ADVANCED NEUROEVOLUTION: A GRADIENT-FREE ALGORITHM TO TRAIN DEEP NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this paper we present a novel optimization algorithm called Advanced Neuroevolution. The aim for this algorithm is to train deep neural networks, and eventually act as an alternative to Stochastic Gradient Descent (SGD) and its variants as needed.We evaluated our algorithm on the MNIST dataset, as well as on several global optimization problems such as the Ackley function. We find the algorithm performing relatively well for both cases, overtaking other global optimization algorithms such as Particle Swarm Optimization (PSO) and Evolution Strategies (ES).

1 INTRODUCTION

Gradient Descent (GD) and its variations like stochastic gradient descent Bottou (2010) are the de facto standard for training deep neural networks (DNNs) for tasks in various domains like Object Detection Liu et al. (2016), Robotic Grasping Lenz et al. (2015) and Machine Translation Bahdanau et al. (2014). Most of the field of Deep Learning is centered around algorithms similar to variants of Gradient Descent to find the optimal weights given desired input/output pairs Goodfellow et al. (2014), Finn et al. (2016), Mnih et al. (2013). However, there are also some limitations to using gradient-based optimization. For example, the neural network and the loss function have to be differentiable end-to-end. As a consequence, there are a number of problems that can not be directly modeled or solved without some alterations such as Formal Logic and Hard Attention (Luong et al., 2015). Note that throughout this paper, we will refer to gradient-based methods collectively as SGD. Similarly, we will refer to Advanced Neuroevolution with the acronym AdvN.

For those reasons, we developed a new algorithm which we call Advanced Neuroevolution. It is not a single algorithm, in truth. It is an ensemble of low-level algorithms, layered on top of each other. Those low-level algorithms have different scopes of operations addressing different levels of abstraction in the search process. For example, the perturbation mechanism addresses the introduction of noise into the models, the most basic operation. In contrast, the minimum distance mechanism addresses the global scale properties, i.e. the search regions. The goal is to traverse the search space as efficiently as possible without use of gradients. In the case of neural networks the search space is the weight space, including biases.

Indeed, while this algorithm was developed primarily for training of deep neural networks, it can be used for other optimization tasks. In essence, we present the algorithm as an evolutionary optimization algorithm, with a focus on DNNs.

There are many global optimization algorithms such as Evolution Strategies Michalewicz (1996), Particle Swarm Optimization Kennedy (2011) and Simulated Annealing Van Laarhoven & Aarts (1987). Each has its merits and limitations. Our aim is not to compete directly with those algorithms but rather to complement them and offer another option with its own merits and limitations. To evaluate the performance of such algorithms we can use well-known benchmark functions such as the Rastrigin or Ackley function. We recognize those functions and test Advanced Neuroevolution against them to assess its performance.

In addition, there have been other approaches to using evolutionary optimization techniques to train DNNs, see (Such et al., 2017) and (Salimans et al., 2017) as recent examples. It reflects the awareness within the broader research community about the potential of such algorithms, and the need for alternatives to SGD. We don't see our algorithm replacing SGD, especially in fields where it is already quite successful such as Computer Vision. Our aim is to complement it, by offering another option. Furthermore, there is no reason why both can not be used in tandem as part of a grander learning strategy.

1.1 Background

We noted that there has been a limitation in learning potential due to the employment of SGD. Specifically, SGD is prone to losing the model's top performance throughout training iterations. This is usually due to dynamic elements in the environments, and sometimes can also arise from batch training. To elaborate, at one point in training t_i the agent may achieve a certain high score, say 9000. At a later point in time, t_{i+1}, there is no guarantee that the agent will not regress to a lower score when faced with the exact same conditions. While this behavior is a natural part of SGD, and happens frequently during the training progress, the algorithm eventually overcomes these temporary regressions.

