EqPropNEAT
Extending NEAT with equlibrium propagation

Cs. Karikó, M. Kovács, Zs. Tornai

***Abstract*—Artificial neural networks have been employed successfully to solve a great variety of tasks over the past decade. In most applications, network architecture is usually decided with a trial-and-error process, relying on empirical experience. The neuoroevolution of augmenting topologies algorithm (NEAT) [1] proposes a solution to finding well fitting network architectures to a given task, however for learning weights, it only uses a simple genetic mutation rule.**

This paper proposes an extension of NEAT with an additional learning step employing equilibrium propagation [2] for learning edge weights. We claim that the resulting novel technique needs less iterations for convergence, as weights are adjusted by a method which was specifically invented for such task. Our experimental results show a significant reduction in training times on common neural network benchmark tasks, while achieving similar prediction quality.

***Index Terms*—neuroevolution, neat, equilibrium propagation.**

I. Introduction

THE process of designing neural network models always involves deciding on a particular network architecture. In order to avoid both overfitting as well as subpar prediction quality, the right amount of network complexity has to be found for a particular learning task. Over the years, a great amount of experience has accumulated regarding this decision, however these rules are more about empirical know-how, than proven theorems. For our experiment we took well tested and proven algorithms NEAT and Equlibrium Propagation(EP) [2] and tried to design a neural network which best represents a biological solution to a given task. For this reason EP was chosen rather than Backpropagation(BP) [3] to further fine tune the network in a way which would be plausible in a biological network. Due to the similarities between BP and EP we hope to capture all the positive aspects of BP while still maintaining the properties of a biological network. Due to the nature of EP our networks learns slower than a comparable network would utilizing BP, but as BP is considered "biologically implausible" we accept this shortcoming. In *Performance* chapter we will discuss the performance compared to BP and we will explain why the difference in real-world applications would not be significant.

II. Materials and Methods

A. Background

NeuroEvolution of Augmenting Topologies (NEAT) [1] is a genetic algorithm for evolving neural networks. The following is a brief description of the most important steps so we can give a clear picture on how our method differs from the original.

1) Take an initial population of genomes representing (exact details of this representation can be found in [1]) mini neural nets with no hidden layer and random connections from the input layer to the outputs.
2) Divide this population into species based on a network similarity metric. Historical markings introduced by the NEAT method can be utilized for comparing networks efficiently.
3) Repeat the following iterations until the networks' performance reaches a level deemed satisfactory, or a predefined maximal number of iterations is reached.
   a) Create the neural networks corresponding to each genome of the population. The original paper calls these networks "organisms".
   b) Divide these organisms into species based on a network similarity metric. Historical markings introduced by the NEAT method can be utilized for comparing networks efficiently.
   c) Evaluate each organism on the given task resulting in a fitness score.
   d) Calculate species-wide scores by averaging or taking the maximum score of each member. From now on, each organism's score is considered equal for a given species. This step may also employ multiple heuristics such us increasing scores of "young" species for additional diversity.
   e) For each species, sort its organisms and mark the genomes from a given lower percentile for removal at the end of the current generation.
   f) Calculate the target number of off springs for each organism, with a weight favoring those with a higher fitness score.
   g) Create some genomes via crossover. Parents should be chosen based on the weights calculated in the previous steps. Additionally, genomes from the same species should receive a higher weight.
   h) Create other new genomes by mutating randomly selected genomes. For the selected genome, one of the following mutations is applied:
      - Connect two previously unconnected nodes (respecting the feed forward structure of the network). The new connection receives a random weight.
      - Perturb the weight of the connections, or assign entirely new random weights. The ratio of affected connections and the probability of the two

disclaimer: TODO

Write Exercise Figure Results

methods (and their ranges) are hyper parameters.
- Disable or enable a connection.
- Create a new node by splitting a randomly selected connection in half. The old connection is disabled.

i) Assign the newly created genomes to one of the existing species or create their own if they are distant from any current species (by the previously mentioned metric).

