NEATpy: A Python Package Implementing NEAT with a C++ Back End

Gage Golish
Department of Mathematics and Computer Science
Indiana State University

Spring 2019

Contents

1 Project Overview

1.1 Introduction

NEATpy is an original implementation of NEAT (NeuroEvolution of Augmenting Topologies) written in C++, and wrapped into a user friendly Python 3 package. The training of neural

networks via genetic algorithms is commonly referred to as neuroevolution. Many strategies for achieving this exist. The simplest form of neuroevolution is treating the weights of a group of neural networks, whose topologies have already been decided, as the population in a genetic algorithms experiment. NEAT, developed by Kenneth O. Stanley at the University of Texas in 2002, takes a slightly more complex approach in which the topologies of the neural networks are encoded and modified by genetic algorithms as well. The back end of this project was implemented entirely based on Stanley's original paper.

The code for this project can be divided into three categories: genetic algorithms, neural networks, and python module. The code that falls into the former two categories is part of the C++ back end, and deals with implementing the NEAT algorithms. The latter category of code deals with wrapping the back end in order for it to be used as a Python 3 package, and with extending the functionality and usability of the package for end users. Please note that this project is still in its infancy, and will continue to undergo radical changes in the near future.

1.2 Genetic Algorithms

Before genetic algorithms can be used to evolve a neural network's weights as well as its topology, a suitable encoding scheme must be chosen to represent a neural network. Just as in NEAT, in this project a neural network is represented by a list of neurons and a list of connections. In the context of genetic algorithms, we will refer to an encoded neural network as a *genome*, and its individual neurons and connections will be referred to as *genes*. This simple encoding scheme, along with some other innovations made in NEAT [1], enables genetic algorithms such as mutation, crossover, and selection to be efficiently applied to a neural network.

1.2.1 Mutations

There are two main types of mutations that are performed on a genome. The first type is connection gene mutations. A connection gene can undergo multiple different mutations:

- **Connection Weight Perturbation**: A connection can have its weight perturbed, which simply means that some random noise is added to the weight. Currently, a weight, w_i, is perturbed using the equation $w_{i+1} = w_i + \delta$, where δ is a random value from a normal probability distribution. Other possible perturbation algorithms are planned to be tested.
- **Connection Weight Replace**: A connection gene will have its weight replaced by a new random weight.
- **Connection Weight Toggle**: A connection gene contains a flag that determines whether the gene is enabled or not. When a gene is disabled, it is not used when activating the genome. This mutation switches a gene's disabled flag from `true` to `false`, or vice versa.

[1] Pun intended.

Another connection gene mutation that is planned, but not currently implemented, is connection gene removal. A random gene is removed from the genome, and the nodes in the genome are adjusted accordingly.

The other type of mutation is topological mutation. These mutations differ in that they change the topology of the genome. These changes are marked by historical markings referred to as innovation numbers. Each gene in the genome is marked with an innovation number. Topological mutations include:

- **Add Connection**: A brand new connection gene is added between two node genes in a genome. The connection gene is assigned an innovation number based on the innovation numbers of the two node genes it resides between.
- **Add Node**: A new node gene is added to the genome. This is achieved by choosing a random connection gene [2], and splitting it into two connection genes, each between one of the old nodes and the new node. The node is assigned an innovation number based on the two nodes it has been placed between. Note that the old connection is only disabled, not removed. It may be enabled again later due to a connection weight toggle mutation.

1.2.2 Crossover

The innovation numbers in the previous section are essential to the success of the crossover algorithm used in this project. A global hash map is maintained for connection innovation numbers, as well as for node innovation numbers. A new innovation number is only assigned if the innovation in question has not been seen before in the history of the current experiment. The current crossover scheme has the child inherit all of the node genes from the best fit parent. The connection genes from each parent are lined up, and if the parents share an innovation, it is randomly chosen from the parents. Excess and disjoint genes [3] are inherited only from the best fit parent. There are plans to add other types of crossover, in which genes are inherited from both parents.

1.2.3 Selection

Selection in this project is achieved via speciation. Each node is grouped into species using the distance function $d(g_1, g_2) = c_1E + c_2D + c_3\overline{W}$, where E and D are the numbers of excess and disjoint genes between g_1 and g_2 respectively, and $\overline{W}$ is the mean of the differences of the weights of the matching connection genes between g_1 and g_2. c_1, c_2, and c_3 are constants that allow preference to placed on any one of the measurements. If the distance between two genomes is under a set threshold value the genomes are grouped into the same species.

Each species is represented by a random genome within it. Before selection takes place, the population is sorted into species using these representatives and the distance function

[2] There is currently a bug in which a connection can be split twice, resulting in duplicate connection genes in the genome. This will be fixed in a future release.

