EDEN: Evolutionary Deep Networks for Efficient Machine Learning

Emmanuel Dufourq
African Institute for Mathematical Sciences
Maths & Applied Maths, University of Cape Town
Email: edufourq@gmail.com

Bruce A. Bassett
African Institute for Mathematical Sciences
South African Astronomical Observatory
Maths & Applied Maths, University of Cape Town
Email: bruce.a.bassett@gmail.com

***Abstract*—Deep neural networks continue to show improved performance with increasing depth, an encouraging trend that implies an explosion in the possible permutations of network architectures and hyperparameters for which there is little intuitive guidance. To address this increasing complexity, we propose Evolutionary DEep Networks (EDEN), a computationally efficient neuro-evolutionary algorithm which interfaces to any deep neural network platform, such as TensorFlow. We show that EDEN evolves simple yet successful architectures built from embedding, 1D and 2D convolutional, max pooling and fully connected layers along with their hyperparameters. Evaluation of EDEN across seven image and sentiment classification datasets shows that it reliably finds good networks – and in three cases achieves state-of-the-art results – even on a single GPU, in just 6-24 hours. Our study provides a first attempt at applying neuro-evolution to the creation of 1D convolutional networks for sentiment analysis including the optimisation of the embedding layer.**

***Index Terms*—neuro-evolution, genetic algorithm, neural network**

I. Introduction and Rationale

Deep neural networks are powerful but unintuitive beasts whose wrangling requires experience, significant trial and error to achieve good performance. The performance of such networks continued to improve as the depth is increased, e.g. [1]. This along with the rising influence of deep learning in all fields means it is becoming more and more important to develop methods to automatically design optimal or near-optimal network architectures and hyperparameters. Deciding on the exact nature and order of the layers, choice of activation functions, number of units in fully connected layers, number of filters in convolutional layers and other variables in creating deep neural networks is non-trivial. Given huge computing resources it is possible to simply try a vast number of possible combinations. Is there a way to be competitive with only a small amount of computing power, such as a single GPU?

One solution, which we pursue here, is to evolve good neural networks through the use of evolutionary algorithms [2]. Such neuro-evolutionary algorithms are not new, spanning nearly three decades, see e.g. [8], [9], [10], beginning with a study that evolved the weights of the neural network [3].

Here we briefly summarise recent related work on neuro-evolutionary algorithms, which, by contrast to this study, have used very significant computing resources. Real et al. [4] proposed a neuro-evolutionary approach to optimise neural networks for image classification problems using a parallel system executed on 250 computers and achieved considerable success on the CIFAR image problems. Zoph and Le [5] instead use recurrent neural networks along with reinforcement learning to learn good architectures. Eight hundred networks were trained on 800 GPUs.

Miikkulainen et al. propose CoDeepNEAT [6] in which a population of modules and blueprints are evolved. The blueprints are made up of several nodes which point to particular modules representing neural networks. Thus their proposed approach allows for the evolution of repetitive structures by enabling the blueprints to reuse evolved modules. Desell [7] proposed EXACT, a neuro-evolutionary algorithm for deployment on a distributed cluster which they executed across 4500 volunteered computers and evolved 120,000 networks to tackle the MNIST dataset. Their approach did not use pooling layers and was limited to two dimensional input and filters.

Finally we note that with a single GPU we have recently evolved deep networks to accurately identify whether a supervised machine learning challenge requires regression or classification [13], achieving an average 96% accuracy across a diverse set of tasks. This is a direct precursor of the current work and, given sufficient computing resources, can be seamlessly integrated into the network optimisation we discuss here.

In this work we propose Evolutionary DEep Networks (EDEN), a neuro-evolutionary algorithm that combines the strengths of genetic algorithms and deep neural networks to explore the search space of neural network architectures, their associated hyperparameters and the number of epochs to be applied. In our study, we explore additional features – such as the optimisation of embedding layers – and increase the complexity on the existing research. With EDEN we are interested in addressing two questions: can we evolve generally good architectures and hyperparameters for a broad range of problems (not just image classification)? Can this be successfully achieved on a single GPU, as opposed to the very large clusters used in previous studies?

