Machine Learning and Video Games Using Snake Game with Resilient Back Propagation

Philip Akkerman
dept. of Computer Science
Brock University
St Catharines, Canada
pa13xb@brocku.ca

David Hasler
dept. of Computer Science
Brock University
St Catharines, Canada
dh15pd@brocku.ca

***Abstract*—This paper explores the use of machine learning in video games using the classic Snake game and a neural network trained with Resilient Back Propagation (R-Prop). The three primary objectives of this paper were to determine which inputs to use when training a neural network to play Snake, to determine which training parameters resulted in the most effective learning, and finally to build a neural network which can play Snake better than in previous research. First, It was found that a small amount of meaningful inputs were more effective than large amounts of general data as inputs to the neural network. Secondly, a set of training parameters were found to deliver the best training results. Lastly, a neural network was created which vastly out-performed neural networks created in previous research.**

I. Introduction and Problem Definition

A. Introduction

This research paper is about finding the best way to train video game agents with neural networks, creating a neural network that is able to learn how to play the Snake game using R-Prop, and determining which training parameters contribute to the most efficient learning with R-Prop. In this paper we will explain the background of the classic game of Snake and neural networks using R-Prop, and describe the evaluation function used to autonomously train the neural network. Furthermore, we will discuss the setup of our experiments to train and test the neural networks. We will then discuss the results of training and testing and come to conclusions about the three main goals above. Other studies will also be compared when evaluating the performance of the neural network.

B. Problem Definition

Artificial intelligence (AI), also known as machine learning, can be used to solve many types of problems. Most often, it is used in classification problems, but the field is growing to encompass other practical applications as well. One such application is in the video game industry, where machine learning can be used to replace the traditional game AI that developers must hard-code themselves. this application of machine learning has the potential to save time and money by allowing AI to learn the game on its own without the need for exhaustive situation management through traditional coding.

Some challenges are present however when trying to apply machine learning to video games. As opposed to traditional classification problems, where inputs are often easily defined, determining which features to select in video games can be very difficult. In the case of a game like Snake game, inputs could be a grid of pixels, object locations, sensory information from the snake's perspective, among others. Determining what information to feed a neural network has significant impact on its performance. This research aims to not only solve this problem in the case of Snake game, but also to determine learning parameters that yield the best results, and to create an agent which can play the Snake game autonomously, thereby demonstrating that machine learning can be used effectively in video games.

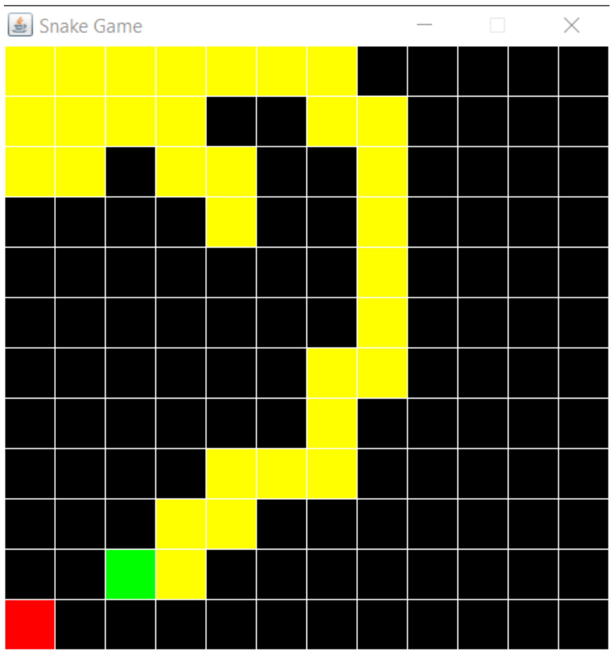

Figure 1. Snake Game

II. Background

A. Snake Game

Snake is a classic video game which has been around for decades. It is available on mobile devices, internet browsers, and classic video game consoles. The game is played by controlling a snake as it travels around a fixed-sized gameboard.

The purpose of the game is to collect as many apples as possible while avoiding the walls and the snake's body. If the snake runs into its own body or a wall, the game is over. The game begins with the snake facing east in the middle of the gameboard, with an apple randomly placed on an empty space in the gameboard. The gameboard used in this research had a size of 12 by 12 tiles, for a total of 144 tiles. Since the snake initially takes up three tiles, the maximum score possible (and the win condition for the game) is a score of 141, measured in apples eaten.