As mentioned above, the NEAT algorithm uses a simple genetic mutation rule for evolving edge weights. With a sufficiently large population this method can evolve correct connection weights, however this process usually requires lot of generations. A number of extensions have already been proposed for improving weight search by incorporating BP to the original algorithm [4]. Such attempts include Learning-NEAT [5], DeepNeat and CoDeepNeat [6]. A common property of these techniques is that a limited number of training epochs are run during evaluation for adjusting edge weights and determining network performance.

A brief summary of already existing methods based on NEAT As it is mentioned above, there are many types of NEAT. HyperNEAT is a kind of extension of NEAT that uses a Compositional Pattern-Producing Network, which is a form of indirect encoding. It an attempt to decrease the difference between the result producedFirst of all, there are two versions for Deep Neural Network. The first type, DeepNEAT is an extension of NEAT for deep networks. It has the same base process as NEAT: In the first step, a chromosome population (each represented by a graph) with minimal complexity is created. Through generations, the structure is added to the graph incrementally due to the mutation. To determine how genes of two chromosomes can be lined up, historical markings are used. That is called crossover. In this whole population, there are more species, which are based on the similar metric. Each species evolve proportionately to its fitness and evolution happens separately in each of them.

DeepNEAT is different from NEAT in the aspect, that each node in the chromosome represents a layer, instead of a neutron, in a Deep Neural Network. Each note contains a table of hyperparameters, which determine type and property of that layer. The edges simply indicate how the nodes (layers) are connected. To construct Deep Neural Network from a DeepNEAT chromosome just simply needs to replace each node with the corresponding layer. The chromosome contains a set of global hyperparameters as well, which can be applied to the entire network. During the evaluation, each chromosome is converted into a Deep Neural Network, which is then trained for a fixed number of epochs. After training, a fitness value that indicates the performance is returned to DeepNEAT and assigned as suitable to the corresponding chromosome. The resulting structures are complex and unprincipled.

In addition, for deep neural networks to get more successful, they should be composed of modules that are repeated multiple times. These modules have complicated structures with the branching and merging of various players. A variant of DeepNEAT, which is using this theory, called Coevolution DeepNEAT (CoDeepNEAT). In CoDeepNEAT there are separately evolving populations of modules and blueprints, but they are using the same methods as DeepNEAT, which is described above. A chromosome which is a graph, where each node contains a pointer to a given modul species, is called a blueprint chromosome. On the other hand, each module chromosome is a graph, which is representing a small deep neural network. During the evaluation, the modules and blueprints are merged to establish a larger assembled network. CoDeepNEAT can develop repeating modular structures efficiently. Additionally, the cause of the small mutation in the modules and blueprints frequently leads to large changes in the assembled network structure. CoDeepNEAT can discover more various and deeper architectures, than DeepNEAT.

Another really important point about NEAT, that it is a great solution at global search, but it is not proper for fine-tuning local search. In this case, NEAT can be more efficient if it is combined with a local search scheme. One of the most effective local search algorithm is back propagation, that can derive correct weight. The primary aim of NEAT is not to evolve the most optimal solution, but to develop network that can react great to back propagation training. Back propagation has some disadvantages as well, e.g. difficult to work effectively, sensitive initial condition of the network. Learning-NEAT (L-NEAT) is combining NEAT and back propagation for data classification problems. It is a proper solution for both global and local search, since it is using the advantages of both methods, so it efficiently and effectively produce classifying neural network with well generalization ability. In this way it performs better than NEAT.

Equilibrium propagation [2]

Equilibrium propagation proposes a new approach to training neural networks. Unlike conventional artificial neural network models, equilibrium propagation models the network as a continuous-time dynamic system. Training and evaluation is accomplished with a numerical simulation of changing energy states converging to an equilibrium. After the simulation has arrived at a fixed point, the results can be read from the state of output nodes. Training is achieved by ”nudging” the states of the output nodes towards the correct prediction. This change than propagates throughout the network and weight updates can be computed from the two states of a given node during the evaluation and the training phase.

