İHSAN DOĞRAMACI BİLKENT UNIVERSITY

CS 464 Introduction to Machine Learning
Project Final Report

Section: 01
Team Number: 08

Team Members:

Pelinsu Acar	21602073
Emre Uludağ	21602492
Furkan Burak Mutlu	21601598

Introduction

The purpose of the project was to develop a neural network that can play a Snake game from scratch. Since there is not any target and information of the best movement for all states in the game, traditional machine learning algorithms are not proper for this problem. Therefore, implementing a Deep Reinforcement Learning algorithm utilizing Keras Tensorflow is the best option. The game was based on an existing implementation of a Snake game which is coded in Python and it tries to teach playing with one hour of training. The original game is extended to play on a base of 28 x 26 screen. The evolution of the snake is based on rewards, which are given credit about the snake's actions and helps it to learn. During this project, the aim is to achieve the highest score. For this purpose, we have tried different methods that have a variety of parameters and scores that get from them. The results and scores will be discussed according to different variations.

Problem description

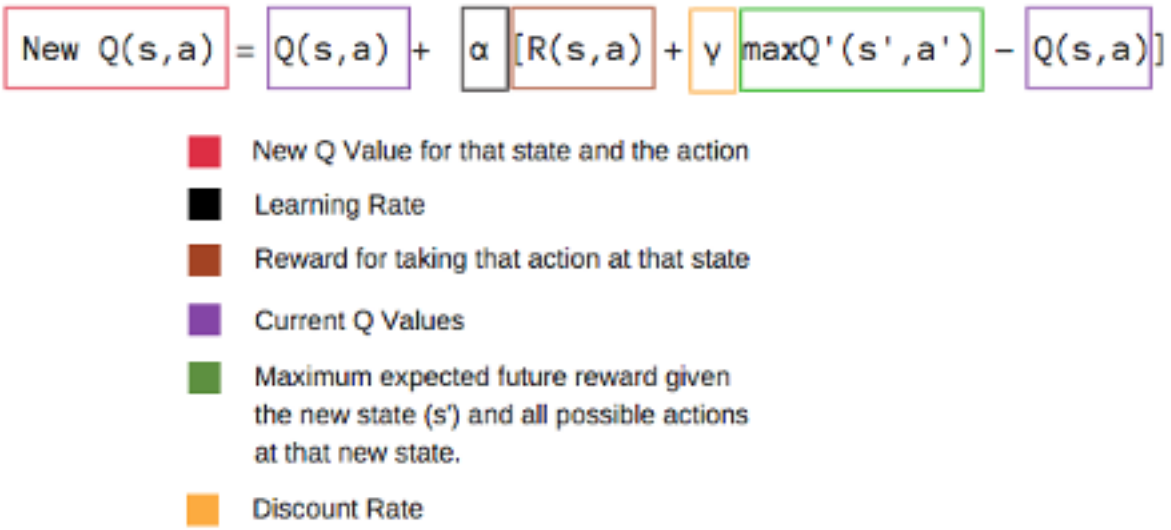
The problem is whether can snake game can play by itself without any human interrupt. To do so, Deep Reinforcement Learning will be used. The learning algorithm has 3 layers that are connected to decide the next move of the snake. These layers have an extra input layer which contains 11 different inputs about the snake's next move. After the inputs go through the 3 layers of the Deep Reinforcement Learning algorithm it concludes at one of the possible 3 outputs. Snake can choose to go straight, turn right, or turn left. For the first 75 episodes, the snake goes with random outputs where the learning happens. On the other remaining episodes, the snake will take action according to the algorithm mentioned before. The score of the current episode is shown on the screen with the game, also the output decisions shown on the screen as well to show how the snake is playing. The primary goal of the project is to survive the snake without hitting the walls, independently from the bait. The next goal is to teach the snake how to eat and make it grow. In the beginning, the snake is only trying to learn and making random decisions which leads to low scores. Snake gets positive feedback when it eats the bite and gets negative feedback when it crashes into the wall or itself. When the snake starts to play with the algorithm after 75*** episodes, the score increases dramatically. The aim is to achieve the highest score as 75 after random moves.

Methods

When the snake makes an action, our game gives a positive or negative reward considering if the action was good or not. We determined three types of rewards for our game. If the snake crashes something (wall or itself), it will get -10 and if the snake manages to eat the food, it will get +10. Lastly, if the snake makes 100 actions without eating any food, it will get -10. We created this third reward to motivate the snake more to the food and prevent it from taking unnecessary actions without going to the food.

We have 11 states which are Boolean variables to represent the observations at each iteration. 3 of them consider the location of a possible danger close to the snake (straight, right and left), 4 of them consider the direction of snake's movement which can be right, left, up or down and the rest four states are for the food location according to the snake (right, left, above or below).

Since we have 11 different inputs for our algorithm, it would be difficult to update Q values based on the Q table consisting of 11 rows. Therefore, we decided to use Deep Q Learning that increases the opportunity of Q learning since it uses a Deep Neural Network instead of a table. In this policy, Belmann equation updates the Q values for each action.