Snake game serves as a good test bed for AI because of its simplicity and its increasing difficulty over time. Each time the snake eats an apple it gets longer, so it becomes more difficult to survive, as the snake can get trapped by its tail. The snake moves one space at a time each frame, and the player (or AI) can choose to turn the snake to the left, to the right, or to go straight. This means that there are only three outputs for a neural network that plays Snake. The inputs are more difficult to determine, and will be discussed later in this report.

B. Neural Networks

Neural networks is one form of artificial intelligence. A neural network uses a layered network of neurons with connections containing weights and biases to each consecutive layer. The first layer in a neural network is the input layer, this contains the input values to the network. For example, in this project the network was given the values achievable by going left, straight, or right, overall an input of 12 neurons was used because for each direction the snake could go it needed to know the 4 evaluation heuristics listed in the section below. The next layers in the network would contain the hidden layers with the hidden neurons and different architectures were used during testing. The last layer of the neural network is the output layer, this layer contains the outputs of the network and as an example this network used 3 outputs: left, straight, or right.

Once the network is built, the connections of weights and biases are randomized for the initial network, and then the network feeds forward the input neurons to reach the output neurons. This calculation of feeding forward from layer to layer is done by taking the sum of the weights times the inputs plus a bias, which is put through a logistic function to normalize the output between 0 and 1. Once the network calculates the final layer, it then must use a learning method to teach itself based upon the error. The error is calculated as the difference between the network's move decision (represented as an array of three values between 0 and 1) and the evaluation function's move decision. For this neural neural network we used R-Prop to handle the learning. The R-Prop learning rule takes the error of the network to alter the weight and bias connections of the network until it reaches the maximum epochs set. R-Prop uses an increase and decrease rate to determine the rate at which the neural network modifies weights and biases. It also uses a maximum and minimum step to determine the maximum and minimum change it can make when modifying weights and biases, so it will bound the step within the maximum and minimum step. It updates weights and biases based on the direction of the current gradient compared to the previous gradient, depending on the direction of the previous change. Eventually the R-Prop function reaches the max epochs where hopefully it is close to the global minimum of the error function of the network.

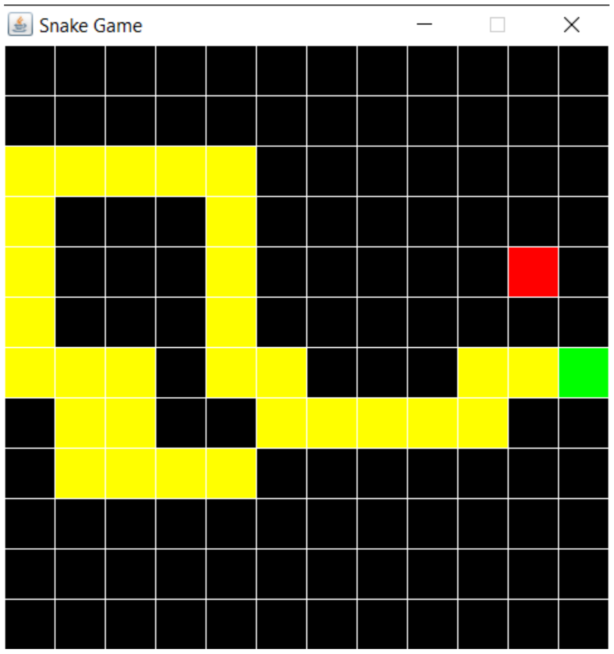

Figure 2. Snake Splitting Board in Half

C. Evaluation Function

In order to provide supervised learning to the R-Prop training algorithm, an evaluation function was made to determine the optimal move from a given gameboard using four heuristics. The first heuristic used was the number of available spaces the snake could reach, the second heuristic was the Euclidean distance between the snake's head and the apple, the third heuristic was whether the snake collided with an obstacle, and the fourth was whether an apple was eaten. In order to determine the best move using these heuristics, each of the three moves was attempted (left, straight, right) and the heuristic values were returned. To improve performance, the function was allowed to look one extra move into the future, to avoid splitting the available spaces in half as shown below in image 2. This meant that for each evaluation, the function searched up to nine possible moves, calculating the available spaces and distance from apple after each move.

