# Learning to Learn: Generalization of Evolved Plasticity Rules in Recurrent Neural Networks

Wesley Joosten
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
w.joosten@student.utwente.nl

## ABSTRACT

In machine learning, one often looks towards biology to find mechanisms to replicate. One of these mechanisms we observe in biology is synaptic plasticity rules. Plasticity rules allow for an artificial neural network to learn new tasks. By changing synapse weights based on these rules, an agent can adapt itself. Choosing proper rules then becomes a new challenge. Previously, gradient descent has been used to optimize one such set of rules. We use an evolutionary algorithm, specifically evolution strategies, to learn rules for recurrent neural networks. These are algorithms based on biological evolution which do not require a measure of error and may thus be applicable to a wider set of problems. Our results show that this approach may have potential, obtaining better-than-chance performance on classification tasks. This results in a proof of concept that enables further research in this area.

## Keywords

Synaptic plasticity, neuroevolution, neural networks, machine learning, evolution, meta learning, recurrent neural networks, evolution strategies

## 1. INTRODUCTION

Biology has produced agents capable of performing and learning complicated tasks. It is no surprise then, that in the past decades, research has often looked towards biology to solve machine learning problems. A well-known example is artificial neural networks (ANNs), which are inspired by biological neural networks (BNNs) [12]. These ANNs can be seen as a very basic model of a brain, and analogous terminology is used when talking about BNNs and ANNs. An ANN consists of a number of neurons that are interconnected. Each connection has a certain synaptic weight. By adapting the weights of the connections, an ANN can be trained to perform a task [1].

ANNs are often trained using the back-propagation algorithm [13]. While very high accuracies can be achieved on classification tasks [4], this method has a few limitations. Firstly, a network trained using back-propagation on one classification task will not do well on another. For instance, a network trained to recognize letters can not recognize flowers. Would you later want to retrain the network on a new task, it will forget how to do the first [9, 8]. A second issue is that back-propagation needs information about the correctness, or rather error, of the predicted result to train [13]. In certain tasks, this is either very difficult or even impossible to determine. Furthermore, although not a problem in itself, back-propagation is not biologically plausible, i.e. it is highly unlikely BNNs learn in this way [7].

*35th Twente Student Conference on IT* July 2nd, 2021, Enschede, The Netherlands.

One proposed solution to the second problem (needing a measure of correctness) is evolutionary algorithms. These algorithms, based on biological evolution, can be used to adapt the weights of an ANN without knowing anything about the correctness of an individual action [19]. All one needs is a measure of fitness, that is, how well the ANN performs overall. Since evolutionary algorithms use similar principles as those that drive biological evolution, they are also more biologically plausible. However, using evolutionary algorithms on the weights of the network directly does not solve the generalization problem. A network trained on a certain task or dataset will still not be able to learn another. After all, the trained model is still the same as when using back-propagation.

Looking towards biology again, a way of providing more generalizability is synaptic plasticity rules [14]. Synaptic plasticity is a process wherein the weight of synaptic connections changes over the lifetime of an individual. We then speak of a plastic network. Applying plasticity rules to an ANN introduces a way to learn a new task [5]. If plasticity rules are determined that allow an ANN to adapt itself, it could generalize from one problem to another, being able to teach itself in a way. A question is then, what form do these plasticity rules take?

An early proposed rule from biology is Hebb's rule [6]. It states that if a neuron is involved in the firing of another neuron, the synaptic weight between these neurons increases. Using this rule in ANNs has problems (weights keep growing for instance [15]), but it can serve as an inspiration. We can use rules of this same form, i.e. determining a change in the weight of a connection based on the pattern of firing of the neurons it connects [18]. A simple abstraction gives four different rules for the change in weight of the connection between neuron A and B, one for each possible combination of neurons A and B firing or not. An issue is that this might be too simple, the space of possible rules is very small.

A way to increase the complexity of possible rules is to consider recurrent neural networks (RNNs) [5]. Take an RNN with a hidden layer that fires a certain amount of times before propagating an output signal. We can now determine rules based on the history of firing over these iterations. This allows for more complex relationships between firing patterns and the change in weights.

The problem we are left with is determining these plasticity rules. Previously, these rules have been learned using gradient descent [5], however, this reintroduces the problem of needing a gradient measure of error for a single action. A solution to this would be to use evolutionary algorithms to learn these rules, needing only an overall fitness.

From this problem we can derive the following research questions:

1. How do evolutionary algorithms perform compared to gradient descent for learning plasticity rules of a neural network?
2. How well do plasticity rules learned by evolutionary algorithms generalize to unseen problems when compared to ones learned by gradient descent?

