Comparing Evolution Strategies with model-free Reinforcement Learning in Boxing (Atari 2600)

Alexander Egorov, Daniel Gradinaru, Rohil Gupta
{aegorov, dgradina, rvgupta}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Introduction

Many papers have presented different AI models that can act on high-dimensional sensory inputs such as sight and hearing. Most of these papers describe their algorithms and then demonstrate their effectiveness in their given environments. However, fewer efforts have been made to compare the studies' results. This paper will select two heavily studied models and attempt to compare their performance against their training time in-game learning.

This is a particularly interesting and challenging problem since algorithms are often introduced, improved upon or presented in a different light. However, in this paper, the relative learning rate of the NEAT and Q-learning algorithms is explored. Both algorithms presented in the paper will be trained on similar data-sets, and will be compared periodically. Since the chosen game, Boxing Atari 2600, can be adjusted to be 2-player, the two algorithms can be both theoretically and physically pitted against one another too.

The first algorithm this paper will study is the Genetic Algorithm (GA), NeuroEvolution of Augmenting Topologies (NEAT). NEAT works by repeatedly generating new neural networks from the highest performing ones in the previous generations (Stanley and Miikkulainen 2002). This approach somewhat mirrors natural evolution by allowing the highest performing networks to move on and discarding the less successful ones (Stanley and Miikkulainen 2002). In some cases, this is easier to implement since it does not require a value function (Stanley and Miikkulainen 2002).

This paper will also study the Temporal Algorithm, Q-learning (Whiteson and Stone 2006), with a primary focus on Deep Q-Learning. This is a popular variation of a Q-learning, created by Google Deep Mind, which approximates its Q function using a convolutional neural network (Ohnishi et al. 2019). This approach has been seen to be highly successful since in many cases its Q-function has been observed to converge and stabilize even when Q-learning fails to do so (Ohnishi et al. 2019).

Contributions

This paper will focus on answering the following question *How does the learning rate of NEAT algorithms compare to the learning rate of Q-learning algorithms in an arcade boxing game?* The Atari 2600 game, *Boxing* will be used in this paper.

The research question will be answered by training two models - one using a NEAT algorithm and one using a Q-learning algorithm - and comparing them at specific time intervals. The two models will be trained individually against the game's existing AI, to avoid discrepancies in the training data. Time intervals will be used instead of training cycles to avoid giving preference to the algorithm with the longer cycle. With this method, it will be possible to compare the algorithm's relative learning speeds against training time as well as their final results.

NEAT algorithms are expected to learn faster as multiple papers have shown that they generally outperform Q-learning algorithms (Whiteson and Stone 2006; Taylor, Whiteson, and Stone 2006). Q-learning has a stronger theoretical foundation than NEAT, however, NEAT tends to do better in practice (Whiteson and Stone 2006; Taylor, Whiteson, and Stone 2006). For more details refer to the *Results* section below.

Related Work

The usage of genetic algorithms to evolve neural network topologies dates back to several decades ago. Among the first usage of genetic algorithms in reinforcement learning problems is using "genetic reinforcement learning" in a simulated inverted-pendulum control problem with an algorithm referred to as GENITOR (Whitley et al. 1993). The inverted pendulum problem, also commonly referred to as cart-centring/pole-balancing, involves a cart with an inverted pendulum attached to it via a hinge (depicted in Figure 1). The goal is to balance the pendulum by applying an appropriate force on the cart, as without this the pendulum will simply fall over. The paper found that, when running through several different comparative tests against an Adaptive Heuristic Critic (AHC) algorithm used by Charles W. Anderson, the genetic algorithm provided competitive results despite not requiring state information at each time step.

Several years later David Moriarty and Risto Miikkulainen presented a new method using genetic algorithms to form neural networks from evolved individual neurons - SANE (Symbiotic, Adaptive Neuro-Evolution) (Moriarty

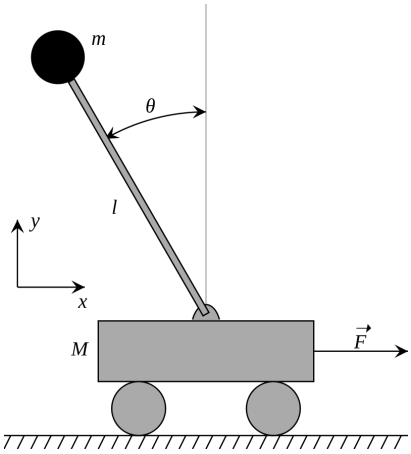

Figure 1: A schematic drawing of an inverted pendulum on a cart

and Mikkulainen 1996). This was also tested using the inverted pendulum problem, this time proving considerably faster and more efficient than AHC, Q-learning, and the aforementioned GENITOR. More specifically, SANE on average required far fewer balance attempts than Q-learning and GENITOR and spent roughly a half of the CPU time (whereas 1-layer AHC required the lowest balance attempts on average but had very long CPU times resulting in a 20x slower solution, and 2-layer AHC had by far the highest balance attempts with similarly poor results).