The benefit of this algorithm over backpropagation is that it is possible to implement it in analog hardware [7] giving way to a high performance hardware implementation of our method, which will be discussed in a later section.

However, equilibrium propagation can be implemented in software as well. To accomplish this, one has to compute two fixed points at each iteration. One for determining the network's prediction with the give input, and another one for calculating the necessary weight changes.

The following is a brief overview of the formulae used from [2]. First, consider the following energy function:

$$E(u) := \frac{1}{2}\sum_i u_i^2 - \frac{1}{2}\sum_{i\neq j} W_{ij}\rho(u_i)\rho(u_j) - \sum_i b_i\rho(u_i)$$

Writing Exercise Five Results

Where u denotes a node, u_i its value, $\theta = (W, b)$ the network parameters (weights and biases) and ρ the activation function. The cost function:

$$C := \frac{1}{2}||y - \hat{y}||^2$$

Where y is the prediction of the network for an input x, and $\hat{y}$ is the correct output. EP tries to minimize the following total energy function:

$$F := E + \beta C$$

where $\beta \in \mathbb{R}, \beta \geq 0$.

As described in [2], each iteration of the algorithm consists of the following three steps:

1) Take a data point v, feed it into the network and find a local minimum of F with $\beta = 0$ (the free fixed point, $s^0_{\theta,v}$). Collect $\frac{\partial F}{\partial \theta}(\theta, v, 0, s^0_{\theta,v})$.
2) Run the "nudged" phase by setting the input to the same v and finding a local minimum of F with $\beta \neq 0$, $|\beta|$ should be "small". At this "nudged" fixed point ($s^\beta_{\theta,v}$) collect $\frac{\partial F}{\partial \theta}(\theta, v, \beta, s^\beta_{\theta,v})$.
3) Update θ according to:

$$\Delta\theta \propto -\frac{1}{\beta}\left(\frac{\partial F}{\partial \theta}\left(\theta, v, \beta, s^\beta_{\theta,v}\right) - \frac{\partial F}{\partial \theta}\left(\theta, v, 0, s^0_{\theta,v}\right)\right)$$

B. EqPropNEAT

Our method substitutes BP with equilibrium propagation. During the mutation stage of each generation, instead of assigning random weights to some connections, we employ equilibrium propagation iterations in order to adjust edge weights.

As the network's complexity increases with more and more generations, the search space for weights increases rapidly. In order to get the most out of each network architecture proposed by the NEAT algorithm, we suggest adjusting the number of equilibrium propagation steps run between generations according to network complexity. Note that this means that the number of iterations varies even between species of the same generation.

During the evolution process, networks only change gradually between each generation. This allows us to split the weight learning process between generations, so that the number of EP iterations does not have to be enough for convergence as the training process will continue in the next generation. Too many EP iterations could also cause overfitting which limits the network's ability to generalize, It even might limit the adaptability of species in future generations.

We have tested a number of different formulae for determining the number of necessary simulation iterations:

$$N_1 = V \tag{1}$$
$$N_2 = V + E \tag{2}$$
$$N_3 = V \cdot E \tag{3}$$

Where V and E denote the number of nodes and edges respectively. Figure 1 shows a comparison in a classification task.