We interface EDEN to TensorFlow [11] and thus new layers, functions and other features can easily be incorporated and controlled by EDEN as these represent function calls to the respective TensorFlow functions. Additionally, EDEN is not limited to TensorFlow, other modern deep neural network platforms can be interfaced. Figure 1 illustrates an example of a neural network architecture encoded by an EDEN chromosome.

The associated video[1] illustrates the evolution of the chromosomes during the execution of EDEN on the MNIST image classification problem, showing the population converging towards an efficient solution made up primarily of two-dimensional convolutional layers.

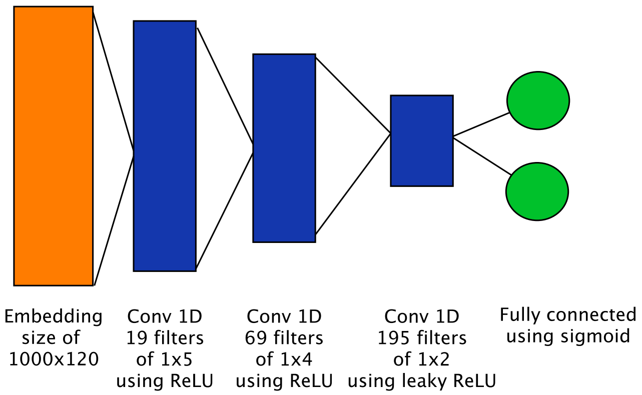
Embedding size of 1000x120
Conv 1D 19 filters of 1x5 using ReLU
Conv 1D 69 filters of 1x4 using ReLU
Conv 1D 195 filters of 1x2 using leaky ReLU
Fully connected using sigmoid

Fig. 1. Each EDEN chromosome contains two genes, encoding the learning rate and a neural network. The figure illustrates an example of a neural network evolved using EDEN for a sentiment analysis task. EDEN created an embedding layer with an output dimension of 120, followed by three 1D convolutional layers. EDEN evolved the number of filters, each filters' dimension along with each corresponding activation function. For the last layer, the selected activation function which EDEN determined was the sigmoid function. The learning rate for this chromosome is 0.0023.

II. Genetic Algorithm

A genetic algorithm (GA) [12] is an evolutionary algorithm which can be applied to solve optimisation problems. A population of chromosomes is randomly initialised. Each chromosome represents a candidate solution to the optimisation problem. A fitness function is used to evaluate each chromosome to determine the extent to which the chromosome can solve the problem. In a generational model, the GA iterates multiple times, known as generations, until some predetermined condition is met (for example, a maximum number of generations). Each chromosome is made up of several genes, and these genes are altered using a genetic operator. The resulting chromosome after the application of a genetic operator is known as an offspring. Multiple offspring are created based on the population size. The offspring replace the current chromosome population in each generation. In this study we used the traditional GA. We additionally increment the number of neural network epochs along with the number of generations to explore the best value for the number of epochs. Algorithm 1 presents the GA used.

Algorithm 1: Modified genetic algorithm used in this study

```
input: epochs: number of neural network epochs
input: population_size: population size
input: generation_max: maximum number of GA
       generations
1  begin
2      generation ← $0$.
3      epochs ← epochs.
4      population_size ← population_size.
5      Create an initial population of chromosomes.
6      Evaluate the initial population.
7      while generation $\leq$ generation_max do
8          if generation $\neq$ 0 then
9              epochs ← (epochs $+1$).
10             population_size ← (population_size $-10$).
11         Select the parents.
12         Create offspring using the genetic operators.
13         Replace the current population with the new
            offspring created in step 12.
14         Evaluate the current population.
15         generation ← generation $+ 1$.
16     return The best chromosome.
```

We choose to use GAs since the complexity of the chromosomes can be increased or decreased based on the number of genes which are encoded. GAs provide a further key advantage over other optimisation algorithms: they fluently handle complex combinations of discrete (e.g. layer type) and continuous (e.g. learning rate) search spaces, making them ideal for neuro-evolutionary studies; e.g. [4], [13].

III. Proposed Chromosome

Each EDEN chromosome is made up of two genes, and these genes constitute the required components to optimise a single neural network on some given input classification dataset. The two genes encode the learning rate and the network architecture. The learning rate denotes the value which is applied during the training optimisation. The architecture represents the exact order of the neural network layers and operations.