We attempt to answer these questions by using the model proposed by Cristian et al. [5] and learning the rules using both back-propagation as well as evolution strategies. We train on a classification problem, using both generated datasets as well as MNIST and compare results.

We begin by discussing similar approaches to solve machine learning problems in the Related Work section. In our Background section, we explain the necessary state of the art concepts, and then introduce our novel approach in Methods. Under Experimental Setup, we introduce our experiments and then discuss the outcomes in the Results section. We finally discuss the limitations of our paper and future research directions in Discussions, ending with Conclusions.

## 2. RELATED WORK

Recently, plastic networks have been explored, usually with some form of evolutionary algorithms to learn rules with which the network can adapt during its lifetime. Soltoggio et al. [14] explore these evolved plastic artificial neural networks (EPANNs).

Multiple recent papers build upon their work with new concepts. Such as Najarro and Risi [10] who use synapse-specific Hebbian-like learning rules learned by evolution strategies in networks with randomly initialized weights. A later work by Pedersen and Risi [16] uses similar rules but merges these rules during learning so multiple synapses use the same rules. The decrease in the number of rules shows better adaptability.

Yaman et al. [18] introduce the concept of deterministic Hebbian-like rules, trained using genetic algorithms and show that these can produce networks that learn to adapt to changes in environments. The deterministic nature and small search space allow for interpretable rules.

Cristian et al. [5] build upon these principles but attempt to increase the rule space by collapsing feedforward networks with multiple hidden layers into recurrent networks in which a single hidden layer fires a certain amount of recurrent steps before readout. Rules based on the firing patterns over these steps are then trained using gradient descent.

In [17], Yaman et al. introduce a different way of considering the firing history of neurons into plasticity rules, namely neural activity traces (NATs). They use RNNs which keep track of the firing pattern frequency during an epoch and apply rules afterwards based on these frequencies.

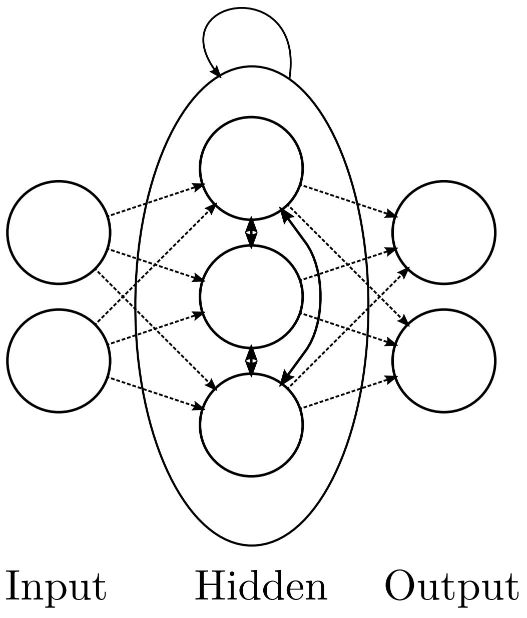

**Figure 1. Visual representation of the network, here with 2 input neurons, 3 hidden neurons and 2 output neurons. First the input layer propagates to the hidden layer, then the hidden layer has its recurrent steps and lastly the hidden layer propagates to the output.**

## 3. BACKGROUND

### 3.1 The Network

We consider RNNs such as described by Cristian et al.[5], consisting of an input layer, a recurrent hidden layer and an output layer (See Figure 1). After computing the activation of the hidden layer from the input layer (as in Equation 1), the hidden layer performs a number of recurrent steps. During such a step, the activation of a neuron is determined by summing the original activation of all hidden layer neurons with their last activation, applying the weight of the connections and summing all resulting values as in Equation 2. A hidden layer neuron is connected to all hidden layer neurons apart from itself. After the recurrent steps, the output is computed as in Equation 3.

$$a_{T=0} = f(w_{input} a_{input}) \tag{1}$$

$$a_{T=n} = f(w_{hidden}(a_{T=0} + a_{T=n-1})) \tag{2}$$

$$a_{output} = f(w_{output} a_{T=N}) \tag{3}$$

Where $f(x)$ is the activation function, $a_{T=n}$ are the activations of the hidden layer neurons at a certain timestep where $a_{T=0}$ denotes the activation before any recurrent step and $a_{T=N}$ after the last recurrent step, $a_{input}$ are the activations of the input layer (and thus the input), $w_{input}$ are the weights of the connections between the input layer and the hidden layer and $w_{hidden}$ are the weights of the connections between the hidden layer and itself, $w_{output}$ are the weights of the connections between the hidden layer and output layer.

