Evolving a Neural Network with the Genetic Algorithm to Play Snake

Anthony Beetem
Northeastern University, Massachusetts
beetem.a@northeastern.edu

Abstract

In this paper we examine the ability of a Genetic Algorithm (GA) to train neural networks for playing the game Snake. In this approach we specifically use the GA to train the weights and bias of a neural network with a given structure, a technique referred to as conventional neuroevolutionary. We start by explaining the Canonical Genetic Algorithm and the specific variants of its internal processes that we use. We run the algorithm on multiple neural network architectures, Snake enviornment features, and with sets of GA parameters and explore the performance of the GA in each case. Finally compare the results that our algorithm obtained to the results of other algorithms, and show that GAs can be effective and even produce comparable results to state of the art approaches for small neural networks up to certain performance benchmarks, but seem to not be viable for larger network structures.

Introduction

Snake is a classic arcade game where a player controls a constantly moving series of connected squares that represent a snake as they attempt to collect as many apple objects as possible. The game is played on a square grid of uniform square tiles. A snake has a constant forward velocity, but a player can direct the snake to turn its movement direction by 90 degrees to either its right or left. At any given moment the board contains one tile that contains an apple object. A player's score is increased by one point for every apple object that the snake comes into contact with, or "eats." Each time the snake eats an apple, a single new body segment is added to the snake, effectively reducing the amount of free space that is available on the grid. The game ends if the snake runs into a wall or into its own body while navigating the grid.

We chose the Snake game as our simulator for testing neural networks for several reasons. Snake provides a fully observable enviornment where all relevant state information can be very easily extracted. The action space of snake is very low, consisting of turn left, turn right, and go forward, simplifying the process of coming up with solutions for an agent trying to solve it. There are several viable sets of features that can be extracted from a state for the purpose of determining next actions, allowing for solutions of varying complexities and efficiencies. It features an element of stochasticity with its randomly generated apples, meaning that it encourages solutions that can generalize to any possible state. Its difficulty increase as a player's score increases, providing a straightforward way to compare the effectiveness of different solutions based on the scores they were able to achieve.

We chose to use neural networks as a solution for determining the optimal action for any given state that a snake could be in. Neural networks are portable, easy to set up, have several training methods that can fit many different data acquisition circumstances, and can be trained to give increasingly optimal behavior in return for increasingly intense levels of training.

In this paper, we will be training out neural networks using the Canonical Genetic Algorithm. GAs are local search algorithms that discover solutions to problems through a process that emulates natural selection (Russel and Norvig 2010). GAs depend on a random search to explore a solution space to gather several potential solutions to a problem. From there the solutions are evaluated and the best solutions are combined to ideally create better solutions. GAs offer the benefit of not having to store or have already stored training data in order to train an agent, models they train can simply be run against a simulator and evaluated based on performance. GAs have been shows to be able to find solutions to smaller neural networks efficiently (Kitano 1990).

In following sections, we will examine the viability of applying the GA to a neural network for Snake. We will explore several different neural network architectures and networks with different input features entirely to demonstrate whether they can generalize to a variety of neural networks. We will examine the effects of altering several GA factors

such as mutation rates, population sizes, and amount of training generations. Finally, we will compare conventional neuroevolution to other methods of training agents, and even to other methods of training the exact same neural networks we trained with the GAs.

Problem Description

Given an untrained neural network structure, we want to train the weights and biases so that an agent using it to make decision can maximize their utility in their enviornment. We want to train the network without previously having collected or labeled data, so we also need a simulator to observe the behavior of an agent employing a neural network in an enviornment. We will also need an evaluation function to evaluate the neural networks performance based on the agent's behavior in the simulator. In summary, given a neural network structure, a simulator, and an evaluation function, we want an algorithm that will return a trained neural network with optimal weight and bias values.

Since the GA is considered a local search algorithm, we can formalize this as a search problem. A state consists of a set of weight and bias values associated with a neural network. The initial state will consist of a set randomly generated values that represent weights and biases. The actions that a GA performs to produce new states include trading weight and bias values between sets (crossover) and probabilistically replacing values within these sets with random values (mutation). A fitness function includes a simulator and an evaluation function to determine the utility of a neural network. The objective is to find a state, or neural network, that produces the snake that can obtain the highest fitness value when placed in a random seeded snake enviornment.

The Snake Enviornment

We used a Snake environment made using the OpenAI gym interface. The snake enviornment consists of an 8 by 8 grid that has a single apple the randomly generated into a new unoccupied tile each at initialization and every time an apple is eaten. The snake object has a head and starts with 4 body segments to start. Empty tiles have an RGB value of (0,0,0), apples have (255, 0, 0), body segments have (0,255,0), and head components have (0,0,255). The snake can choose one of three choices at each timestep, turn left, continue forward, or turn right. All actions are relative to the snake's current movement direction, which differ from standard snake environments where actions turn the snake north, east, west, or south relative to the game's grid. Each time the snake's head encounters an apple, the snake gains an extra body segment. The game halts at every timestep until the agent playing the game decides on an action. After each action taken, the enviornment returns an observation consisting one overall array representing the game board, 8 inner arrays representing the columns of the game grid, and each column has 8 arrays of 3 integer rgb values between 0 and 255. The rgb value in each array indicate whether the tile is occupied by an apple, body segment, head, or nothing. An example enviornment can be seen in Figure 1.

