Reinforcement Learning 2020
Assignment 3: Function Approximation

Meng Yao (s2308266), Shutong Zeng (s2384949), Auke Bruinsma (s1594443)

12-04-2020

Abstract

Abstract

1 Introduction

This assignment consists of two parts. In the first part we apply deep learning on the Mountain Car problem and in the second part on the game Breakout.
The deep learning method we introduce here is a neural network. In our two cases, we used supervised learning. We need labeled data so that the neural network is able to apply some operations on our inputs and let the outputs approximate the true labels.
Both of these sections apply convolutional neural network, as know as a CNN. CNNs consist of three important types of layers. Firstly, convolutional layers convolve the image with a number of filters or kernels and then are activated by a function, which can be either 'sigmoid' or 'Relu' and so on in order to make the network nonlinear. Secondly, pooling layers downsample the output of convolutional layers. Two general types of pooling are max pooling and mean pooling. Thirdly, dense layers are fully connected layers between previous hidden layer and the output layer [1].
To find the actions the car can take at each state, we introduce DQN (Deep Q-net or Deep Q-Learning). To start, we define some fundamental terms such as current state, state at the next step, action, policy and reward. We also define the state-action-value function Q, whose output is known as *Q value*. We want to train the agent to take actions in each step given the current state, that maximizes the Q value, A.K.A. the reward(van Hasselt (2015)).

A RL task is generally an agent training task, which takes different actions in different states, A.K.A. actions. Each action corresponds to a reward and the goal of the agent is to maximize the total reward.

In Q-Learning, we assume that the expected reward of each action is known. The maximum total reward Q-value will be achieved by a sequence of actions. This strategy can be generally depicted as $Q(s,a) = r(s,a) + \gamma \max_a Q(s', a)$. We calculate the cumulative quality of the possible actions the robot might take $Q(s,a)$ as (Mnih et al. (2013)):

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} \left(P(s,a,s') \max_{a'} Q(s',a') \right)$$

In this equation, the maximum value caused by taking a certain action is taken into account for the next action. We select the maximum of all the possible values of $Q(s,a)$ and add it to the reward. DQN aims at approximating Q and updating by the back propagation mechanism. It is the combination of RL and deep learning. It uses a semi-gradient to approximate the real value of Q. However, due to the frequent changes of the real value by updates, it tends to be unstable. To solve this issue, a target network, which is a copy of the training model, is introduced to decrease the frequency of updates. In this case, the real value will be much more stable.

1.1 Mountain Car

The Mountain Car example is a famous reinforcement learning model first introduced in 1990 (Moore, 1990). The goal is to get the car over the hill, but the car doesn't have enough power to get over the hill in one time. It needs to drive up the hill until it can't go further, release the gas so that it will fall back, and when it is on

[1]https://github.com/MengYao-astro/NN_2/blob/master/ASG2_NN20.pdf

the other hill and start going forward again, hit the gas again so that it will have more power. See figure 1 for a visualization[2].

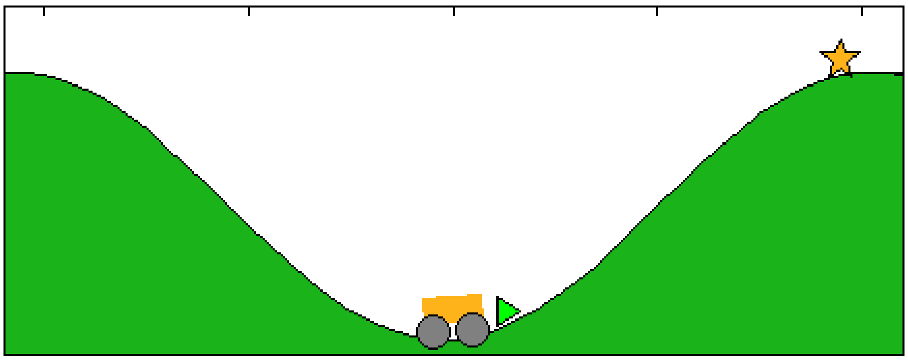


Figure 1: The mountain car problem, visualized.

1.2 Breakout

Breakout is a computer game that was released in the arcade halls in 1976. The concept of the game is very simple. As a player you are in control of a platform, you can move this platform to the left or to the right. In the game there is a ball moving. If this ball hits certain blocks, the blocks will disappear. The goal is let the ball bounce on your platform so that you will be able to hit all the blocks with the ball. Since the initial release of the game, it has been reworked with additional features multiple times on other platforms. See figure 2 for how the game looks when playing[3].