For the hidden layer, the activation function is in two parts. First, a conventional activation function is applied (such as ReLu) and then a cap[11] is applied. The activations are ranked on their intensity and only the highest ones get propagated. The post-synaptic neuron is then said to have been activated. The lower activations are set to zero. The network keeps track of the activation patterns during the recurrent steps.

### 3.2 The Rules

We consider rules as proposed by Cristian et al.[5]. These rules are based on the activation patterns of the hidden layer. After every iteration through the whole network, the weights of the network are updated based on these rules. They are real-valued numbers that are summed with the existing weight as follows:

$$w = w + \eta \Delta w$$

where $\eta$ is a given step size and $\Delta w$ is the value that follows from the rule.

For each layer. $\Delta w$ is determined by a mapping between the activation patterns of the hidden layer and a real value. The activation patterns are determined by taking the activations of each timestep ($a_{T=0}$ through $a_{T=N}$)) and discretizing them with $s(x)$ as in (4). We then, for each hidden neuron, take its discretized activation from each timestep and thus get $N+1$ values, where $N$ is the number of recurrent steps. We label these $a_0$ through $a_N$.

$$s(x) = \begin{cases} 0 & x = 0 \\ 1 & x \neq 0 \end{cases} \tag{4}$$

For the connections between the input layer and hidden layer, we only consider the hidden layer neuron activation pattern. For connections between the hidden layer and itself, we consider the activation pattern of both the pre-synaptic and post-synaptic neurons. For the connections between the hidden layer and output layer, we consider the activation pattern of the hidden layer neuron, as well as whether the output layer neuron is the correct label. Tables 1, 2 and 3 show an illustration of the input, hidden and output layer rules respectively, for an example network with 2 recurrent steps ($N = 2$). Here $x_k$ are some real value, $a_i$ denote pre-synaptic neuron activations and $a_j$ denote post-synaptic neuron activations.

**Table 1. Input layer rules**

| $a_0$ | $a_1$ | $a_2$ | $\Delta w$ |
|---|---|---|---|
| 0 | 0 | 0 | $x_1$ |
| 0 | 0 | 1 | $x_2$ |
| ... | ... | ... | ... |
| 1 | 1 | 1 | $x_8$ |

**Table 2. Hidden layer rules**

| $a_{i,0}$ | $a_{i,1}$ | $a_{i,2}$ | $a_{j,0}$ | $a_{j,1}$ | $a_{j,2}$ | $\Delta w$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | $x_1$ |
| 0 | 0 | 0 | 0 | 0 | 1 | $x_2$ |
| ... | ... | ... | ... | ... | ... | ... |
| 1 | 1 | 1 | 1 | 1 | 1 | $x_{64}$ |

**Table 3. Output layer rules**

| $a_0$ | $a_1$ | $a_2$ | Correct label? | $\Delta w$ |
|---|---|---|---|---|
| 0 | 0 | 0 | No | $x_1$ |
| 0 | 0 | 0 | Yes | $x_2$ |
| 0 | 0 | 1 | No | $x_3$ |
| ... | ... | ... | ... | ... |
| 1 | 1 | 1 | Yes | $x_{16}$ |

## 4. METHODS

For our evolutionary algorithm we choose evolution strategies [3]. These are applicable for real valued parameters [2]. Furthermore, they have been shown to produce good rules in similar scenarios [10, 16]. For the genotype, we choose the mapped real value for all of the rules and put these all in a vector. This leads to a search space of $2^N + 2^{2N} + 2^{N+1}$ real values (where $N$ is the number of recurrent steps). For our fitness function, we first let the network go through all samples once and then compute the accuracy on a test set.

To allow for a fair comparison between gradient descent and evolution strategies, we first set a baseline. We obtain this baseline by reproducing the results of Cristian et al [5].

To compare the performance of both approaches, we train the network on a simple generated dataset as well as a common machine learning dataset. The network is trained using both gradient descent and evolution strategies. Achieved accuracies on both the training and test set, as well as speed of convergence are then considered to make a comparison.

To compare generalizability, after training using the above method, we generate a new dataset en then show this dataset to the network, all the while keeping track of the accuracy. Here we no longer train the rules, only apply them.