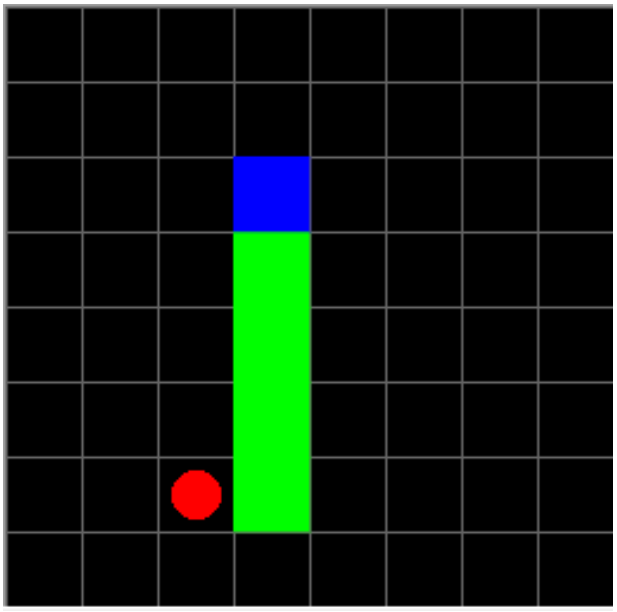

Figure 1: An example of a randomly generated snake enviornment

Neural Networks and Input Features

In order to demonstrate the GAs ability to evolve neural networks of different shapes, sizes, and even inputs, we created multiple different network architectures as well as feature extractors to fit data into them. Every neural network has the same three outputs, positive floating values indicating the probability that which action: turn left, move forward, or turn right is the most optimal choice to make given the state information given to the net.

The simplest feature extractor allows neural networks with only four input: three are binary values, 0 or 1, indicating whether a wall or a snake segment is directly adjacent to the right, left, or front of the snake's head, and one floating point value that represent the angle between the snakes head relative to the snakes forward direction. This feature set has been shown to work well in a similar project (Bialas 2019). We will refer to this feature set as DSFA (danger sense and food angle) in this paper. The neural network architecture we used for this feature set had 4 input neurons, 8 neurons in its only hidden layer, and 3 output neurons, giving us a total of 4*8 + 8 + 8*3 +3 = 67 weight and bias values to evolve.

The most complex feature set gave 3 inputs values for each of 8 directions around the snake's head. Each direction would include an integer value distance to the wall in that direction, and two binary values indicating whether a body segment existed on that path, and whether an apple existed on that path. We refer to this feature set as EDB (eight direction binary) The main structure we used for this feature set included the 24 neurons in the input layer a single hidden layer with 12 neurons, and the 3 output neurons, giving us a total of 339 weight and bias values. We also ran test with a similar architecture that included an extra hidden layer of 6 neurons right after the first hidden layer, which had a total of 399 weight and bias values to evolve.

The Genetic Algorithm

The algorithm we use to evolve our neural network is based around the Canonical Genetic Algorithm. Figure 2 depicts the Canonical Genetic Algorithm and each of its phases (Aljahdali and El-Telbany 2009). Each phase of the algorithm must be fit to work with our neural networks, and beyond that there are several variations on certain phases, such as crossover and mutations, which we will explain our choice of variants in the sections below.

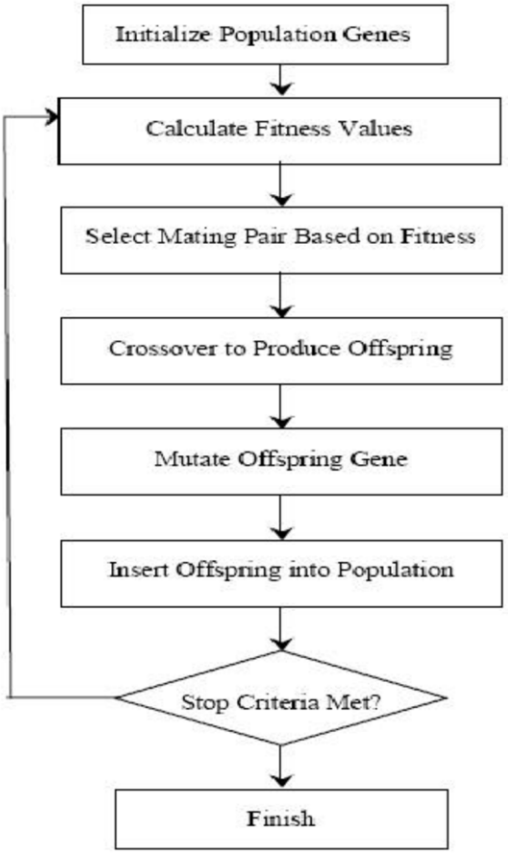

Figure 2: The Canonical Genetic Algorithm

Initializing the population

After taking in a neural network structure, the GA will produce a specified number of neural networks, each with randomly initialized weight and bias values. In our implementation we hold one neural network model at a time and simply iterate over ever weight and bias value in our neural network, setting them to random values between -1 and 1, and then saving the weights as nested arrays that can be loaded into the model once they are needed. We use the keras api to create neural network. The pseudocode for this procedure can be seen in Figure 3.

```
define Initialize-Population(neural-network, population-size):
    nn-weights = []
    for i = 0 to i = population-size:
        for layer in neural-network:
            for j in length(layer):
                layer[j] = Random-Float(-1, 1)
        weights = Get-Weights(neural-network)
        nn-weights =  Append(nn-weights, weights)
    return nn-weights
```

Figure 3: Pseudocode for initializing neural network population.

Determining Fitness

The fitness of an individual is directly related to its performance in the problem enviornment. In our case fitness is a calculation determined by the number of apples and steps that a Snake agent employing a neural network was able to obtain in a single round of gameplay. We start by giving our neural network model to a simulator that generates a snake enviornment with randomly generated apples and snake starting positions. We also give the simulator a feature extractor if the neural network is built to take in certain feature inputs. The simulator will use the feature extractor to give inputs to the neural network, which determines the probabilities that each direction, left, right, or straight, is the most optimal. The simulator advances the snake in the direction with the highest probability. The process repeats until the snake runs into a wall, runs into itself, or until there are free spaces on the board to place an apple.