[3] Disjoint genes are those that have innovation numbers less than that of the highest innovation number in the partner genome. Excess genes are those that have innovation numbers higher than that of the highest innovation number in the partner genome.

described above [4]. Each species is assigned a portion of the next population based on its collective fitness. Finally, this portion is filled by choosing one genome at a time via a tournament, and subsequently applying mutation and crossover algorithms to produce an offspring. Note that a genome can only be crossed over with another genome from the same species. Also, the best performing member of a species with more than five genomes is copied to the next generation without change. If a species has not increased its fitness after a certain number of generations, it becomes stagnant and is no longer allowed to reproduce, causing it to die out.

1.3 Neural Networks

The ultimate goal of running an experiment in this project is to produce a suitable neural network for the problem at hand. Each neural network begins as a simple fully connected feed forward neural network, containing a bias node and no hidden nodes. As an experiment progresses, the networks evolve a much more complicated structure.

1.3.1 Topology

The neural networks produced by the genetic algorithms described in the last section evolve a topology that is inconsistent with a traditional neural network topology. These networks are not organized in layers. For instance, an input node can have connections to multiple hidden nodes as well as a direct connection to an output node. This is an advantage, as such free-form neural networks may not be thought of when one is designing a neural network topology. It is also possible for recurrent connections to be evolved. The only restriction placed on a network's topology is that a connection's input must not come from an output node, and a connection's output may not be fed to an input node.

1.3.2 Activation

With such interesting topologies, these networks are not activated via the common feed forward algorithm. Instead, a depth first search is launched from each output node, propagating backward through the network until an input or bias node is reached. Each hidden and output node's value is passed through the activation function $a(x) = \frac{1}{1+e^{-4.9x}}$, a slightly steepened sigmoid curve. In order to handle the recurrent connections, each network is activated multiple times before returning the output.

1.4 Python Module

NEATpy is intended to be used as a Python 3 module with a simple set of classes and function calls. This will allow the user to integrate NEATpy into their project with relative ease, and to use NEATpy in conjunction with other powerful Python 3 machine learning packages such as `numpy` and `scikit-learn`.

[4] In a future release, a specified portion of each species will be eliminated via stochastic universal sampling at this point.

1.4.1 Wrapping Back End

The back end is wrapped into a Python 3 module using Swig, a wrapper and interface generator. Only code that is essential to the end user is wrapped for use in the module. Currently the only class available to the Python 3 user is `NEAT` and its associated methods. `Genome`, `Node`, `Connection`, `Species`, and NEAT hyper parameters will be available in a future release.

1.4.2 Child Classes

Each class in the module generated by Swig will be hidden from the end user. Instead, child classes will be created in order to hide all interface code, and to extend the abilities of the parent classes. This will allow a more powerful layer of functionality to be laid on top of the relatively simple back end.

1.4.3 Visualization

Another important piece of functionality added to the Python 3 module is the ability to visualize a neural network in the population. This is accomplished via the incorporation of existing Python 3 modules that excel at visualizing networks. Currently, the only visualization function implemented stores the image generated in PNG format.

2 Project Organization

This section will describe the organization of the project, as well as the purpose of each file. Documentation will not be included for how to use the project. This will be added in the near future.

2.1 Tree View

```
MyNEAT/
|-- neatpy
|   |-- __init__.py
|   |-- neat_backend
|   |   |-- include
|   |   |   |-- Connection.h
|   |   |   |-- Genome.h
|   |   |   |-- InnovationMap.h
|   |   |   |-- NEAT.h
|   |   |   |-- Node.h
|   |   |   |-- options.h
|   |   |   |-- Species.h
|   |   |   `-- utilities.h
|   |   |-- __init__.py
|   |   |-- Makefile
```

```
|   |   |-- neat_backend.i
|   |   |-- setup.py
|   |   |-- src
|   |   |   |-- Connection.cpp
|   |   |   |-- Genome.cpp
|   |   |   |-- InnovationMap.cpp
|   |   |   |-- NEAT.cpp
|   |   |   |-- Node.cpp
|   |   |   |-- options.cpp
|   |   |   |-- Species.cpp
|   |   |   `-- utilities.cpp
|   |   `-- tests
|   |       |-- and.cpp
|   |       |-- avg_xor.cpp
|   |       |-- or.cpp
|   |       `-- xor.cpp
|   |-- NEAT.py
|   `-- visualize.py
|-- test-xor.py
|-- visualize_xor.py
`-- xor.py
```