## 5. EXPERIMENTAL SETUP

### 5.1 The Datasets

For our generated dataset we use *scikit-learn* [?] to create a classification dataset. The dataset has 10 features which are all informative, and 4 classes. This creates 8 clusters of points normally distributed about vertices of a 10 dimensional hypercube, with sides of length 20. Each class gets assigned 2 clusters. Interdependence between features as well as further noise is then added to the data. We generate a total of 5000 samples and use 4000 for the training set and 1000 for the test set. For the dataset we use to test generalizability, we generate a new dataset in the same way.

We also perform the experiment with the MNIST dataset. MNIST is a common dataset for evaluating machine learning performance. It consists of 70.000 28x28 pixel images of handwritten digits (See Figure 2 for some examples. 60.000 of these form the training set, and 10.000 form the test set.

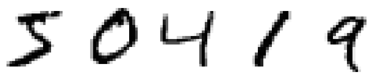

**Figure 2. Some example images from the MNIST dataset.**

### 5.2 The Hyperparameter Settings

The choice of hyperparameters is based on both the work of Cristian et al [5] as well as preliminary testing with a range of parameters. Both performance and training time are taken into account.

We consider a network with 3 recurrent rounds. All weights are initialized randomly using a standard normal distribution. The same goes for the plasticity rules. We use a step size of 0.01 when updating the weights. For the MNIST dataset, we use a network with 1000 hidden layer neurons and a cap of 500 activated neurons. For the generated dataset we use a network with a hidden layer of 20 neurons and a cap of 10 activated neurons. For the input and hidden layer we use a sigmoid activation function before the cap. On the output layer we apply a softmax activation function.

For both approaches, we present the samples in batches of 100 at a time. The weights of the network are reset after every batch during training of the rules. The weights are then updated using the plasticity rules after each shown sample. When subsequently showing a new dataset, we do this one sample at a time, allowing the network to change its weights after each sample. We show the whole dataset 50 times, never resetting the weights of the network.

For the gradient descent method, we train the rules using Adam optimization with cross-entropy loss and a learning rate of 0.01. The loss is computed by, after having gone through the batch while updating the weights, showing the batch to the network once more without updating the weights.

For the evolution strategy, we use a population size of 10 and a mutation rate of 0.01. We generate an initial population by picking from a standard normal distribution for the vector values. We calculate fitness by first showing the batch to the network while updating weights and then computing the accuracy over the batch without updating weights.

## 5.3 The Comparison

We train the RNN with both gradient descent as well as evolution strategies. We try to keep the model and training scenario as similar as possible to allow for a fair comparison.

## 6. RESULTS

## 6.1 Generated Dataset

The results of both approaches on training with the initial generated dataset are shown in Figure 3. After training rules using both methods, we then also apply these rules to another generated dataset without training the rules further. The results of this experiment are shown in Figure 4.

### *6.1.1 Gradient Descent*

The accuracy of the gradient descent approach stays at 25% during training. The reason being the model almost always classifying all samples under the same label. An effort was made to solve this problem, and multiple issues were identified, but no conclusive solution was found. For a more detailed discussion on this see Appendix A. When we look at the loss of the gradient descent training, we can see it decreasing in the beginning but then flattening out. When, after training, we show the model a new dataset, perhaps unsurprisingly, it still classifies everything under the same label.

### *6.1.2 Evolution Strategy*

When training using evolution strategies, the model does better in our experiments. The training accuracy quickly gets to around 40% with the test accuracy being slightly lower (See Figure 5). We can observe a continuing slightly increasing trend. Perhaps with more training epochs, the model can improve. When looking at the loss, the situation is similar. There seems to be a slight downwards trend, even slightly increasing towards the end. It seems that the model needs some time before improving. When we show the model a new dataset after training, the accuracy significantly decreases. It hovers around 25%, which would be the same as random classification. The accuracy is thus similar to the gradient descent approach. However, where in the gradient descent approach it always classifies every sample under the same label, in the evolution strategy trained model it is different. Even when the total accuracy is around 25%, this is not always due to 100% in one class and 0% in the rest.

## 6.2 MNIST

The results of both approaches while training on MINST are shown in Figure 6.

### *6.2.1 Gradient Descent*

With gradient descent the results are much the same as with the generated dataset, namely hovering around the equivalent of classifying randomly. Also here it seems to always classify all samples under the same label.

### *6.2.2 Evolution Strategy*

For the evolution strategies, it is also much the same as with the generated datasets. It has more promising results than gradient descent but still not very high performance. It seems however though, that compared to the generated dataset, it does not increase performance over time.