[1]

To determine the best action, we should take the biggest Q value. In general, the algorithm applies the following procedure:

1. The Q value is determined randomly with the start of the game.
2. The algorithm understands the current state.
3. Then, it chooses an action considering the current state. Although these actions are determined randomly at the first stages of the training, as the system starts gaining experience, it begins to rely more on its Deep Q Network and gets rid of randomness.
4. For each action that snake performs, the game gives a reward and the system passes to the next state updating the Q value. Furthermore to create the data that is going to be used for training, the algorithm stores the current state, next state, its movement and the game's situation (finished or not)
5. 3. and 4. steps are repeated unless the game is over.

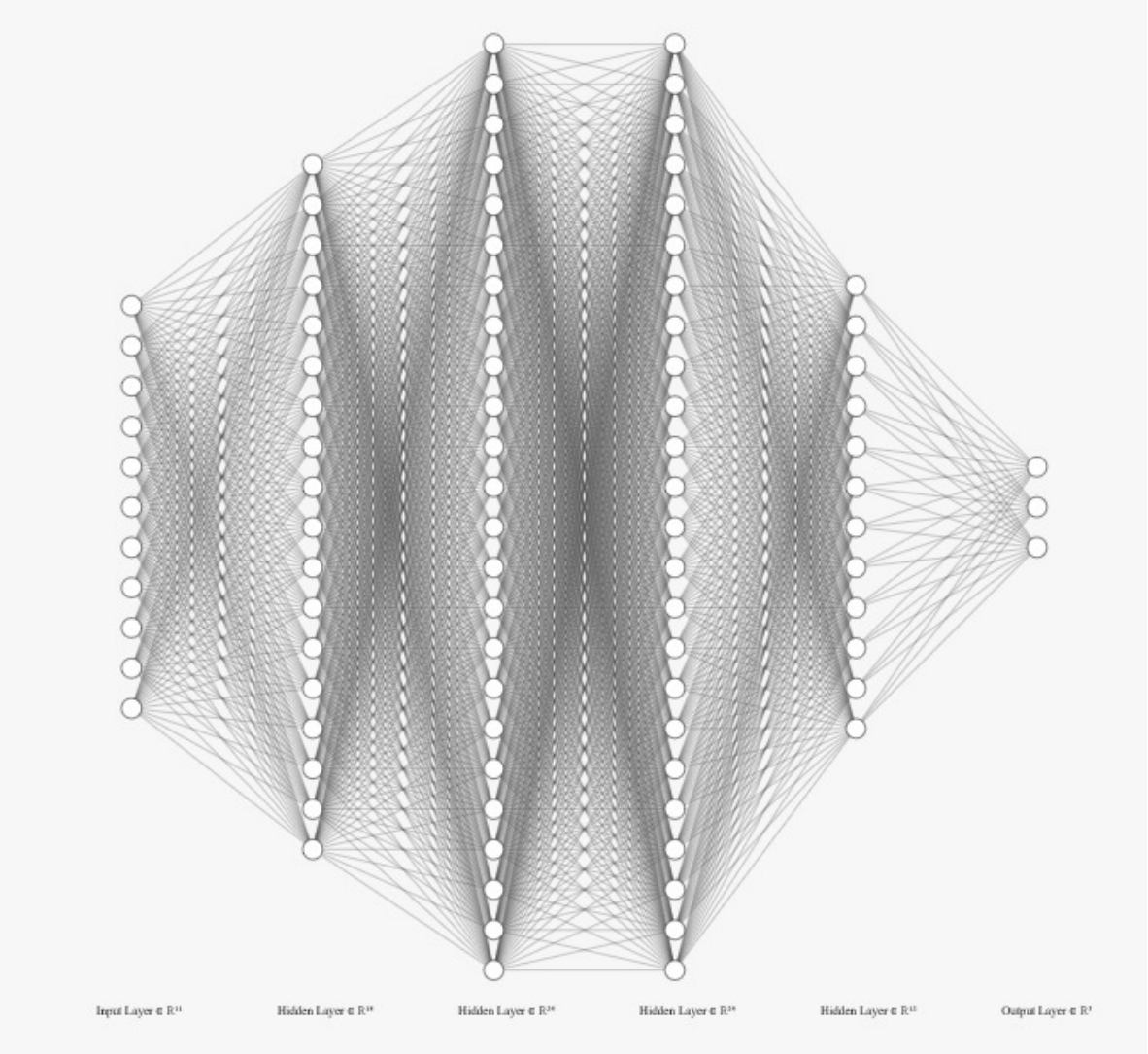
As it mentioned in step 4, we have data that consists of several states and movements and is used to train our neural network. This data creates a memory which has a size of 2500. Since every action is connected with the next state and there can be high correlation among these sequence of memories, our model is at risk of being affected by this correlation. Therefore, each time we sampled these data and chose a small batch that has a size of 500 randomly with a method called Replay Memory. We also created another memory which contains only the last experiences to ensure that our neural network is trained with the more up-to-date information since the long term memory doesn't ensure this by selecting a batch randomly. This short memory contains only the last ***??? In total, game consists of 200 episodes and the first 75 of them are used for training meaning that the snake chooses its movements randomly to gain experience rather than considering the old states.