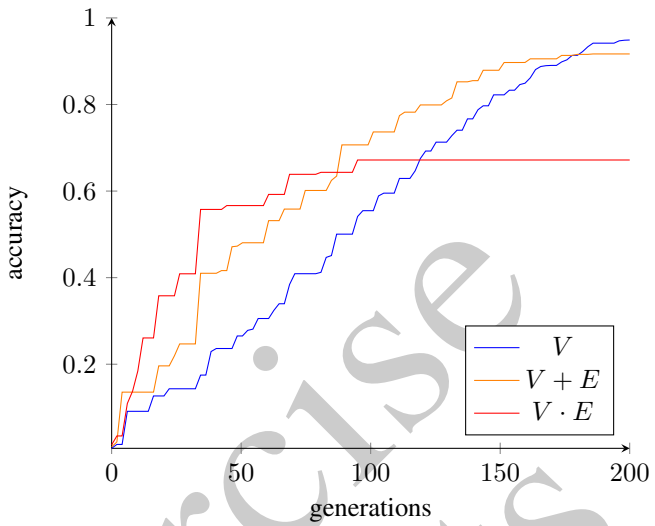

Fig. 1. A comparison of the accuracy of networks with different formulae determining the number of EP iterations per generation. The task used for testing was the classification of Iris flowers into three classes: setosa, versicolor and virginica.

Note that more EP iterations do not translate directly to better accuracy. On one hand, initially networks learn faster as more time is spent on optimizing connection weights per generation. However, after some generations the simple newtork structures become an obstacle for further improvement. As mutations increase structural complexity the potential of a given network rises and with lots of EP iterations, close to optimal parameters are quickly reached over a few generations. On figure 1 this behaviour is clearly visible as short but steep segments on the $V \cdot E$ line.

C. Hardware acceleration

As mentioned in the introduction, EP was designed with effective hardware implementations in mind. It has been shown [7], that the EP training of purely analog neural networks is indeed, possible. The missing piece for accelerating most of the EqPropNEAT stages is the capability to dynamically create the analog neural network representations for the current population. This would allow the use of hardware implemented EP for high-speed training.

Fortunately, field-programmable analog arrays (FPAA) provide an off-the-shelf solution for reprogramming the interconnects between different analog devices, enabling the creation of hardware based analog neural networks [8] while our algorithm is running. However, implementing EP requires the ability to efficiently measure voltages in our network, as well as additional circuitry for digital operations such as computing the gradient. Field-programmable gate arrays (FPGA) are a much more popular digital counterpart to FPAAs. For our use case, a mixture of both systems is needed. While such devices have been already proposed in technical literature [9], to the best of our knowledge, they have never been manufactured in a meaningful quantity yet. In theory, a mostly hardware accelerated implementation of our algorithm will be possible,

Writing Exercise
Fictive Results

once the above mentioned hardware becomes widely available. Figure 2 shows an outline of this algorithm.

Although at first glance the proposed hardware acceleration seems to improve runtimes significantly, we must mention a couple of potential performance limiting bottlenecks. Genetic algorithms in general require a population with thousands of genoms to provide enough diversity. Without this the algorithm does not work, or at least needs a lot more generations to arrive at a solution. It is also even more susceptible to overfitting. However with thousands of neural networks, with each of them potentially having at least hundreds of nodes, the required circuitry becomes problematically large. There is a high chance that even if the required hardware becomes available, it probably will only be able to contain a fraction of the population at a time. Of course this problem can be somewhat alleviated by splitting the population into groups which are evaluated and trained in separate passes (yielding slower runtime), however this way the number of occasions when the FPMA hardware is reconfigured with the new networks grows. In the case of modern day FPGAs the time it takes to reconfigure the entire board is usually in the realm of seconds, although by sacrificing some parts of the board for as a cache, this can be dramatically improved [10]. In any case it is best to avoid reconfiguration as much as possible. At the time of writing this paper it is impossible to tell how much of a limiting factor this will be.

Another potential issue is the lifetime of the hypothetic hardware as many frequent reconfigurations might degrade the board significantly. Runtime partial reconfiguration can also be employed to correct appearing hardware errors as the board ages [11]. Unfortunately this comes at a cost of reduced usable board area and as a result, a smaller limit on the number of neural networks we can evaluate in parallel.

