Project Part 2
736 Neural Networks and Machine Learning
Nic Manoogian
Robert Bond III
Zach Lauzon

Executive Summary

In the second part of our project, we maintain our original problem and data sets: create an optimal agent for playing Nintendo's *Super Mario Bros.*. In this document, we describe the selection of different hyper-parameters in the NEAT algorithm, the experimentation methods used to test different values for these hyper-parameters, and the results of our experiments and conclusions.

Overall, we notice the following trends in our study:

- Sparser networks (those with a higher ratio of neurons to connections) reach a higher fitness during the computation time.
- A smaller learning rate (later referred to as *step size*) leads higher variance and average performance across experiments.
- A high maximum stagnation period seems to provide little benefit.

Requirements

The original requirements for this part of the project outline different experiments for learning our data set. Specifically, we were instructed to test hyper-parameters for both Multilayer Perceptron (MLP) and Radial Basis Function (RBF) neural networks.

This posed a problem because these methods are used in supervised learning and our data requires unsupervised reinforcement learning. In learning from our data set, we employ an evolutionary algorithm called *NeuroEvolution of Augmenting Topologies (NEAT)*. In this algorithm, neural networks are generated in each generation by crossing over the highest scoring networks and mutating them (K. Stanley, 2015).

Like MLP and RBF neural networks, the NEAT algorithm has hyper-parameters that can be adjusted as well. For this part of the project, we select hyper-parameters relevant to our problem and defend this selection, we outline our experimental design and process, and we discuss our results.

Specification

There are 16 primary system parameters that can be adjusted in the NEAT algorithm (K. O. Stanley, 2004). Due to computational constraints and combinatorial explosion, we have selected five parameters to test:

1. *Population size*: The number of genomes present in each generation.
2. *Maximum stagnation*: The maximum number of generations an individual species is permitted to not progress (achieve a higher max fitness) before being removed.
3. *Add link chance*: The average number of weights added between neurons to each genome per generation.
4. *Add node chance*: The average number of neurons added to each genome per generation.
5. *Step size*: The maximum amount by which one weight may be mutated per generation.

To keep the computation reasonable, we selected two values for each of these parameters (also referred to as "high" and "low").

Parameter Justifications

Population size is an important parameter in almost all genetic algorithms. We view population size as a trade-off: larger populations have greater diversity due to more opportunities for mutation while smaller populations converge quickly. Theoretically, a very large population size might introduce too much randomness and leave the computation divergent and a very small population size might fall too quickly into a local maximum. Practically, there is another concern: larger populations require more computation time per generation. In one implementation, SethBling uses a population size of

300 (SethBling, 2015). Keeping this reference in mind, we select high and low values of 700 and 300, respectively.

The NEAT algorithm separates the population of networks into different groups (called *species*) that develop quasi-independently. This process allows the algorithm to identify "innovations" that evolve independently across species and preserve them (K. O. Stanley, 2004). The number of species, as well as the number of networks per species, is variable. If a species doesn't reach a new maximum fitness in the maximum stagnation time, it is removed to allow for more growth in more promising species. SethBling selects a maximum stagnation value of 15 generations. However, SethBling only allows the algorithm to run for tens of generations but, using computational optimizations, we have the capacity to run the algorithm for thousands of generations. For this reason, we select maximum stagnation high and low values of 200 and 50 with the assumption that letting species "live longer" might allow for the emergence of more novel strategies.

Most genetic algorithms include some notion of a "mutation rate." Because the NEAT algorithm generates neural networks, we have a number of topology factors to mutate. One such factor is called the "add link chance." This is the probability of adding a new connection between any two random nodes with a random weight. If the value is greater than 1.0, the whole number of links will be created and the remaining decimal will be the probability of adding an additional node. SethBling chooses a value of 2.0; thus, his implementation adds 2 nodes exactly. We use high and low values of 2.5 and 1.0.

Another topology factor is called the "add node chance". This is a the probability of adding an entirely new hidden node to the network topology. If the values is greater than 1.0, the same rule applies here as it did in the "add link chance parameter." SethBling uses a value of 0.5. We use high and low values of 0.7 and 0.3. Clearly the number of nodes and links are associated; we study this relationship in a later section.