The aim of our neural network is to optimize the action for each given state by maximizing the possible reward. To measure how close the prediction is to reality, we used Loss function. Then the neural network is trying to reduce this loss by minimizing the difference between its prediction and the target.

$$loss = \left(\mathrm{r} + \gamma \max_{a`} \hat{Q}(s, a`) - Q(s, a) \right)^2$$

We have a mean square error loss function used for regression tasks from Keras Sequential model. Since we sum the error values from all the nodes, we take the square of the differences between the prediction and the target to prevent the cancellation of positive and negative terms. In this way, MSE gives more emphasis meaning higher weights to the extreme values by squaring them and in our case these outliers are important and should be detected by the algorithm to learn better.

Our deep neural network has 3 hidden layers of 18, 24 and 12 neurons respectively and the learning rate is … as an origin. It is not fixed and decreases to …. at the end of the 200 episodes. We tried different learning rates and found the best one for our model as an origin. There are 68 nodes and 954 edges. The schematic of the network is below:

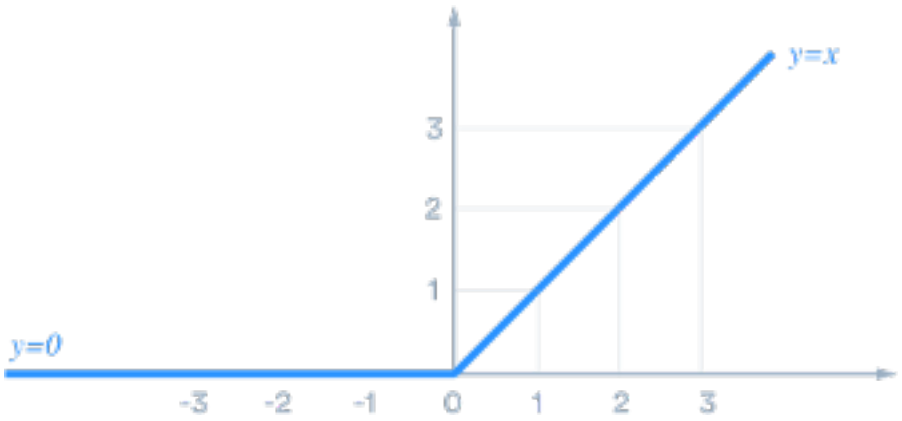


[2]

To optimize the learning rate, we used Adam (Adaptive Moment Estimation) optimizer which is a first order gradient based Stochastic optimization method from the Keras Sequential model methods. In addition to being an efficient and straightforward method, it is suitable for problems with large scale data and noisy gradients. It can work well with only little tuning of hyper parameters and it requires relatively low memory. In the classical stochastic gradient descent algorithms, a unique learning rate is maintained for each weight updates and it is stable throughout the training. On the other hand, Adam optimizer changes the network weights repetitively according to the training data. Therefore, the learning rates are computed separately for different network weights.

The network takes these 11 different states as inputs and return three outputs that represents three different actions (move right, move left and move straight) for the snake.

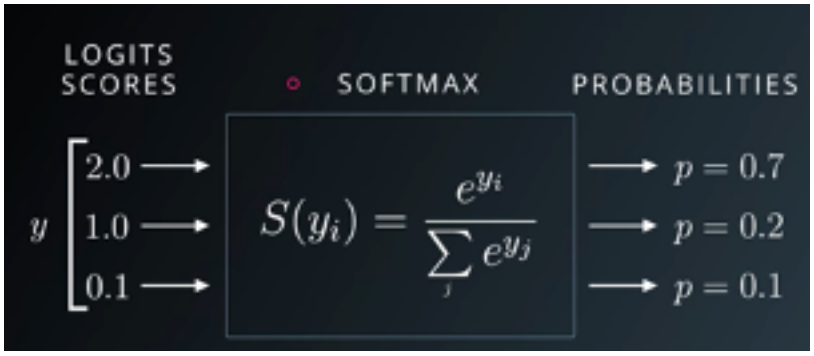
For the activation functions of the layers, we used ReLU (rectified linear unit) for the first two hidden layers, and linear*** for the last one. ReLU can be defined as y = max(0,x) mathematically and its graph is below:



[3]

Since ReLU is zero for negative values and linear for positive values, it is cheap in terms of computation because of the simple math and it's sparsely activated. The training or run time is less and it converges faster. Another advantage of ReLU is that the slope does not saturate as the x becomes larger and the vanishing gradient issue seen in other activation functions does not exist in ReLU. Lastly, the maximum level of error information can be transmitted during the back propagation through the network, since its gradient is always one.

For the final layer output, we used a softmax activation function to turn each output into probability values between zero and one.



[4]

As it seen from figure …, Softmax function calculates the exponent of outputs and divides the sum of these exponents to normalize the each output so that the output vector meaning all probabilities adds up to one.

Results

For this section, the results will be analyzed on the change of the following parameters:

- Learning rate————-
- Out Activation—————
- Gamma—————
- Epsilon decay————çalışıyo hala
- Number of layers—-buna bakıcam
- Layer sizes———pelin bakıyor

Different results obtained from the exchange of these data will be displayed as a score-episode, learning rate-score graph.