A. Network Layers

The following layers and operations were made available to EDEN: two-dimension convolution [14], one-dimension convolution [15], fully connected, dropout [16], one-, and two-dimension max pooling [17] and embedding [18]. Inappropriate choices (such as using a 2-D convolution for a text sentiment problem) are penalised as described in [13].

For the sentiment analysis tasks, instead of using pre-trained vectors such as Word2Vec [19], or setting a pre-determined embedding dimension size, we decided to allow EDEN to learn the dimension of the word embeddings as part of the optimisation. We created a dictionary by mapping each unique words their frequency count in the training data. We took the

[1] https://vimeo.com/234510097

top 1000 most frequent words and used this to encode the text into vectors of integers. Enabling EDEN to optimise both the vocabulary size and the embedding would result in significant computation time and hence this was not included in this study.

B. Activation Functions

When a layer is randomly generated an activation function is also randomly selected. Convolutional layers can choose between the following functions: {linear, leaky relu, prelu, relu}. Fully connected layers choose from: {linear, sigmoid, softmax, relu}. The last fully connected layer in the network can use any of: {linear, sigmoid, softmax}. These functions were selected as they are commonly used in literature. It is however possible to include a larger number of activations functions.

IV. EDEN

A. Initial Population Generation

The first phase in an evolutionary algorithm is to create the initial population of chromosomes. These chromosomes denote the first generation of solutions to the optimisation problem, i.e. in this case to generate neural networks that can correctly classify data. The number of chromosomes to create in the initial population (initial population size) is a user-defined parameter. Once each chromosome is created it is evaluated to determine how close it is to the optimal solution (100% classification accuracy), see section IV-D.

The initial population generation method used in this study was inspired by the ramped-half-and-half method proposed by Koza [20] which enables the creation of candidate solutions of various sizes. In a similar manner, we implemented an initial population generation method that would create neural network architectures of various sizes to increase the amount of diversity in the initial population (as opposed to a population that is skewed towards a particular size).

Algorithm 3 outlines the pseudocode on how a chromosome was randomly generated and algorithm 2 presents the initial population generation method used in this study. In certain cases, invalid architectures can be generated, these invalid architectures are discarded and a new one is generated.

Given our computational limitations, we had to limit the search space by setting bounds on certain variables. Real et al. [4] did not implement these limitations, however it is worth noting that in their study they used 250 machines. The *keep* probability for dropout was randomly generated between 0 and 1 as these are the only acceptable values.

The bounds for each variable are listed below.

- number of filters in 1D and 2D convolution: [10, 100]
- filter size for 1D and 2D convolution: [1, 6]
- kernel size for 1D and 2D max pooling: [1, 6]
- number of units in fully connected layers: [10, 100]
- embedding layer output size: [100, 300]

Algorithm 2: Creating initial population of chromosomes of various architecture sizes

```
input: population_size: population size
begin
    for $i \leftarrow 0$ to population_size do
        Generate chromosome with size = $(\lfloor \frac{i}{10} \rfloor + 1)$
        Determine number of parameters
        Evaluate chromosome's validation accuracy
        Add chromosome to initial population
```

Algorithm 3: Creating an EDEN chromosome.

```
input: chromosome_size: maximum number of genes
       in chromosome
begin
    Initialise an empty chromosome.
    layer_type ← 'cnn'
    for $i \leftarrow 0$ to chromosome_size $- 1$ do
        if $i = 0$ then
            dropout_allowed ← false
        else
            dropout_allowed ← true
        new_layer ←
          $CreateLayer$(dropout_allowed, layer_type)
        Append $newlayer$ to chromosome
        if $newlayer$ is fully connected then
            layer_type ← 'non-cnn'
    Randomly create fully connected layer and append to
      chromosome
    return chromosome.
Function CreateLayer (dropout, type)
    if $type = \text{'}cnn\text{'}$ then
        if $dropout = true$ then
            Randomly create convolution, fully connected
              or dropout operation
        else
            Randomly create convolution layer
    else
        if $dropout = true$ then
            Randomly create fully connected layer or
              dropout operation
        else
            Randomly create fully connected layer
```

B. Parent Selection

During each generation of the GA, parents must be selected to create offspring using a genetic operator. Parents are obtained using a parent selection method. Three common parent selection methods are fitness-proportionate, rank and tournament selection [21]. For this study, tournament selection (algorithm 4) was used given that it was shown to be a successful method by Zhong et al. [22].