The final parameter that we test is called the "step size." This is the amount by which the NEAT algorithm will mutate the weights associated with each connection in the network. In a way, this is analogous to a "learning rate" in traditional supervised learning techniques. If the step size is too large, the algorithm may lack the resolution to find the optimal solution. If the step size is too small, the algorithm may fall too quickly into a local maximum. SethBling's implementation uses a value of 0.1. We test high and low values of 0.1 and 0.05. Much like an ANN learning rate, there is

unfortunately no way to know if one has selected a good step size.

Implementation

One significant advantage of the NEAT algorithm is that it lends itself well to parallelization. In our system, one central *server* generates the population of neural networks by implementing the NEAT algorithm. The population is then divided among one or more *clients* which play a fixed set of levels in the game using an assigned neural network. They report game play data (such as frames played, victory or defeat, mode of defeat, etc.) back to the server. The server collects this information from the clients and generates new networks using the NEAT algorithm. This process repeats over a number of generations. Using a number of clients including personal computers, Amazon Web Service Cloud Computing, and the Google Compute Engine, we are able to achieve a maximum client computation rate of 3.6 billion frames per hour. For reference, SethBling's implementation used a single emulator that ran at 5.4 *million* frames per hour

Another module, called the *facilitator*, manages an *experiment pipeline* which is essentially a queue of experiments to conduct. An example is provided in *Listing 1.* The keys in this experiment file correspond to various parameters in the experiment. For example, the *Population* key corresponds to the population size described in the previous section. The experiment file also contains keys associated with when to stop the experiment. For example, $StopFrames = 3000000000$ is interpreted by the facilitator to mean "stop this experiment after 3 billion frames." The facilitator dispatches each experiment to the server and aggregates the results into CSV files which can be easily plotted in a graphing program.

Experiments

We generate 32 experiments (named $e0$ through $e31$) by combining all possible high and low values for the selected NEAT hyper-parameters. Refer to the following table for individual experiment specifications. Each experiment was allowed to run for 3 billion frames.

ID	Population	Max Stagnation	Add Link Chance	Add Node Chance	Step Size
e0	300	50	1.0	0.3	0.05
e1	300	50	1.0	0.3	0.10
e2	300	50	1.0	0.7	0.05
e3	300	50	1.0	0.7	0.10
e4	300	50	2.5	0.3	0.05
e5	300	50	2.5	0.3	0.10
e6	300	50	2.5	0.7	0.05
e7	300	50	2.5	0.7	0.10
e8	300	200	1.0	0.3	0.05
e9	300	200	1.0	0.3	0.10
e10	300	200	1.0	0.7	0.05
e11	300	200	1.0	0.7	0.10
e12	300	200	2.5	0.3	0.05
e13	300	200	2.5	0.3	0.10
e14	300	200	2.5	0.7	0.05
e15	300	200	2.5	0.7	0.10
e16	700	50	1.0	0.3	0.05
e17	700	50	1.0	0.3	0.10
e18	700	50	1.0	0.7	0.05
e19	700	50	1.0	0.7	0.10
e20	700	50	2.5	0.3	0.05
e21	700	50	2.5	0.3	0.10
e22	700	50	2.5	0.7	0.05
e23	700	50	2.5	0.7	0.10
e24	700	200	1.0	0.3	0.05
e25	700	200	1.0	0.3	0.10
e26	700	200	1.0	0.7	0.05
e27	700	200	1.0	0.7	0.10
e28	700	200	2.5	0.3	0.05
e29	700	200	2.5	0.3	0.10
e30	700	200	2.5	0.7	0.05
e31	700	200	2.5	0.7	0.10