One issue that occurred frequently was that of infinite loops, where the function would repeat the same series of actions over and over, resulting in an unending game. To solve this, we implemented a hunger system, where the weight attributed to the distance to the apple was increased by 0.01 after each move, and reset upon eating an apple. This was enough to modify the snake's behaviour over time and break from these infinite loops.

In this heuristic function, a higher score was better. The available spaces heuristic was added to the score with a weight multiplier of four. This value was determined in initial testing, and was increased to four to make the snake prioritize keeping itself alive. The second heuristic, the distance to the apple, was subtracted from the score with a weight of one, multiplied by the hunger multiplier. This penalized the snake for being too far from the apple for too long. The third heuristic, death by collision with a wall or itself, was implemented by replacing the score with -1,000,000. Eating an apple increased the evaluation score by 10.

In the case of giving inputs to the AI (also knows as input rows), a similar system was used as above. Instead of applying weights to the available spaces heuristic however, a normalized value between 0 and 1 was given representing the available spaces from the move. Since the maximum available spaces was 141, the number of spaces was simply divided by 141 to come up with the value given to the AI. The distance function was also divided by the maximum Euclidean distance possible to normalize between 0 and 1 (although the hunger factor could increase it past 1). Collision (death) and eating an apple were given to the AI as binary data, where 1.0 meant a collision, and 1.0 meant an apple eaten for their respective input nodes. To reduce on input size, the optimal values from each of the three directions (left, straight, right) were given as inputs to the neural network. This meant that the input size was 12 (4 heuristics for each direction) instead of the 36 if all 9 future possibilities were fully considered. Values from future nodes were only considered if they did not result in a collision.

III. Experiment Setup

The experiments began by deciding which parameters to evaluate learning on the Snake game. A total of 12 experiments were selected with one parameter modification each. The default parameter setup was as follows:

1) Learning Rule: R-Prop
2) Number of runs per experiment: 10
3) Maximum epochs: 1000
4) Maximum moves per epoch: 1000
5) Hidden layers: two, with 100 and 50 nodes respectively
6) Increase Rate: 1.2
7) Decrease Rate: 0.5
8) Maximum Step: 50.0
9) Minimum Step: 0.0001
10) Random Seed: same for each game: 0

In the neural network background section, most of these parameters were already explained, the difference we hope to see is that we will find an optimal set of network parameters for an autonomous agent using neural network R-Prop to learn how to play a game of snake. One variable not discussed so far is the random seed, this variable allows the AI to train consistently on the same game or to train on a different game each epoch. The list below shows the parameter modifications for each experiment.

Experiment 1 -	Increase rate lowered to 1.1
Experiment 2 -	Increase rate raised to 1.6
Experiment 3 -	Decrease rate raised to 0.7
Experiment 4 -	Decrease rate lowered to 0.2
Experiment 5 -	Maximum step raised to 100
Experiment 6 -	Maximum step lowered to 10
Experiment 7 -	Minimum step raised to 0.01
Experiment 8 -	Minimum step lowered to 0.000001
Experiment 9 -	Random seeds each game
Experiment 10 -	No hidden layers
Experiment 11 -	One hidden layer with 100 nodes
Experiment 12 -	Two hidden layers with 1000 and 500 nodes

After all of the training experiments were completed, the neural networks that had the lowest error from each experiment were saved. These neural networks were then tested in 1000 games, to determine how each network performed when evaluated based on game scores instead of similarity to the evaluation function, which is how they were trained. The goal was to determine the relationship between training and testing, as well as to find the neural network which plays the Snake game the best.

IV. Results and Discussion

After all the experiments were completed, the resulting error values were compiled into graphs so that the error of each run could be viewed across the epochs. Some of the runs did not complete as many epochs because the stopping criteria of an error change of 0.00001 was hit, which indicated that that run was not changing enough anymore. For example, in experiment 2, the fifth run stopped at epoch 753 since there had been no significant change in error between epoch 752 and 753.

