The Evolution of Evolving Neural Networks

Elliot Arbuthnot, Hamiz Jamil, Edward Ng, Kevin Zhang

Abstract

In this paper, various neuroevolution algorithms that eventually led to the creation of CoDeepNEAT are compared and analyzed. This algorithm is built on the foundations of NEAT and aims to coevolve populations of blueprints and modules to search for the optimal architecture for a deep neural network. The other genetic algorithms that are reviewed are NEAT, and SANE that were early developments of neuroevolution and served as inspiration for CoDeepNEAT.

1. Introduction

Genetic algorithm (GA) is a method for solving optimization problems using natural selection by initializing a population, performing fitness evaluations, then selecting the desired individuals to undergo crossover and mutation. Over the years there have been many applications of GAs, ranging from optimizing routing, scheduling and parameter tuning. Another application of combining such algorithms with a neural network was proposed by John Holland in the 70s. However, around 20 years later, David Fogel introduced the idea of evolutionary programming, laying the foundations for evolving neural networks.

Fogel's work described the use of evolutionary programming, such as GAs, to tackle finite state machine problems for predicting environments. His work proved that using these algorithms, more complex and dynamic problems can be optimized (Fogel, 1998). In 1997, Moriarty evolved a population of individual neurons, which are connected in input and output layers of a neural network (Moriarty et al. 1996). It was then proposed by Stanley and Miikkulainen in 2002, the idea of evolving a population of simple artificial neural networks using GAs by updating their topology and weights (Stanley et al. 2002). Finally, in 2019, Miikkulainen and his team came up with an extension through a coevolutionary process of evolving populations of blueprints and modules that can be assembled as a neural network architecture. This algorithm, CoDeepNEAT, would allow evolution of deeper and more complex neural networks (Miikkulainen et al. 2019).

The paper goes into detail about each stage described above, walking through the history of various neuroevolutionary methods. These methods ultimately set the stage for evolving deep neural networks through CoDeepNEAT.

2. Evolving Artificial Intelligence

This paper aims to describe the development of each of the three procedures in simulated evolution over the past 35 years: genetic algorithms, evolution strategies, and evolutionary programming. The evolutionary process to be discussed can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results, allowing a far wider domain of problems to be tackled with artificial intelligence (Fogel, 1998).

Genetic algorithms are systems produced to simulate genetic systems. The problem to be addressed is defined in an objective function describing the fitness of a potential solution. After this objective function is defined, a population of candidate solutions is initialized as a *chromosome* given the specified constraints for the problem (Fogel, 1998). These chromosomes are decoded in a designated form appropriate for evaluation and assigned fitness scores and a probability of reproduction, which is used to generate a new population of chromosomes by selecting strings from the current population through crossover and mutation. The process is complete if a suitable solution is found or if the available computing time has been exceeded (Fogel, 1998).

On the other hand, evolution strategies and evolutionary programming is an alternative approach to simulating evolution, they emphasize the behavioural link between parents and offspring (reproductive populations), rather than the genetic link as seen by genetic algorithms. The problem is defined as finding the real-valued n-dimension vector that is associated with the functional representation of the problem, without loss of generality. An initial population of parent vectors is selected from a feasible range in each dimension with (typically) a uniform distribution of initial trials (Fogel, 1998). An offspring vector is created from each parent using a Gaussian random variable, and selection is used to determine which of these vectors to maintain by comparing error function outputs. The vectors with the least error become new parents for the next generation, and this process continues until a sufficient solution is reached or available computing time has expired (Fogel, 1998).

While genetic algorithms simulate models of genetic operators found in nature through crossover, mutation, and inversion, evolutionary programming focuses on mutational transformations that emphasize linkage between parents and offspring (Fogel, 1998). No model here can be a complete description of a true system, however, they have been demonstrated to be of practical use when applied to difficult optimization problems where heuristic solutions are not readily available. This makes the use of evolution in artificial intelligence a robust and efficient problem-solving technique, with a lot of room to grow in the future.

3. SANE

Symbiotic Adaptive Neuro-Evolution (SANE) is a reinforcement learning method which evolves a population of neurons through genetic algorithms to form a neural network capable of performing a task. Using symbiotic evolution discourages the convergence to suboptimal solutions by promoting both cooperation and specialization, resulting in a fast and efficient genetic search (Moriarty et al. 1996).