We use an evaluation function suggested by a similar project, where early on in the evolution process we want to reward snakes that survive longer by avoiding collisions, give large rewards for collecting food, and give slightly negative snakes that take large amounts of steps without collecting apples (C. 2020). The pseudo code for running the simulator and the evaluation function can be seen in Figure 4.

```
define Run-Simulator(neural-network, feature-extractor):
    while enviornment.snake.isAlive:
        inputs = feature-extractor(env)
        direction-probs = neural-network.predict(inputs)
        choice-direction = argmax(direction-probs)
        env = Advance-Env(env, choice-direction)
    return env.apples-eaten, env.steps-taken

define Evaluate-Fitness(apples, steps):
    val = taken
    val += 2^apples + 500*apples^2.1
    val -= 0.25 * steps^1.3 * apples^1.2
    return val
```

Figure 4: Pseudocode evaluating a neural networks fitness.

Selection

In the selection process we choose pairs of two neural networks at a time based off their fitness so that we can combine the solutions in the cross over phase. In our algorithm each pairing produces two offspring, so the number of pairings is equal to half of the population size. We use a function that selects each parent probabilistically, where an individual's chance of being selected is directly related to its fitness value relative to the fitness values of the other networks.

In addition to selecting parents, we include the concept of elitism in our GA. Elitism allows us carry over the best solutions from one generation into the next generation without subjecting them to crossover or mutations, reducing the liklihood that the quality of solutions degrade between one generation and the next. We felt this was important for neuroevolutionary specifically since adjusting any weights of a network could have drastic and often negative effects on its performance. Elitism chooses solutions purely based off of their fitness values, where the networks with highest fitness are always chosen; this process differs from the parental selection process described above as it is not probabilistic.

```
define Elitism-Select(score-sorted-population, amt-to-preserve):
    networks = []
    for i in range(amt-to-preserve):
        networks =Append(networks, score-sorted-population[i])
    return networks

define Parent-Select(population, scores):
    parentA, parent = Probabilistic-Select(population, scores, 2)
    return parentA, parentB
```

Figure 5: Pseudocode selecting neural networks.

Crossover

Once we have selected a pair of parent neural networks, crossover is where we combine solutions to populate the next generation. This process is meant to mimic the crossover phase of meiosis in biology, so we need a way to represent our neural networks as sets of chromosomes and genes. Our neural network weights are stored as nested arrays. The top-level array contains arrays that represent layers of weights or bias arrays of a neural network. Bias array are just arrays of floating-point values. Layer arrays contain arrays of floating-point weight values. We decided that each array of floating point values, both bias and weights, would represents chromosome. Within each array or chromosome, a single floating point value represents a gene of that chromosome. The pseudo code in figure 6 demonstrates how we can combine these solutions given this encoding.

We use the two-point crossover method in our algorithm. This two-point crossover allows any continuous segment of arbitrary length of a weight array to be swapped with the corresponding segment of another network. Other crossover methods exist, such as single-point crossover where each weight array is split into two parts, one of which is swapped with another networks corresponding part. We chose the two-point crossover as it provides a more diverse range of results and can always simulate single point crossovers when the first crossover point generated is zero.

```
define Crossover(networkA, networkB):

    # convert network to an Array of Arrays of Floats
    weightArrsA = Get-Weight-Arrs(networkA)
    weightArrsB = Get-Weight-Arrs(networkB)

    for i =  0 to i = len(weightArrsA):
        crossover-point1 = Random-Integer(0, len(weightsArrA[i]))
        crossover-point2 = Random-Integer(0, len(weightsArrA[i]))
        for j = 0 to j  = len(weightArrsA[i]):
            if crossover-point1 < j < crossover-point2:
                weightArrsA[i][j] = weightArrsB[i][j]
                weightArrsB[i][j] = weightArrsB[i][j]

    # Create new neural networks from altered weights
    childNetwork1 = Network-From-Weights(weightArrsA)
    childNetwork2 = Network-From-Weights(weightArrsB)

    return childNetwork1, childNetwork2
```

Figure 6: Pseudocode Crossing over network weights.

Mutation

Mutation is the process that the GA uses to explore the solution space past what is available in its current population. In our algorithm, we iterate over each weight or bias value in a network and decide randomly based on a given mutation rate whether we should change that value to a random value between -1 and 1. Figure 7 shows the pseudocode for this process.

```
define Mutation(networkA, mutation-rate):
    weightArrs = Get-Weights(networkA)
    #itrate over each layer of the network
    for i = 0 to i = len(weightsArrs):
        #iterate over each weight array of a layer
        for j = 0 to j = len(weightsArrs[i])
            if Coin-Flip(mutation-rate):
                weightArrs[i][j] = Random-Float(-1, 1)
    return Network-From-Weights(weightArrs)
```

Figure 7 Pseudocode Mutating network weights.

Baseline Training Algorithms

In order to truly evaluate the performance of the algorithm, we included tests run by other algorithms that are capable of solving the same problem, which are currently considered state of the art. We applied the Adam optimizer algorithm and the PPO algorithm to our problem. First we trained our same neural networks and feature sets using the Adam optimizer. Next we used the PPO algorithm to train the networks using full state information from the Snake enviornment, rather than selected features.

Adam Optimizer

The Adam algorithm is an extension of SGD (Kingama and Ba 2019), and is included as a neural network optimizer for keras models. To train our neural networks with Adam, we needed training data to fit to the model. To collect training data, we employed an agent that made decisions based off of sampling the action space. We saved the observations featuures, and correlated actions taken by the random agent for every step until its snake died. We discarded any runs where an agent did not meet a minimum score of one apple. To collect data that we could compare to the GA, we gathered by running batches of games comparable to population sizes of our GA training sessions. After fitting collected data, we ran the trained models 5 times and tracked their scores. We ran another batch game to collect data from and the process would repeat for an amount of iterations comparable to the amount of generations that we trained our GA networks with. The goal of this process was to be able to track the learning curve of the algorithm that we could then compare to the learning curves generated from the GA sessions.