Following this and another upgraded approach Enforced Sub-populations (ESP) tested on a much more difficult case of the inverted pendulum problem involving two poles (Gomez and Miikkulainen 1999). Risto Miikulainen worked with Kenneth O. Stanley on an improved method of neuroevolution, NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen 2002). They succeeded in introducing a more efficient approach as a result of "employing a principled method of crossover of different topologies, protecting structural innovation using speciation, and incrementally growing from the minimal structure", and proved that each of these components is necessary for improving NEAT through a series of ablations (removing the parts and testing to ensure that the algorithm became weaker). The paper discussing this algorithm concluded that performance can be greatly increased by evolving structure along with connection weights (if done correctly), and that NEAT is several times more powerful and efficient than previous attempts in neuroevolution.

Since then, there have been extensions to NEAT, such as real-time NeuroEvolution of Augmenting Topologies (rt-NEAT) that allows for evolutions to occur in real-time and was used with a video game (similar to a real-time strategy game) built around this principle, NeuroEvolving Robotic Operative (NERO), where the player trains a team of robots to fight enemy against another player's team (Stanley, Bryant, and Miikkulainen 2005), and HyperNEAT which uses NEAT to evolve Compositional pattern-producing networks (CCPNs) that represent spatial patterns in hyperspace (Stanley, D'Ambrosio, and Gauci 2009).

Q-learning was first introduced in a thesis submitted for Christopher Watkins's Ph.D. (Watkins 1989). It is a reinforcement learning algorithm that has its roots in the study of animal intelligence, specifically rewarding/punishing for appropriate behaviour (operant conditioning). This algorithm uses the Markov decision process (MDP) as a model for the problems it solves. MDPs are made up of a state-space, a function that returns a list of viable actions for a given state, a transition function that is the probability that an action in a state will lead to another state, and a reward received when we transition from state to state. The "Q" stands for quality, and the algorithm works by computing the quality of state-action combinations. Q is initially some arbitrary value, and as the algorithm progresses we update it for each state-action pair iteratively (usually with the Bellman equation). We can store this data in what is called a Q-table, which is essentially a lookup table that tells us at a given state the maximum expected reward.

Q-learning is a type of temporal difference (TD) learning, a class of model-free reinforcement learning methods. One of the more impressive advances in TD research was Gerald Tesauro's TD-Gammon, a computer program that learned to play backgammon at the Grandmaster level entirely through reinforcement learning (Tesauro 1995). Shortly following this some attempts were made to tackle more complex games such as GO using this method, but they did not have the same exceptional results.

More recently, however, a team from DeepMind Technologies made some significant advancements in a variant of Q-learning, deep Q-learning (Mnih et al. 2013). It was heavily inspired by TD-Gammon's architecture, and made use of the advancements in hardware, modern deep neural network architectures, and scalable Reinforcement Learning algorithms that had happened in the previous 20+ years. Using only the pixels as input in a variety of Atari 2600 video games, deep Q-learning was able to become very skilled at Breakout, Enduro, Pong, and more. DeepMind was acquired by Google in 2014 and 2016 became known worldwide for beating Lee Sedol, one of the best Go players in the world with the AlphaGo program.

Methodology

The algorithms will be compared and tested in the Atari 2600 video game Boxing (See Figure 2). Each model will be trained independently and then at specific time intervals, the models' performances will be evaluated through a head-to-head game. It should be noted that these head-to-head games will not be used to train the models, but rather to determine which is superior. With this data, we will be able to adequately compare the two algorithms at various benchmarks. This approach will not only allow us to determine which algorithm has learned more at given points in time but will determine which algorithm has the better final performance. *OpenAI's Gym Retro* platform will be used to facilitate the training and testing of the algorithms as it provides a useful interface with the Boxing game.

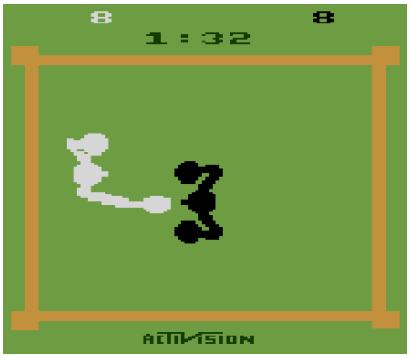

Figure 2: Boxing, 1980 top-down video game interpretation of boxing developed by Activision for the Atari 2600

Evaluation Method

The *OpenAI's Retro Boxing (Atari 2600)* environment (`https://github.com/openai/gym/blob/master/gym/envs/atari/atari_env.py`) will be utilized to interface with the game. The environment provides a standardized interface by which one can perform actions for an agent. Since data collection in a game can be a challenging task, the environment handles the data collection and data normalization. The environment provides a set of actions that an agent can take, such as punch, and move left, and returns both rewards and observations. It achieves this by purposefully and carefully reducing the level of complexity of a game to allow an agent to better understand its environment. One of the ways this is done is by removing several unneeded portions of pixels from the game to produce the screenshot in Figure 2.