## 7. DISCUSSIONS

## 7.1 Limitations

While effort was put in to optimize hyperparameters, this was not approached very systematically, mostly due to time constraints. We can not exclude that our choice in hyperparameters is the cause for the difference observed between gradient descent and evolutionary algorithms. Previous research has gained significantly better results with gradient descent on very similar problems with very similar models and hyperparameters. This may be due to the method being very sensitive to changes in the model, or our having overlooked something in our experimentation. The cause behind the network converging towards classifying all samples under the same class has been studied with some effort, however, no conclusive reason has yet been found. Further systematic testing could be done. Appendix A gives some pointers towards mechanisms that might play a role, but the problem still persists. Without any rigorous analysis, it cannot be concluded that the training methods are the cause for the issue.

## 7.2 Future work

Our work can be replicated with significant effort put into hyperparameter optimization for both methods. This way, a best-case scenario can be tried for both methods to allow for a better comparison. In our experimentation, we were unable to try for generalizability on more complicated datasets. An interesting experiment would be to apply learned rules on a dataset as MNIST to other datasets such as MNIST or notMINST, or perhaps even datasets with different dimensions such as CIFAR-10 or CIFAR-100. An interesting question is what the influence of the number of recurrent rounds is on the performance of the network. The number of rounds in our experiments was somewhat arbitrarily chosen, based on some preliminary testing and the tests of Cristian et al. [5]. More systematic testing would prove useful in determining a proper number of rounds. While our research only considered classification tasks, one of the benefits of using evolutionary algorithms is the lack of need for a measure of error. It would be interesting to apply the method to reinforcement learning tasks. The model can be easily adapted for such