Reinforcement learning methods require less a priori knowledge than supervised learning techniques, which means they undergo extensive training and require cost-intensive CPU time (Moriarty et al. 1996). This has limited reinforcement learning to laboratory-scale problems, and this limited scale is what this paper aims to solve. SANE is a neuro-evolution system that forms effective neural networks quickly in domains with sparse reinforcement, through symbiotic evolution, where each individual in the population represents only a partial solution to the problem (Moriarty et al. 1996). This method evolves individual neurons to form complete neural networks. This allows SANE to find solutions faster than previous reinforcement learning methods, as the population remains diverse and the genetic algorithm can search different areas of the solution space in parallel (Moriarty et al. 1996).

In SANE, the population consists of individual neurons (partial solutions), and full solutions (complete neural networks). Since single neurons rely on others to achieve high fitness levels, they must maintain a symbiotic relationship. The fitness for an individual is calculated by summing the fitness values of all possible combinations of that particular individual with all other partial solutions in the population, and dividing by the total number of combinations (Moriarty et al. 1996). This results in a fitness value which reflects the average fitness of all full solutions formed by each of the individuals, without having to form these complete solutions through evolution prior to. To give a concrete example of this in practice, we take the task of animal classification for example. Partial solutions specialize towards one aspect of the task, so one specialization may learn to recognize a mammal, whereas another recognizes a reptile. As individuals they perform no meaningful task, but when combined to form a complete solution, they produce an effective animal classification solution (Moriarty et al. 1996).

The implementation of SANE requires evolving a population of hidden neurons for a given type of architecture (such as a 2-layer feed-forward network), until each neuron has participated in a sufficient number of networks. The average fitness of each neuron is computed, and the selection is based on neurons with a high average fitness that have cooperated well with other neurons in the population (Moriarty et al. 1996). Crossover operations are used to combine the chromosomes of the best-performing neurons, and mutation introduces genetic materials that may have been missing from the initial population. Where the mutation component of SANE sets itself apart is by not being used to create diversity, rather, only used as an insurance policy against missing genetic material (Moriarty et al. 1996).

To effectively evaluate SANE against previously state-of-the-art reinforcement learning algorithms, this paper compares its performance in the inverted pendulum (or pole-balancing) problem to state-of-the-art methods such as the single & two-layer Adaptive Heurisitc Critic, the Q-learning method and the GENITOR neuro-evolution system. In the inverted pendulum problem, SANE formed effective networks 9 to 16 times faster than the Adaptive Heurisitc Critic and 2 times faster than Q-learning and the GENITOR neuro-evolution approach without loss of generalization (Moriarty et al. 1996).

Overall, SANE outperforms previous state-of-the-art reinforcement learning methods in a highly complex reinforcement learning problem domain. While there is still room to improve by testing this algorithm against other reputable problems in the realm of reinforcement learning, the initial discovery of SANE has provided ample evidence of the effectiveness of evolution in reinforcement learning.

4. NEAT

Evolving topologies and weights was introduced by Stanley and Miikkulainen in their method of NeuroEvolution of Augmenting Topologies (NEAT). This algorithm beat the baseline fixed topology methods on a reinforcement learning task at the time. This was attributed to the higher efficiency due to crossing over topologies, while preserving diversity using speciation, and increasing the search space from a minimal structure . This was done using the principle of genetic algorithms to optimize and find complex solutions through evolving a population over many generations (Stanley et al. 2002).

In order to implement an evolving neural network, minimal solutions should be evolved to reduce the number of parameters that are searched. Therefore, the initial population is more deterministic and does not contain random topologies due to unnecessary nodes and connections (Stanley et al. 2002). To determine minimal topologies so larger networks do not dominate the population, the fitness function must incorporate the size of the network. However, the chosen method is to simply initialize a population with no hidden nodes, such that it evolves to only gain solutions that are beneficial. This will minimize the intermediate solutions along with the final solution and minimizes the search space to greatly increase efficiency (Stanley et al. 2002).

Genomes are then linear representations of network connectivity, with a connection gene that shows the relation between two node genes. The node genes can vary from inputs, hidden nodes, and output. Mutation can change the weight of the connection or network structure through two methods; either a connection is added between two existing nodes or a node is added to an existing connection (Stanley et al. 2002). These two methods enable the network to grow while keeping speciation, sometimes leading to unbounded sizes of genomes that could mismatch in structure during crossover. This problem would be solved with a historical marking represented as a global innovation number. Only individuals of matching innovation numbers can be selected together to crossover connection genes or node genes (Stanley et al. 2002).