Furthermore, the OpenAI environment automatically collects valuable data points, such as each player's score and the remaining time, from the simulated game, which it returns as rewards. Note that as the Open AI environment directly interfaces with the game and collects data, there is no further pre-processing that can be conducted. Thus, one is no longer concerned with how game data should be collected, how often it should be collected, and how it should be pre-processed. The focus can instead be placed on the efficacy of the algorithms being implemented instead.

Algorithms

The algorithms that will be used to answer the proposed research question are NEAT (NeuroEvolution of Augmenting Topologies) and deep Q-learning.

NEAT (Stanley and Miikkulainen 2002) is a type of genetic algorithm, a search heuristic that mimics some aspects of biology such as mutations and crossovers (reproduction between two parents to generate offspring). NEAT keeps track of a collection of genomes, each of which contains a list of *connection genes*. These *connection genes* specify two connected nodes, the weight of their connection, and something Stanley's paper refers to as an *innovation number* that is incremented with every added gene. This latter part is used to track the ancestry of every gene, important for matching them for crossover. The *innovation number* solves the problem of competing conventions for disparate topologies (which arises when there are multiple ways of representing the same information in different networks, see Figure 3), an important innovation for the field of neuroevolution. Networks are then mutated by either adding a new connection between two previously unconnected nodes, or adding a new node to an existing connection, splitting it into two new connections.

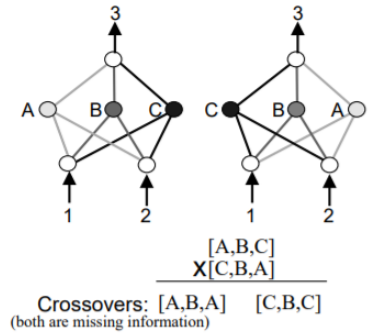

Figure 3: Competing conventions problem. Here the two networks contain the same information but in different order, [A,B,C] and [C,B,A], which is problematic as the result of their crossover, ex. [C,B,C], will be missing a third of the main components

The user creates a fitness function that is used to determine the *fitness* (quality) of an individual genome. This is used to determine how well a neural network produced by a genome solves the given problem or how close it is to solving it. With each generation the fittest individuals are chosen to mutate and reproduce, leading to increasingly complex and powerful neural networks.

The implementation of NEAT that will be used is **neat-python** (McIntyre et al.).

The game will be scaled down and converted from RGB to grayscale. This is done to minimize the number of inputs into our neural network and to reduce the complexity of computations. Next, using the neural network's `activate` function, this data will be used as input and will produce an array as output, representing the (game) controller input. This array will either be of size 8 or 16, depending if we are in one-player or two-player mode. We call the **retro** environment's `step` function with this to produce game info (player 1's score, player 2's score, time left), whether the game is over or not, and the reward. Making use of a fitness function of choice, the current fitness of the genome being tested is computed. If this exceeds some predetermined threshold, we stop. Otherwise, we continue running on the next frame of the game until the game is over or we have met some other exit condition. This work is done for each genome in a generation, and each generation will choose the best genomes to mutate and reproduce. This is done mostly by the **neat-python**, though we can modify many values that control factors such as genome compatibility options, connection add/remove rates, network parameters, node activation options, and more in a configuration file that is used by

our main program.

Deep Q-learning (Mnih et al. 2013) is a variant of the Q-learning algorithm, a model-free reinforcement learning algorithm that computes and iteratively updates the *quality* (maximum expected reward) of state-action combinations. The issue here is clearly that if there are many states and/or actions per state for a given problem, the respective Q-table will be large which will lead to increased memory required to store the table, as well as increased computation time to explore each state. The solution to this problem brought forward by DeepMind is to approximate Q-values with the use of neural networks, specifically deep Q-learning networks (DQNs) (Figure 4). Deep Q-learning utilizes a technique referred to as *experience replay*, which involves storing the agent's experiences over time into a *replay memory*. Q-learning updates are then applied to random samples of experiences, after which actions are chosen and executed according to an epsilon-greedy policy.

Below is the pseudocode for the algorithm.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 4: Pseudocode for deep Q-learning with experience replay (Mnih et al. 2015)

The implementation of deep Q-learning that will be used is **keras-rl** (Plappert 2016), a library that includes several deep reinforcement learning algorithms with Keras (an open-source neural-network library written in Python). This will be appropriate to use as it has built-in support for the OpenAI Gym environments that will be used for evaluation. We will further need to utilize and possibly modify the existing implementations of the above algorithms to make them work with OpenAI's Retro Boxing environment.