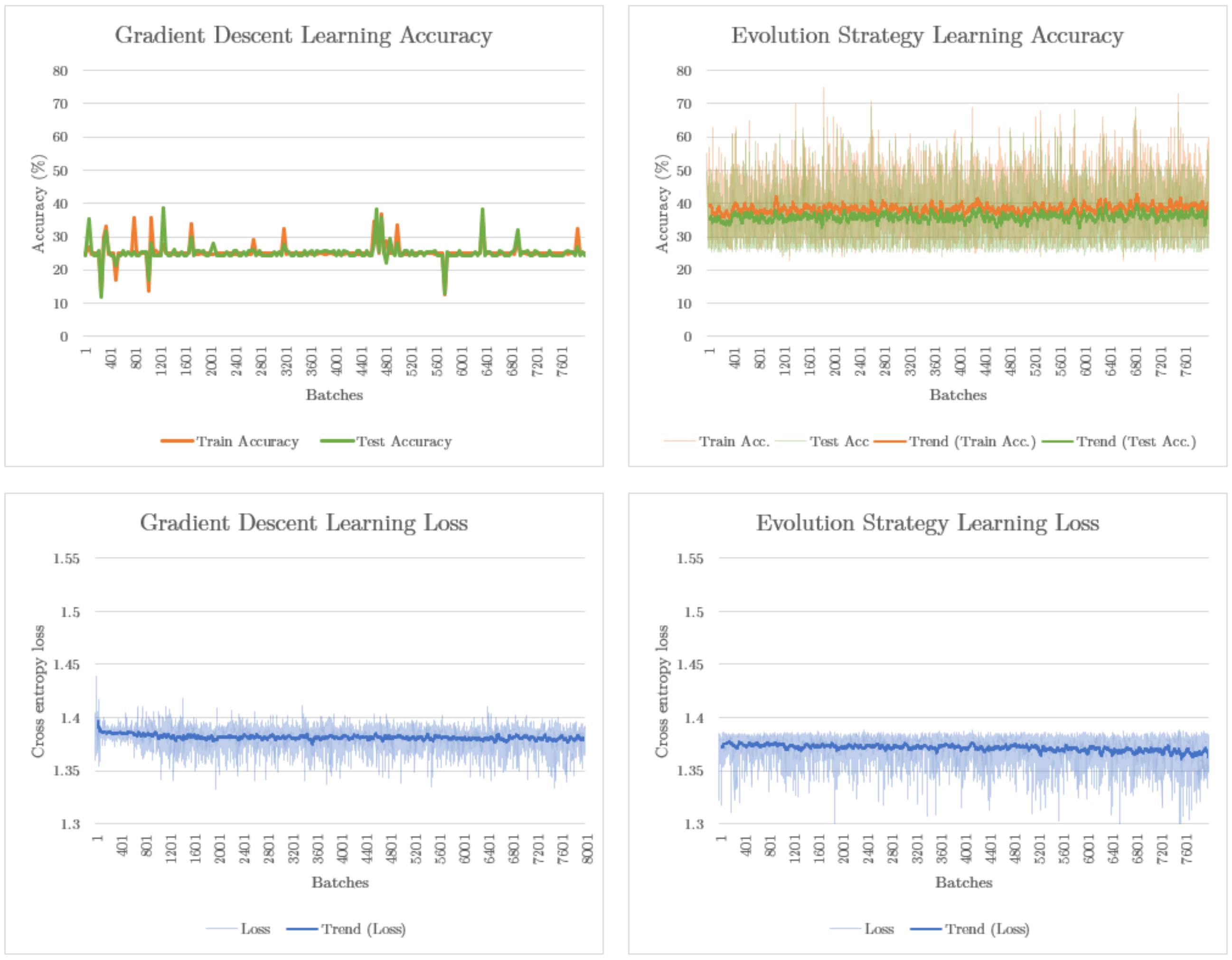

**Figure 3. Results of training on the generated dataset. The trendlines are moving averages over 40 batches. For the evolution strategy we take the best performing sample of each generation. For the gradient descent approach accuracy measurements were only done very 40 batches.**

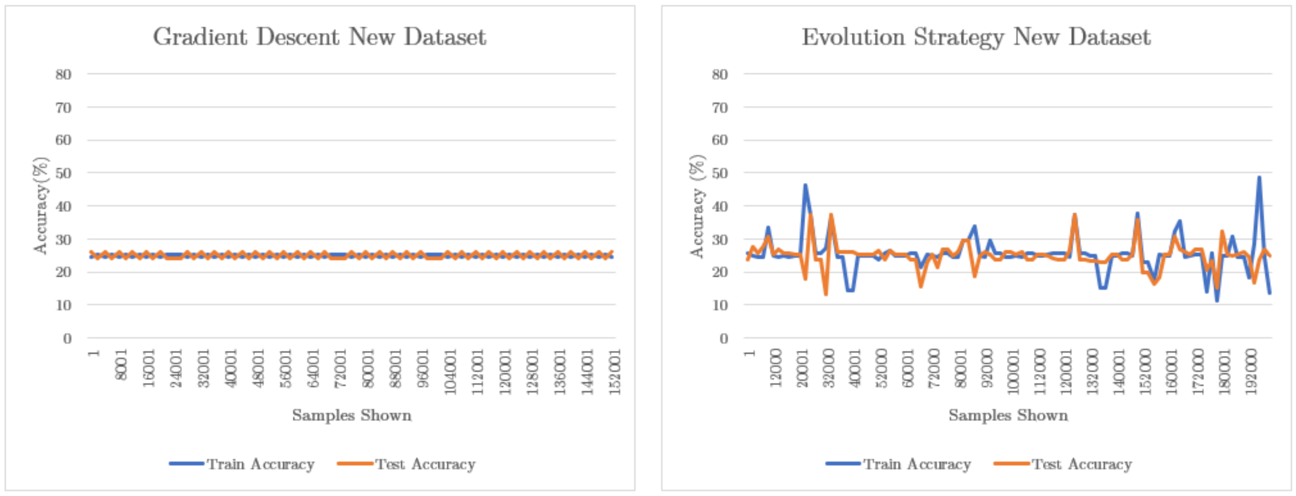

**Figure 4. Results when trained on one generated dataset and shown a new generated dataset.**

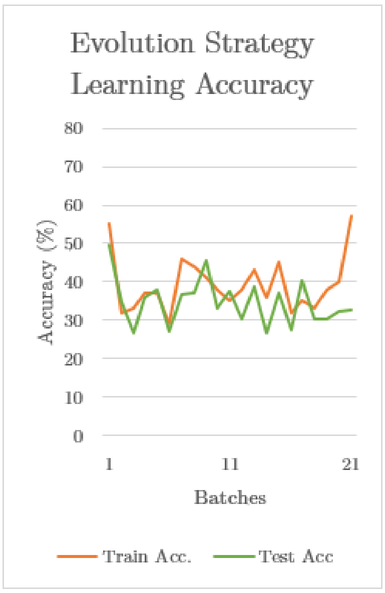

**Figure 5. Results of the best individual in the first 20 batches when trained on one generated dataset with evolution strategies. The first population seemed to have a lucky randomly generated individual.**

scenarios by removing the correct label part of the output rules and perhaps introducing a modulatory signal.

