Deep Neuro-Evolution in TensorFlow Eager

Cristian Bodnar
University of Cambridge
cb2015@cam.ac.uk

Word Count: 2441

Abstract

Deep Neuroevolution is a class of Artificial Intelligence algorithms that evolve Artificial Neural Networks (ANNs) using genetic algorithms. Recent research has shown that Deep Neuroevolution algorithms are a competitive alternative to learning-based algorithms in Reinforcement Learning (RL) tasks. An important subclass of Neuroevolution are the algorithms that also evolve the topology of the networks such as NEAT, HyperNEAT, and Adaptive HyperNEAT. However, until recently, these algorithms could not be implemented in TensorFlow, whose static graphs were incompatible with the topological changes in the architecture. This paper introduces TensorFlow-NEAT, a library for the NEAT algorithm and its extensions, written in TensorFlow Eager. The performance of this library is benchmarked against the PyTorch-NEAT library from Uber Research on simple RL environments. Furthermore, starting from the comparison between the two neuroevolution libraries, a more general comparison between PyTorch and TensorFlow Eager is carried out.

1 Introduction

Neuroevolution algorithms are an essential class of algorithms in Machine Learning. Even though they have been shadowed in recent years by Deep Learning approaches, recent research has demonstrated that they can be competitive in many RL environments, including robotic tasks [Such et al., 2017] as well as Atari games [Wilson et al., 2018]. The interest in Neuroevolution algorithms has also increased recently due to novel attempts to combine evolution with standard Deep Reinforcement Learning algorithms such as Deep Deterministic Policy Gradients [Khadka and Tumer, 2018] or Proximal Policy Optimisation [Hämäläinen et al., 2018].

A special type of Neuroevolution algorithms are those that also evolve the topology of the networks, and not just the weights. The most popular such models are NEAT Stanley and Miikkulainen [2002] and its extensions HyperNEAT [Stanley, 2009] and Adaptive HyperNEAT [Risi and Stanley, 2010]. Nonetheless, implementing these algorithms efficiently in a scientific computing framework like TensorFlow, which is fundamentally based on static computation graphs, can be inconvenient and sometimes extremely inefficient. I believe this is the main reason an open-source mature TensorFlow library for NEAT and its variants does not exist. Given the momentum that is currently accumulating around neuroevolution, a library like this would be of great utility for the research community and other interested developers.

So far, PyTorch-NEAT from Uber Research has been the only NEAT library written in a scientific computing framework. This is because PyTorch, unlike TensorFlow, was designed with dynamic computation graphs in mind. In 2018, TensorFlow released an "eager" execution mode, equivalent to the PyTorch one, which supports the imperative execution of tensor