The algorithm works as follows. A number (tournament size) of chromosomes are randomly selected from the current population. The tournament size is a user-defined parameter. Once the chromosomes have been randomly selected they are each evaluated using the fitness function. The chromosome with the smallest fitness (a smaller fitness denotes a better performing chromosome since the validation error is used in the computation of the fitness) is then returned as the parent to be used by the genetic operator.

C. Mutation

The recombination genetic operator was not included in our study, similar to Real et al. [4]. For each execution of the mutation operator a single parent is obtained using tournament selection. The mutation operator is applied to the parent to generate offspring$_1$. The mutation operator is then applied to offspring$_1$ and consequently creates offspring$_2$. The fitness of offspring$_1$, offspring$_2$ and the original parent chromosome is compared. The chromosome with the lowest fitness is returned and placed into the new population. Preliminary experiments revealed that performing mutation once on a parent prohibited the algorithm from sufficiently exploring the search space. It was for this reason that we repeated the mutation operator to generate two variations of offspring.

The details about how the mutation operator changes a chromosome are as follows. For a given chromosome, the operator randomly changes either the chromosome's learning rate or the neural network layers. In the case that the learning rate is selected, then a new value for the learning rate is randomly generated as was discussed in section III. In the case where the neural network layers is selected, then the operator either adds a new layer, deletes a layer or replaces one. The choice is made based on the size of the architecture. If the size of the chromosome's architecture has reached the maximum size (predetermined), then a layer can either be deleted or replaced. However, if the size is less than the maximum allowed size then a layer can either be added, deleted or replaced. A constraint was put in place so that the mutation operator cannot remove the first or last layers.

Deletion is performed by randomly selecting any layer (excluding the first or last layers) and removing it from the network architecture in the chromosome. *Replacement* is performed by randomly selecting a layer within the architecture and removing it. An entirely new layer is generated and inserted in the same position as the one which was removed. *Addition* generates a new layer and adds it anywhere in the architecture.

It is possible that the randomness within the mutation results in invalid neural network architectures. After each application of the mutation operator, a check is performed to assess the validity of the resulting architecture. If mutation generates an invalid architecture, then the mutation operator is applied again until a valid one is generated.

The number of neural network parameters is computed for each offspring created. The parameters represent the number of trainable weights in the neural network. Larger values denote more complex models, and small numbers consequently denote less complex ones.

Algorithm 4: Pseudocode for tournament selection.

```
input  : size: size of the tournament
output: The best chromosome which will be used as a parent
begin
    current_best ← null
    for $i \leftarrow 1$ to size do
        random_chromosome ← randomly select a chromosome from the population
        Evaluate random_chromosome
        if fitness of random_chromosome $<$ fitness of current_best then
            current_best ← random_chromosome
    return current_best
```

D. Chromosome Evaluation

A fitness function is required to steer EDEN towards an optimal solution. This function computes a fitness score – a numerical value which denotes how 'good' a chromosome is. For this study, we chose a fitness function that makes use of the error on the validation set (the dataset was split into training, validation and testing subsets) as well as the number of trainable parameters. The relative importance of these two is controlled by α, a complexity parameter. This fitness function rewards less complex and more accurate models compared to more complex, less accurate ones. Furthermore, it helps to break ties when two chromosomes have the same validation error. We fix $\alpha = 1$, but this can be changed depending on the relative importance of performance versus the need for small networks in the problem at hand.

$$\text{fitness}(\text{Net}) = \text{val}_{\text{error}} + \alpha\left(1 - \frac{1}{\text{N}_\text{p}}\right) \tag{1}$$

where

Net = the neural network being evaluated
$\text{val}_{\text{error}}$ = the validation error
N_p = the number of trainable parameters
α = complexity parameter (default $\alpha = 1$)

Once the mutation operator generates an offspring, the architecture and hyperparameters which are encoded in the chromosome are used to train a neural network using TensorFlow. The training data is used in the training of the network. The categorical cross entropy loss function is used during training. Once the network is done training then the neural network is evaluated on the validation data using the fitness function. The fitness obtained from the function is then stored as the chromosome's fitness. The fitness function used in this study is presented in equation 1.