Timeline

The first step in implementing these algorithms is to first initialize the OpenAI Retro Boxing environment and make the player character move randomly. Once this is completed, the NEAT algorithm can be used to train an agent. Since this process is time-consuming it can be optimized by computing

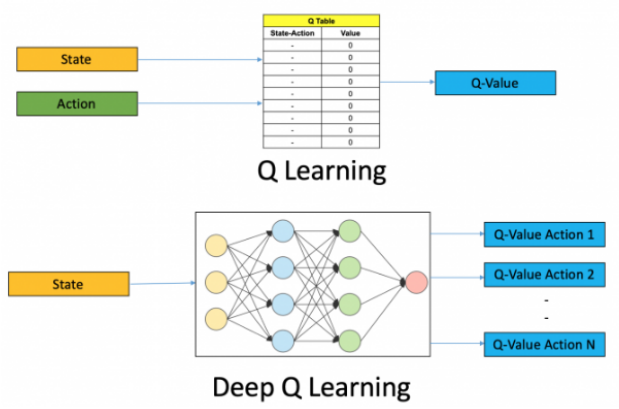

Figure 5: Comparison between Q-learning and deep Q-learning. Q-learning takes a state and and action and outputs a single Q-value according to what's in the Q-table, whereas deep Q-learning uses a neural network to produce the Q-values of all possible actions given a state.

the fitness of each genome concurrently. However, note that each round is not deterministic, therefore two identical runs of a neural net can return different fitness values. This inconsistency can be avoided by playing multiple games with each neural net (once again, this can be parallelized).

After this NEAT framework is set up with the OpenAI Retro Boxing environment, a fitness function must be selected for the algorithm. This process will have some amount of trial and error as the most obvious solution may not be practical. Furthermore, the screen holds a large amount of data that must be fed into the neural net (number and colour of pixels). Since this will greatly slow down the learning speed for every fitness function, the size of the input must be adequately reduced. For this study, gray-scaling and resolution reduction were the two methods utilized to reduce the impact of this problem. In particular, resolution reduction by a factor of 8 and 16 will be separately tested and analyzed with various fitness functions.

Several reasonable fitness functions will be tested with the NEAT algorithm using scaling down factors of both 8 and 16. For each of these runs, a population size of 30 genomes per generation will be utilized and the algorithm will be run for a minimum of 20 generations to allow for sufficient training and data collection. Data dumps will be generated periodically throughout the training process alongside a final chart showcasing the best and average species.

To implement Q-Learning, we will specify a reward function to maximize the reward given some state and action. Similar to the NEAT algorithm, several reasonable reward functions will be trained with different scaling down factors. There will be a minimum threshold set on the number of episodes run for each of these scenarios. Once again, data dumps will be generated periodically alongside a final chart summarizing the learning speed.

Data will be collected regarding the NEAT and Deep Q-learning algorithms' average win-loss ratio and training

speed. These algorithms will then be compared and contrasted with each other before a conclusion is reached.

Expected Results

It is expected that NEAT will outperform Deep Q-Learning. While Temporal Algorithms have a deeper mathematical foundation, NEAT has been shown to learn faster especially when a deterministic domain function is available (Whiteson and Stone 2006; Taylor, Whiteson, and Stone 2006). Furthermore, it has been shown that there are more difficulties in implementing a Temporal Algorithm then a Genetic Algorithm as difficulties can arise with properly estimating the function approximators (Salimans et al. 2017; Whiteson 2005).

Results

To implement the NEAT algorithm a fitness function must be selected along with a scale to the gray-scaled screen. Two fitness functions were compared along with two scales. The first fitness function subtracted the opponent's score from the agent's score and added 100 if the agent won. The second fitness function returns the square of the difference between the players' scores and then negates the value if the agent lost. The two scales that were chosen for the screen input are 1/8 and 1/16. All training was done with a population size of 30 and run for a minimum of 20 generations.

When the screen is scaled by 1/8, the NEAT algorithm is given more information and thus should be able to create a better agent. However, in the data computed there is only a marginal increase in the best score when using 1/8 over 1/16. On the other hand, completing a generation takes approximately 300 seconds on average when using 1/8, on the other hand, it only takes 70 seconds on average when using 1/16 (Figures 7, 9, 11, and 13). This is because NEAT takes longer to train with a larger number of inputs as each node must compute more information. Therefore, using a larger inputs size was found to lead to a marginal increase in score and a significant increase in the time to train.

The squared difference function was selected as it converts marginal increases in point difference to large score changes. Therefore, there is a high reward for landing punches and a high penalty for taking punches. This incites the algorithm to maximize its score and minimize its opponent's score. Furthermore, squaring the difference has the additional effect of encouraging the algorithm to maximize the difference between the agent and the opponent's score.

The difference function was selected as it heavily favours neural networks that win regardless of the score difference. With this function, any loss leads to a negative fitness value, while wins lead to significantly larger positive fitness values. This idea creates a clear distinction between winning and losing neural networks. Using this fitness function causes the NEAT algorithm to prefer networks that win, regardless of the score difference.