1.3 Tools

In this assignment multiple new tools are used. We will give a short introduction on them.

- `Gym`[4]: Gym is a library used for the programming and development of different reinforcement learning algorithms. In the mountain car problem it is able to provide a nice visual representation in which you can see the car trying to get up the hill.
- `Keras`[5]: Keras is a Python deep learning libary. It is very useful for building different neural network models.
- `Tensorflow`[6]: Tensorflow is an open source machine learning platform, useful for training machine learning models. It is easy to run in a collaboration notebook in the browser.

2 Mountain Car

2.1 Method

When using the module `gym`, it is possible to use certain environments[7], such as the 'acrobot' environment or the 'cartpole' environment. The one we are using is called the 'mountaincar-v0' environment, and looks like figure 3[8].

```
env = gym.make('MountainCar-v0')
```

In order to let the mountain car move, one needs to go forward in time. This is what the step function allows you to do. The environment's step function returns 4 important values. These are `observation`, `reward`, `done` and `info`[9].

[2] https://en.wikipedia.org/wiki/Mountain_car_problem
[3] https://en.wikipedia.org/wiki/Breakout_(video_game)
[4] https://gym.openai.com/
[5] https://keras.io/
[6] https://www.tensorflow.org/
[7] https://gym.openai.com/envs/#classic_control
[8] https://gym.openai.com/envs/MountainCar-v0/
[9] https://gym.openai.com/docs/#environments

Figure 2: The game Breakout on the Atari 2600

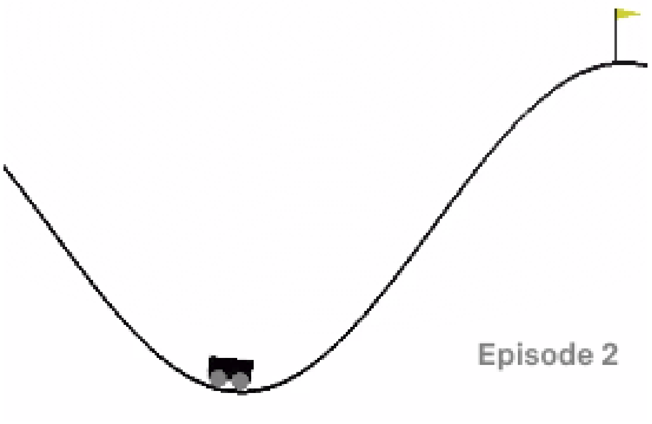


Figure 3: MountainCar-v0 visulization

1. `observation` This is a variable that is different for each environment. For each environment it gives information on its representation. In our situation it will give data on the car's position and velocity.
2. `reward` Returns the reward obtained by its previous action.
3. `done` Determines if the simulation is completed and should be reset again.
4. `info` Useful information for debugging.

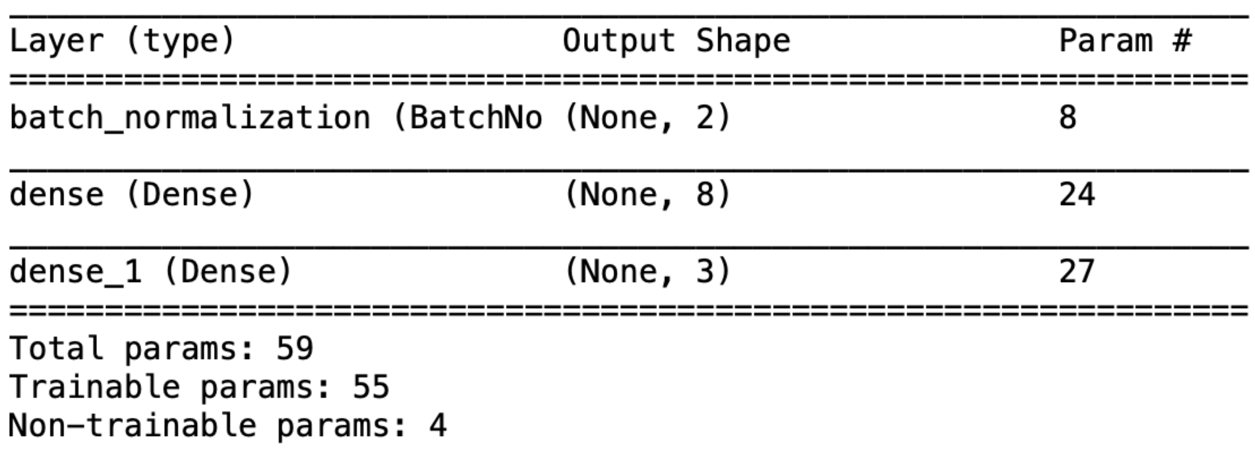
Our approach to this part of the assignment was inspired by the blog post provided to us in the assignment's instruction[10]. To get ourselves familiar with the libraries we are supposed to use, we first played a bit around with the code to see what everything does and how everything works. Just as the blogpost, we tried a `random_game` function, which outputs the variables described above: action, position, velocity, reward, done and info. We initialised each step with a random move to see what happens. This helped gaining a better understanding. This code however is not used in running the network; it is only for us to see how things operate, all other functions are reworked and adjusted independently by us. Using the `env.render()` allows us to easily to see what happens. As expected, applying random moves to the mountain car will not get it to the top of the hill.