PPO

To explore a much difficult variation of our problem, we wanted to train an agent that took in the full state information rather than features of the environment and would require a very large neural network. We used the Proximal Policy Optimization algorithm (Schulman et. al. 2017). PPO is a reinforcement learning algorithm that was developed by OpenAI and was integrated to work very well with most OpenAI gym enviornment, such as the Snake game we use.

Results

In the following graphs we plot a single data point for each generation, which represents the maximum number of apples that the neural network with the highest fitness was able to obtain.

Genetic algorithms depend heavily on randomness, which could potentially imply that the results generated could be heavily skewed by luck. In order to order get an idea on how much data could change between identical training sessions, we ran our EDB 24 input neural network with constant parameters three separate times. We evolved the networks over a period of 100 generations with populations sizes of 100 each, elitism rates of 10%, and mutation rates of 10% to each. Figure 8 below shows the highest number of apples that any individual neural network was able to achieve in its population during each generation of the process. From these results we can see that there is in fact a lot of variation in the shapes of the learning curves and the in the amount of time it takes each session to reach certain milestones For example, the first trial took 67 generations to produce a neural network that could score 10+ apples, but the third trial took 18 generations to reach the same milestone.

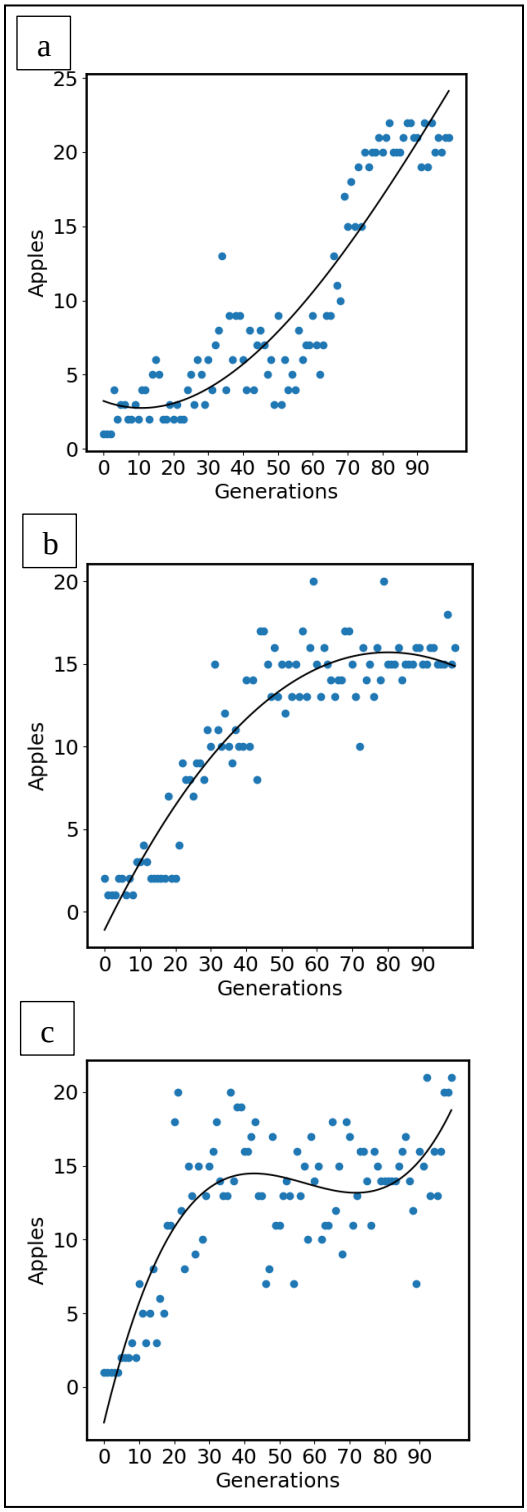

Figure 8: Three EDB sessions run for 100 generations with identical parameters of size 100 population, 10% mutation rates, and 10% elitism rates

Population size

Next, we explored the effects that altering the population size could have on the ability of a model to evolve using the GA. For each session we held all parameters constant except for population size. The population sizes were 30, 60, and 90. All mutation rates were 5%, and elitism was 10% of the population size, and each population was evolved over 100 generations.

The graphs in figure 9 show the number of apples that the highest scoring snake of each population was able to obtain. The session with the largest population size of 90 seemed to evolve at the fastest rate. The sessions with the lowest population of 30 seemed to take significantly longer to start showing any progress. Early on the smaller neural network did appear to have relatively high scoring neural networks. It is most likley the case that these nets were lost from the population just due to poor luck. Several strategies that evolve, especially early on, will work well if the snake is given favorable starting conditions, but it may fail completely if it is given put in other random circumstances. An example of this is a neural network that will never tell the snake to turn left starting out against the right side wall and getting eliminated immediately; this is a strategy that can obtain high scores, but due to the random generating conditions, a individual networks like this will likely be eliminated. In the lower population sessions, the “genes” of these snakes will be scarcer and more prone to extinction due to unlucky circumstances.

Finding better solutions also takes a greater amount of generations on average in smaller populations. This is because there is a lower chance that all the genes needed for an optimal network are present in the initial population and will need to appear through random chance mutations.

The benefit to having smaller populations is that they are much faster to run for sequential implementations of the GAs such as ours. The amount of time our algorithm takes to run depends mostly on the simulation phase, since every neural network gets to play an entire game each generation.