V. Experimental Setup

For each dataset, we executed EDEN 5 times and averaged the results (similar to [4]). EDEN was evaluated on a single machine with a MSI GeForce GTX1070 and 16GB of CPU RAM. During the evolutionary process an experiment used between 4GB to 7GB CPU RAM based on the dataset, and the GPU utilisation varied from 50 to 99 percent during the training of the neural networks. The algorithm was developed in Python 3.6.1, TensorFlow 1.2.1 and Keras 2.0.6 [23]. Keras was used to determine the number of parameters for the neural networks contained in each chromosome. The operating system was Ubuntu 16.04 LTS.

A. Datasets

Table I presents the datasets for which EDEN was evaluated on. IMDB [24] and Electronics [25] are sentiment analysis datasets. The other datasets – namely, MNIST [26], CIFAR-10 [27], Fashion-MNIST [28] and the two EMNIST datasets [29] – were image classification problems. For each dataset, EDEN was trained on the training data, the validation set was used to evaluate the performance of the chromosomes and the test set was used when reporting the results. The training and testing split is presented in the table. The datasets did not contain any missing values, and the class values were converted into their respective one-hot encoded values.

TABLE I
The 7 datasets used in this study. The number of training and testing samples are provided along with the number of classes for each dataset. The IMDB and Elec datasets are sentiment analysis problems, and the remaining datasets are image classification problems.

Dataset	**Training**	**Testing**	**Classes**
CIFAR-10	50,000	10,000	10
Elec	25000	25000	2
EMNIST - Balanced	112,800	18,800	47
EMNIST - Digits	240,000	40,000	10
Fashion-MNIST	60,000	10,000	10
IMDB	25000	25000	2
MNIST	60,000	10,000	10

B. Parameters

Table II present the GA and neural network parameters used throughout this study. Preliminary runs were performed to obtain these values. The purpose of EDEN was to, amongst other things, evolve the neural network's hyperparameters and thus the parameters presented in the table were the only values which were input into EDEN.

VI. Results and Conclusion

Table III presents the average test accuracy and number of trainable parameters. The best results (for the population size which we used) were obtained when using Adam. EDEN was initially configured to include the optimiser function in the chromosome, but the results revealed that this did not improve the results. It is possible that a larger population size would have yielded interesting results by searching for the most optimal network optimiser.

TABLE II
The GA and neural network parameters used in this study. We conducted additional experiments to select these parameters. The number of generations was not set to a high value to avoid extreme runtimes. The neural network parameters were used by EDEN during the training of the neural networks.

Parameter	**Value**
Number of generations	10
Initial population size	100
Tournament size	7
Number of epochs	starting value of 3, incremented by 1 every generation
Weight initialisation - mean & standard deviation	0.00, 0.01
Batch size	1024
Optimiser	Adam [30]

TABLE III
The average best EDEN test accuracy (%) after 10 generations and 13 epochs of training (standard deviation shown in parentheses). The average number of trainable EDEN parameters and the previous state-of-the-art results and reference are also shown alone with the average learning rates (denoted LR) which were evolved for the best chromosomes.

Dataset	**Test Accuracy**	**LR**	**Params**	**State of the art (%)**
CIFAR-10	74.5 (3.1)	0.0024	172,767	**97.14** [33]
Elec	87.2 (0.5)	0.0040	26,625	**93.17** [31]
EMNIST-Bal.	**88.3** (0.8)	0.0019	1,688,43	78.02 [29]
EMNIST-Digits	**99.3** (0.1)	0.0027	3,001,576	95.90 [29]
Fashion-MNIST	**90.6** (0.5)	0.0059	4,624,447	89.7 [28]
IMDB	85.8 (0.6)	0.0053	319,185	**93.34** [31]
MNIST	98.4 (0.3)	0.0031	1,857,601	**99.79** [32]