The first two experiments used different increase rates for the R-Prop function. The graph in figure 3 shows that experiment 1 was able to learn more quickly and able to reduce the error further than that of experiment 2. This means that by using the smaller increase rate, the neural network was able to learn the Snake game quicker for most runs. Another observation is that the graph of experiment 1 in figure 3 appears to deviate less than the graph for experiment 2. From these experiments alone it is not possible to conclude which

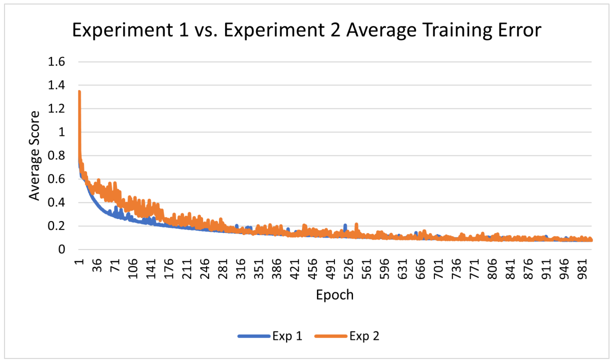

Figure 3. Neural Network Experiment 1 vs Experiment 2 Error

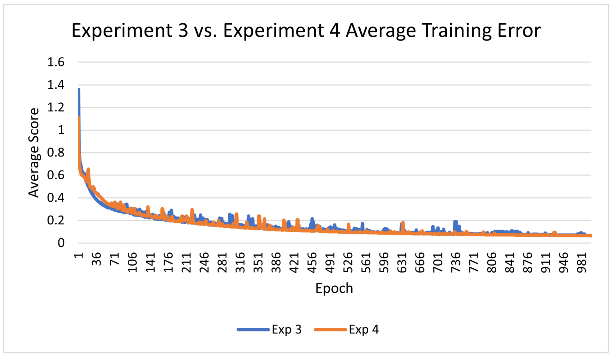

Figure 4. Neural Network Experiment 3 vs Experiment 4 Error

parameters for the neural network provide the best results because they both converge at the same point, but the runs for experiment 2 did provide lower final error values on average. During testing, however, as seen in figure 9, experiment 2 performed far better than experiment 1, with average scores of 5.1 and 2.3, respectively.

The next two experiments tested decrease rates of 0.2 and 0.7. Looking at the graph in figure 4, the third experiment tends to show a smaller deviating graph. The fourth experiment shows outliers at certain points throughout the run, but overall lower error averages earlier on. Although the runs were similar in learning, the fourth experiment had quicker learning mid-way through training when using a decrease rate of 0.7 instead of 0.2. Experiment number 3 obtained a higher average score in the 1000 testing games than experiment 4, even though experiment number 4 had a lower average error rate. This result means that when using a decrease rate of 0.7 the neural network was able to better learn the game of Snake.

The third set of network parameters tested was increasing and decreasing the maximum step allowed in R-Prop. Figure 5 shows the experiment 5 vs experiment 6 average error graph over 10 runs each. One observation from the graph is that the deviation seems to be smaller in the fifth experiment when the max step is increased to 100. This means the error appears to be more consistent for several runs and that the learning is also more consistent. It is also noticeable that the sixth experiment has a smaller error by the end, meaning it was able to learn more from the evaluation function. In the 1000 testing games played, both neural networks achieved similar average scores, so there was no distinct conclusion to make from changing the max step parameter.

Our fourth set of experiments was assessing the affect of raising and lowering the minimum step allowed for R-Prop. In figure 6 there is a noticeable difference at the end of the graph which shows that the error when decreasing the min step to 0.000001 is lower than for experiment 7, which raised the min step to 0.01. The graphs followed the same trend until an error of 0.3 was achieved. This indicates that with a min step that is too high, the function is unable to fine-tune its weights and biases in order to lower the error past a certain threshold, in this case 0.3. The results of the 1000 testing games show that experiment 7 had a significantly higher average score, at 11.3 compared to 5.9 for experiment 8. In this case, a lower error did correspond to higher game performance.

The ninth experiment used fully random seeds when training instead of the consistent game used to train in the other experiments. To visualize the error difference in training, experiment 9 was compared to experiment 1, because the first experiment had the closest to default parameters. The graph in figure 7 shows that the error is clearly smaller for the experiment 1. This is because it is much more difficult for the neural network to learn consistently on random games. But this also means the neural network will be able to get a better variety of moves and different situations, which could improve generalization. When viewing figure 7, the error goes down at the same rate in the beginning, but after 30 epochs the first experiment's error begins to lower faster than experiment 9. At first the conclusion would be that the neural network performed poorly when trained on a random seed. When testing the networks on 1000 random games however, experiment 9 was able to achieve the best score by a wide margin, see figure 9.