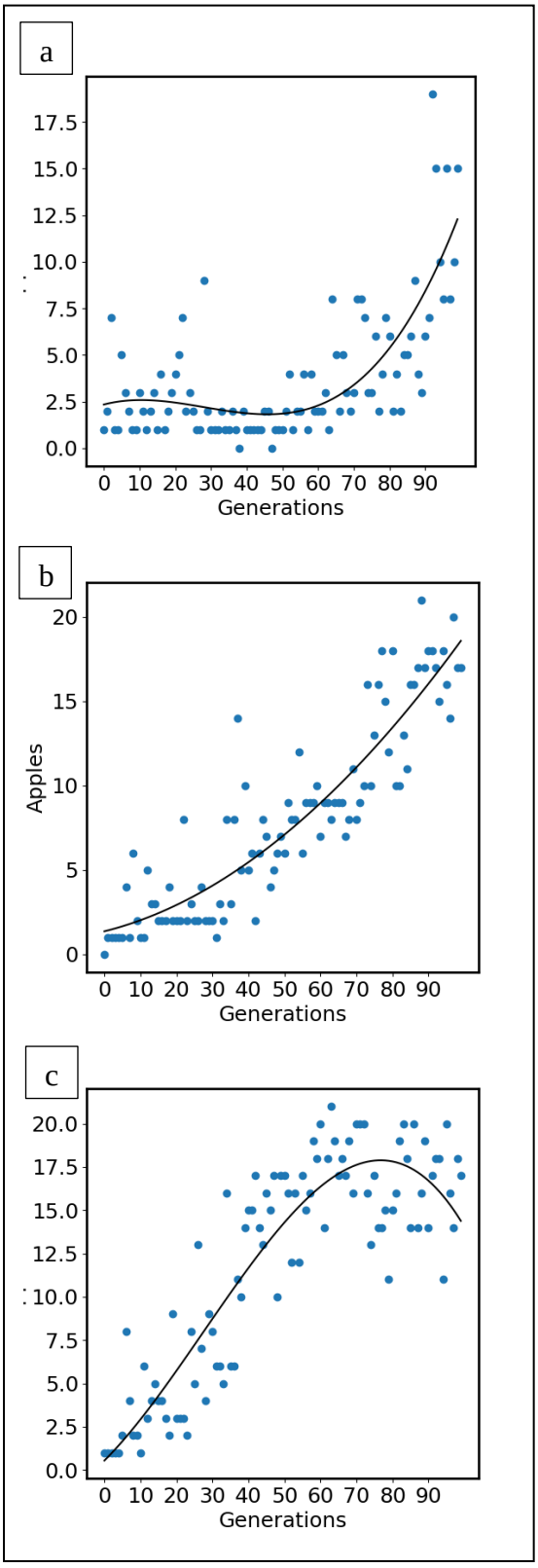

Figure 9. Separate EDB Sessions with varying population sizes a) 30 individuals, b) 60 individuals, c) 90 individuals. Mutation 5%, elitism 10%

Mutation rates

We wanted to examine how adjusting the mutation rate would affect the evolution rate of the neural networks, and to find if setting the rate too high or low could have detrimental effects on a session.

We used the EDB feature set with the (24,12,3) shaped neural network architecture for all sessions. We gave every session a population size of 100 and evolved them over the course of 100 generations. They all had an elitism rate of 10% of the population. Each of the 5 sessions was given a different mutation rate, one of 0%, 1%, 10%, 25%, and 50%.

Figure 10 below shows the maximum scoring networks for each generation in each session. We see that in figure 10a, the session with a mutation rate of zero was able to achieve scores of up to 12 apples with crossover operations on the initial population alone, but was unable to discover any better solutions and the progress halts very early on. Figures 10c and 10d had mutation rates of 25% and 50% respectively, and they saw little to no progress throughout the trials. This is likley because the mutation rate was far too high to allow incremental improvements and is starting to resemble a random search. Figures 10b and 10c with mutation rates of 1% and 10% respectively showed relatively steady learning rates that made steady progress over the 100 generations. The 1% mutation rate seemed to allow for the most stable growth and was also able to reach the highest scores out of any rate presented. Further tests would need to be run to determine if a 1% mutation rate is consistently a better option that a 10% rate for the given enviornment and network.

Elitism

We ran three sessions varying only in elitism rates to examine whether it was able to prevent solutions from degrading over generations. We ran every session with a population of 100, for 100 generations, all with a mutation rate of 5%. The sessions have elitism rates of 0%, 1% and 10% respectively. Figure 11 below shows these results.

The figures show that having no elitism generally seemed to slow the learning rate in figure 11, since the maximum number of apples obtained was 10. Figures 11b and 11c with rates of 1% and 10% respectively showed similar learning trends. The model trained with a 10% learning rate was able to obtain higher scores, but due degree of randomness that the GA is subjected to, we would need more evidence to decide what rates are best to employ with our enviornments.