The network structure is then evaluated with a Double Pole Balancing method, with evidence that NEAT can evolve structures when needed and efficiently at this task (Stanley et al. 2002). In Figure 1, the results of different NEAT ablations are used, with the full NEAT algorithm explained above giving the greatest success and performance. It is also noted that diversity is preserved during the experiments and solutions of low fitness have survived, meaning winning species will not overtake the population. In conclusion, NEAT progresses the evolution of neural networks by creating an efficient way of evolving weights and topology to find the optimal solution.

Method	Evaluations	FailureRate
No-Growth NEAT(Fixed-Topologies)	30,239	80%
Non-speciated NEAT	25,600	25%
Initial Random NEAT	23,033	5%
Nonmating NEAT	5,557	0
Full NEAT	3,600	0

Figure 1. Different variations of NEAT, known as ablations.

5. CoDeepNEAT

The culmination of the previous papers have led to the creation of CoDeepNEAT, a method to explore different deep learning architectures to find optimal configurations of both module layers and architecture blueprints through coevolution (Miikkulainen et al., 2019). Historically, the challenge of optimizing deep neural networks has been limited due to computational demands. As progress in computational power and availability continued to develop, Miikkulainen et al. created this systematic approach to evolving deep neural networks.

To begin this approach, NEAT was first extended to include evolution of both topologies and hyperparameters in the method known as DeepNEAT. The process of DeepNEAT mimics that of NEAT through mutation adding edges to nodes and crossover combining chromosomes, but differs through the introduction of each node of a chromosome representing a layer of a deep neural network as opposed to a single neuron (Miikkulainen et al., 2019). These nodes contain tables of hyperparameters which determine the type of layer represented by the node, including convolutional, fully connected, and recurrent layers (Miikkulainen et al., 2019). Additionally, the hyperparameters determine the properties of the layer including number of neurons, kernel size and activation function (Miikkulainen et al., 2019). These hyperparameters are mutated using uniform Gaussian distribution for real hyperparameters and random bit-flipping for binary-valued hyperparameters (Miikkulainen et al., 2019). From here, each layer is connected to another layer to construct a deep neural network in which fitness evaluation occurs by training the network for a number of epochs, and then evaluated based on its performance (Miikkulainen et al., 2019).

This approach was the first step to evolving deep neural networks, but caused overly complex and unprincipled networks. To help remedy this problem, the introduction of modules and blueprint evolution was used, and additionally incorporates principles from some of the more successful networks by repeating modules multiple times while maintaining a complex structure (Miikkulainen et al., 2019). This is what CoDeepNEAT is and it took its algorithmic inspiration from SANE by Moriarty and Miikkulainen. Populations for modules and blueprints are evolved similarly to DeepNEAT, with the blueprint chromosome representing a graph of nodes pointing to a module species and the module chromosome representing a small deep neural network (Miikkulainen et al., 2019). For fitness evaluation, the combination of a blueprint and modules is done with blueprint nodes being assigned a random species, with a corresponding module graph that is substituted in to create a full network (Miikkulainen et al., 2019). Evaluation occurs similarly to DeepNEAT, but fitnesses are calculated using the average fitness of all networks the blueprint/module was involved in. Interestingly, due to the combinatorial nature of CoDeepNEAT, small mutations to individual modules and blueprints can drastically change the structure of a network, allowing for diverse and deep architectures (Miikkulainen et al., 2019).

CoDeepNEAT was tested on the CIFAR-10 benchmark, and was found to have an error of 7.3%, compared to a benchmark of 6.4%, but with a much faster convergence of 120 epochs compared to 200 (Miikkulainen et al., 2019). The network was found to utilize multiple of the same modules in a row similar to many state-of-the-art deep neural networks (Miikkulainen et al., 2019). CoDeepNEAT was further tested on evolving LSTM architectures, with an extension on the model allowing for mutations that allow for more repetition of LSTM module connectivity. CoDeepNEAT attempted to search for new LSTM units while finding optimal connectivity across multiple layers. The model would generate a population of network graphs made from LSTM modules which allow for skip connections (Miikkulainen et al., 2019). The benchmark this was tested against was language modelling, in which CoDeepNEAT was tested using the Penn Tree Bank dataset in which the best found network improved performance by 5% compared to the vanilla LSTM used for the Penn Tree Bank dataset (Miikkulainen et al., 2019).