In Robotics, and associated fields such as Reinforcement Learning, this beahvior can amount to being sample-inefficient. That is, the agent usually takes a relatively large number of samples, possibly in the order of 10^6 to 10^8, to achieve competitive learning(Mnih et al., 2016). In such cases, a large amount of labeled data is required which is often costly to create. Alternatives like simulation have other potential issues with producing training data which is representative for reality. This is another reason why we wanted to investigate gradient-free approaches.

The algorithm we propose always maintains its best performing agent. Indeed it is wholly different from SGD as an optimization step. Advanced Neuroevolution seeks to train neural networks based on the concept of evolving a pool of agents, whereas SGD iteratively optimizes a single agent. One advantage for this approach is that it inherently achieves exploration and exploitation, whereas SGD usually struggles on exploration due to its gradient-following constitution.

We see this paper as a starting point, for the whole Robotics community to engage in developing of alternative optimization for DNNs used in Robotics and RL. Training neural networks using optimization algorithms is not a novel idea, see for example (Meissner et al., 2006). However, we aim to improve upon the shortcomings of other algorithms with the mechanisms we introduce here. Our algorithm should be able to handle sparse-rewards, exploration, exploitation, high-dimensionality, computational efficiency, pool-efficiency and sample-efficiency.

The field of evolutionary computation is rich with spectacular implementations and algorithms. For example, (Real et al., 2017) use neuro-evolution as a proxy algorithm to evolve the architectures of the networks, rather than the networks themselves. They also use computer vision is a starter task, though they apply their method on the more challenging sets of CIFAR10 and CIFAR100. The networks themselves are trained using SGD. In contrast, we evolve the weights of the networks themselves rather than the architectures. Similarly, the work in (Fernando et al., 2016) also uses gradients to evolve the networks. The authors in (Such et al., 2017) train models of 4M parameters, and cite it as one of the largest models ever trained with evolutionary algorithms. Following this interesting path, we decided to use a model on a similar scale for the MNIST classification task, to see whether our algorithm can solve as large a network or not.

1.2 Contributions

We summarize the contributions of the paper as follows.

1) Introduce an evolutionary optimization algorithm called Advanced Neuroevolution (AdvN).

2) Benchmark AdvN algorithm and other optimization algorithms, on solving global optimization functions.

3) Test AdvN against functions with special properties such as the sparsely-rewarding Easom, to evaluate its behavior. For example, Easom is a function that has near-zero gradient everywhere except a small region compared to the search space. Gradient-following in this scenario would be an implausible strategy.

4) Test AdvN and SGD on hand-written digit classification dataset MNIST. We impose special conditions to see how the algorithm can adapt to typical training challenges.

2 Algorithm Design

The algorithm is actually a set of smaller, locally-aware algorithms and functions all operating to achieve a global synergistic behavior. We describe each in turn as follows, and eventually integrate all of them so as to portray the entire algorithm. Please note that we prefer to use nomenclature that is different from traditional evolutionary optimization. It reflects our way of thinking about the problem. Also, using standard evolutionary/genetic nomenclature such as ”Mutations”, sometimes hinders the mental grasp on the problem because of the association with biological processes. We do not wish to crystallize around or closely subscribe to the mechanisms of Biological evolution.

For those reasons, we developed our own notation, which we share in the proceeding sections. Another note is that we employ a fixed-size pool, and the fixed network structure. This is in contrast to other evolutionary algorithms where the network structure may be changed throughout the training process.

2.1 Sample Pool

For the first iteration of the algorithm the pool will consist of a set of random samples chosen chosen according to the weight initialization scheme, e.g. Xavier Normal Glorot & Bengio (2010). The subsequent iterations of the pool will be made up of the following components: Elite, Anchors, Probes and Blends. The Elite is the historically best-performing sample. Anchors are the best-performing N samples in a generation, where N is the desired number of anchors. These samples are copied over to the new pool as is. From each anchor M probes are spawned, where M is the number of probes desired. Those probes are clones of the anchor, each randomly perturbed. In addition to the anchors and probes, the Elite is copied over as well. The remaining slots are filled with blends. Blends can occur between an anchor and any other sample in the pool.