Table III shows our results. EDEN achieved new state-of-the-art results on the EMNIST-balanced, EMNIST-digits and Fashion-MNIST datasets. For the two sentiment analysis tasks (Elec and IMDB) EDEN evolved neural networks which produced good – but sub-state-of-the-art – accuracy despite EDEN's ability to optimise the embedding layer. In future work, we will determine the effect of also allowing EDEN to optimise the vocabulary size. The average evolved learning rate ranged between 0.00186 and 0.0059. Average execution times, in hours, for a single EDEN experiment of ten generations were 9, 7, 18, 24, 12, and 6 for IMDB, Elec, EMNIST-balanced, EMNIST-digits, CIFAR-10 and Fashion-MNIST respectively. In addition, we enforce the constraint that no networks receive more than 13 epochs of training. As a result EDEN took, on average, 12 hours for the MNIST dataset (accuracy 98.4%); significantly less than the 2 month execution time of EXACT which achieved a similar accuracy of 98.32% [7].

EDEN did not, however, produce competitive results on CIFAR-10, obtaining an average test accuracy of 74.5% after 13 (80.5% after 100 epochs) training epochs of the final network evolved after 12 hours, compared to the current 97.14% state-of-the-art [33]. This is primarily due to the 7-

layer depth constraint we imposed due to running EDEN on a single GPU. As a result the best model that EDEN evolved had only 172,767 parameters, only 0.7% of the 26.2 million used in the state-of-the-art [33]. Figures 2 and 3 illustrate the change in fitness and learning rate over the evolutionary process. Both figures show the convergence of the population. The fitness rapidly decreases from the random initial population to generation 3, after which the fitness decreases at a slower rate.

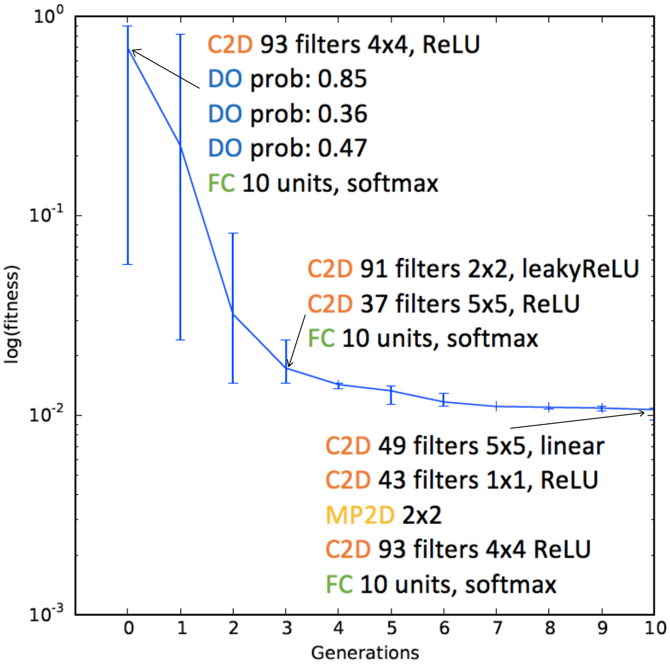

Fig. 2. Illustrating the change in mean fitness over the GA generations for the MNIST data. Error bars show the 5% and 95% percentile values in fitness across the population. Initially there is significant variance in the fitness which reduces as the solutions improve and the population converges. We also show three networks sampled from the initial, mid-point and final generations, along with their associated hyperparameters. We show the best evolved network at three stages during the evolution. Here C2D, MP2D, DO, FC represent 2D convolution, 2D Max Pooling, Drop Out and Fully Connected layers respectively.

Determining optimal or efficient deep neural network architectures and hyperparameters is a challenging task. Researchers and practitioners who are new to the creation of deep neural networks can benefit from algorithms which automatically create architectures and determine hyperparameters. In our study, we propose EDEN, a neuro-evolutionary algorithm that interfaces with TensorFlow – or any other deep neural network platform – to automatically create architectures and optimise hyperparameters. Here EDEN was evaluated on classification problems, but can easily be applied to regression problems.

EDEN is designed to evolve efficient deep networks and for each dataset was executed on a single GPU running for 24 hours or less. The findings reveal that competitive results can be obtained using significantly less computational power than has been deployed in other neuro-evolutionary studies. Evaluated on image classification and sentiment analysis problems, EDEN achieves state-of-the-art results in three of seven datasets. Our study is also a first attempt at applying neuro-evolution to the creation of 1D convolutional networks for sentiment analysis, optimising an embedding layer for sentiment analysis. In future work, we intend on extending EDEN to evolve generative adversarial networks architectures, as well as exploring parallel implementations.