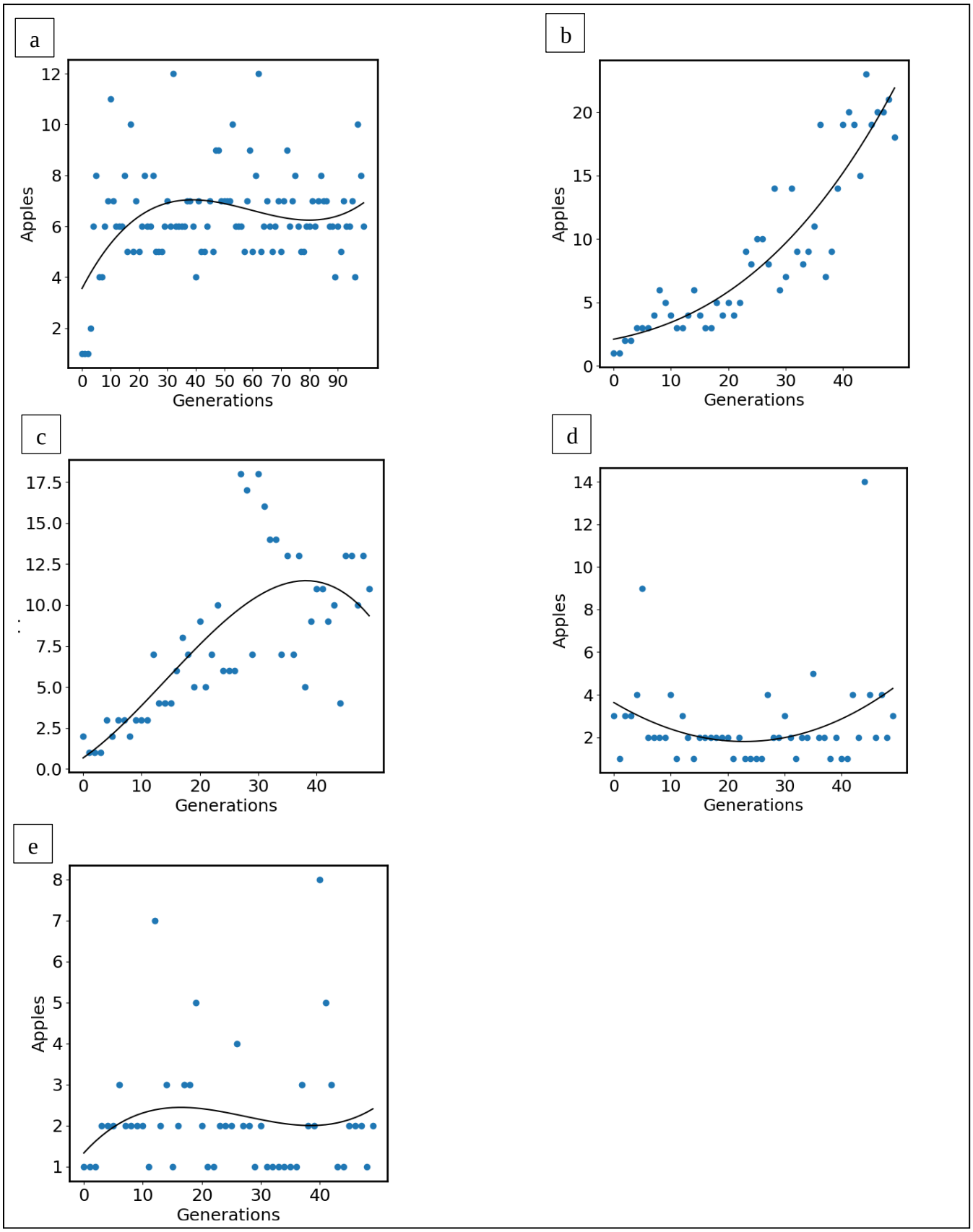

Figure 10: Five separate EDB sessions with varying mutation rates a) 0%, b) 1%, c) 10%, d) 25%, e) 50%. Population: 100, elitism 10%.

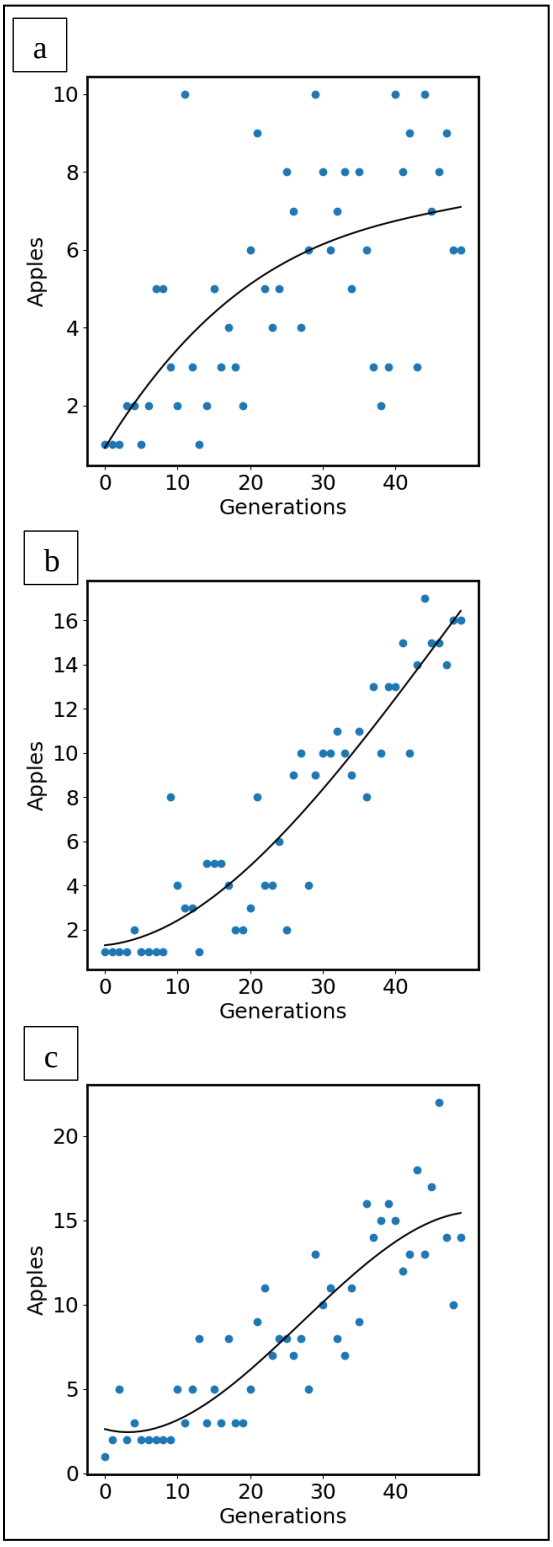

Figure 11: Three EDB sessions with separate elitism rates a) 0% individuals, b) 1% individuals, c) 10%. Population size 100, mutation 5%