These figures game stats are same expect the batch size and out activation. The figure 2 and 3 are the example of difference between linear and softmax activation.

*	Epsilon Decay	Memory Size	Batch Size	Gamma	Out Activation	Average Score
Figure 6	1/100	2500	500	0.9	Linear	17.2
Figure 7	1/100	2500	500	0.9	Softmax	13.9

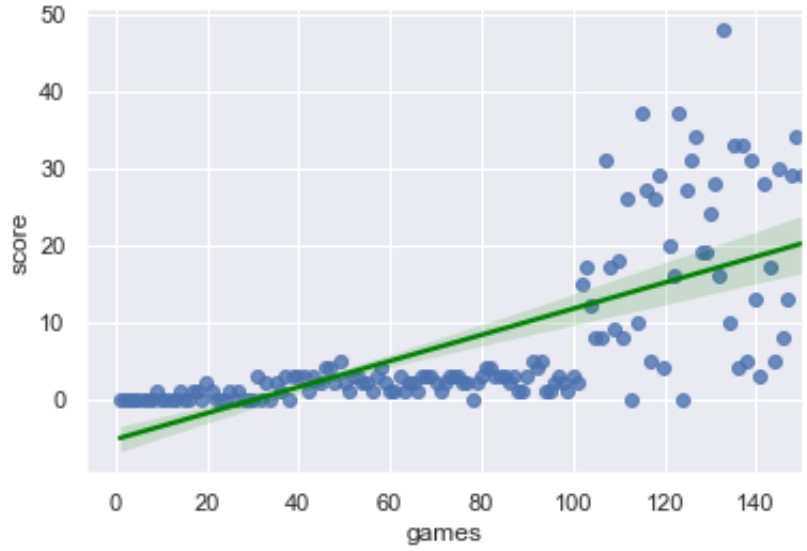


Figure 6

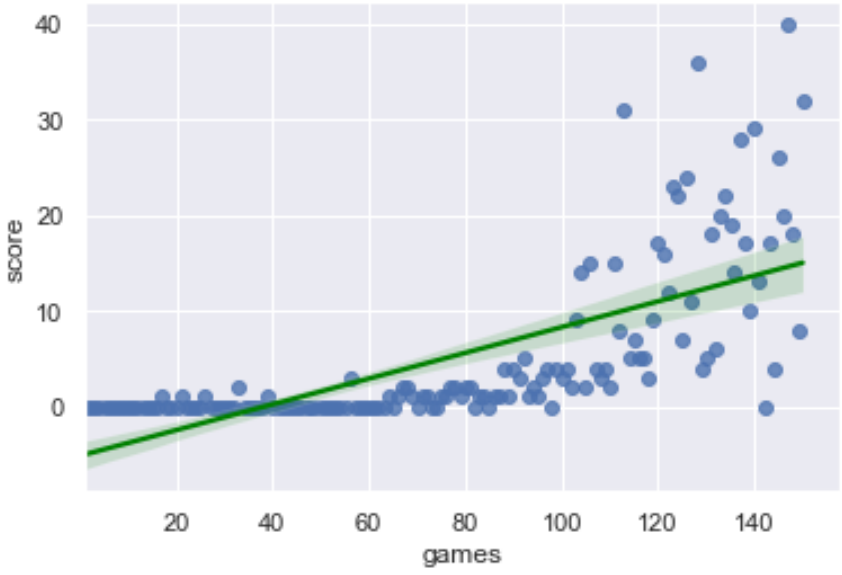


Figure 7

Figure 9 and 10 are the comparison of epsilon decay and its effects on score.

*	Epsilon Decay	Memory Size	Batch Size	Gamma	Out Activation	Neuron Numbers	Learning Rate	Average Score
Figure 8	1/50	2500	750	0.6	Linear	18-24-24-12	0.0004	22.11
Figure 9	1/100	2500	750	0.6	Linear	18-24-24-12	0.0004	23.62