In the function `make_train_sets()` we store successful runs of the simulation in a training set. A successful run means a run where the final score of the mountain car game has above a certain threshold. There are a number of hyperparameters that will play an important part during the making of training sets. The parameter `num_games` determines the number of games/runs that will be played. Each run has a number of steps, which is determined by the `num_steps` parameter. In each run and and in each step, the action that will be taken is still determined randomly. From this action all the variables (observation, reward, done, info) are obtained. The crucial step is the reward function. This is a simple function that gives a reward of 1, if the position value is

[10]https://medium.com/coinmonks/solving-curious-case-of-mountaincar-reward-problem-using-openai-gym-keras-tensorflow-in-python-d031c471b346

larger than a certain threshold value. Throughout all steps the sum of the rewards is stored in the `sum_reward` variable. After each run there is a check of the sum of all the rewards is above the minimum score. If yes, this run will be stored as training data. In this way all the runs which random moves have produced successful runs are stored as training data.

The neural network was built on the platform of *TensorFlow* and the summary of architecture is shown in Fig. 4. Our input layer has two nodes which take in the two values from observation of mountain car environment. We used batch normalization on the input layer order to make the training converge faster. For the first hidden layer, we built a fully connected layer with 8 nodes, which is a dense layer with the output size of *(None,8)*. Next, we fully connected it to our output layer which has 3 nodes representing 3 action options. In this neural network, we only used one hidden layer and a small amount of trainable parameters because of the fewness of our input nodes and output nodes. More hidden layers and nodes will be redundant and will cause over-fitting problems. A major change made in our implementation was that we considered this task as a 3 classes classification problem because we have 3 action options. In this case, our loss function is defined as cross entropy. We used the '`Adam`' algorithm as the optimizer which includes a momentum term and is able to jump out from the local minimum.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization (BatchNo (None, 2)                 8
_________________________________________________________________
dense (Dense)                (None, 8)                 24
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 27
=================================================================
Total params: 59
Trainable params: 55
Non-trainable params: 4
```

Figure 4: Summary of the neural network for mountain car game

2.2 Result

We trained our neural network for 20 epochs. The reward threshold value was tuned to -0.15 which makes it easier to score and will lead to more rewarded training data. Note that our testing games are played with the original threshold, which is equal to -0.1. In this case, our scores are comparable with others. We also tuned the threshold of being a 'successful' game to -190, which will make the training data have more 'good' actions in it. In this case, the accuracy is shown in Fig. 5. Next, we tried 10 games with our trained model and the results are listed in Tab. 1

Game	Score
1	-97
2	-76
3	-92
4	-110
5	-79
6	-77
7	-84
8	-86
9	-79
10	-86
avg	-86

Table 1: Mountain car scores

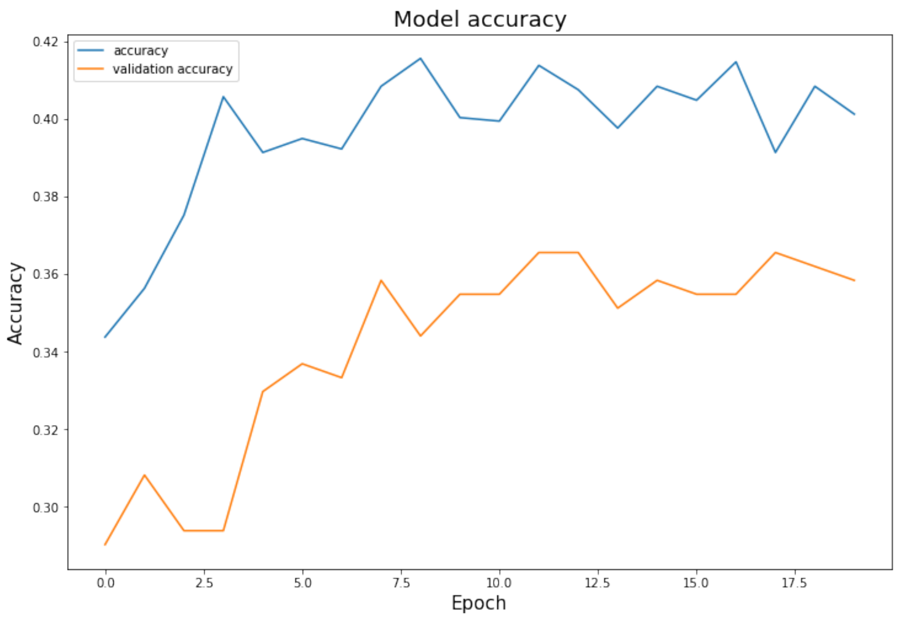


Figure 5: Accuracy w.r.t epoch for mountain car network

2.3 Conclusion

We accomplished the task of training a neural network to play the mountain car game. In this task, we constructed a network, tuned the parameters so that we reached higher scores then the previously mentioned blog post. Moreover, our construction of the neural network also makes the training faster due to less amount of trainable parameters.

3 Atari BreakOut

In this section, we accomplished a basic DQN for Atari break out training and an improvement of experience replay.

3.1 Method

3.1.1 Environment

We used a *BreakOut* environment called 'BreakoutDeterministic-v4' in 'gym' to simulate the game. This environment skips 4 frames when taking an action, which is different from 'Breakout-v0'.

When applying this environment, which is similar to 'mountain car', four variables will be produced in each step: `frame`, `reward`, `done` and `info`.[11]

1. `frame` The current frame of 'BreakOut'.
2. `reward` Returns the reward obtained by its previous action. Note that this is not the score on the board.
3. `done` Determines if the simulation is completed and should be reset again.
4. `info` Useful information for debugging.

Another important variable in this environment is the action. It has 4 possible options, which is checked by:

