADES	beta_1	beta_2		82.04 ± 0.20%	81.85 ± 0.29%
	0.92226	0.69285			
Sign	lr			74.97 ± 0.40%	75.09 ± 0.50%
	0.0009				
Adam	lr	beta_1	beta_2	81.93 ± 0.21%	81.56 ± 0.26%
	0.00163	0.81344	0.71023		
RMSprop	lr	rho		80.97 ± 0.27%	80.65 ± 0.45%
	0.00085	0.64813			
Nesterov	lr	momentum		82.45 ± 0.25%	82.03 ± 0.25%
	0.00907	0.98433			

adjust ideas from older solutions to create new, better optimizers. ALR's ability to create unique, competitive, evolved optimizers may help researchers improve human-made optimizers.

It is also worth highlighting that the evolutionary conditions did not incentivize the creation of an optimizer that performed well outside its native task. ALR created ADES while evaluating optimizers strictly based on their performance in the Fashion-MNIST dataset but the optimizer still operates successfully in a different task. We consider that this result motivates research into other applications of ALR. Specifically, it may be interesting to use ALR with a fitness function that evaluates optimizers based on their performance on several tasks, promoting general applicability. The evolutionary setup used also enforced minimal limitations on the solutions created. While this led to the creation of a novel solution, it is relevant to investigate the potential of ALR when using additional rules that guarantee evolved optimizers are closer to strong, established human-made optimizers. In fact, it could be interesting to utilize ALR starting from the population of human-made optimizers. Evolving solutions closer to human-made optimizers may work as an alternative to hyper-parameter optimization, where the entire optimizer is tuned for a specific task.

5 Conclusion

This work presents an adaptive implementation of the AutoLR framework. This framework is capable of producing novel, specialized ANN optimizers through an EA. Specifically, the framework [2] is used to evolve optimizers for an image classification task. The setup used included no incentive to imitate traditional optimizers as solution were rated solely based on their performance.

Some of the optimizers evolved under these circumstances can compete with the established human-made optimizers. This optimizer, called ADES, showed

other exciting properties. Despite being evolved in a specific task, this optimizer could compete with human-made optimizers even in a different task. As such, the results obtained with ADES indicate that the AutoLR framework could create new general ANN optimizers that can be employed on a breadth of problems. To summarize, the contributions of this paper are as follows:

- The proposal of ALR, an implementation of AutoLR capable of producing adaptive ANN optimizers.
- The evolution, benchmark, and analysis of two new ANN optimizers.
- The discovery of ADES, an automatically-generated adaptive ANN optimizer capable of competing with state-of-the-art human-made optimizers in two relevant image classification tasks.

The results obtained in this work also warrant further study into a few topics. The evolved optimizer ADES remained competitive with human-made solutions when moved outside of its native task. However, all tasks considered are image classification problems. It is vital to understand whether ADES retains its utility when applied to different problems. Furthermore, in this work we only compared optimizers based on their final test accuracy. However, there are other properties that are desirable in optimizers such as convergence speed and sensitivity to hyper-parameters. Studying evolved optimizers in this lens may reveal additional advantages and disadvantages to this approach. Finally, the success of ADES outside its native task suggests ALR may serve as a tool for the creation of optimizers in other applications. In this work, the experiments were designed to explore the benefits of specializing optimizers, however we found that the applicability of the solutions produced were also interesting. It would be relevant to study the benefits of using ALR to create generally applicable optimizers (e.g., by assigning optimizer fitness based of their performance in several different tasks).

Acknowledgments. This work is partially funded by: Fundação para a Ciência e Tecnologia (FCT), Portugal, under the grant UI/BD/151053/2021, and 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.

References

1. Bottou, L.: On-Line Learning and Stochastic Approximations, pp. 9–42. Cambridge University Press, Cambridge (1999)
2. Carvalho, P., Lourenço, N., Assunção, F., Machado, P.: AutoLR: an evolutionary approach to learning rate policies. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO 2020, pp. 672–680. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3377930.3390158
3. Chollet, F., et al.: Keras CIFAR10 architecture (2015). https://keras.io/examples/cifar10_cnn_tfaugment2d/

4. Chollet, F., et al.: Keras MNIST architecture (2015). https://keras.io/examples/mnist_cnn/
5. Farabet, C., Couprie, C., Najman, L., LeCun, Y.: Learning hierarchical features for scene labeling. IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1915–1929 (2012)
6. Hinton, G., Srivastava, N., Swersky, K.: Overview of mini-batch gradient descent. University Lecture (2015). https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
7. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
9. Jacobs, R.A.: Increased rates of convergence through learning rate adaptation. Neural Netw. **1**(4), 295–307 (1988)
10. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
12. Lopez, M.M., Kalita, J.: Deep learning applied to NLP. arXiv preprint arXiv:1703.03091 (2017)
13. Mockus, J., Tiesis, V., Zilinskas, A.: The application of Bayesian methods for seeking the extremum, vol. 2, pp. 117–129 (2014)
14. Nesterov, Y.: A method for unconstrained convex minimization problem with the rate of convergence o (1/k̂ 2). In: Doklady an USSR, vol. 269, pp. 543–547 (1983)
15. Pedro, C.: Adaptive AutoLR grammar (2020). https://github.com/soren5/autolr/blob/master/grammars/adaptive_autolr_grammar.txt
16. Pedro, C.: AutoLR (2020). https://github.com/soren5/autolr
17. Pedro, C.: Keras CIFAR model (2020). https://github.com/soren5/autolr/blob/benchmarks/models/json/cifar_model.json
18. Pedro, C.: Keras MNIST model (2020). https://github.com/soren5/autolr/blob/benchmarks/models/json/mnist_model.json
19. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
20. Senior, A., Heigold, G., Ranzato, M., Yang, K.: An empirical study of learning rates in deep neural networks for speech recognition. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6724–6728. IEEE (2013)
21. Snoek, J., Larochelle, H., Adams, R.P.: Practical Bayesian optimization of machine learning algorithms. In: Proceedings of the 25th International Conference on Neural Information Processing Systems, NIPS 2012, vol. 2, pp. 2951–2959. Curran Associates Inc., Red Hook (2012)
22. Suganuma, M., Shirakawa, S., Nagao, T.: A genetic programming approach to designing convolutional neural network architectures. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, pp. 497–504. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3071178.3071229