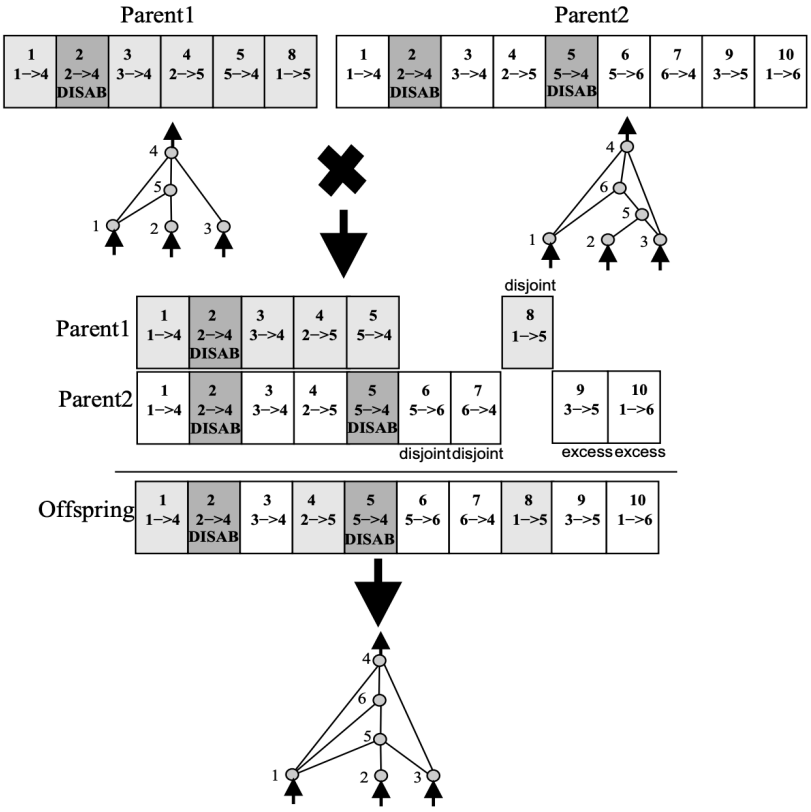


Figure 1: Crossover of the genomes of two parent networks (image from Stanley and Miikkulainen [2002]).

operations. This execution mode enables the implementation of dynamic models such as NEAT. This paper introduces the TensorFlow-NEAT library which uses eager execution.

The goal of this paper is twofold. First, it introduces and describes the TensorFlow-NEAT library. Second, it aims to be a more general comparison of PyTorch and TensorFlow Eager, but illustrated by detailed comparisons between Tensorflow-NEAT and PyTorch-NEAT. Section 2 gives an introduction to NEAT, HyperNEAT and Adaptive Hyper NEAT. Section 3 describes the library and Section 4 evaluates it in comparison with PyTorch-NEAT.

2 Background

This section gives a short introduction to Neuroevolution that is required before introducing the library.

2.1 Neuro-Evolution of Augmenting Topologies (NEAT)

Neuro-Evolution of Augmenting Topologies (NEAT) uses a direct genetic encoding. This means that every edge and weight is explicitly represented in the genome. The genome consists of an array of connections, where each connection is defined by the input node, the output node, the weight of the connection, an on/off bit that determines if the gene is expressed or not, and an innovation number, whose purpose will be explained shortly.

The weights of existent connections are modified by adding some noise to their values during the mutation phase (e.g. Gaussian noise $\mathcal{N}(0, 1)$). Mutations also modify the topology in two ways. The *add connection* mutation connects two existent nodes which are not connected and initialises the weight with a random value. The *add node* changes the topology by "splitting"

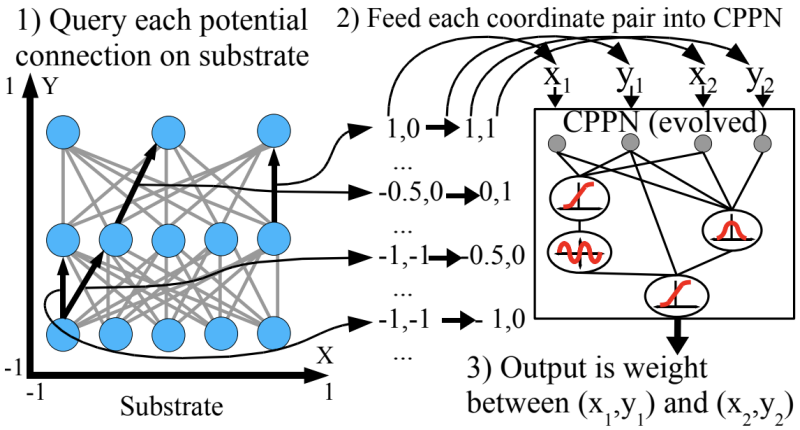


Figure 2: An ANN produced by HyperNEAT and its CPPN. The bold arrows are example of queried connections. The coordinates of these connections are given as input to the CPPN to determine the weight of that connection. The image is taken from Risi and Stanley [2010].

an existent edge into two edges and by adding an intermediary node in the middle. The old connection is deactivated, the weight from the input node to the new node is initialised to one, and the weight from the new node to the output node is initialised with the weight of the old connection. The reason the first edge is initialised to one is to prevent significant shocks in the network after a mutation.

