Robotics Report Draft

Sam Wheeler

March 2020

1 My sections

1.0.0.1 The Keras Python Library

Sam Wheeler

Developed by Google engineer François Chollet, Keras is a Python API for creating and manipulating neural networks. Rather than acting as a standalone platform, Keras was designed to utilise other powerful machine learning libraries with a more generalised and intuitive interface[1], namely (at the time of writing): Microsoft's CNTK[2], Theano by the Montreal Institute for Learning Algorithms[3] and Google's TensorFlow[4]. TensorFlow is perhaps the most recognisable of these platforms as it has been adopted by many high profile artificial intelligence research organisations, such as Alphabet's DeepMind, since its initial release in 2015. Keras has since become TensorFlow's default API.

TensorFlow provides a low-level system for processing dataflow computation graphs, or 'models', optimised for use in machine learning research. Each computation graph consists of 'nodes' and 'tensors' (vectors between nodes along which data can flow). Nodes are representations of a particular operation, in the form of an abstract mathematical computation. Additionally, there are 'control dependency' node connections which simply prevent the destination node from 'firing' until the source node has completed its operation. This basic structure for performing graph computations has been implemented such that it is generalised to a variety of different platforms, meaning that it is designed to run on several different operating systems and in a wide range of hardware configurations; TensorFlow models can therefore be written and constructed on, for example, a system with a single GPU, but can then perform or be trained on a different system with potentially many more processing cores far and greater computing power. As a result of its generalisation and optimisation for machine learning applications, TensorFlow has been adopted rapidly both in industry and in the research community[5].

In our development of neuroevolutionary learning algorithms we used the 'Sequential' model type. This is the most basic type of neural network that Keras offers, consisting of input and output layers with an arbitrary number of intermediary 'hidden' layers; these layers are densely connected, meaning that each node is connected to every node that resides in an adjacent layer. Using this model type, we were able to create a proof of concept 'simple evolutionary' algorithm for evolving neural networks. This uses an asexual mutation process capable of altering connection weights, adding or deleting layers and increasing or decreasing the number of nodes in each layer. Combined with a simple function to select the best performing networks by fitness, this algorithm was able to solve the CartPole Gym environment by OpenAI.

In the case of more complex neuroevolutionary methods however, such as GNARL and NEAT, the Sequential model type is insufficient. Both of these algorithms require a more abstract network structure, which is irreducible to discrete layers; they are also based on 'sparse' rather than dense networks (nodes are not necessarily all interconnected) since one of the main mutation types is to link two previously unconnected nodes. In order to cater for this, therefore, it was necessary to write a higher level API running on top of Keras to provide a more usable interface. This replaces the standard Sequential interface, which is based around the defining of entire layers, with a set of classes and functions to allow for the creation of individual nodes and connections - all of which are stored as attributes of a central object which contains all of the information needed for a network. The resulting set of nodes and connections is then compiled into a usable, Sequential Keras network. This essentially consists of adding nodes to bridge non-adjacent layers and using connections with weights of zero to simulate absent links, as shown in Figure 1.0a.

1.0.1 NEAT

Sam Wheeler

[1] Chollet et al., *Keras*.
[2] Microsoft, *CNTK*.
[3] MILA, *Theano*.
[4] GoogleBrain, *TensorFlow*.
[5] Martín Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*.

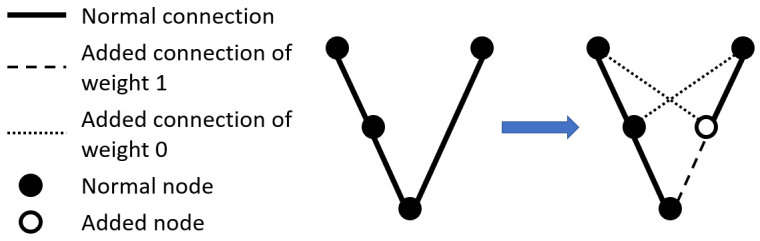

Figure 1.0a: Visualisation of the process used to compile a sparse network into a Keras Sequential computation graph. The left network represents an example computation graph produced by our custom interface, while the right is the resulting Sequential model following compilation.