2.2 Perturbation, Search Radius and Blends

The most basic step in the algorithm is the Perturbation step. This is equivalent to genetic mutation, prinicipally referring to the introduction of noise into the makeup of the system. It is the primary searching mechanism in the algorithm. Two important properties are the magnitude and shape of the noise distribution. We choose uniformly-distributed noise, centered at the origin. We refer to limits of the uniform distribution as Search Radius. It can be thought of as defining a virtual radius around the anchor where the probes will be cast in random directions. The greater the radius the farther away the probes will be casted. This tunes the explorative vs. exploitative properties of the algorithm. The limit of the uniform distribution, ie. Search Radius, is calculated with respect to the integrity value and 2 pre-defined constants. Search radius is calculated as

$$SearchRadius(p) = (tanh((\lambda p) - 2.5) + 1) * lr, \tag{1}$$

where p = (1-integrity), λ and lr are scalar constants. The learning rate, lr, scales the function, controlling the search radius. The shifted, scaled hyperbolic tangent function has attractive properties in the range [0,1]. It has an almost-flat slope near 0 and an almost-flat slope near 1. This allows the algorithm to spend more time searching low-energy regions where the most likely rewards are, ie. exploitation. It also prevents the algorithm from searching exceedingly high-energy configurations, ie. controlling exploration.

Similar to the Search Radius, is the number of selections. This variable determines how many weights will be perturbed. It is calculated as

$$Selections(p) = (\frac{\alpha}{1 + \frac{\beta}{p}}), \tag{2}$$

where p = (1-integrity), and α and β are scalar constants. In the range [0,1] this function starts with at the origin and increases until it saturates near top. This saturation limits the number of modifications in the network according to the chosen constants. Intuitively, making too many adjustments to a neural network model in one step is usually un-rewarding. In addition, when searching high-energy regions, ie. low integrity, the number of selections can exceed reasonable limits. This creates a lot of wasted cycles where the search is not profitable. The experimenter chooses the constants introduced in equations 1 and 2 before starting the algorithm.

Blending mechanism takes randomly-chosen weights from 2 components to yield a blend. The first component is always one of the anchors, picked at random. The second component can be any sample in the pool. This is commonly referred to as Crossover. The first component is cloned, to form the basis. Then a number of weights are randomly picked from the second component. This number is calculated by equation 2, using the same constants. The importance of blending is that it potentially allows us to explore regions in the search space which are otherwise unreachable without effort. Intuitively, blends have the potential to transport samples farther away from the current search region, shortcutting the general mechansim of perturbations. Blends help to extend the actively searched area in the search space based on already known favorable samples.

2.2.1 Elitism

There are two sorts of elites, a generational elite, and a historical elite. The generational elite is the best-performing sample of the current pool. The historical elite is the best performing sample across all generations. In our algorithm the generational elite is one of the Anchors, and the historical elite is simply called the Elite.

2.2.2 Adaptive Integrity and Backtracking

The integrity concept defines the scope and magnitude of perturbations. The lower the integrity, the greater the magnitude and the scope of perturbations introduced into the model. The magnitude and scope of perturbations follow a non-linear trajectory with the integrity. The governing functions are equations 1 and 2. Generally, neural networks are not updated in bulk. Rather, updates should come as small incremental changes to the network. For those reasons, intuitively, we saturate the magnitude and scope of perturbations and blends in order not to exceed thresholds. Those thresholds are parameters of the algorithm. They control how aggressive the perturbations and blends are allowed to get.