In the data, the linear function performed slightly better than the squared one. The linear function could win by 12 points while the squared one could only win by 6. It should be noted that both functions plateaued - reached a local maximum - after large amounts of generations (Figures 7, 8, 11, and 13). It should also be noted that the squared difference function reached its plateau much faster than the linear function. Therefore, while the linear fitness function performs better, the squared function reaches a similar performance in much fewer generations. In terms of overall training time, it took many hours to create an agent that could win, but only by a small margin.

For Deep Q-Learning the change in the score difference per action was used as the reward function and the screen size was not scaled. We note that with this configuration the reward per episode rapidly increased into mainly positive values - wins (Figure 17). Furthermore, it only took about an hour of training on one of our machines to create a consistently winning agent. However, it should be noted that this agent's wins were by very small margins.

It is noted that not scaling the input screen did provide the Deep Q-Learning algorithm with more data and a result increased the training time. However, the increased training time was not a problem as the algorithm was still able to consistently create winning agents after about an hour of training. That is why it was decided to not scale the input and instead provide the algorithm with all the full data set.

After all the models finished training, the best performing NEAT agent and the best performing Deep Q-Learning agent played 20 full games. The difference between the final scores is shown in Figure 6.

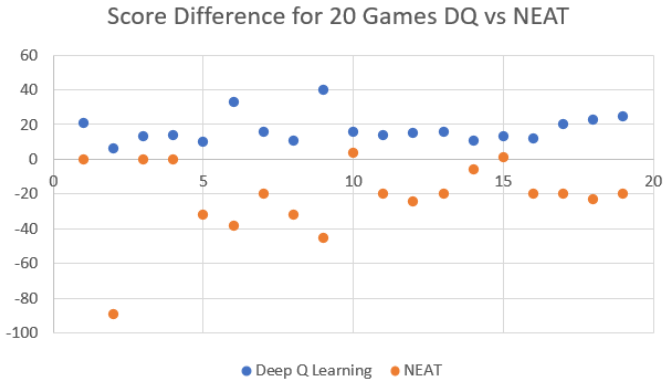

Figure 6: Score Difference between Player 1 (controlled by the AI being tested) and Player 2 (controlled by game) for 20 Games DQ (after 1000000 steps) vs NEAT (after 86 generations)

The NEAT agent performed worse than the Deep Q-Learning agent. The NEAT agent only won one game and either tied or lost the rest (1 win, 4 ties, 15 losses). It even had a monumental loss by almost 90 points! On the other hand, the Deep Q-Learning agent won every game. Furthermore, its lowest score difference was still greater than the NEAT agent's highest score difference. In fact, in our tests, it took the Deep Q-Learning algorithm less than an hour to generate an agent about as good as the best NEAT agent. Note that it took roughly 10 hours, to train each agent.

In Figure 6, there is a variance in the score differences for each agent (NEAT had a significantly higher variance than Deep Q-Learning). This is attributed to the inherent randomness of the opponent in the game which means that the same agent will never play the same game twice. This is very good for training and validation (as over-fitting the data is no longer a concern), however, it is not great for comparing agents (as they will play different games).

The NEAT implementation used for this paper was not real-time; it did not decide on the fitness value on an action by action basis, instead, it only looked at the final scores. This makes it difficult for the algorithm to distinguish between good actions, and bad or useless actions. Therefore, random useless or bad actions can appear in late species and the NEAT algorithm will deem them as good because they are in the same network as some good actions. Thus, it can be difficult for the NEAT algorithm to be able to trim the useless and bad moves. This may explain why the NEAT algorithm performs this badly relative to the real-time Deep Q-Learning algorithm. It can also explain the greater variance in the difference of the scores, as sometimes the randomness of the game's AI may be able to exploit the useless or bad moves still embedded in the NEAT agent's neural network.

The Q-Learning implementation used for this paper is real-time; it evaluates the reward function on an action by action basis. Therefore, the algorithm can better distinguish the good moves from the useless and bad ones. This makes the final agent better at playing against random opponents as it knows which moves are good and which ones are bad. This may explain the Deep Q-Learning agent's success over the NEAT agent as well as the decreased variance.

Conclusion and Future Work

The problem this paper aimed to examine was to compare the effectiveness and learning rate of NEAT and Deep Q-Learning when trained with the Atari 2600 game of Boxing. The motivation for choosing this as the game to test the algorithms on was that unlike 2D platform games like Mario or arcade games like Breakout, where the levels are unchanging and predictable, Boxing has the player face off against a "computer-controlled" opponent who is constantly performing wildly flailing, random punches. We decided to choose two relatively popular and widely-used (yet very different) algorithms to see which one was better at playing this difficult game.