The NEAT (NeuroEvolution of Augmented Topologies) method for evolving neural networks was first proposed in 2002 by Stanley and Miikkulainen[6]. Unlike many of the neuroevolutionary methods developed contemporaneously to NEAT, a crossover mechanism is utilised for the evolution of network populations in place of the simpler 'asexual' reproduction favoured in, for example, GNARL.

Variants of this original approach have proven successful in a multitude of virtual environments, primarily in the development of artificial intelligence for use in games. Examples include the controlling of a simulated car in The Open Car Racing Simulator (TORCS)[7], starting from a position of zero a priori knowledge, as well as the real time learning demonstrated in the robot combat game NERO[8] (a later project of Stanley and Miikkulainen). Expansions to the base algorithm have also been used to extend the functionality of NEAT machine learning[9], optimising it for finding solutions in more complex environments that may require fractured strategic decisions (in which the optimal action varies discontinuously as a function of state[10]).

These additional techniques applied to NEAT for use in highly fractured action spaces appear to be of little utility to this project, however. This is because the swinging motion is essentially a simple optimisation problem which depends only on the deterministic physics of the pendulum and damping forces - therefore, the set of actions taken by the robot need only be reactive to the current state. For this reason the only NEAT based approach explored during this project is the one described in the original publication, although it may be interesting to pursue some of the more complicated extensions to this base algorithm for comparison in future projects.

1.0.1.1 Theory

The formalism through which NEAT is described in its initial proposal is deliberately borrowed from biology, since the method is heavily informed by natural selection processes observed in nature. In this formalism, connections between nodes in a neural network are the 'genes' attributed to a particular 'genome' – where a genome contains all of the information necessary to describe a full network. This is an example of a genetic encoding system. Although nodes are referred to as 'node genes' by Stanley and Miikkulainen, the objects referenced here are not actually specific to an individual genome; rather, the node genes are stored in a list which is common to all genomes created by the NEAT process, thus providing information about which are input or output nodes and which can be connected in a network. In each genome, a given gene can additionally be marked as 'deactivated', meaning that it is not represented in the neural network but remains part of the genetic information.

This encoding is necessary for the crossover mechanism to function. During the crossover process, through

[6]Kenneth O. Stanley and Risto Miikkulainen, "Evolving Neural Networks through Augmenting Topologies".
[7]Cardamone, Loiacono, and Lanzi, "Learning to Drive in the Open Racing Car Simulator Using Online Neuroevolution".
[8]K. O. Stanley, Bryant, and R. Miikkulainen, "Real-time neuroevolution in the NERO video game".
[9]Whiteson, Taylor, and Stone, "Empirical Studies in Action Selection with Reinforcement Learning".
[10]Kohl and Risto Miikkulainen, "2009 Special Issue: Evolving Neural Networks for Strategic Decision-Making Problems".

which novel genomes for a new population are generated by combining the genetic information of two 'parents', the parent genomes must be 'lined up' to identify which of their genes are matching (note that this requires the encoding system to be linear in nature). To identify matching genes - in other words, genes which reference connection vectors between the same two nodes - NEAT records the order in which new genes are created in the form of 'innovation numbers'. Each time a gene is created, a global innovation number is incremented and the new value is attributed to that gene. Hence, two genes which share the same innovation number are known to have the same historical origin and can be identified as matching. If a gene exists in either genome with no matching counterpart that gene is referred to as 'disjoint', or 'excess' if its innovation number exceeds the maximum innovation present in the other genome.

Once the genomes have been aligned using innovation numbers, the crossover process can occur. In the case of matching genes, a gene from one of the parents is selected at random to be added to the child genome; if the matching gene is deactivated in either of the parents, there is a predefined chance that it will also be deactivated in the child. For all non-matching genes, both disjoint and excess, the child genome inherits from the parent with the superior fitness value. An example of a crossover between two different genomes can be found in Figure 1.0b.