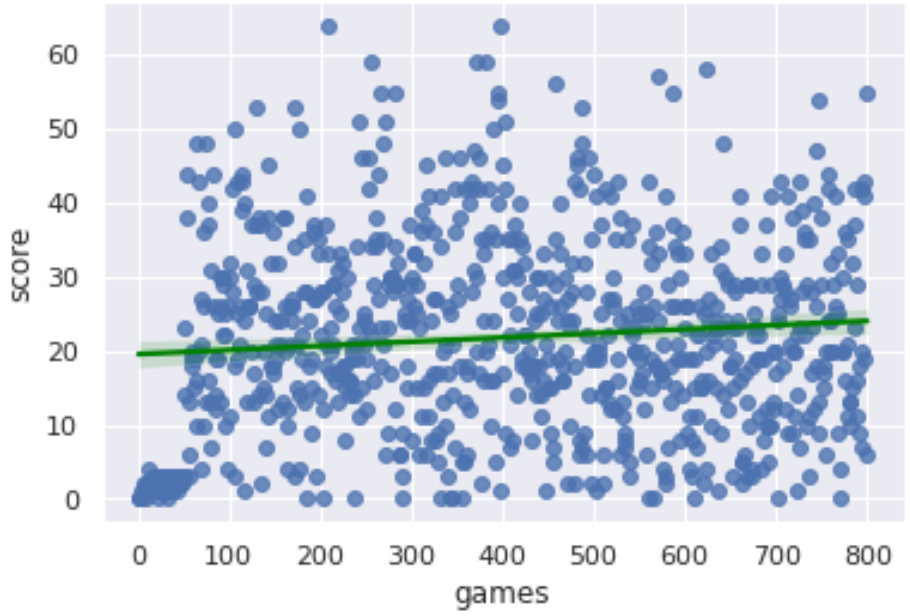


Figure 8

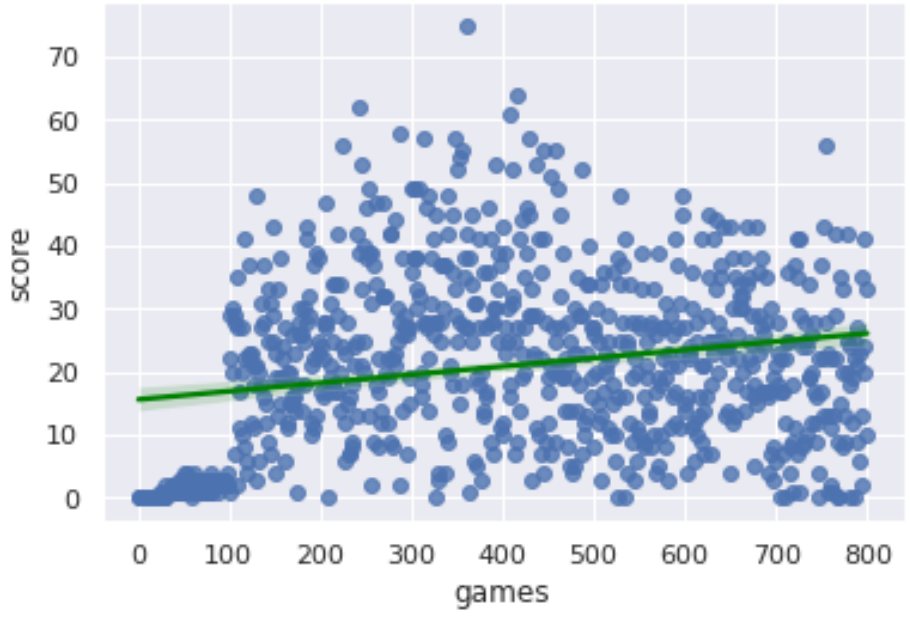


Figure 9

The figures 10 and 11 are shows the effect of number of neurons in the layers. Figures 11 and 12 are the example of layer size difference.

*	Epsilon Decay	Memory Size	Batch Size	Gamma	Out Activation	Neuron Numbers	Learning Rate	Average Score
Figure 10	1/100	2500	750	0.7	Linear	100-120-120-60	0.000075	34.35
Figure 11	1/100	2500	750	0.7	Linear	18-24-24-12	0.000075	45.7
Figure 12	1/100	2500	750	0.7	Linear	18-24-12	0.000075	33.46