We made use of OpenAI's Gym Retro platform to obtain insight into our boxing game as it allowed us to interface with the controls of the game, keep track of each player's score and, in some cases, leverage the environment's defined reward function. These features were used extensively to both train and test NEAT, using a modified version of **neat-python** (McIntyre et al.), and deep Q, using the **keras-rl** reinforcement learning library (Plappert 2016). To plot and graph the obtained data, both matplotlib and Weights and Biases (wandb) were used.

Our results showed that although both algorithms showed improvement as training progressed, only Deep Q-Learning was able to quickly able to produce games where the AI performed at a level equal to or higher than the in-game opponent. However, irrespective of the changes made in the configuration for the NEAT network and the fitness function and the time spent on training the network, NEAT was never able to consistently even tie the in-game opponent. This is evident in Figure 11 and Figure 13. Although in the best case, NEAT was able to win, it consistently had an average negative fitness value despite the scaling factor utilized. Conversely, as evident through Figure 17, Deep Q-Learning was able to quickly improve and within hours of training (200 intervals) was able to defeat the in-game opponent! The staggering difference between the two algorithms is most succinctly illustrated in Figure 6 where, despite hours and hours of training, NEAT is merely able to produce 1 game in 20 where it marginally edges out its opponent. In comparison, the Deep Q-Learning algorithm wins all of its games with the score difference going as high as 40 in one of the games!

However, the results are most likely not as impressive as they could have been. These algorithms can be rather finicky and can require a lot of hyper-parameter tuning to achieve the best possible results. Both of these algorithms typically perform better if they happen to have a lucky, random, good start. They are also at risk of getting stuck in local optima, or to undergo sub-optimal speciation and extinction.

For future work, spending more time on optimizing parameters and simply running our scripts on more powerful machines and for longer periods of time would yield more data and interesting results. It may have also been enlightening to see these algorithms face off against each other head-to-head in the 2-player mode in Boxing. And finally, testing other variations of NEAT (such as rtNEAT, hyperNEAT, odNEAT) and Deep Q-Learning (double DQN, continuous DQN, duelling network DQN), and using different video games from other genres would be a great next step.

Appendix

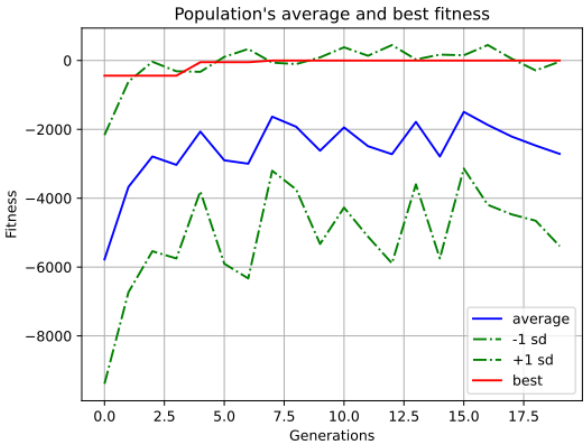

Figure 7: Square of Difference of Scores at 1/16 Scaling (Fitness)

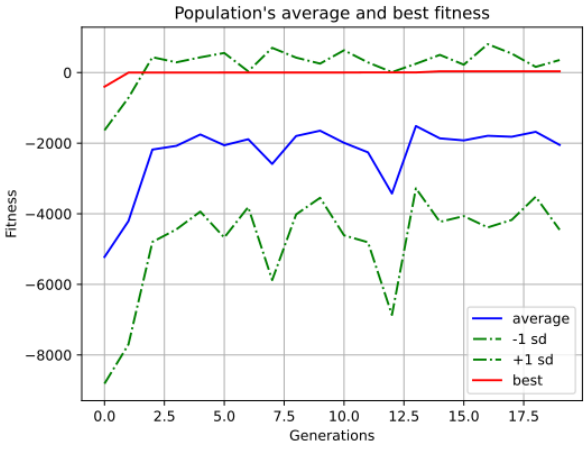

Figure 9: Square of Difference of Scores at 1/8 Scaling (Fitness)

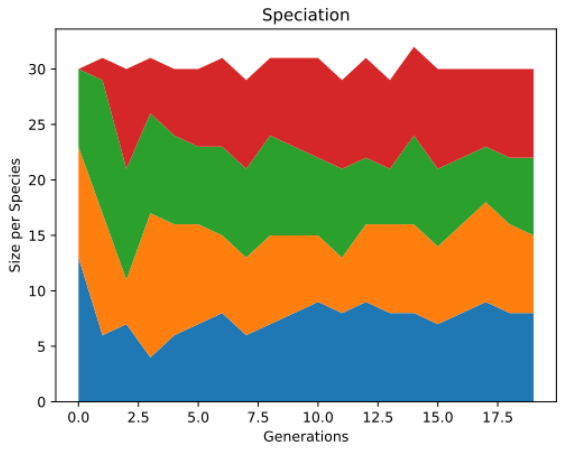

Figure 8: Square of Difference of Scores at 1/16 Scaling (Speciation)