2.2 Files

- **C++ Backend**:
  - **Connection.h/cpp**: Contains the `Connection` class definition. This class represents a connection gene.
  - **Genome.h/cpp**: Contains the `Genome` class definition. This class represents a genome, which has references to both connection genes and node genes in order to encode a neural network.
  - **InnovationMap.h/cpp**: Contains the `InnovationMap` class definition. This class is used to keep track of both node gene and connection gene innovation numbers throughout the duration of the experiment. The hash map enclosed requires the key to be a pair of integers, so the hash function from the boost library is used.
  - **NEAT.h/cpp**: Contains the `NEAT` class. This class is the main class that is interacted with by the end user. It contains the population of genomes, the innovation number hash maps, as well as functions for repopulation and activating networks.
  - **Node.h/cpp**: Contains the `Node` class definition. Represents a node gene. Also contains a type definition for the type of a node.
  - **options.h/cpp**: These files contain global variables wrapped in the `neat_options` namespace that serve as the hyper parameters for the current experiment. Cur-

rently, it is impossible to set these hyper parameters when using the Python 3 package. The plan is to pass a dictionary of options to the `NEAT` class constructor.

  - **Species.h/cpp**: Contains the `Species` class definition. Represents a species during the speciation process. The primary reason for creating a separate class for a species instead of simply keeping track of a vector is to facilitate a simple way to keep track of stagnation.
  - **utilities.h/cpp**: Contains several namespaces, with functions that are meant to be used internally by the C++ back end. These functions are not intended to ever be available in the Python 3 package.
  - **tests/*.cpp**: These files contains tests that were used during development to track the progress of the project. Each trains a network to solve a simple logic function.
  - **Makefile, neat_backend.i, setup.py**: These files are used to build the back end module. `neat_backend.i` is a Swig configuration file that tells Swig what to wrap. `setup.py` is used to compile the project after Swig is called. The make file build the project when `make` is run.

- **Python Package**:
  - **NEAT.py**: Contains the `NEAT` class that will be used when using the package. It acts as an interface between normal Python 3 code and the Swig generated `NEAT` class in the back end module. It also implements a few new methods that are not in the original `NEAT` class.
  - **visualize.py**: Contains a function for generating a visual representation of a network when passed a `Genome` object. The image is stored as a PNG file.
  - **__init__.py**: Decides what gets imported when `neatpy` is imported.

- **Examples**:
  - **xor.py**: An example of using `neatpy` in conjunction with `numpy` to evolve a network that solves the XOR problem.
  - **visualize_xor.py**: Also evolves a network that can solve XOR, but visualizes the solution space of the best fit genome during each generation.
  - **test-xor.py**: Calculates the average number of generations required to solve XOR over n epochs.

3 Tests and Results

3.1 XOR Test

NEATpy has only been successfully tested on the XOR problem so far. Over 100 epochs, it found a solution in 5368 generations on average. Compared to the results of the original NEAT paper, this is a very poor score. Some examples of networks found are shown in Figure 1.

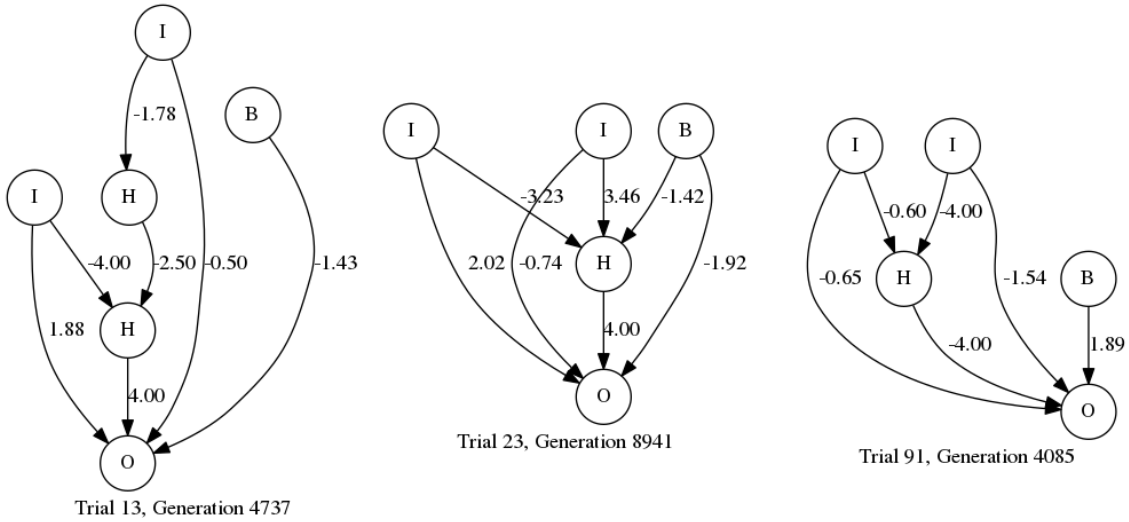

Figure 1: Three solutions found when running the XOR test. These networks have only one or two hidden nodes, making them good XOR solutions. Most networks found in the tests, however, were much more complex. This suggests some changes need to be made to the evolutionary algorithms.