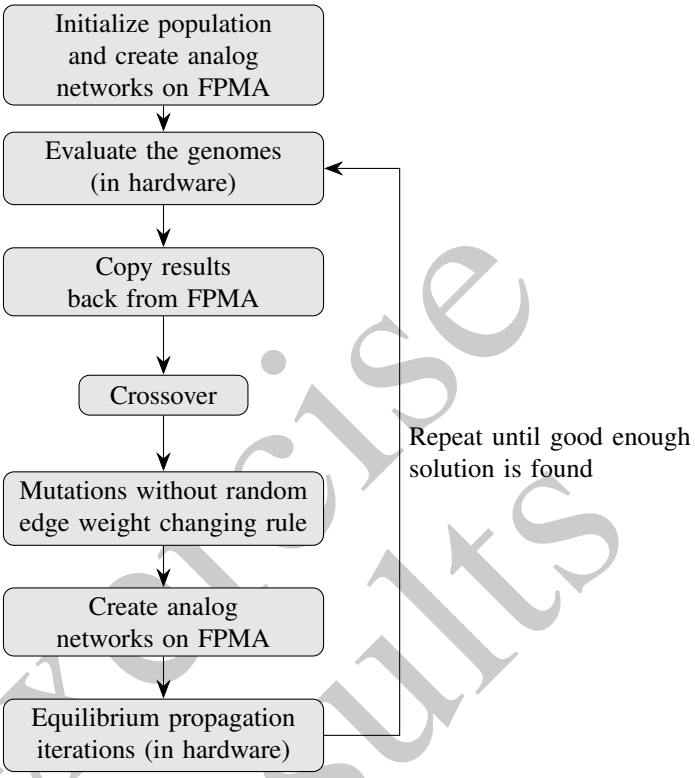

Fig. 2. An outline of the hardware accelerated EqPropNEAT algorithm. Note that the evaluation and weight training of genomes is entirely accomplished on the hardware level.

D. Biologically plausible algorithm

One of the goals of this study is the creation of a biologically plausible neural network model. For the generation of the network we went with a well tested solution, namely: NEAT. For the fine tuning of weights the obvious solution would have been the BP which offers great performance and is well tested and widely used to solve various task. However because of the aforementioned criteria of having a biologically plausible algorithm BP is off the table. For this reason we choose EP which utilizes a very similar concept with one key difference: it is considered biologically plausible. The fact that EP uses the same algorithm for both phases of training from one perspective made the hardware acceleration possible, but looking at it from another direction we can see that this also makes it a biologically plausible solution. Knowing this it's easy to see why BP's usage of different algorithms for these phases leads to the fact that it is considered biologically implausible. This way it's possible for the model to accurately represent the neural network of a living creature.

Of course this whole idea only makes sense if the performance remains at similar levels to those of conventional solutions. In general the performance of EP is subpar compared to BP due the unique restriction of only using one algorithm for both training phases. This problem can be mitigated by the aforementioned hardware acceleration technique. We will discuss the results further in the *Performance* chapter. As we have proven the performance can be a match or even an improvement compared to contemporary solutions thus we are left with only the benefits and no downsides. Benefits being the reduced complexity due to only one function being used for training and as a consequence of this the biological plausibility.

Considering the results and implications of EqPropNeat we should be able to accurately reconstruct a living creature's neural network and with the proper hyperparameters. In theory we would be able to create a neural network which would engage the neurons in the same place and in the same order as the biological network it was based on. This would get us as closer to understanding how the brain works and also would raise some ethical questions but that is beyond the scope of this paper and this is only in theory so far.

III. Results

A. Performance

We evaluated our method on several benchmarks, which are commonly used for comparison of neuroevolution techniques. Comparison was made with the NEAT implementation of the original author (K. Stanley) [12] and a self made implementation of the Learning-NEAT method [5] (as the original authors did not share their implementation). Tha latter is particularly important, since our method is mostly derived

Written Exercise Figure Results

	NEAT	EqPropNEAT	Learning-NEAT
Successful	3	10	12
Failed	17	10	8
Success rate	15%	50%	60%

Fig. 3. Success rates of the different NEAT derived techniques at the 2D pole balancing task. An attempt was deemed successful if the network managed to keep the pole balanced over 6000 simulation steps (10 seconds). All methods were run 20 times, the percentages tell the ratio of successful runs where the algorithm managed to train a network that could solve the problem.