Integrity decreases when the current generation of samples does not yield a score that is incrementally better than the previous best score. There is a parameter in the algorithm that defines the required minimum percentage change in the score, we call it Minimum Entropy. For example, in the global optimization scenario, we set the minimum acceptable percentage improvement, ie. Minimum Entropy, to 1%. Thus, if the current generation does not improve upon the previous generation's score by at least 1% then the integrity is decreased by a fixed step size. The experimenter determines the appropriate step size.

Backtracking curbs the behavior of integrity. It can be thought of as a layer on top of integrity. If integrity keeps decreasing, without improvement in the reward signal, then backtracking interferes and resets integrity re-inserts the Elite. Essentially, as integrity decreases we search higher-energy regions of the search space. Eventually, the model becomes ”hot” from applying high-magnitude perturbations, and the weights start exploding. To fix this problem, backtracking resets the integrity back to maximum, and inserts the historical elite in place of the generational elite. When the elite is inserted, and the probes are spawned with high integrity, essentially copying the elite with minor modifications, then the entire pool ”cools down”. This mechanism makes it possible to explore high-energy configurations more safely. It is somewhat reminiscent of the simulated annealing.

2.2.3 Anchors, Probes and Distance

Anchors constitute the main search mechanism. Essentially, anchors constitute the best N-performing agents, in order. Thus for example if there are 5 anchors, those will be the 5 best-performing agents. Anchors are updated each generation, as well as being carried over to the next generation in case no improvement was made. The best anchor is also the generational elite.

We spawn probes as clones of the anchors, but with the perturbations applied to them. Intuitively, what this means is that probes are spawned to search the local neighborhoods of the anchors. By having multiple anchors we search multiple regions in tandem. By having multiple probes, we search multiple local neighborhoods within those regions.

Distance is a metric that represents how far the anchors can be from each other. It is calculated in terms of magnitudes and positions of difference between the weights of the anchors. Essentially, it's inefficient for the anchors to be close to each other. It would be practically searching the same region. Thus a minimal distance is defined. For that reason we choose the Canberra distance as given below:

$$d(x, y) = \sum_{i=1}^{n} \left(\frac{|x_i - y_i|}{|x_i| + |y_i|}\right), \tag{3}$$

where x,y represent the two anchors.

2.2.4 Radial Expansion

Similar to distance where we don't want the anchors to collapse to a single region, we also introduce the concept of radial expansion to achieve the same goal. Radial expansion adaptively adjusts the perturbations magnitude and scope. We do this in order to cast the samples far away from each other so that we can maintain a full roster of anchors. For example, if we have a roster of 5 anchors and we're only using 3, then it means that all the other anchors are not far enough according to the distance metric. Thus we lose 2 anchors, and operate only 3 anchors.

The remainder of the slots do not necessarily go to waste, they are given to the blending algorithm. However, this is not the ideal situation, since we expect the algorithm to be use the number of anchors allocated. Therefore, the mechanism of radial expansion increases the search space by increasing the value of the parameters governing the perturbations magnitude. This in turn forces the probes to be casted farther from the anchors, thereby allowing a greater diversity (and consequently distance) within the pool. We find that this mechanism is significantly involved in the training of networks solving the MNIST task.

2.2.5 Collective Algorithm

Collectively, the algorithm operates on two steps. The first is a conditioning step that takes into account the current and previous states of the algorithm, and decides the integrity to use, who are the anchors and the elite for the upcoming generation. The conditioning step makes use of the Distance and Expansion mechanisms described above. The second step is the execution step, the actual formation of the pool through the perturbation and blending steps.

3 Implementation

This section defines the implementation details of the algorithm, as well as the benchmarking processes. Across the entire experiment set, we use only a pool size of 50 samples. This remarkably low number emphasizes the objectives we are trying to achieve with this algorithm. The algorithm is written and implemented in PyTorch framework Paszke et al. (2017). The networks used are common architectures. On this end, however, we think there is a vast opportunity to develop network architectures that are tailored to this sort of algorithm. Thus far, the architectures used have been tailored for SGD featuring layers such as Dropout. In our experiments, we find that the performance is sensitive to the architectural choice. However, optimizing the architecture is beyond the scope of this work.