The final experiments were done to test different hidden layer architectures for the neural network. When first looking at the graph in figure 8 it is easy to see that the error of experiment 12 jumps frantically throughout the graph not showing much consistency until the end. Though the twelfth experiment is not very consistent, it does end with the best error overall

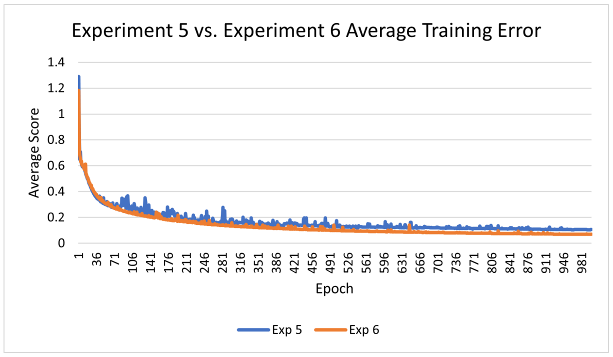

Figure 5. Neural Network Experiment 5 vs Experiment 6 Error

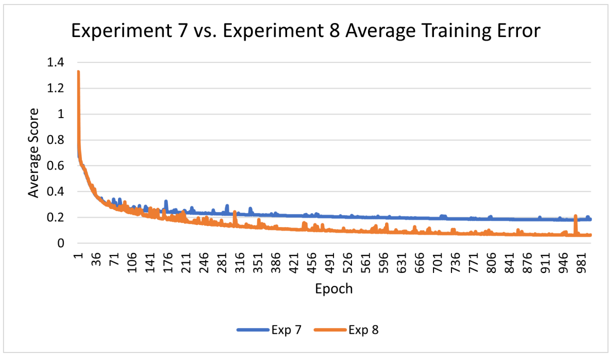

Figure 6. Neural Network Experiment 7 vs Experiment 8 Error

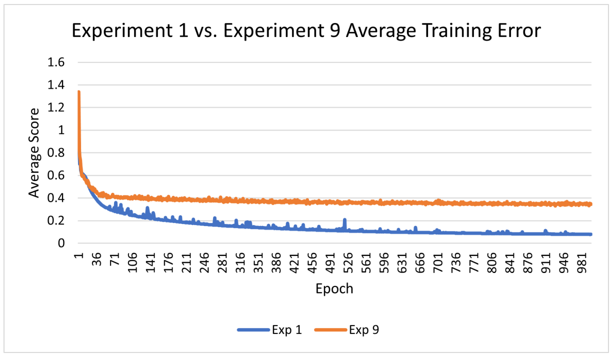

Figure 7. Neural Network Experiment 1 vs Experiment 9 Error

out of any of the experiments. It is worth noting however that due to the high number of nodes used in experiment 12 and how long it took to perform a single training run, only one run was able to be performed, which would contribute to the visibility of outliers. Another observation to be made is that experiment 10 has the highest error, experiment 11 the middle amount of error, and experiment 12 has the least error, which corresponds to the increasing number of nodes and layers in the three consecutive experiments. This means that the neural network performed best when increasing the number of hidden layers to 2 and also when the number of hidden neurons was increased to a large amount, 1000 and 500 for each layer respectively.

Since the training results showed that experiment 12 had the lowest error, during we expected it to perform the best in testing. However, this was not the case as shown in figure 9 where the box plot shows a comparison of the experiments' scores over 1000 games. As seen in the box plot, the experiment with the best mean score was experiment 9 when a random seed was used to train every game. One reason that experiment 12 did not produce the best results is that it was not trained on random seeds as in experiment 9. On the other side of the box plot is the network that got the worst score overall, which is when the network had a reduced increase rate of 1.1. The network expected to perform the best was experiment 12, but in the end it an had average performance of snake games on random

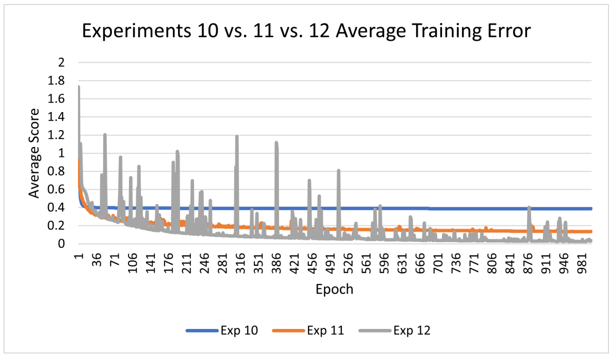

Figure 8. Neural Network Experiment 10 vs Experiment 11 vs Experiment 12 Error

seeds over 1000 games. This demonstrates that similarity to the evaluation function, as measured by error, does not always determine game playing performance.