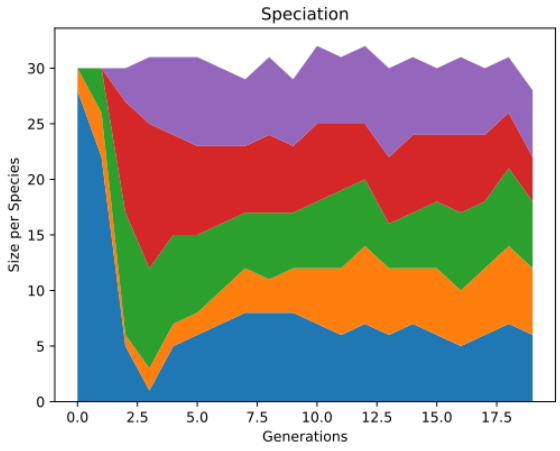

Figure 10: Square of Difference of Scores at 1/8 Scaling (Speciation)

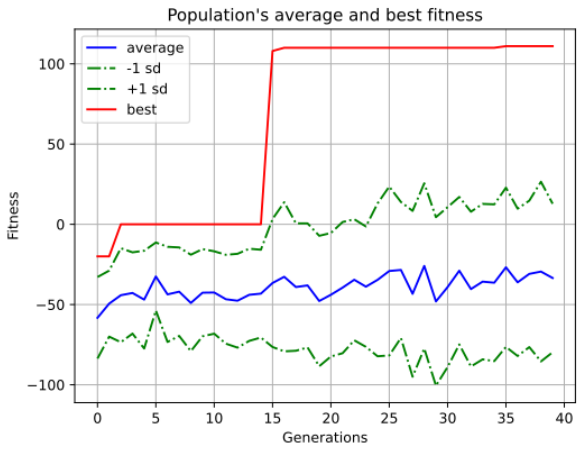

Figure 11: Difference of Scores at 1/16 Scaling (Fitness)

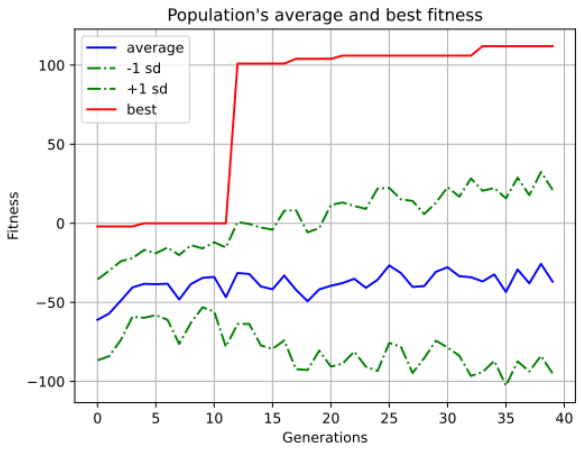

Figure 13: Difference of Scores at 1/8 Scaling (Fitness)

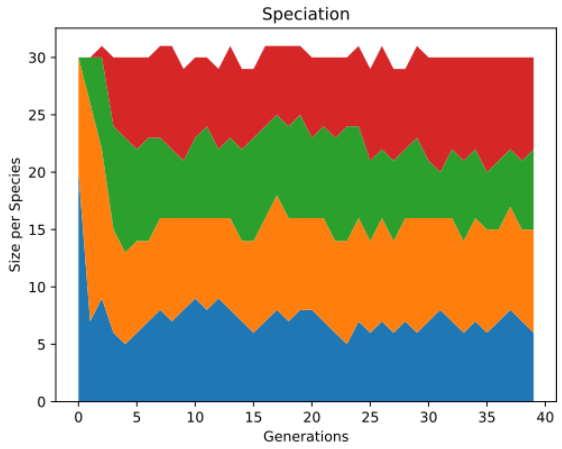

Figure 12: Difference of Scores at 1/16 Scaling (Speciation)

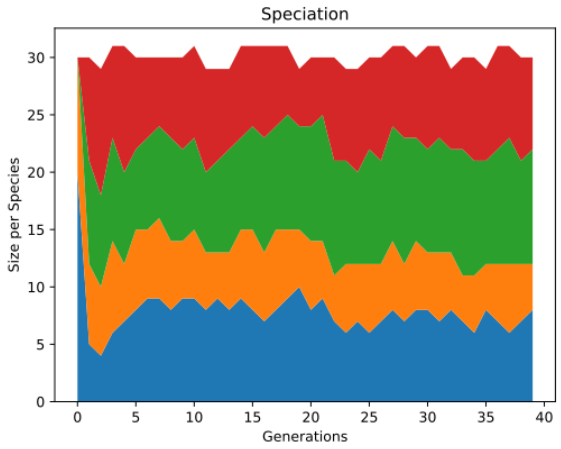

Figure 14: Difference of Scores at 1/8 Scaling (Speciation)