We use 4 Nvidia Titan V GPUs for data parallelization. That is, the models for copied into each GPU and the inference is run by dividing the dataset into 4 chunks, and then the loss is aggregated. For all our experiments we use a pool size of 50, 4 anchors, and 8 probes per anchor.

3.1 Global Optimization

Our aim was to showcase the algorithm on a number of tasks which are relatively simple. We decided to benchmark our performance against that of typical global optimization algorithms. We use the Inspyre and PyBrain libraries implementions Garrett (2017) Schaul et al. (2010), along with the default parameters. It is typical that any optimization algorithm is sensitive to the choice of parameters. However, tuning those parameters is beyond the scope of this paper.

We test on a set of 7 well-defined functions. Each function is tested 5 times, and the average number of generations required to converge is presented. Note that we define convergence as reaching a value within 0.06 of the global optimum. For example, if the global optimum is 0, then when the algorithm reaches 0.06 or less, it terminates. We solve the global optimization problem as a regression problem. The architecture we used is a single hidden layer of size 128. The functions are all 2-dimensional, therefore the network accepts two real numbers as the (x,y) origin and outputs 2 real numbers as optimal (x,y). The value of the function at the predicted (x,y) is the network's cost, and the aim is to minimize that cost. The input coordinates to the network are picked from a uniform distribution for each run, but remains constant throughout that run. The uniform distribution's limits differ for each function. Specifically, they are [-5,5] for Ackley, [-5.2,5.2] for Rastrigin, [-2,2] for Rosenbrock, [-500,500] for Schwefel, [-15,-5,-3,3] for Bukin, [-20,20] for Easom and [-512,512] for Eggholder. For fair comparison they are the same for all the algorithms in the benchmark.

We run the algorithms for a maximum of 5000 generations. If the algorithm has not converged, it is assumed that it is stuck in a local minima which it can't escape. If this is the case, then we calculate this particular run as taking 5000 generations. If all the runs don't converge within 5000 generations, we note this with a ”+” sign, e.g. 5000+.

In addition to regular benchmark functions, we wanted to test the algorithm against some special functions with special properties. For example, the Easom function is flat throughout except one regions where it is depressed. A typical gradient-following algorithm such as SGD will struggle to solve this function. It reflects an aspect of the sparse-reward problem in domains such as RL. Those special functions are not used for benchmarking, however, because the benchmarking libraries do not carry their implementations.

3.2 Image Classification

In this set of experiments, we run the algorithm against a simple CNN to solve the MNIST digit classification problem. The model we trained had 4 convolutional layers of size 32, 64, 128 and 128 respectively with stride 2 and max-pooling in between, and a single fully-connected layer of size 512. The activation used was the Parametric ReLu function in PyTorch. The model has 4.7M parameters. To speed up the computations we switched to using half-precision floats.

Interestingly, in our experimentation we found that the algorithm does not need much data at all to train. For that reason we used only a randomly chosen subset of the MNIST dataset, of 2000 images. When we run validation, however, we run against the entire validation set of 10,000 images. This not only tests how well the model generalizes, but also how much it can learn from a limited-dataset. This problem is of relevance to domains with relatively small volume of annonated data, such as Medical Imaging (Altae-Tran et al., 2017).

The test is terminated once the algorithm achieves a loss of 0.15 or lower, or if the number of generations exceeded 5000. We assert this limit in order to prevent runs of diminishing returns. Our goal is not to achieve or exceed the state of the art in MNIST classification. The system would need to be tuned beyond the casual effort, and then the training would take a different, more aggressive approach. Our aim is to showcase how the algorithm performs in an appreciatively high-dimensional search space.