Listing 1: p2_e0_300_50_1.0_0.3_0.05.experiment

```
{
  BiasMutationChance = 0.40000000000000002,
  BoxRadius = 6,
  ButtonNames = {
    "A",
    "Left",
    "Right"
  },
  ClientCode = "normalize_velocities.lua",
  CrossoverChance = 0.75,
  DeltaDisjoint = 2,
  DeltaThreshold = 1,
  DeltaWeights = 0.40000000000000002,
  DisableMutationChance = 0.40000000000000002,
  EnableMutationChance = 0.20000000000000001,
  InputSize = 169,
  Inputs = 172,
  LevelAugmenter = 0.10000000000000001,
  LinkMutationChance = 1.0,
  MaxNodes = 1000000,
  MutateConnectionsChance = 0.25,
  Name = "p2_e0_300_50_1.0_0.3_0.05",
  NodeMutationChance = 0.3,
  Outputs = 3,
  PerturbChance = 0.90000000000000002,
  Population = 300,
  Port = 56509,
  StaleSpecies = 50,
  StepSize = 0.05,
  StopFitness = -1,
  StopGeneration = -1,
  StopTimeSeconds = -1,
  StopFrames = 3000000000,
  VERSION_CODE = "ZMQ",
  WorldAugmenter = 0.20000000000000001
}
```

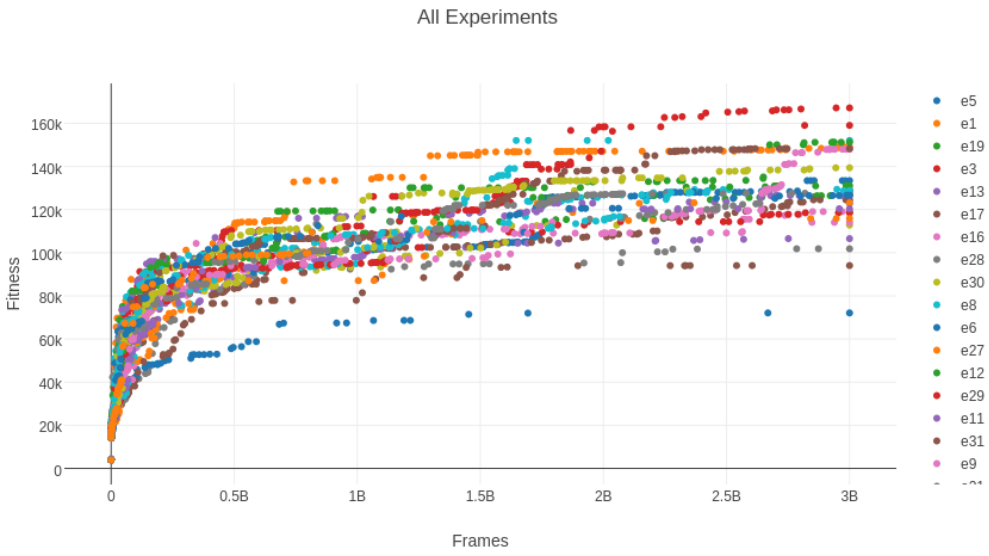
Results

Due to the large and multi-dimensional nature of the data, we include graphs for only relevant findings. All data may be viewed and downloaded in CSV format at `https://plot.ly/~ZachLauzon/71/`.

All Experiments

We first plot all experiments (distinguished by color) on one graph (see *Figure 1*). This graph (and all graphs in this analysis) plot maximum fitness achieved as a function of frames played. We use frames played so that our results are independent of population size and hardware available.

Figure 1: All Experiments - `https://plot.ly/~ZachLauzon/74/all-experiments/`



Population

We first examine the experiments in terms of the population parameter. In *Figure 2*, we separate the experiments into two groups: the 300 and the 700 population size experiments. We take this approach in each parameter

analysis. In doing so, we hope to observe the effects of each parameter individually.