The mutations just described determine an unbounded increase of the genomic array and the population of networks will contain individuals with genomes of different sizes. This is where the innovation number proves its purpose. When a novel gene appears, a global innovation number is incremented, this number is assigned to that gene, and it is never modified again. If the same gene (e.g. the same structural mutation) occurs during the same generation in more than one individual, then all these genes are assigned the same innovation number. During the crossover phase, the two genomes are aligned based on their innovation numbers. The matching genes (those with the same innovation number) are randomly inherited by the offspring from one of the parents, while the genes that do not have a corresponding matching gene (disjoint or excess genes in Figure 1) are copied from the parent who is the fittest.

A possible problem that can occur is that structural innovations might not perform well immediately after they occur. Often, they require some time to develop. The risk of such early innovations to be eliminated from the population due to low fitness exists. To avoid this problem, NEAT uses explicit fitness sharing. A linear distance function between the genomes of the form $\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W$ is used where N is the size of the larger genome, E is the number of excess genes, D is the number of disjoint genes, W is the average weight difference between matching genes and c_1, c_2, c_3 are adjustable coefficients. If the distance between two genomes is less than some threshold δ_t, then they are assigned to the same species. During reproduction, the fitness of an individual is divided by the number of individuals in the same spaces. Therefore, the competition is focused on individuals with similar topologies.

2.2 HyperNEAT

Because NEAT uses a direct encoding, it is unable to evolve large ANNs. Furthermore, it is unable to exploit the geometrical relationships between the inputs. HyperNEAT is an extension of NEAT that addresses these issues. It uses an indirect encoding in the form of a Compositional Pattern Producing Network (CPPN) [Stanley, 2007]. NEAT is used to evolve this CPPN. First, the nodes of the ANN that HyperNEAT will produce are placed in layers on a coordinate grid (also called the substrate). Then, all the possible connections between nodes on this grid are queried. Assuming a 2D grid, for each queried

connection $(x_1, y_1) \rightarrow (x_2, y_2)$, the CPPN produces an output value $CPPN(x_1, y_1, x_2, y_2)$ that determines the weight of that connection. The algorithm is also illustrated in Figure 2.

2.3 Adaptive HyperNEAT

Adaptive HyperNEAT is an extension of HyperNEAT that draws inspiration from neuroscience, by trying to model brain plasticity. The ANN in Adaptive HyperNEAT does not just evolve, but it also adapts during its lifetime. This is achieved by letting the CPPN produce different plasticity rules in different parts of the network. A learning rule specifies how the weights should change as a function of the presynaptic activity o_i, the postsynaptic activity o_j and the current weight w_{ij}:

$$\Delta w_{ij} = \phi(o_i, o_j, w_{ij}) \tag{1}$$

The exact form of the function ϕ depends on the chosen plasticity model. In the experiments from this paper, the most general model is used where the plasticity rule of a connection $(x_1, y_1) \rightarrow (x_2, y_2)$ could be any function evolved by NEAT via the CPPN:

$$\Delta w_{ij} = CPPN(x_1, y_1, x_2, y_2, o_i, o_j, w_{ij}) \tag{2}$$

3 TensorFlow-NEAT

The TensorFlow-NEAT library is a TensorFlow Eager equivalent of PyTorch-NEAT. There is an almost one to one mapping between them. The similar internal structure and API make comparisons between the two easy to perform. Both libraries use at the core Python-NEAT which handles the core algorithmic tasks of NEAT and its extensions. The Python-NEAT core has the following functions:

- Keeps track of the genomes in the population.
- Mutates the genomes.
- Performs crossovers between genomes.
- Computes the δ distances between genomes and splits the population into species.
- Evolves the next generation from the current one.
- Executes (custom) loggers and other statistics reporters which collect information about the population being evolved.

TensorFlow-NEAT uses the genomes computed by Python-NEAT and transforms them into dense tensors of weights. TensorFlow then executes the computationally heavy tensor operations for computing the output of the evolved network on the device specified by the user (CPU or GPU). In the case of HyperNEAT, TensorFlow executes both the computations of the CPPN and that of the evolved network.