A final test was performed using CoDeepNEAT to develop a deep neural network which performed image captioning using the MSCOCO image-captioning dataset (Miikkulainen et al., 2019). CoDeepNEAT evolved the network to contain fully connected layers, LSTM layers, sum layers, concatenation layers, and specific hyperparameters for both the global architecture and individual layers (Miikkulainen et al., 2019). The fitness metrics used to evaluate the performance were the mean of BLEU, METEOR, and CIDEr, as there is no single best metric for caption evaluation (Miikkulainen et al., 2019). The evaluation was capable of exceeding the baseline values in all three metrics, as well as achieving a manual evaluation of 63% mostly correct accuracy on captioning iconic images from outside of the dataset (Miikkulainen et al., 2019).

Overall, CoDeepNEAT utilizes the increase in computing power to achieve comparable performance for its created deep neural networks which can be used in a number of different fields, and it can generally do so at a much faster convergence than the baseline models (Miikkulainen et al., 2019). As computing power continues to increase, CoDeepNEAT will look to create deeper and more robust architectures and topologies for the deep neural networks of the future.

6. Comparison and Analysis

CoDeepNEAT evolves on both NEAT and SANE by allowing deeper networks and the reuse of different species of modules in the same network. While NEAT attempts to evolve a whole network, SANE and CoDeepNEAT evolve portions of the network, with each section collaborating together to improve individual fitness. DeepNEAT is designed to add additional hidden layers to the network, which CoDeepNEAT builds upon by making these layers more modular to allow the repetition of basic components (Miikkulainen et al., 2019).

While SANE and CoDeepNEAT both involve evolving sections of the neural network, an important distinction is how the sections are pieced together. In SANE, a fixed number of neurons are incorporated into a full solution. In the implementation by the researchers, the nodes would only be utilized in a single hidden layer. The number of nodes is a hyperparameter that would be tuned by the researcher. CoDeepNEAT gets around this by evolving a population of blueprints also using DeepNEAT, which describes the flow of the neurons and how different modules are connected (Miikkulainen et al., 2019).

Fitness in NEAT is shared by similar individuals leading to individuals in the same niche competing. Fitness in SANE and CoDeepNEAT are cooperative, with each component being given a fitness that is the average of all full solutions that the component is involved in. For CoDeepNEAT, this applies to both blueprints and module (Miikkulainen et al., 2019)s.

For crossover, NEAT and CoDeepNEAT both utilize a version of crossover that protects innovation, whereas SANE utilizes a simple one-point crossover. Both are suitable for their uses. NEAT and CoDeepNEAT evolve both the connections, and nodes, and their crossover method helps protect innovations whereas SANE does not need the same since the number of nodes is fixed. CoDeepNEAT diverges from SANE as each individual is not forced into a single hidden layer, and the output of the previous component would affect the input of the next component in the blueprint.

Mutation is significant in NEAT and CoDeepNEAT as it can generate new structures not seen in the population, whereas mutation in SANE is designed to reintroduce genetic material. In NEAT and CoDeepNEAT, mutations can alter the structure through either adding a connection or adding an intermediate node. Adding a node is designed to minimally disturb the network, with the new connection leading into the neuron having a weight of one and the new connection leading out of the connection having the old weight (Stanley et al., 2002).

7. Conclusion

What started out as a concept of evolving a population to find optimal solutions based on genetics, grew to solving programming problems by evolving a population of neural networks. In a span of 50 years, researchers have adapted multiple iterations of evolving neural networks starting from the neurons, to evolving the weights, to the topology itself. It has been seen as a cutting edge invention, beating baseline models through neuroevolution, and has culminated into a coevolutionary process to handle deep neural networks. This advancement has proven to show potential of genetic algorithms in the complex field of Deep Learning.

References

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., & Hodjat, B. (2019). Evolving Deep Neural Networks. *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, 293–312. https://doi.org/10.1016/b978-0-12-815480-9.00015-3

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

Fogel, D. (1998). Artificial Intelligence through Simulated Evolution. 227-296. https://doi.org/10.1109/9780470544600.ch7

Moriarty, D. E., & Miikkulainen, R. (1996). Efficient Reinforcement Learning through symbiotic evolution. Machine Learning, 22(1–3), 11–32. https://doi.org/10.1007/bf00114722