Figure 2: Population - `https://plot.ly/~ZachLauzon/75/population/`

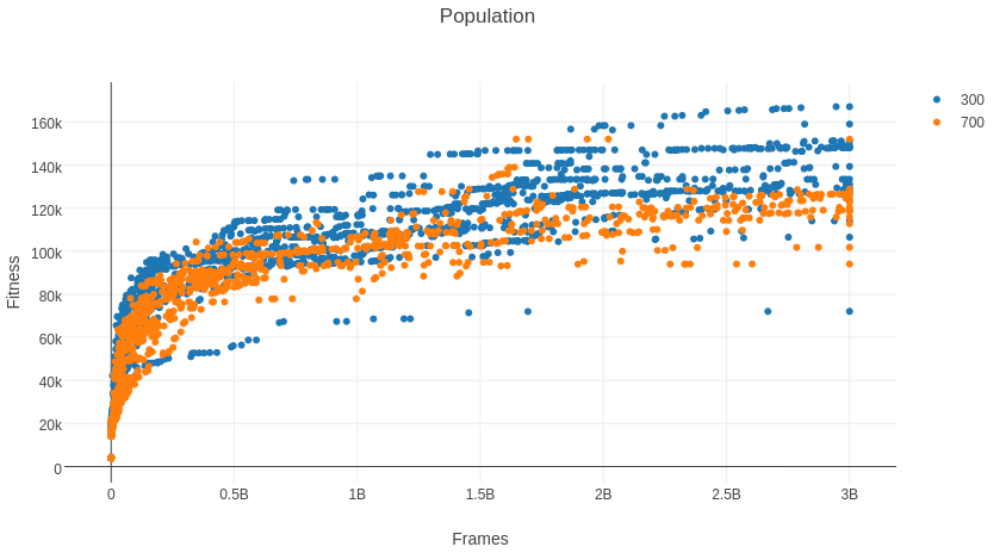


Figure 2 shows that a population size of 300 achieves significantly better performance over a 3 billion frame period. This may be due to less noise in smaller populations, or that smaller populations can evolve more generations over a fixed period than larger ones. A longer running experiment may be necessary to further explore this relationship.

Number of Neurons and Connections

Figure 3: Add Link - `https://plot.ly/~ZachLauzon/81/add-link/`

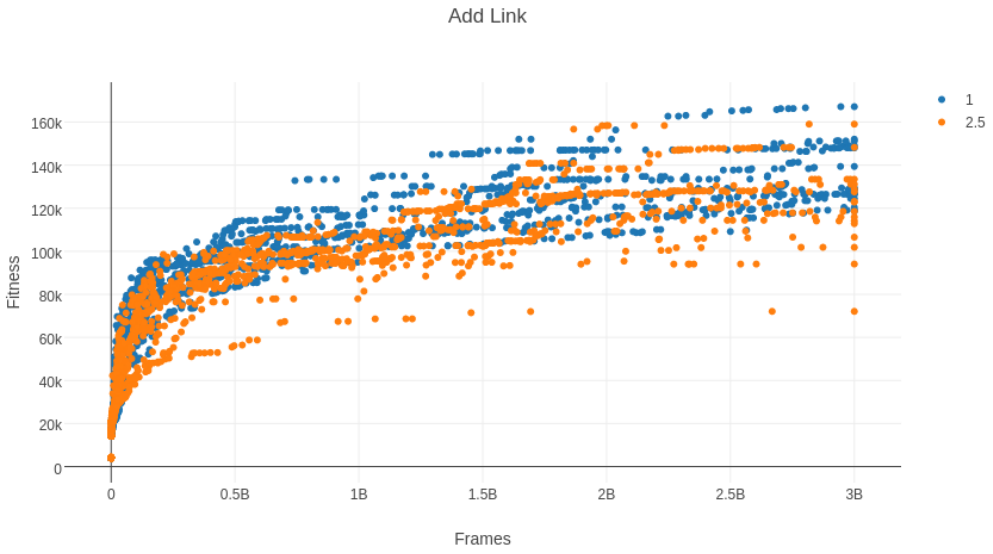


Figure 3 strongly suggests that adding fewer connections between neurons every generation produces better networks over time. Adding more connections seems to increase variance while reducing average performance.