3.1 Running and Interaction with OpenAI Gym

Like PyTorch-NEAT, the API of TensorFlow-NEAT easily allows the user to create a NEAT RecurrentNetwork and to attach it to an OpenAI gym environment. A MultiEnvEvaluator exists that allows each member in the population to be attached to a separate instance on the environment so that they can all interact with the environment in parallel. A sample code of how TensorFlow NEAT can be run on the CartPole environment is given below:

```
from tf_neat.multi_env_eval import MultiEnvEvaluator
from tf_neat.recurrent_net import RecurrentNet

def make_net(genome, config, batch_size):
    return RecurrentNet.create(genome, config, batch_size)

def activate_net(net, states):
    outputs = net.activate(states).numpy()
```

```
    return outputs[:, 0] > 0.5

def make_env():
    return gym.make("CartPole-v0")

evaluator = MultiEnvEvaluator(
    make_net, activate_net, make_env=make_env,
    max_env_steps=max_env_steps, batch_size=batch_size,
)
fitness = evaluator.eval_genome(genome)
```

4 Evaluation

In Subsection 4.1, the performance of PyTorch-NEAT and TensorFlow-NEAT is directly compared. In Subsection 4.2, PyTorch and TensorFlow are compared from a code perspective with illustrative examples from the two libraries.

4.1 Benchmarking

All the experiments in this subsection are run on a Linux machine with an 8 core Intel Core i7-6900K CPU 3.20GHz. No other processes were run in the background, except for the usual Linux processes.

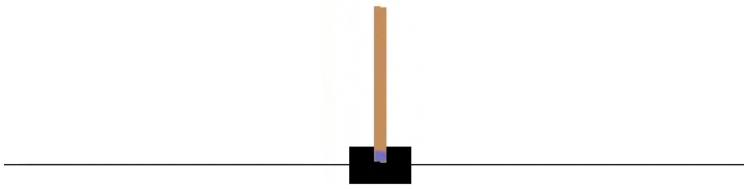
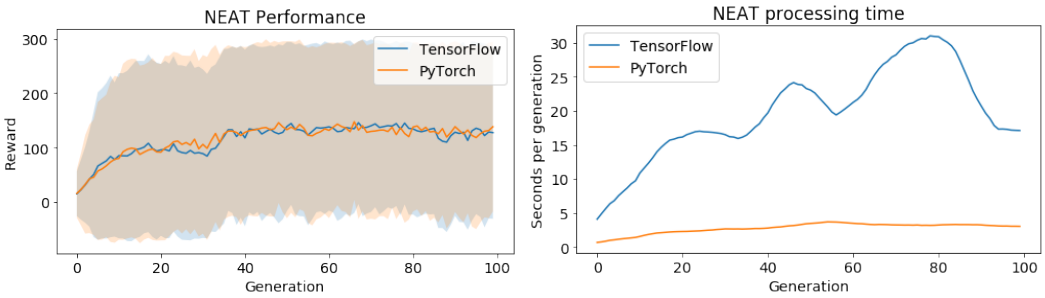


Figure 3: The CartPole-v0 environment from the OpenAI gym.

The first experiment evolves 100 generations of NEAT Recurrent Networks on the CartPole-v0 environment from the OpenAI gym [Brockman et al., 2016]. The experiment validates that TensorFlow-NEAT and PyTorch-NEAT networks have similar performance in the environments and compares the execution speed of the two. The Cart Pole environment consists of a pole attached through a un-actuated joint to a moving cart (Figure 3). The state of the environment consists of a vector of 4 numbers: the position of the cart, the velocity, the angle of the pole and the angular velocity. At each time step, the agent can take two actions: move the cart either to the left or to the right. The agent receives rewards as long as the pole remains balanced and the cart remains in a certain range. If the pole is more than 15 degrees from vertical or the cart moves too far from the centre, the episode ends.