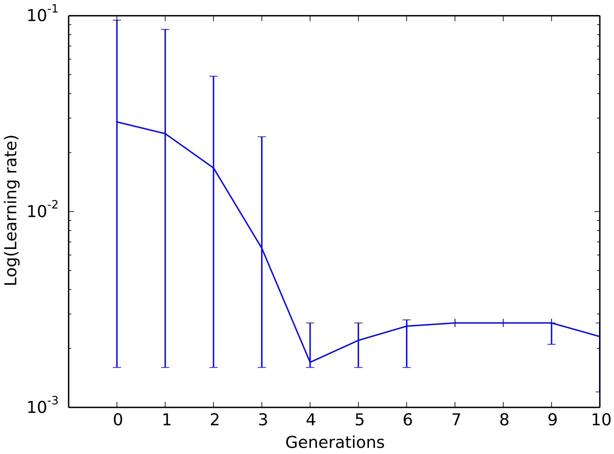

Fig. 3. Change in mean learning rate over the GA generations. Error bars show the 5% and 95% percentile value in terms of the learning rate variance in the population. Initially the chromosomes are random so there is a lot of variance in the learning rate. This changes as the population converges towards better solutions.

Acknowledgment

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the authors and are not necessarily to be attributed to the NRF.

References

[1] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training very deep networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 2377–2385. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969442.2969505

[2] G. I. Sher, *Handbook of Neuroevolution Through Erlang*. Springer Publishing Company, Incorporated, 2012.

[3] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 379–384., 1989

[4] E. Real *et al.*, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[5] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[6] R. Miikkulainen *et al.*, "Evolving deep neural networks," *arXiv preprint arXiv:1703.00548*, 2017.

[7] T. Desell, "Large scale evolution of convolutional neural networks using volunteer computing," *arXiv preprint arXiv:1703.05422*, 2017.

[8] B. T. Zhang and H. Mhlenbein, "Balancing accuracy and parsimony in genetic programming," *Evolutionary Computation*, vol. 3, no. 1, pp. 17–38, 1995.

[9] J. Arifovic and R. Genay, "Using genetic algorithms to select architecture of a feedforward artificial neural network," *Physica A: Statistical Mechanics and its Applications*, vol. 289, no. 3, pp. 574 – 594, 2001.

[10] M. A. J. Idrissi *et al.*, "Genetic algorithm for neural network architecture optimization," in *2016 3rd International Conference on Logistics Operations Management (GOL)*, pp. 1–4., 2016.

[11] M., Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[12] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[13] E. Dufourq and B. A. Bassett, "Automated problem identification: Regression vs classification via evolutionary deep networks," in *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, ACM, 2017.

[14] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, pp. 143–155, 1989.

[15] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[16] N. Srivastava *et al.*, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[17] Y. T. Zhou and R. Chellappa, "Computation of optical flow using a neural network," in *IEEE 1988 International Conference on Neural Networks*, pp. 71–78 vol.2., 1988.

[18] A. L. Maas *et al.*, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, ser. HLT '11. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 142–150., 2011.

[19] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[20] R. Poli *et al.*, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[21] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[22] J. Zhong *et al.*, "Comparison of performance between different selection strategies on simple genetic algorithms," in *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, vol. 2, pp. 1115–1121., 2005.

[23] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[24] A. L. Maas *et al.*, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, pp. 142–150., 2011.

[25] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," *arXiv preprint arXiv:1412.1058*, 2014.

[26] Y. LeCun *et al.*,"Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[28] H. Xiao *et al.*, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[29] G. Cohen *et al.*, "Emnist: an extension of mnist to handwritten letters," *arXiv preprint arXiv:1702.05373*, 2017.

[30] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[31] D. Vishwanath and S. Gupta, "Adding cnns to the mix: Stacking models for sentiment classification," in *2016 IEEE Annual India Conference (INDICON)*, pp. 1–4., 2016.

[32] L. Wan *et al.*,"Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, pp. III–1058–III–1066., 2013.

[33] X. Gastaldi, "Shake-shake regularization," *arXiv preprint arXiv:1705.07485*, 2017.