```
env.action_space.n
```

In order to monitor the game, the gym library provides a tool called 'env.render()' which presents the frame of your current game. In addition, we modified two functions to show the the frame when needed [12].

[11] https://gym.openai.com/envs/Breakout-v0/
[12] https://stackoverflow.com/questions/40195740/how-to-run-openai-gym-render-over-a-server

3.1.2 Data processing

For training and data preparation, in order to save the storage, we first down-scaled the frame to a smaller size by skipping one for every two pixels on two dimensions. In this case, our frame contains $105 * 80$ pixels. Then we converted RGB color to gray by taking the mean on the last axis of the array.[13] After that, we used the 'threshold' tool in library 'cv2' to accomplish the binarization of the frame, which also gives a normalized image. For the basic implementation of our DQN, we stacked 4 consecutive frames produced by the environment, note that it also skips 4 frames each step, as a state. One state can be seen as just a state, or the next state of its previous one. Besides, each frame will show up 4 consecutive states. For an illustration, imagine we have 6 numbers in an array (0,1,2,3,4,5). In this case, (1,2,3,4) represents a state and it can also be treated as the next state of (0,1,2,3) at the same time, where (1,2,3) are the shared frames.

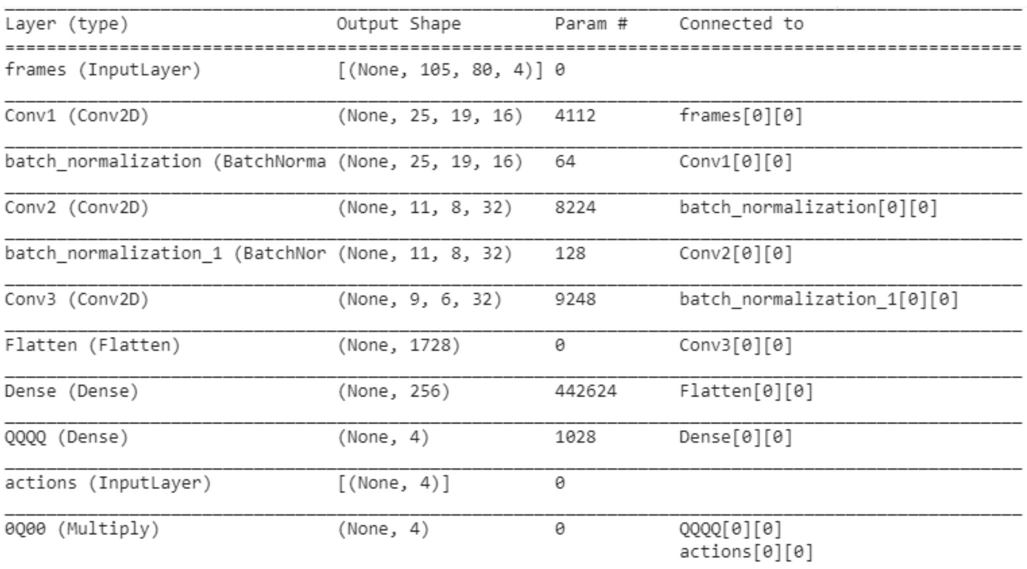
As of actions, they are chosen by an epsilon greedy policy where actions are selected exploratively at the beginning, with epsilon being large (Eq. 1). Then with the training proceeding, epsilon decreases, and we will shift to exploitation. The action will be saved for each step, which means each frame has its own saved action when considering our data structure.

$$\epsilon = \max\Big(0.1, 1 - \frac{i}{n}\Big), \quad i = \text{iterations} \tag{1}$$

In terms of rewards, we define a function that only focuses on the sign of reward provided by the environment. This function is more or less redundant for our case as we only have zeroes or ones as rewards value. We will still include it in our implementation, however.

3.1.3 Neural Network

As the goal is to approximate the Q function, our neural network is constructed to provide the Q value on a given state with possible actions. In this case, the output layer is determined to have 4 nodes, which is equal to the number of action options. The input layer consists of 2 parts. They are 'state' and 'action'[14]. The 'state' input is an array of 4 game frames. The action input is an one-hot array which is converted from action data with a depth of 4. For example, the action we took is '2', the one-hot array should be [0,0,1,0]. Our hidden layers are 3 convolutional layers with different kernel sizes and strides. As an improvement for training, we add an batch normalization layer between two convolutional layers, which normalizes the output of its previous layer. Then we have a flatten layer to put the pixels in one dimension. After that, we built 2 dense layers, which are fully connected, to produce a 4 nodes output. Most importantly, we have to use the action input as a filter to set Q values of actions that we didn't take to 0 and only lets desired Q value pass through. In addition, we are able to get 4 Q values by setting the one-hot array to [1,1,1,1].