## 8. CONCLUSIONS

All in all, it seems evolution strategies may be a promising alternative to gradient descent in training the discussed models. While the answer to our research questions remains largely inconclusive, we have at least proven that evolution strategies can do better than chance. Perhaps with further improvements, they can approach the state of the art. In our experiments, evolution strategies do better than gradient descent every time, so the answers to our research questions seem to tend in favour of evolution strategies. Previous research has, however, attained much better results with gradient descent than we have, so more rigorous experimentation is in order before being able to answer fairly and conclusively.

## 9. REFERENCES

[1] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, USA, 1st edition, 2009.

[2] T. Bäck. Evolution strategies: An alternative evolutionary algorithm. In *Artificial Evolution*, 1995.

[3] H.-G. Beyer and H.-P. Schwefel. Evolution strategies – a comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[4] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, Dec 2010.

[5] R. C. Cristian, M. Dabagia, C. Papadimitriou, and S. Vempala. Learning with plasticity rules: Generalization and robustness, 2021.

[6] D. O. Hebb. *The organization of behavior; a neuropsychological theory.* The organization of behavior; a neuropsychological theory. Wiley, Oxford, England, 1949.

[7] E. Hunsberger. *Spiking Deep Neural Networks: Engineered and Biological Approaches to Object Recognition*. PhD thesis, 2018.

[8] R. Kemker, M. McClure, A. Abitino, T. Hayes, and C. Kanan. Measuring catastrophic forgetting in neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[9] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[10] E. Najarro and S. Risi. Meta-learning through hebbian plasticity in random networks. *arXiv pre-print server*, 2020.

[11] C. H. Papadimitriou and S. S. Vempala. Random Projection in the Brain and Computation with Assemblies of Neurons. In A. Blum, editor, *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, volume 124 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 57:1–57:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, 1996.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[14] A. Soltoggio, K. O. Stanley, and S. Risi. Born to learn: The inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks*, 108:48–67, 2018.

[15] Z. Vasilkoski, H. Ames, B. Chandler, A. Gorchetchnikov, J. Léveillé, G. Livitz, E. Mingolla, and M. Versace. Review of stability properties of neural plasticity rules for implementation on memristive neuromorphic hardware. In *The 2011 International Joint Conference on Neural Networks*, pages 2563–2569, 2011.

[16] J. Winther Pedersen and S. Risi. Evolving and Merging Hebbian Learning Rules: Increasing Generalization by Decreasing the Number of Rules. *arXiv e-prints*, page arXiv:2104.07959, Apr. 2021.

[17] A. Yaman, G. Iacca, D. C. Mocanu, G. Fletcher, and M. Pechenizkiy. Learning with delayed synaptic plasticity. *Proceedings of the Genetic and Evolutionary Computation Conference*, Jul 2019.

[18] A. Yaman, D. C. Mocanu, G. Iacca, M. Coler, G. H. L. Fletcher, and M. Pechenizkiy. Evolving plasticity for autonomous learning under changing environmental conditions. *CoRR*, abs/1904.01709, 2019.