Figure 4: Add Node `https://plot.ly/~ZachLauzon/78/add-node/`

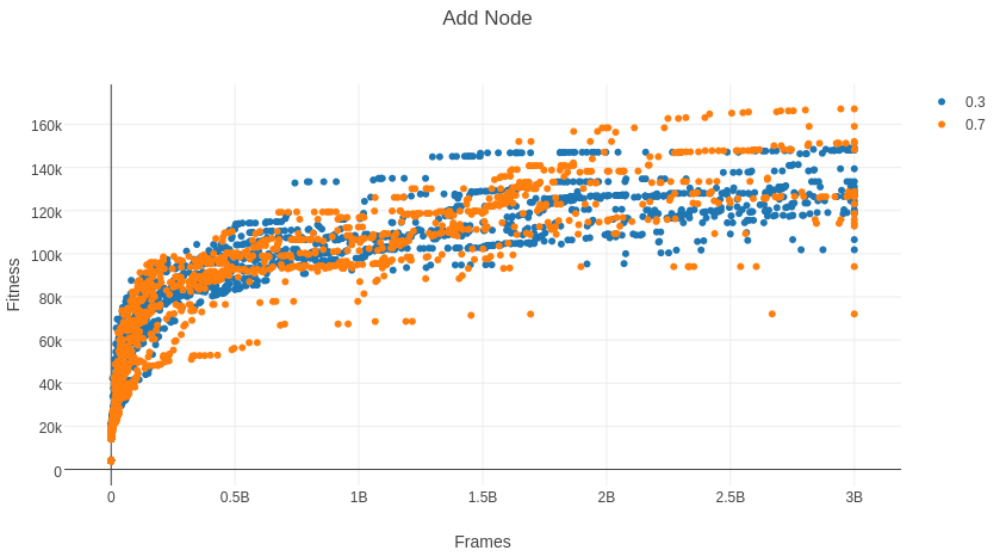
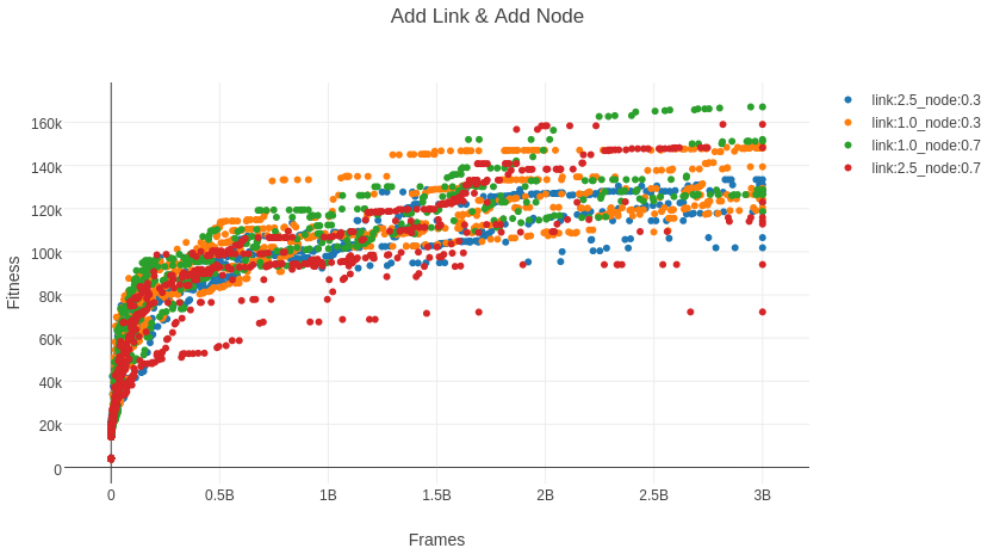


Figure 4 demonstrates that adding more neurons every generation leads to a much higher variance in performance across experiments. Adding 0.3 neurons per generation led to very similar behavior across half of the experiments, despite all other parameters being altered.

Figure 5: Add Link and Add Node `https://plot.ly/~ZachLauzon/79/add-link-add-node/`



To further explore how the frequency of adding neurons and connections affects performance, *Figure 5* plots the 4 unique combinations of add link and add node frequency. As both *Figure 3* and *Figure 4* suggest, adding more neurons and fewer connections produced the highest performing group (*link* : 1.0 *node* : 0.7; in green). The red group, with the same number of neurons as green, displays the highest variance among all groups. We conclude that sparsely connected neural networks are better suited to solve the task at hand, and will further explore increasing the neuron to connection ratio.