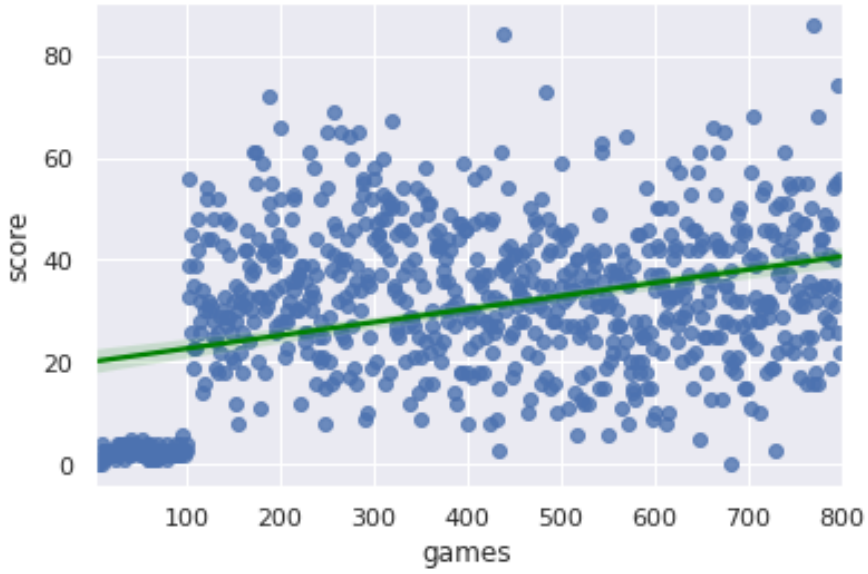


Figure 10

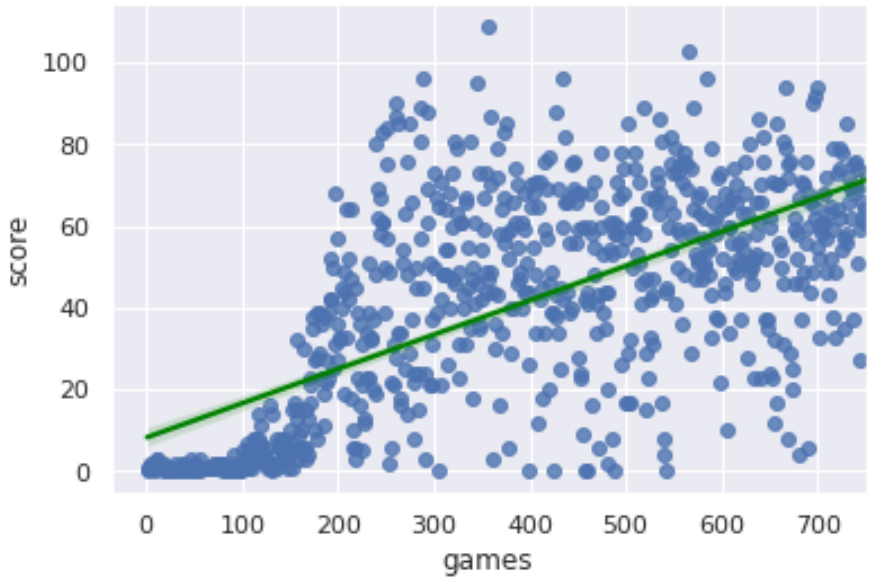


Figure 11

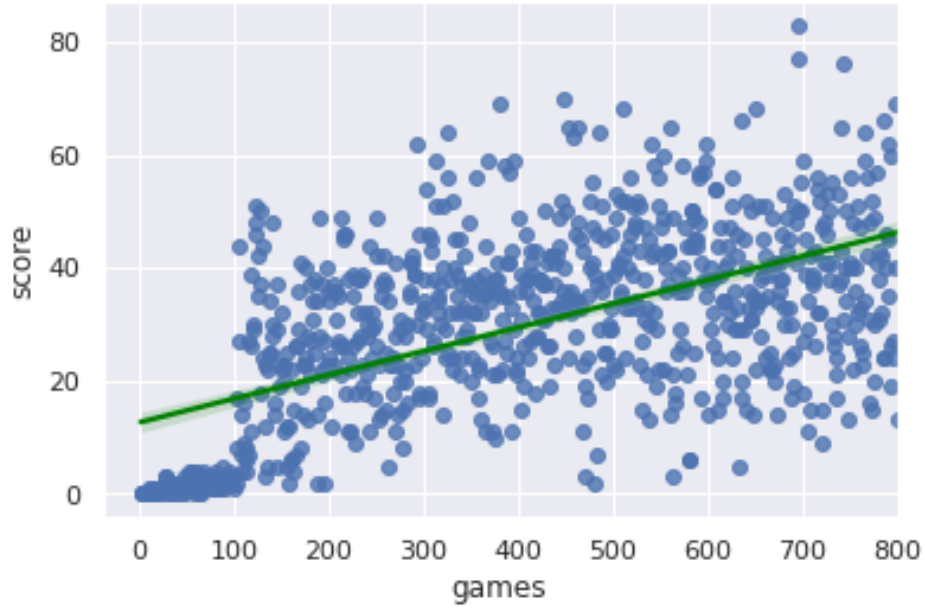


Figure 12

Highest learning rate experiment in the game with rate of 0.001.

*	Epsilon Decay	Memory Size	Batch Size	Gamma	Out Activation	Neuron Numbers	Learning Rate	Average Score
Figure 13	1/100	2500	750	0.8	Linear	18-24-24-12	.001	34.35

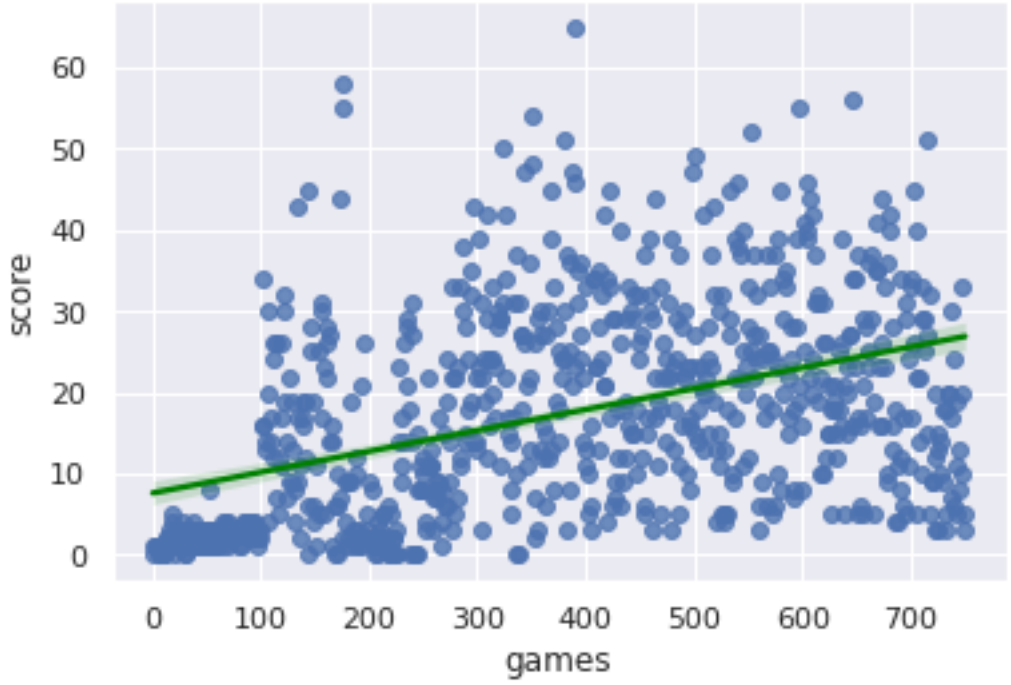


Figure 13

The figures shown in below are comparison of gamma values and learning rate with same variables.