To determine the best network, an assessments of the best scores was made. Clearly the highest score in the box plot of figure 9 is experiment 9, the random seed experiment which yielded a high score of 55. The second highest score was achieved in the sixth experiment with a score of 47. The sixth experiment was one where the maximum step was reduced to 10. It had one of the lowest final errors of any of the networks at 0.06034, which indicates that a more bounded learning step in the R-Prop function contributes to more accurate learning. It was also able to reach a comparatively better score in Snake than other networks, although the relationship between training error and game performance is not consistent among all networks. The lowest high score of an experiment was in the second experiment where its highest score was only 23 apples eaten. Experiment 2 also had one of the lowest final error values, at 0.06189, demonstrating once again that the relationship between learning error and game performance is inconsistent.

The observations of the testing experiment results above show how the different network and learning parameters affect the learning rate and how well the network can play Snake. The best network was found by training using random games, giving the neural network more variety of games while training, which lead to better generalization and therefore better game scores. Even though experiment 9 had one of the worst error values when it finished training, as shown in figure 7, the experiment had the best results of 1000 test games, as shown in figure 9. One takeaway is that the neural network should be trained on random seeds during training because it is able to yield better results. This makes sense because the variety of games used to train the network allows the network to update the weights and biases for many situations. It also allows the neural network to prepare for the testing stage because it is able to play random games better than the rest of the networks which were trained on 1 predefined seed.

Let us compare this model to the refined DQN in the article ”Autonomous Agents in Snake Game via Deep Reinforcement Learning” by Wei et al, where the average score for the refined DQN was 9.04 [1]. The average of 1000 runs on the best performing network using R-Prop, experiment 9, was 22.7. When comparing this value to the refined DQN model, it is clear that the neural network model with R-Prop, which was trained by an evaluation function using random games to learn, achieved a better average score and was able to play the game of Snake better. Another comparison is to see which of these networks provided the best score in any one run. The neural network from experiment 9 returned a best score of 55 apples eaten. The refined DQN by Wei et al however was only able to get 17 as the best score. Therefore, since both the average and best scores of the neural network using R-Prop and trained by an evaluation function was greater than the refined DQN model, the neural network created in this project was able to play Snake better than in the previous research.

Some factors which could have resulted in better scores is that in the "Autonomous Agents in Snake Game via Deep Reinforcement Learning" article, the authors used a random initial placement of the snake in the game. This could cause problems during training because it could place the snake facing towards the wall while starting against the wall. The snake would then go forward and immediately loses without having a chance to train. So when creating our Snake game implementation, it was decided that an improvement from their version of the game would be placing the snake in the center location each time, just as how it is usually played in most other implementations of Snake.

Not included in the experiments above was the process that was undertaken to come to the decision of using the 12 heuristics as inputs for the neural network. Initially, it was our intention to give the snake only basic information such as the entire gameboard of 144 tiles, represented by binary data to determine if a tile was occupied or not. In initial training attempts, it was found that the neural network was incapable of training on this type of data, and the error values would not decrease past 0.6 at best. We had already made a heuristic evaluation function which could play the Snake game effectively, achieving scores up to 81 apples eaten, as shown in figure 10. Since this evaluation function worked by looking one move into the future and considering the number of spaces available and distance to an apple, we decided to give similar information as inputs to the neural network so that it could make more informed decisions. This proved to be an effective technique, as the results above demonstrate.

V. Conclusions

Many conclusions were made throughout this study. A conclusion we were able to make early on was that the neural network needed meaningful inputs in order to learn to play