Varying architectures

Our sessions so far have featured architectures made with a single hidden layer, so here we tested a neural network with two hidden layers to see if the learning rate was greatly altered in any way. Figure12a shows the results of this new architecture, while Figure12b shows the results for previously shown run for reference that has the same GA parameters. Both sessions were run with populations of 100, for 100 generations, with mutations rates of 10%, and elitism rates of 10%. The two sessions shown only differ in their neural network shapes, the first in figure 12a has a shape of (24,12,6,3), and the second has the shape of (24,12,3).

Both figures show comparable learning rates and reached similar scores. This could be due to there not being a large enough difference between the amount of weights that had to be evolved in each network, since they only differed by 60 weight and float values. Adding additional tests with networks that either include more layers or wider layers could potentially expose greater differences in the ability of the GA to train a network.

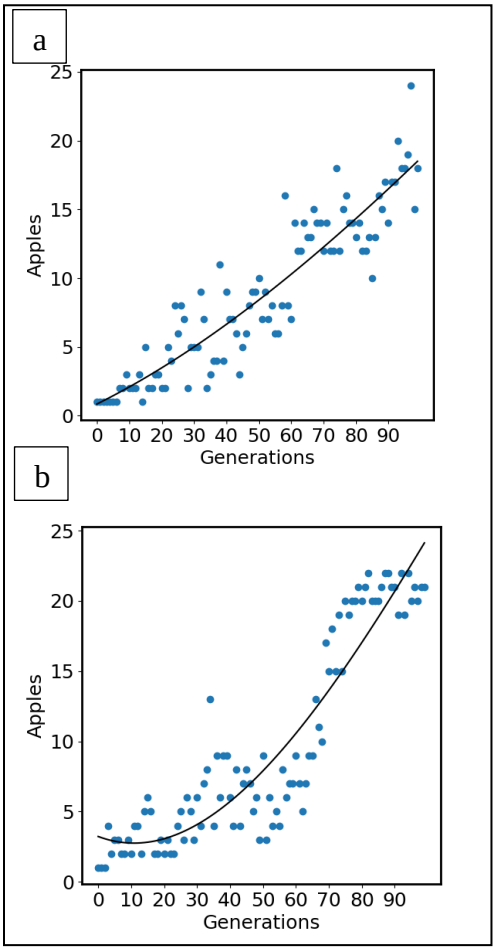

Figure 12: EDB sessions with differing neural network architectures a) has a shape of (24,12,6,3), b) has a shape of (24,12,3).

DSFA

Here we show the results of applying the GA to a completely different neural network with the DSFA feature set. We include one session with our standard protocol of a population of 100, elitism rate of 10%, and mutation rate of 5%. We also include a session with a population of 30 to examine if this relatively simpler neural network can learn with lower population sizes.

Both figures in Figure 13 show that this new feature set and architecture allows these networks to evolve to get high scores very quickly compared to the other results we have seen so far. The network with a population of 30 seemed to have fewer stable scores and achieved a lower score than the population of 100 session. The population of 30 with this feature set seemed to show progress much earlier than the population 30 session run for the EDB feature set. This could be since the much simpler neural network architecture is easier to train and has a smaller solution pace to explore.

Adam

We ran 100 sample games at a time and collected any game that had a score of at least 1 apple and used that to fit our models using the Adam optimizer algorithm in Figure 14. Figure 14a shows the training of the EDB feature set neural network with a shape of (24,12,3). Figure 14b shows the DSFA features set neural network with a shape of (4,8,3). New data was collected and fit 100 times to simulate the generation scheme of a GA. After each fitting the model was run five times and the highest score was tracked on the shown figures.

The results show that Adam was able to train the neural networks to up to scores of about 20 when being limited to running the same amount of simulations as similar GA sessions. Figure 14a is meant to be compared to the results seen in figure 8, as they use the same features, neural network architecture, and ran the same amount of simulations. Figure 14b is meant to be compared to the results of Figure 13a, since they share the same input features, neural network architecture, and amount of games run.

Adam required data to be stored in order to train, and the limitation we imposed on the amount of simulations we could use for training likley cause the slow learning process for the EDB models. Only about 5% of games run would obtain the one apple needed to be included with the training sets, so the algorithm was able to show impressive result considering the low amount of data they were given.

Simulation run for the Adam algorithm employed completely random agents and did not last nearly as long as simulations run in later stages of the GA sessions, which reduced the time cost significantly.

The DSFA run showed steady quick improvement early on but seems to plateau. This can likley be addressed by increasing the minimum score requirement for the data collection phase, but that would only work if we were allowed to run many more training simulations. Overall for these smaller networks, at least within the first 100 generations, the GA shows better learning performance per simulation that has to be run, but the Adam training did suffer from having a very small pool of training samples to use which could potentially be addressed with more complex data collection methods.