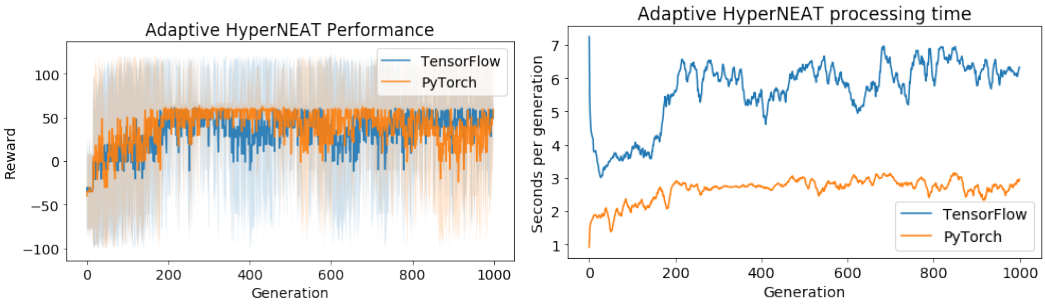
NEAT is run for 100 generations on CartPole-v0. Figure 4a plots the mean average reward of the population and the two standard deviations error bound around the mean. The two graphs are very similar, and this confirms that the two libraries have similar semantics in the case of NEAT. Figure 4b shows the evolution process of NEAT from a processing time perspective. The plot shows that TensorFlow Eager is significantly slower than PyTorch. To confirm that there is no bug in the implementation, the python program was profiled, and indeed the most significant amount of time was taken by Eager's internal execution module.

In the next experiment, the two libraries are compared on the Adaptive HyperNEAT algorithm, this time deployed in a T-Maze environment (Figure 6). T-Mazes are a typical experiment for testing the plasticity of ANNs. The T-Maze environment used in these experiments is discrete. The environment consists of two branches the agent could navigate



(a) NEAT population reward per generation. (b) NEAT processing time per generation.

Figure 4: NEAT Performance and benchmark.

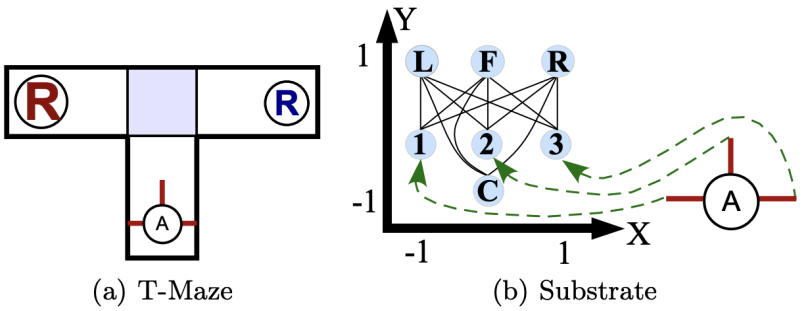


(a) Reward per generation. (b) Processing time per generation.

Figure 5: Adaptive HyperNEAT Performance and benchmark.

to by starting from its position at the bottom of the T. One branch, contains a high reward, while the other branch contains a low reward. During the deployment, the position of the rewards changes from time to time and the agent has to adapt accordingly and explore the other branch. The agent also has to remember where the reward was last time for future trials.

On this environment, Adaptive HyperNEAT is run for 1,000 generations. Again, to confirm that the two libraries have similar semantics, Figure 5a plots the average reward of the population and the 95% confidence bound around the mean. As it can be seen, the graphs for the two libraries are very similar. The periodic reward decreases correspond to changes of the reward from one side of the T-Maze to another. From a processing time perspective, Figure 5b shows that TensorFlow Eager is still slower, but this time by a smaller factor.



(a) T-Maze (b) Substrate

Figure 6: T-Maze environment and the corresponding substrate of Adaptive HyperNEAT. The image is taken from Risi and Stanley [2010].

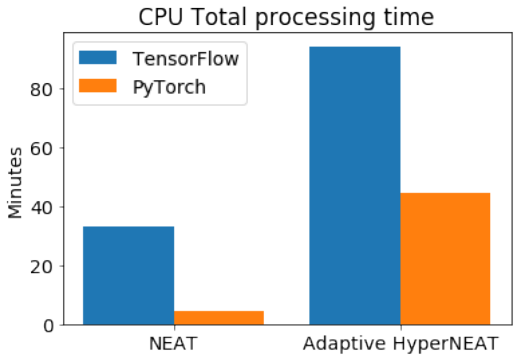


Figure 7: Total processing time of NEAT on CartPole-v0 and Adaptive HyperNEAT on T-Maze.