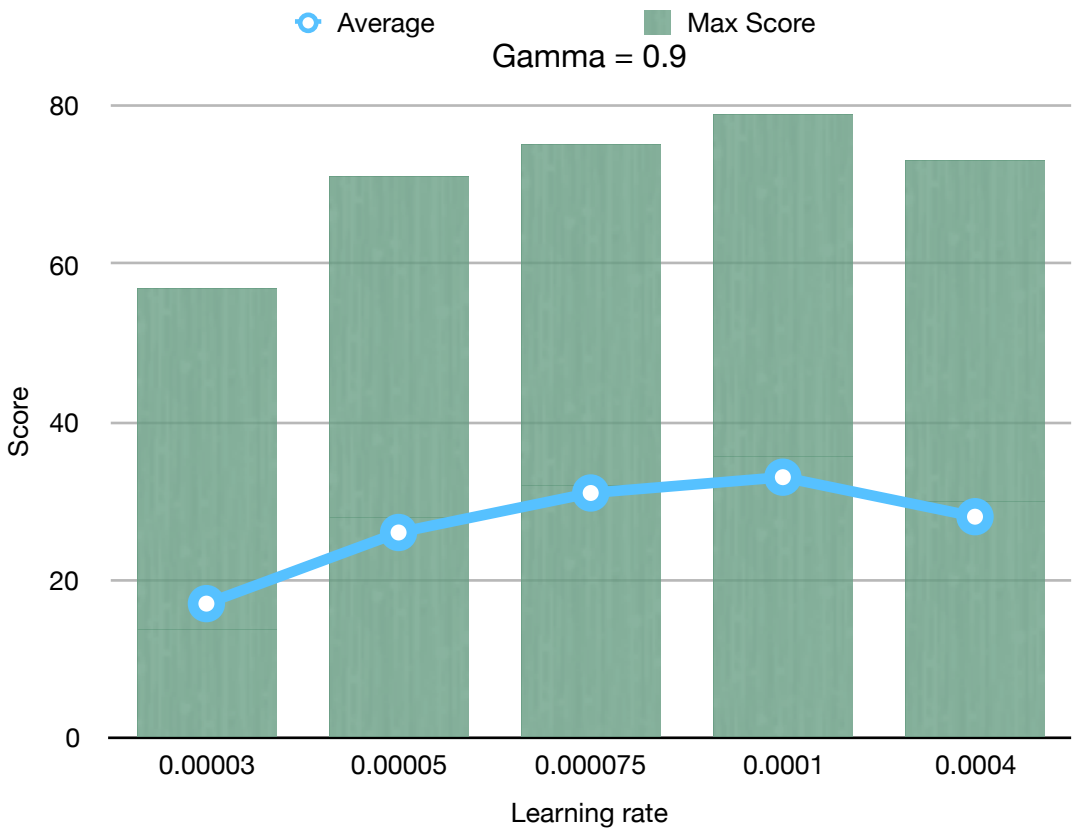


Figure 14

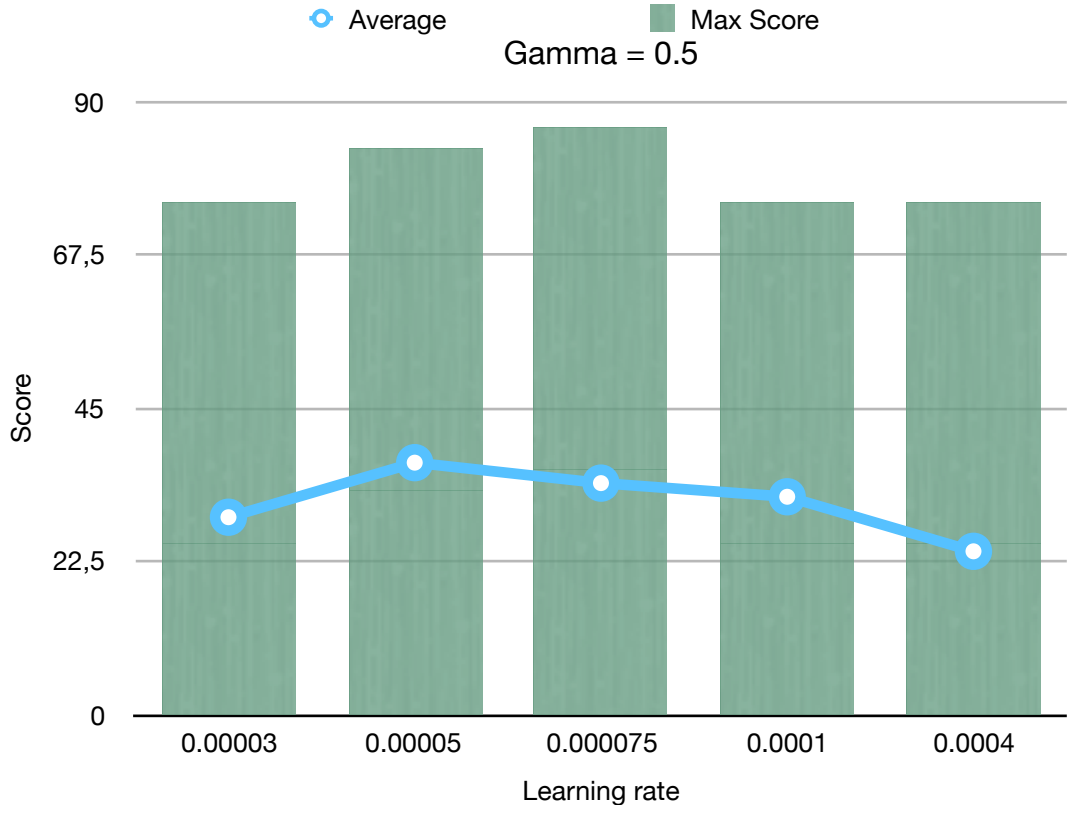


Figure 15

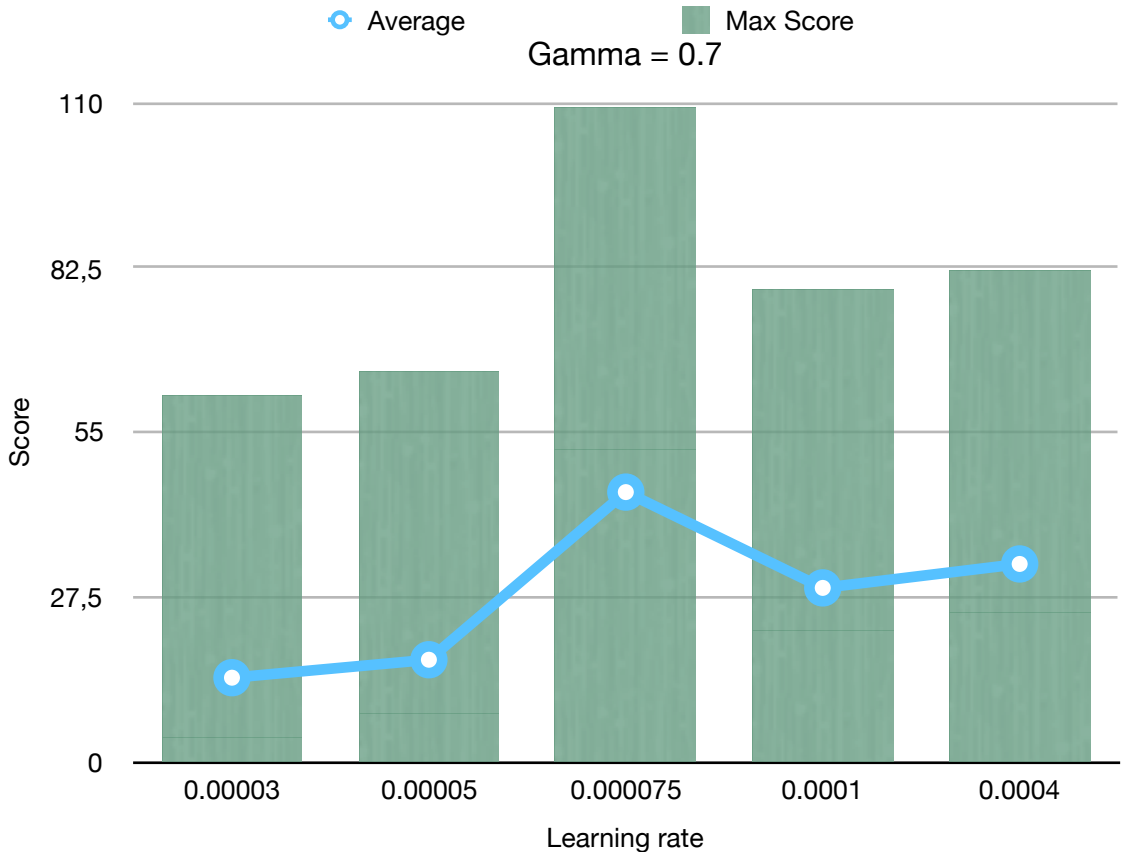


Figure 16

*	Epsilon Decay	Memory Size	Batch Size	Gamma	Out Activation	Neuron Numbers	Learning Rate (interval)	Average Score (interval)
Figure 14	1/100	2500	750	0.9	Linear	18-24-24-12	0.0004-0.000075	17-33
Figure 15	1/100	2500	750	0.7	Linear	18-24-24-12	0.0004-0.000075	24-37
Figure 16	1/100	2500	750	0.7	Linear	18-24-24-12	0.0004-0.000075	17-45

References

1 - https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

2 - http://alexlenail.me/NN-SVG/index.html

3- https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

4- https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d