4 Results and Discussion

4.1 Global Optimization

As mentioned earlier, the algorithm is tested on a number of common global optimization functions. For comparison we test several other optimization algorithms. Namely, Evolution Strategies Michalewicz (1996) (ES), Particle Swarm Optimization (PSO) Kennedy (2011), Differential Evolution (DE) Qin et al. (2009), Simulated Annealing Van Laarhoven & Aarts (1987), Fitness-Maximization Expectation (FEM) Wierstra et al. (2008), Policy Gradients with Parameter Exploration (PGPE) Sehnke et al. (2010) and Random Search. The results are show in Table 1. The results show the Advanced Neuroevolution algorithm outperforming the others by a significant margin. We estimate that the vanilla implementations of those algorithms in the Inspyred and PyBrain libraries are not the best. There are various implementations of Evolution Strategies, for example,

Table 1: Results of running Advanced Neuroevolution and other algorithms against typical global optimization problems, averaged over 5 runs. Some entries for the benchmark algorithms are missing due to limitations in the respective library implementations.

Function	**Number of Generations to convergence**							
	Advanced Neuroevolution	ES	PSO	DE	Simulated Annealing	FEM	PGPE	Random Search
Ackley	17	36	117	659	5000+	287	152	4149
Rastrigin	49	2020	418	632	2368	2389	421	3074
Rosenbrock	20	67	730	2415	2398	-	-	-
Schwefel	113	2019	492	2310	5000+	-	-	-
Bukin	28	-	-	-	-	-	-	-
Easom	9	-	-	-	-	-	-	-
Eggholder	170	-	-	-	-	-	-	-

and some of them can definitely outperform the vanilla implementation shown here. Despite this, the results clearly suggest that AdvN is a strong competitor. Our algorithm performs well and solves all the benchmarking functions in a relatively low number of generations.

In addition, the algorithm solves the special functions we give it. The Eggholder function is notorious for being difficult since the global optimium is in a corner, and the function is not convex. Therefore it is quite difficult for an algorithm that doesn't explore enough to ever find the global optimum, but our algorithm does, and with only a pool size of 50 sampels. Similarly, a gradient-following algorithm would be stuck in the Easom function where the gradient is near zero throughout, save for one small region. We see that our algorithm explores well, and converges upon the reward-bearing region. The Bukin function has a narrow ridge where the global minimum lies. If the exploration steps are too large, the algorithm will generally skip over the ridge.

Again, we highlight that those results were achieved with the same set of parameters across all the functions. The parameters of the algorithm are not changed to solve a particular function. Rather, it has the required agility to deal with the different conditions imposed by the different functions. It is a desirable feature because throughout learning, the problem space may impose varying constraints and dynamics. The algorithm needs to be agile enough to handle those conditions as they arise.

4.2 MNIST HAND-WRITTEN DIGIT CLASSIFICATION

A sample run of the training loss function against the number of inferences is shown in Figure 4.2. The algorithm takes 2333 generations to converge to a training loss of 0.15, averaged over 5 runs. This corresponds to a validation accuracy of 90%. Similarly SGD takes 320 generations (ie. batches of 50) to converge to the same training loss, and achieves also a validation accuracy of 90%. It must be noted that 2333 generations over 2000 images with a pool size of 50 means that our algorithm on average runs inference 233,300,000 times. That is orders of magnitude more than SGD. In our run, SGD takes around 16,000 inferences to converge to a similar training loss. Therefore, we see a clear trade-off in the number of inferences required, in favor of SGD. This may be the case because the gradient is smooth in this task, it is possible that in other tasks, the gradient-space is more challenging to navigate, or uninformative.

With only 2000 images from the training set, the model was able to achieve a relatively acceptable accuracy. Note that we don't use shuffling of those 2000 images. The network sees the same images in the same order. It would be interesting to see what role layers such as Dropout would play if we train other networks using AdvN.