To show precisely by how much TensorFlow Eager is slower, Figure 7 shows the total processing time for NEAT and Adaptive HyperNEAT. When running NEAT on the CartPole-v0 task, TensorFlow Eager is 7.13 times slower. When running Adaptive HyperNEAT, TensorFlow Eager is 2.11 times slower. Another important aspect is that the running time of TensorFlow Eager has much more variance than PyTorch as it can be seen in the plots of both experiments. Some of that variance is also associated with the sparsity of the evolved tensors, which changes as the genomes evolve.

4.2 Code comparison

PyTorch was designed with an imperative mode of execution in mind. The API of PyTorch intentionally mimics the API of numpy and inherits its capabilities and flexibility. This results in a compelling Pythonian way of writing code. Indeed, some small API inconsistencies exist, and the mapping to the numpy API is not exactly one-to-one. TensorFlow also tries to mimic the numpy API, but some core features are missing. Many aspects of Eager execution are still hindered by the workflow of its static graphs version:

1. The distinction between variables and tensors. This distinction propagates all over the API and some operations such as *tf.assign* expect a variable as the first argument. In PyTorch, the Variable API has been deprecated, and tensors are the only building blocks. Variables break the symmetry with numpy, where numpy arrays are the only needed data type.

2. Complicated assignments. TensorFlow Eager does not support assignments using the "=" operator. This makes it impossible to write code similar to numpy. This forces the user to use the not so flexible *tf.assign*.

3. Lack of complex indexing. Probably the biggest disadvantage is that Eager does not support tensor-based indexing, where the indexing tensor could be a tensor of indices or truth values. This forces the user to use either much more complicated APIs or, for more advanced usages, the masking of tensor operations, which is very inefficient.

An example from the two NEAT libraries is given below for building a matrix from the genome of connections. This is an illustration of problems 2 and 3. Furthermore, the TensorFlow code would not have worked if the *mat* tensor had already contained some values different from zero.

```
# TensorFlow
mat = tf.scatter_nd(tf.stack([rows, cols], -1),
                    tf.convert_to_tensor(weights), shape)
# PyTorch
mat[torch.tensor(rows), torch.tensor(cols)] = torch.tensor(weights)
```

A combination of the second and third problem has prevented the writing of some small parts of Adaptive HyperNEAT in TensorFlow. Instead, they were written in numpy because it was more efficient than the TensorFlow workarounds.

5 Conclusion

In this paper, I introduced TensorFlow-NEAT, a neuroevolution library using TensorFlow's eager execution mode. The library is inspired by PyTorch-NEAT and built upon Python-NEAT. The correctness of the algorithms is demonstrated by the performance evaluation of the two libraries on two Reinforcement Learning tasks. From the evaluation, it is evident that TensorFlow Eager is a newcomer in the realm of dynamic computation graphs. In terms of execution time, TensorFlow Eager is significantly slower than PyTorch on standard matrix operations. From an API perspective, the static graphs APIs are still unavoidable and hinder the use of TensorFlow in a numpy-like, imperative fashion. However, some of these problems are likely to be resolved in TensorFlow 2.0 which will contain significant changes in the area of eager execution.

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

Perttu Hämäläinen, Amin Babadi, Xiaoxiao Ma, and Jaakko Lehtinen. Ppo-cma: Proximal policy optimization with covariance matrix adaptation. *CoRR*, abs/1810.02541, 2018.

Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. 2018.

Sebastian Risi and Kenneth O. Stanley. Indirectly encoding neural plasticity as a pattern of local rules. In *Proceedings of the 11th International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, SAB'10, pages 533–543, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15192-2, 978-3-642-15192-7. URL `http://dl.acm.org/citation.cfm?id=1884889.1884945`.

Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8:131–162, 2007.

Kenneth O. Stanley. A hypercube-based indirect encoding for evolving large-scale neural networks. 2009.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

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. *CoRR*, abs/1712.06567, 2017.

Dennis G. Wilson, Sylvain Cussat-Blanc, Hervé Luga, and Julian Francis Miller. Evolving simple programs for playing atari games. In *GECCO*, 2018.