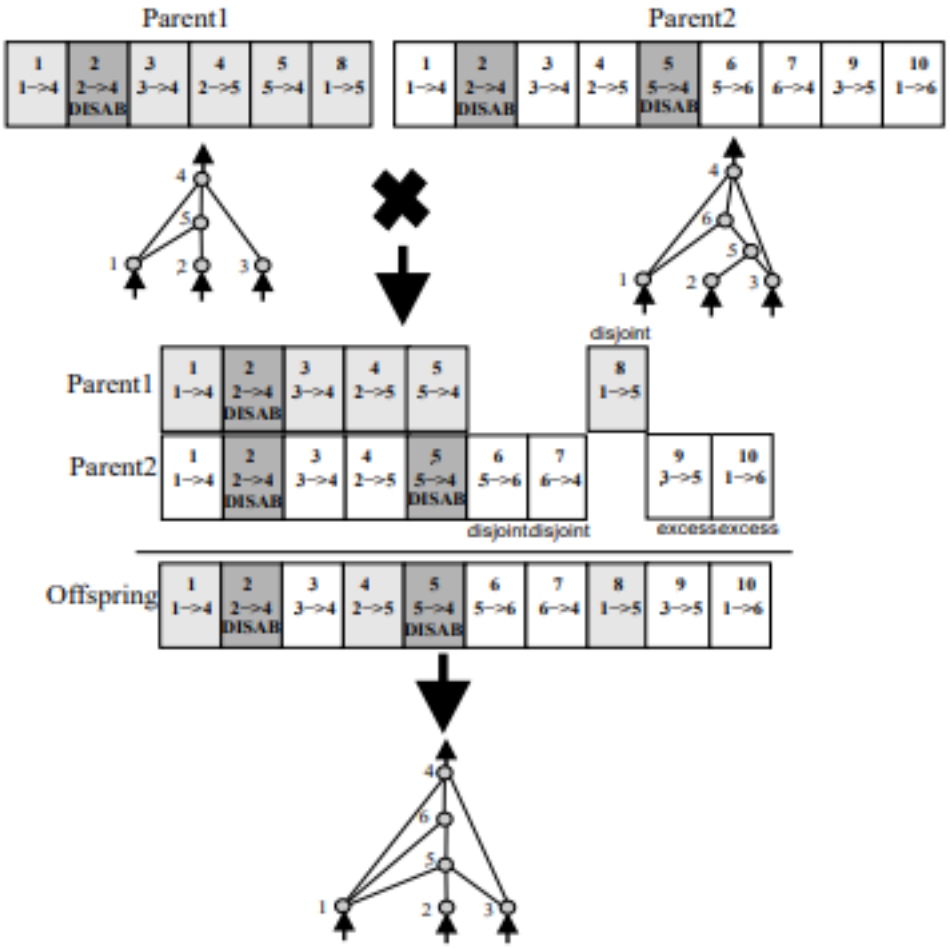

Figure 1.0b: Visualisation of the crossover process in NEAT[11].

[11]Kenneth O. Stanley and Risto Miikkulainen, "Evolving Neural Networks through Augmenting Topologies".

When all of the necessary genes have been added to the child genome, a mutation process is applied to introduce new genetic information to the gene pool. As in other TWEANNs, NEAT mutates network structure via the addition of nodes and connections as well as altering the weights of the connections themselves, each with a predetermined probability. New connections can be made between any previously unconnected nodes, while new nodes are created in place of existing connection genes. In the case of an 'add node' mutation, the old connection is effectively split in two with two new connection genes also created (the connection entering the new node is assigned a weight of 1 while the other is given the same weight as the old connection, thus preventing the new structure from immediately having a negative impact on the performance of the network). The old connection is kept in the genome but marked as deactivated. Finally, connection weights are typically altered by way of Gaussian perturbation so as not to completely randomise the weight and potentially compromise the fitness of the new network; the standard deviation of this Gaussian can be given as one of the predefined variables, although for our implementation it was randomised up to a maximum value of 0.5.

A potential issue with using a crossover algorithm is that many of the novel genomes will be the product of highly genetically diverse parents and, as a result, will have significantly impaired performance. The solution to this is to quantify this diversity - the compatibility distance, δ - between prospective parent genomes and to prevent crossover if the distance is greater than a given threshold value, δ_t. Calculating δ is simply a case of finding a linear combination of the numbers of excess and disjoint genes, as well as the average disparity in weight values for all matching connection genes, denoted E, D and $\overline{W}$ respectively, each with adjustable coefficients[12]:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}. \tag{1.0i}$$

Here, c_1, c_2 and c_3 are variable coefficients and N is the total number of genes in the larger genome.