[19] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

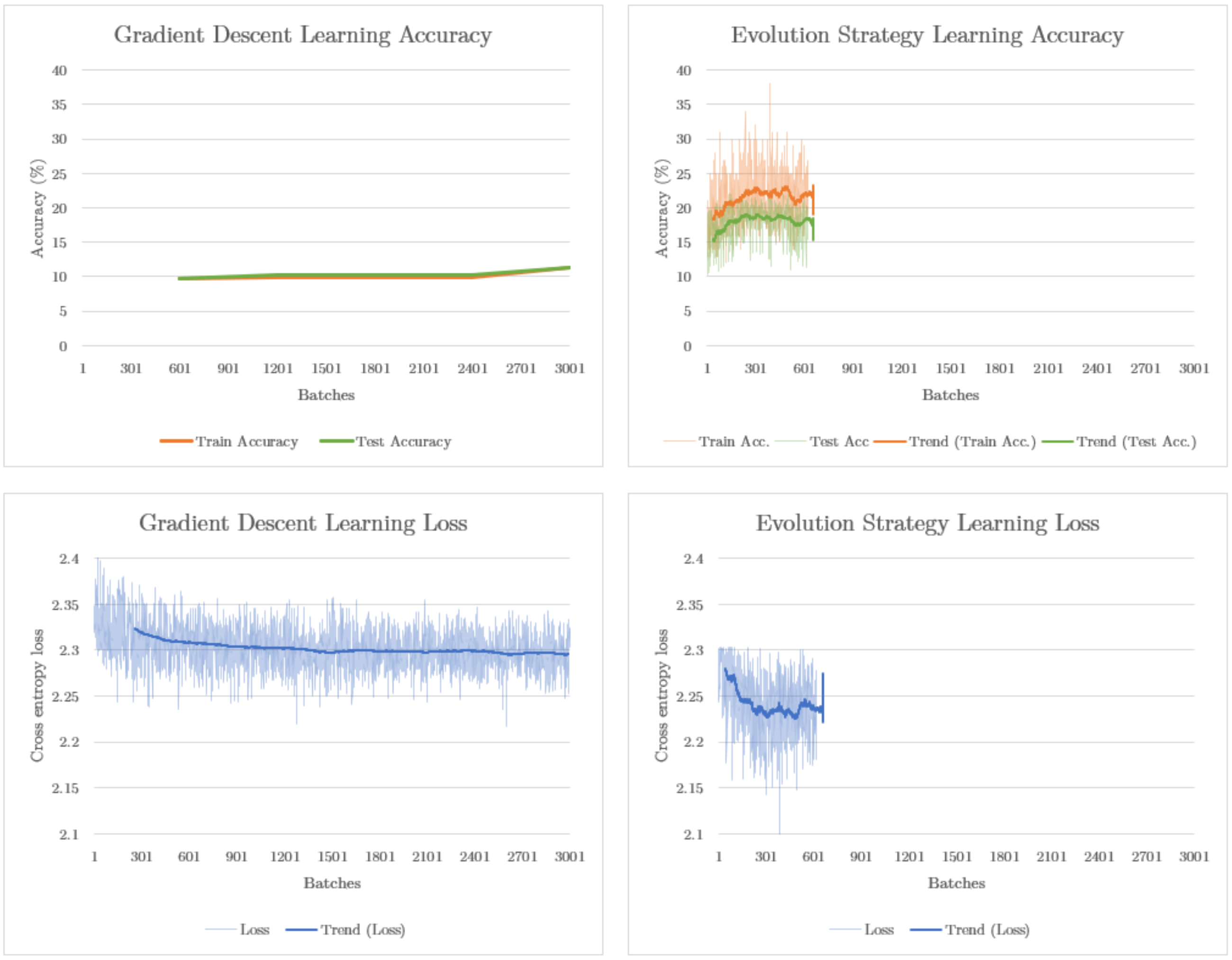

**Figure 6. Results of training on MNIST. The trendlines are moving averages over 40 batches. For the evolution strategy we take the best performing sample of each generation. For the gradient descent approach accuracy measurements were only done every 600 batches. The evolution strategy approach was trained in less batches due to a longer training time per batch.**

## APPENDIX

## A. PROBLEMS ENCOUNTERED DURING TRAINING

When reproducing the results of Cristian et al. [5] using their implementation, a problem was quickly encountered. After a few rounds of training, the network would converge to classifying each sample under the same class. Since this was believed to be an issue with the training method, work on implementing the evolution strategies began before being able to get promising results with the gradient descent implementation. The evolution strategy implementation was observed to have the same problem, although it seemed to happen less quickly. To find the cause of the issue, the training was attempted on very simple generated datasets, to follow the activations and weights through the network. After some digging around, multiple factors were found to be contributing to the issue. The main cause seemed to be that the activations of the output layer were either all enormously big or all zero, causing either the one that just happened to be a little bigger to always be chosen or in the case of zero always the first class. Different causes for both issues were identified. One of them was an initialization of all weights to 0. This caused the same rule to always be applied to many neurons, causing their weights to either ever increase or ever decrease, because they keep on having the same activation pattern. Initializing all weights randomly seemed to make the problem less frequent. The issue of very large activations was tackled by applying softmax over the output layer activations. While not solving the root cause, this did seem to alleviate the problem somewhat. Now we are left with the problem of weights growing still, this is solved by, after each update, scaling the outgoing weights from a neuron by their norm. There was still a problem with activations going to zero. This was expected to be due to the ReLu activation of the neurons. Since having all outputs to zero is pretty close to having all outputs on zero and one at 1, while learning it wants to keep the activations close to zero. However, because of ReLu, this can be achieved by having very negative activations, these get turned into zero anyway. By simply having a linear activation function, the problem seems to be lessened. All this, combined with some tweaking of hyperparameters seems to have alleviated the issue on very simple datasets, now getting results significantly higher than chance. However, on more complicated datasets, such as MNIST or even a slightly more complex generated dataset, the problem seems to prevail. At a later stage, a bug was discovered in the implementation where selecting the rule for the input layer was done incorrectly. This did not seem to contribute to the problem but hindered training in other ways.