It is clear that the algorithm can handle the complexity of the 4.7M parameter network. It suggests that we can use it to train deeper, wider networks, on more complex tasks. Furthermore, we see that half-precision is not a hindrance for the algorithm. This bodes well for future iterations since we train on Volta architecture GPUs, which use tensor cores to execute those operations. It would be interesting to see if we can achieve the same level with even lower precision floats, if any such become available in the future.

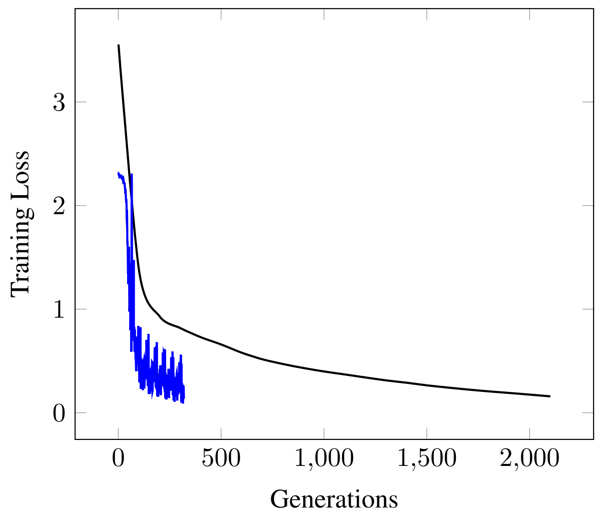

Figure 1: A sample run for the Advanced Neuroevolution algorithm in Black and SGD in Blue on the MNIST dataset classification task. For AdvN, the loss swiftly decreases and then saturates towards the end. For SGD, the typical oscillatory behavior is exhibited. This run was terminated when the loss reached 0.15, at generations 2161 and 320 respectively.

5 Conclusion

We presented the Advanced Neuroevolution algorithm as an alternative optimization step to SGD to train neural networks. The work is motivated by some limitations we perceived in gradient-based methods, such as differentiability and sample-inefficiency. The algorithm is benchmarked against other optimization algorithms on typical optimization problems. It performed satisfactorily well, and improved upon all of them. For fairness, we noted that the implementation of the other algorithms may not be optimized, and they can arguably perform better.

Next our algorithm is tested on the MNIST digit classification task. It achieved 90% accuracy on the entire validation set using only 2000 images from the training set. In all our experiments, half-precision floats are used in order to decrease the time of the computations. The computations are done only on 4 Titan V GPUs instead of thousands of CPU cores as in other evolutionary algorithms papers. This makes training of neural networks with evolutionary algorithms more tractable in terms of resource requirements.

Finally, while not presented in this work, preliminary tests of our algorithm on RL tasks have been promising. It solves the assigned problems, though it takes longer than other approaches. We aim to improve upon the algorithm and the strategies employed in order to achieve competitive results on RL and Robotics tasks.

Acknowledgments

We would like to thank Vik Goel and Qiyang Li for their insightful comments and discussions.

References

Han Altae-Tran, Bharath Ramsundar, Aneesh S Pappu, and Vijay Pande. Low data drug discovery with one-shot learning. *ACS central science*, 3(4):283–293, 2017.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 109–116. ACM, 2016.

Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 512–519. IEEE, 2016.

A. Garrett. inspyred: Bio-inspired algorithms in python, June 2017. URL `http://aarongarrett.github.io/inspyred/`.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pp. 760–766. Springer, 2011.

Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pp. 21–37. Springer, 2016.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Michael Meissner, Michael Schmuker, and Gisbert Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural network training. *BMC bioinformatics*, 7(1):125, 2006.

Zbigniew Michalewicz. Evolution strategies and other methods. In *Genetic Algorithms+ Data Structures= Evolution Programs*, pp. 159–177. Springer, 1996.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE transactions on Evolutionary Computation*, 13(2):398–417, 2009.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pp. 7–15. Springer, 1987.

Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. Fitness expectation maximization. In *International Conference on Parallel Problem Solving from Nature*, pp. 337–346. Springer, 2008.