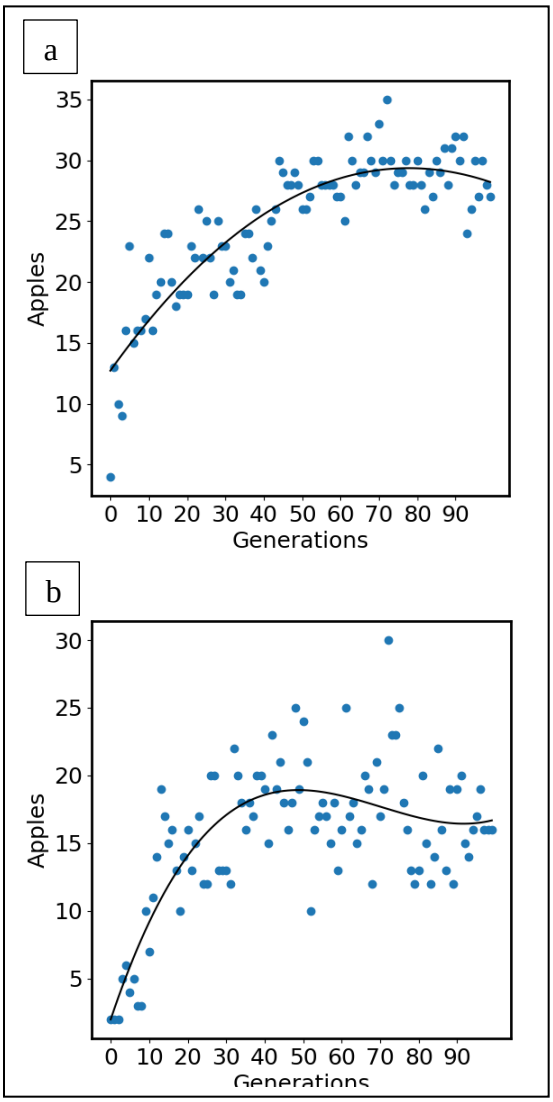

Figure 13. Sessions with DSFA feature set, each differing in population size a) population 100, b) population 30. Elitism 10%, mutation 5%

PPO

The PPO algorithm was trained using the full state information, which was an input size of 8*8 grid units with 3 color channels per grid unit for a total of 192 inputs. We were not able to figure out a way to train the RL algorithm in batches as we had with the Adam algorithm so we do not have a learning curve to show, but the final model was able to reach scores of up to 14 apples. The PPO algorithm was trained over 4 million samples over the course of about approximately 11 hours. To give a fair comparison, we attempted to create GA session that took in the full state data. This resulted in a very large neural network that took in 192 inputs, and the overall shape was (192, 64, 3). Either due to poor neural network architecture choice or due to the inefficiency of GAs training very large neural networks, we were unable to produce any results that suggested the networks were improving at all, even after 1000 generations at a population size of 100. Other papers have noted that GAs are very inefficient when working with larger neural networks compared to other training methods (Kitano, 1990), so we moved on from exploring this path further.

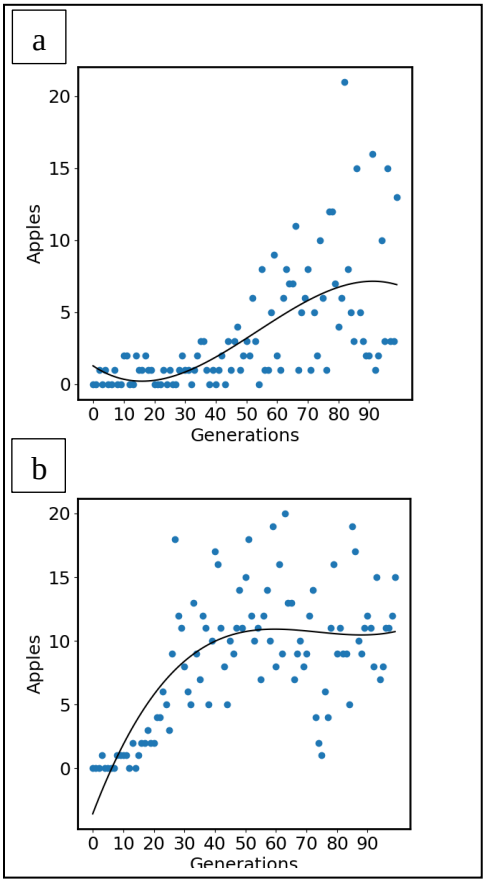

Figure 14. a) EDB, and b) DSFA sessions trained using the Adam optimizer algorithm

Conclusion

Genetic Algorithms seem to be effective at evolving small scale neural networks on a variety of feature sets for the game of Snake. We were able to consistently produce neural networks that demonstrated rational behavior in a Snake game enviornment. The GA could start producing neural networks that demonstrated some rational behaviors within the first few generations. We explored many of the parameters of the GA and their effects on a training session. Finally, we compared our results to the results obtained by other common training methods that could be applied to the same Snake enviornment. The result shown suggest that GAs may be a viable approach training networks to other problem that can be solved with smaller neural networks.

References

Kitano, H. 1990. Empirical Studies on the Speed of Convergence of Neural Network Training Using Genetic Algorithms. In *Association for the Advancement of Artificial Intelligence-90 Proceedings* 789-795.

Bialas, P. 2019. Implementation of artificial intelligence in Snake game using genetic algorithm and neural networks. In *CEUR Workshop Proceedings* vol-2486, 42-6

Russel, S.; Norvig, P. 2010. Beyond Classical Search. In *Artificial Intelligence A Modern Approach*, 126-29. Pearson Education, Inc

C. (2020, May 03). AI Learns To Play Snake. Retrieved December 16, 2020, from https://chrispresso.io/AI_Learns_To_Play_Snake

Kingma, D. P., Ba, J. A. 2019. A method for stochastic optimization. In arXiv 2014. arXiv preprint arXiv:1412.6980, 434.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. In arXiv 2017. arXiv preprint arXiv:1707.06347.

Aljahdali S.; El-Telbany, M.. 2009. Software Reliability Prediction Using Multi-Objective Genetic Algorithm. In IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2009. 293-300. 10.1109/AICCSA.2009.5069339.