Figure 15: Rewards for each episode after 300,000 steps

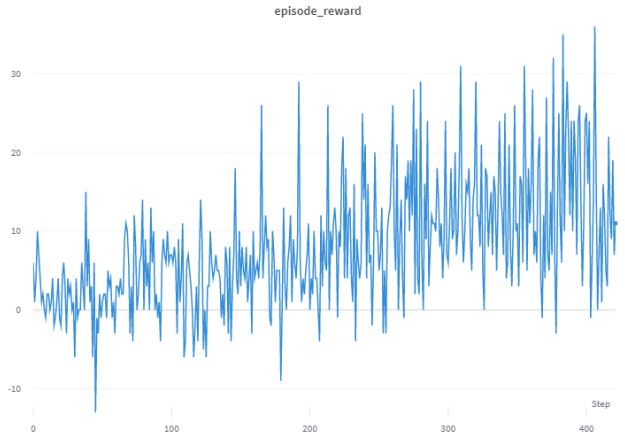

Figure 17: Rewards for each episode after 1,000,000 steps

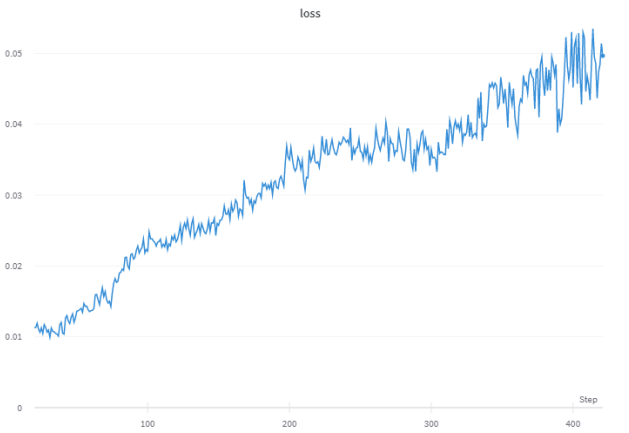

Figure 16: Loss function after 1,000,000 steps

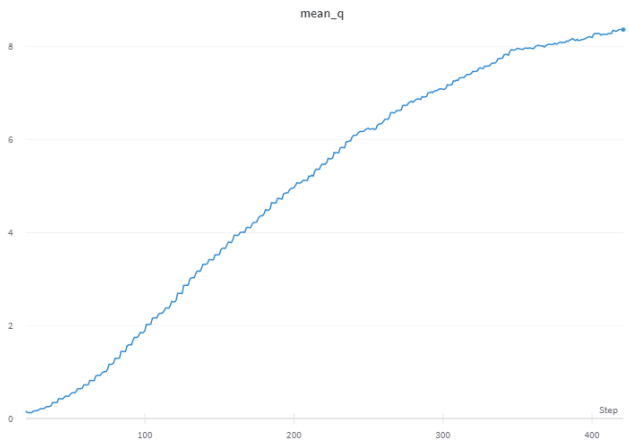

Figure 18: Mean q-value after 1,000,000 steps

References

Gomez, F. J., and Miikkulainen, R. 1999. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, 1356–1361.

McIntyre, A.; Kallada, M.; Miguel, C. G.; and da Silva, C. F. neat-python. `https://github.com/CodeReclaimers/neat-python`.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature* 518(7540):529–533.

Moriarty, D. E., and Mikkulainen, R. 1996. Efficient reinforcement learning through symbiotic evolution. *Machine learning* 22(1-3):11–32.

Ohnishi, S.; Uchibe, E.; Nakanishi, K.; and Ishii, S. 2019. Constrained deep q-learning gradually approaching ordinary q-learning. *Frontiers in Neurorobotics* 13:103.

Plappert, M. 2016. keras-rl. `https://github.com/keras-rl/keras-rl`.

Salimans, T.; Ho, J.; Chen, X.; Sidor, S.; and Sutskever, I. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Stanley, K. O., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10(2):99–127.

Stanley, K. O.; Bryant, B. D.; and Miikkulainen, R. 2005. Real-time neuroevolution in the nero video game. *IEEE transactions on evolutionary computation* 9(6):653–668.

Stanley, K. O.; D'Ambrosio, D. B.; and Gauci, J. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life* 15(2):185–212.

Taylor, M. E.; Whiteson, S.; and Stone, P. 2006. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 1321–1328.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–68.

Watkins, C. J. C. H. 1989. Learning from delayed rewards.

Whiteson, S., and Stone, P. 2006. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research* 7(May):877–917.

Whiteson, S. 2005. Improving reinforcement learning function approximators via neuroevolution. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 1386–1386.

Whitley, D.; Dominic, S.; Das, R.; and Anderson, C. W. 1993. Genetic reinforcement learning for neurocontrol problems. *Machine Learning* 13(2-3):259–284.