Step Size

Figure 6: Step Size - `https://plot.ly/~ZachLauzon/80/step-size/`

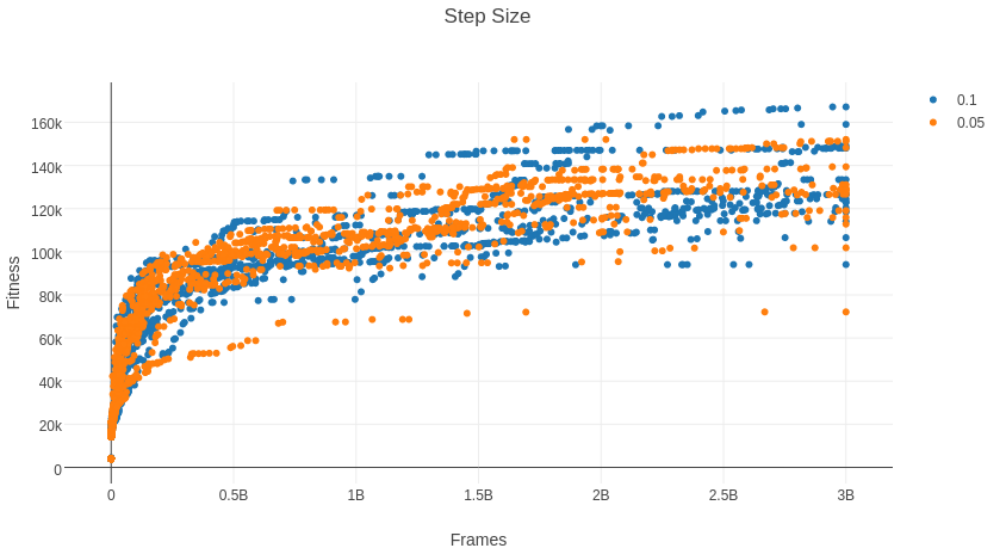
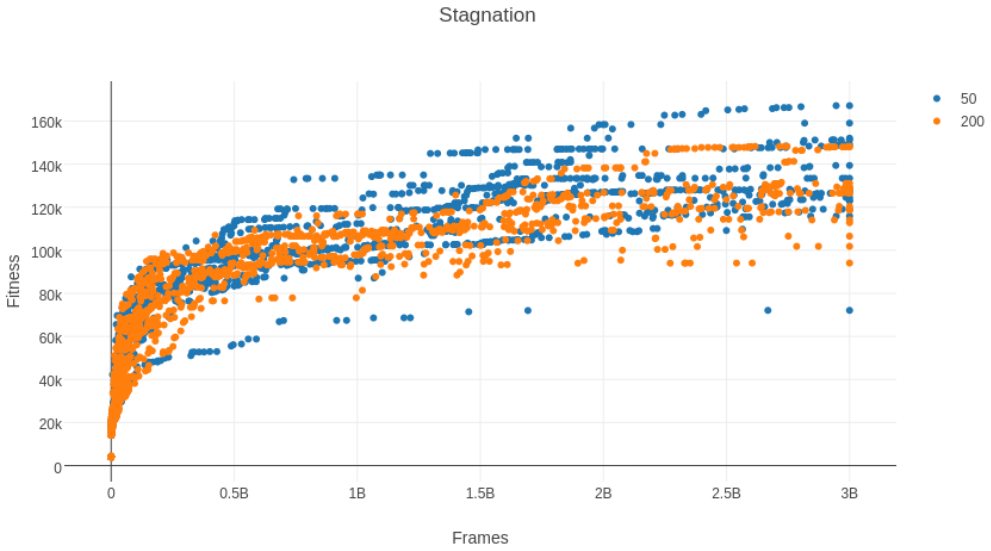


Figure 6 Shows that a higher step size led to the best-performing experiments. Much like learning rate, it seems that a higher maximum mutation rate is necessary to help keep the genetic algorithm out of local maxima.

Maximum Stagnation

Figure 7: Maximum Stagnation - `https://plot.ly/~ZachLauzon/76/stagnation/`



A smaller maximum stagnation rate is shown to achieve higher fitnesses significantly faster, as displayed in *Figure 7*. This suggests that most species take fewer than 200 generations to find more optimal structure. A longer experiment would reveal the long-term implications of this parameter, as a higher stagnation rate could be necessary when reaching thousands of generations of evolution.

References

SethBling. (2015, Jul). *Mari/o - machine learning for video games.* YouTube. Retrieved from https://www.youtube.com/watch?v=qv6UVOQ0F44

Stanley, K. (2015, May). *The neuroevolution of augmenting topologies (neat) users page.* University of Central Florida. Retrieved from https://www.cs.ucf.edu/~kstanley/neat.html

Stanley, K. O. (2004). Efficient evolution of neural networks through complexification.