	NEAT	EqPropNEAT	Learning-NEAT
Successful	3	10	12
Failed	17	10	8
Success rate	15%	50%	60%

Fig. 4. Success rates of the different NEAT derived techniques at the 2D pole balancing task. An attempt was deemed successful if the network managed to keep the pole balanced over 6000 simulation steps (10 seconds). All methods were run 20 times, the percentages tell the ratio of successful runs where the algorithm managed to train a network that could solve the problem.

from Learning-NEAT, by substituting backpropagation with equilibrium propagation.

One of the benchmarks was classic pole balancing. In this problem, the neural network has to keep a pole balanced for a given amount of time.

Another benchmark was the 2D pole balancing task [13]. This is a harder version of the classic pole balancing problem, in which the pole is balanced on top of a movable 2D plane, instead of a line segment. Figure 4 shows the success rates of the tested methods. Note that our method fares batter than the original NEAT algorithm, however it falls a bit short of Learning-NEAT. The reason for this is that due to the poor runtime of software EP , we had to use less EP iterations than ideal. For this task we used feed-forward neural networks, which is the reason why the results are worse then in [13] where recurrent networks were used.

IV. Discussion and conclusion

After conducting our experiments we can positively say that the proposed method performs up to expectations in the accuracy front and it can achieve similar performance to the contemporary algorithms. In terms of learning speeds of course it is currently behind the cutting edge without the specialized hardware. Despite this simulations indicate that with the aforementioned special hardware the EqPropNEAT algorithm can far outperform current rivals in terms of learning speed. This would present new possibilities for applications of AI. The speed of learning and predicting based on simulations could be close to real time, which would mean that with the proper hardware we could have individual devices learning separately without using a centralised dataset or an already created and weight adjusted neural network.

References

[1] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[2] B. Scellier and Y. Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation," *Frontiers in computational neuroscience*, vol. 11, p. 24, 2017.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[4] M. Y. Ibrahim, R. Sridhar, T. Geetha, and S. Deepika, "Advances in neuroevolution through augmenting topologies – a case study," in *2019 11th International Conference on Advanced Computing (ICoAC)*, 2019, pp. 111–116.

[5] L. Chen and D. Alahakoon, "Neuroevolution of augmenting topologies with learning for data classification," in *2006 International Conference on Information and Automation*, 2006, pp. 367–371.

[6] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial intelligence in the age of neural networks and brain computing*. Elsevier, 2019, pp. 293–312.

[7] J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier, "Training end-to-end analog neural networks with equilibrium propagation," *arXiv preprint arXiv:2006.01981*, 2020.

[8] P. Dong, G. Bilbro, and M.-Y. Chow, "Implementation of artificial neural network for real time applications using field programmable analog arrays," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, 2006, pp. 1518–1524.

[9] R. B. Wunderlich, F. Adil, and P. Hasler, "Floating gate-based field programmable mixed-signal array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 8, pp. 1496–1505, 2013.

[10] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 498–502.

[11] G. I. Alkady, N. A. El-Araby, M. B. Abdelhalim, H. H. Amer, and A. H. Madian, "Dynamic fault recovery using partial reconfiguration for highly reliable fpgas," in *2015 4th Mediterranean Conference on Embedded Computing (MECO)*, 2015, pp. 56–59.

[12] K. Stanley. (2010) NEAT implementation in c++. [Online]. Available: https://nn.cs.utexas.edu/?neat-c

[13] F. Gomez and R. Miikkulainen, "2-d pole balancing with recurrent evolutionary networks," in *International Conference on Artificial Neural Networks*. Springer, 1998, pp. 425–430.

Write Effective Results