If this diversity check is employed, the population is automatically 'speciated' into discrete populations which cannot be interbred. In order to improve the speed of the algorithm, NEAT tracks the species of each genome in its population with each genome assigned a species value (in a similar fashion to the system for tracking innovation numbers). Genomes are only allowed to breed if they are members of the same species, meaning that the compatibility distance need not be checked every time a crossover occurs. Each time a new genome is created at the start of a generation, the *delta* value is checked against the first member (or 'mascot') of each existing species until a compatible species is found; if there is no such species, a new species is created with the novel genome as its mascot. This process minimises the number of times that *delta* values must be calculated.

Speciation not only limits the number of impaired genomes produced by crossover, but also acts as protection against the premature deletion of a new topological innovation before its potential has been fully realised. It is unlikely that the original member of a new species represents the greatest possible optimisation of that particular topology, so allowing a species time to reproduce and improve ensures that as many evolutionary routes as possible are explored. An additional mechanism to further this safeguarding effect comes in the form of 'fitness sharing', whereby the fitness value of each neural network is normalised based on the size of its species. This prevents a single species from outcompeting the others and becoming over-represented in the population. The adjusted fitness value for a genome, f'_i, is calculated as follows:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{1.0ii}$$

where sh is a step function, equal to 0 if $\delta(i,j) > \delta_t$ and 1 otherwise. Each species present in the population is given a quota of offspring proportional to the sum of adjusted fitness values over all members. Parent genomes are then selected randomly, each weighted by their fitness (meaning a higher fitness gives a greater probability of selection) and breed via crossover with a member of the same species chosen in the same

[12]Kenneth O. Stanley and Risto Miikkulainen, "Evolving Neural Networks through Augmenting Topologies".

manner; this process is iterated for each species until their quota of offspring is fulfilled.

Conventionally, the initial population of neural networks in NEAT consists of densely connected input and output layers with no hidden layer nodes at all. This means that every input is connected to every output, each network with unique, randomised weights for every connection. Starting in this manner, in combination with incremental topology alterations which are only allowed to propagate if they provide some form of advantage over less complex structures, naturally leads NEAT solutions to tend towards topological minimality. This serves to increase the performance of NEAT, as compared with other TWEANNs, because the dimensionality of the search space is reduced.

1.0.1.2 Implementation and Performance

Our implementation of NEAT uses the custom API outlined in Section 1.0.0.1, which is built on top of the Sequential model type from the Keras machine learning library. One of the key aims for this implementation was to create a flexible function which could be applied to a wide range of different environments, in order to account for the full variety of benchmarks and training methods used in the project. Hence, our final product is generalised to any data size for both input and output and has a wide range of adjustable parameters, including population size, activation function and the probability weights assigned to each mutation type.

Unfortunately, due to time constraints, we were unable to fully implement speciation into the NEAT algorithm. As described above in Section 1.0.1.1, speciation or 'niching' is the method by which NEAT protects new topological innovations against being instantly outcompeted, giving them time to fully optimise through further crossovers and mutations. Stanley and Miikkulainen's analysis[13] indicates that the inclusion of speciation vastly improves the performance of NEAT in control optimisation tasks, with a non-speciated version of their algorithm taking 7 times longer on average to solve the double cartpole problem (similar to the OpenAI CartPole environment used in this project, but with increased difficulty due the additional balancing pole). The improvement in the speed at which speciated NEAT climbs in fitness score would doubtless be useful for a swinging environment, as it would make the training process more efficient and reduce the time taken to find an optimal neural network; hence, we suggest that pursuing a full NEAT implementation with included speciation could be a valuable path for future projects to take.