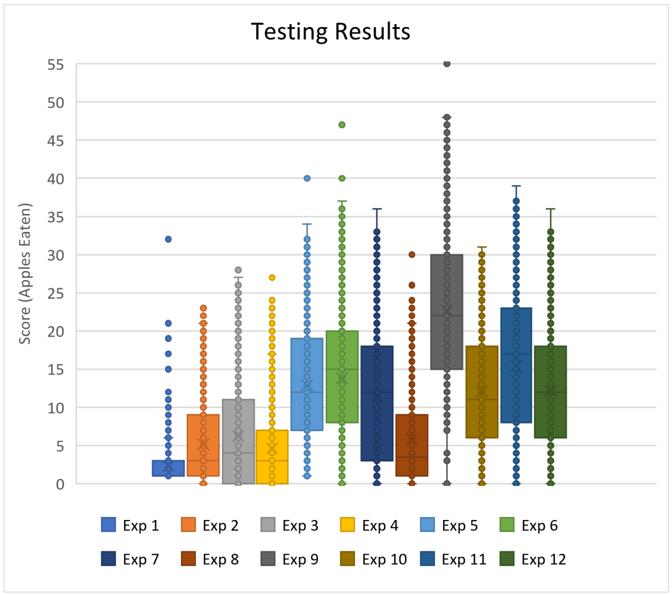

Figure 9. Neural Network Experiment Test Scores

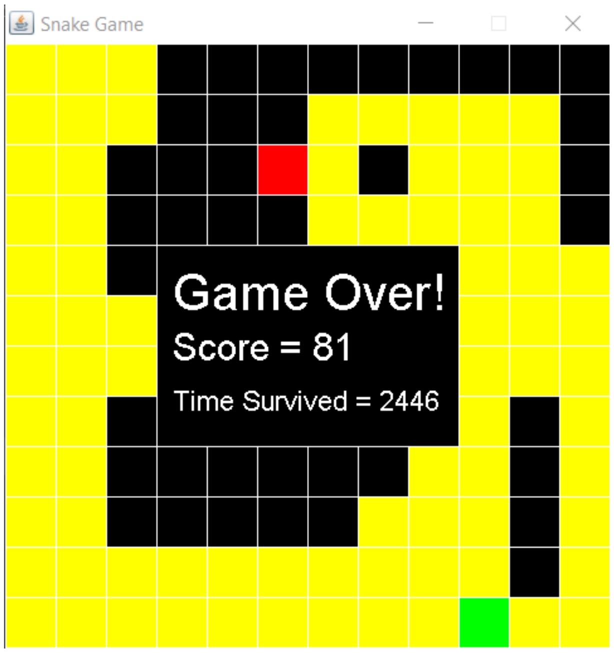

Figure 10. Evaluation Function High Score

Snake, which is why the evaluation function was used. Using raw data of the cells in the gameboard of the Snake game was ineffective, and the neural network was unable to learn. Instead, inputs retrieved from the evaluation function and given directly to the neural network allowed it to better learn whether it was best to move left, straight, or right. This can be applied to using AI in video games in general; if a decision can be made to provide a more processed and useful form of input data, it should be considered as opposed to large amounts of generalized input data.

Another important deduction made was that the evaluation function itself still played better than even the best performing neural network. The best score recorded was 81 by the evaluation function, seen in figure 10, which is much larger than the best score received using the neural network, which was 55. Since the training was done on the evaluation function, it was expected that the neural network would not perform better, but we expected the neural network to have a closer high score. In future research, a useful experiment would be to use a reinforcement learning technique such as neuroevolution to improve upon an already trained neural network, which could refine the model and provide even better results.

The last key result found was that the training results did not translate to testing results. Experiment 12 showed the most promising results since it yielded the lowest error of any network trained, but during testing, experiment 12 did not perform well compared to experiment 9, as seen in figure 9. The hypothesis as to why this happened is simple, because experiment 9 was able to train on random seed games it was able to generalize random games better. However, experiment 12 did not have this luxury and this is a main reason why it did not achieve the expected outcome. Throughout these training experiments, the optimal parameters were found for each situation, based on how well the neural networks were able to reduce error. The optimal parameter options included a higher increase rate, smaller decrease rate, smaller max step, smaller min step, consistent training set, and largest setup of hidden layers and nodes, in this case: 1000 and 500 nodes in two layers. 9.

To recap: a neural network was built to learn with R-Prop on random Snake games using supervised learning from an evaluation function. The neural network was able to decide what the best decision was going forward at each new move. This autonomous agent also played significantly better than the refined DQN model in the article ”Autonomous Agents in Snake Game via Deep Reinforcement Learning” by Wei et al [1]. For future research, it would be valuable to be able to create a more efficient neural network which is able to surpass the evaluation function, or even achieve a perfect score.

References

[1] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, “Autonomous agents in snake game via deep reinforcement learning,” in *2018 IEEE International Conference on Agents (ICA)*, 2018, pp. 20–25. DOI: 10.1109/AGENTS.2018.8460004.