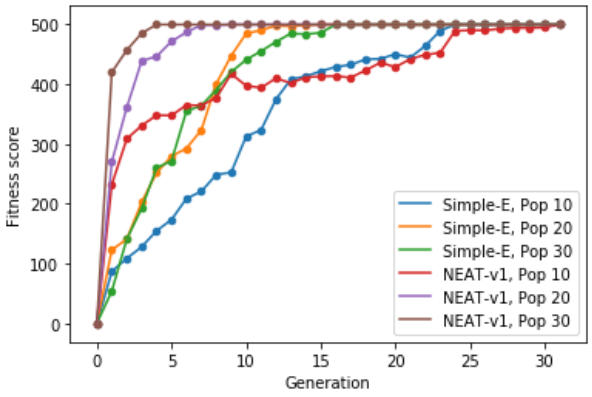

Figure 1.0c: Average generational scores for Simple Evolutionary and NEAT methods in CartPole, using various population sizes.

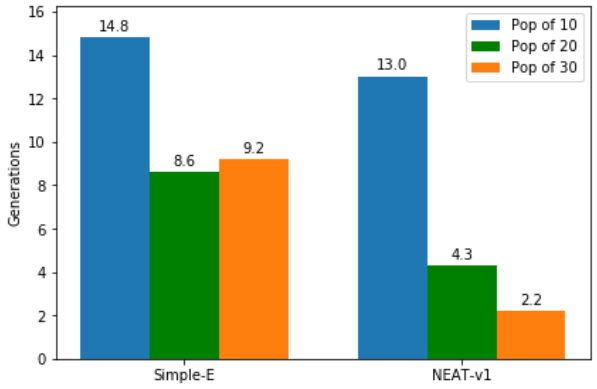

Figure 1.0d: Average number of generations taken for Simple Evolutionary and NEAT methods to solve CartPole.

To assess the performance of our NEAT algorithm we initially used the OpenAI CartPole environment, over a variety of different population sizes (see Figures 1.0c and 1.0d). The Simple Evolutionary algorithm described in Section 1.0.0.1 is used as a benchmark. Like the Simple Evolutionary method, the performance of NEAT seems highly dependent on the population size used. This is to be expected for a intrinsically

[13]Kenneth O. Stanley and Risto Miikkulainen, "Evolving Neural Networks through Augmenting Topologies".

probabilistic process, especially for a very simple control task such as CartPole, as larger population sizes give a greater probability of finding a solution in each generation. The simplicity of the environment used here also makes for a relatively high variance in, for example, the number of generations taken to reach a solution. In order to normalise this and gather meaningful data, it was therefore necessary to average each data point over a set number of training runs (in this case 10).

Overall, NEAT appears to be a significant improvement on the Simple Evolutionary method. Despite both algorithms taking more than 10 generations to solve CartPole with the smallest population size, the required number of generations drops much faster for NEAT as population size is increased. We can also see from Figure 1.0c that the average generational score rises more steeply and uniformly in NEAT populations of 20 and 30 than in their Simple Evolutionary counterparts. Rather than rising erratically in fitness value, which implies an inefficient process for mutating and improving the population members and therefore a tendency to get 'stuck' at a particular score, the NEAT curves at higher population sizes are smoother and more regular as they tend towards the maximum score of 500.

In addition to the improved performance of NEAT in the simple CartPole environment, as compared to the Simple Evolutionary algorithm, our NEAT implementation was able to solve the more complex 'BipedalWalker-v3' environment (also provided by OpenAI Gym[14]). This problem makes 24 data inputs available to the neural network, compared to the 4 provided in CartPole, meaning a much more complicated model is required to find a solution. Despite this, NEAT was able to solve BipedalWalker in approximately 100 generations with a population size of 50. This is evidence that NEAT can perform well at more challenging control problems, which would make it well suited to optimisation of swinging motion.

2 Advice for similar projects

2.1 Advice for Machine Learning Teams

Sam Wheeler

The NEAT method for evolving neural networks proved promising in its ability to optimise for a swinging motion, as well as in other complex control problems, despite lacking a full implementation (including speciation, see Section 1.0.1.2) in this project. It is therefore highly recommended that future projects of this nature pursue a fully functional NEAT algorithm, which is simply a case of adding a speciation function to our existing code, as well as exploring some of the many expansions to the base NEAT method outlined at the beginning of Section 1.0.1.

[14] *BipedalWalker-v2 Gym Environment.*