```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
frames (InputLayer)             [(None, 105, 80, 4)] 0
__________________________________________________________________________________________________
Conv1 (Conv2D)                  (None, 25, 19, 16)   4112        frames[0][0]
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 25, 19, 16)   64          Conv1[0][0]
__________________________________________________________________________________________________
Conv2 (Conv2D)                  (None, 11, 8, 32)    8224        batch_normalization[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 11, 8, 32)    128         Conv2[0][0]
__________________________________________________________________________________________________
Conv3 (Conv2D)                  (None, 9, 6, 32)     9248        batch_normalization_1[0][0]
__________________________________________________________________________________________________
Flatten (Flatten)               (None, 1728)         0           Conv3[0][0]
__________________________________________________________________________________________________
Dense (Dense)                   (None, 256)          442624      Flatten[0][0]
__________________________________________________________________________________________________
QQQQ (Dense)                    (None, 4)            1028        Dense[0][0]
__________________________________________________________________________________________________
actions (InputLayer)            [(None, 4)]          0
__________________________________________________________________________________________________
0Q00 (Multiply)                 (None, 4)            0           QQQQ[0][0]
                                                                 actions[0][0]
```

Figure 6: Model summary of the neural network for BreakOut

As a supervised learning process, our target is the value Q function. It is calculated with reward and the Q value of the next state, which is predicted by our neural network model[15]. The summary of our neural network is shown in Fig. 6.

[13] https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26
[14] 13
[15] 13

We chose Huber loss as our loss function, which is recommended by the blogs[16] [17]. Huber loss is an adaptive loss function which approximates 'mean squared error' for low 'a' values and uses 'mean absolute error' for high 'a' values. The Huber loss function is pre-defined in 'TensorFlow' so that we can use it directly. We set the 'a' value to 1.5 to gain the advantages from both 'MAE' and 'MSE'. Similar to the 'mountain car' game, we also used 'Adam' as the optimizer.

3.1.4 Training process

For the basic DQN implementation on 'Breakout', we train our network during playing. Once we achieve 32 states and 32 next states, we will train the neural network for 1 epoch as the batch size is being equal to 32. Considering the movement of the ball will cause a delay of the presentation of our reward, in another word, when we hit the ball, the reward will be gained after a number of frames, we set the frame difference between a state and its next one to the number of 50 (roughly observed from random games where the ball needs 50 frames to reach the brick from the bottom) [18]. The action we used is the one that is taken at the end of a state and the reward is drawn from its next state.

3.1.5 Improvement: experience replay buffer

As a well-known improvement of DQN, replay buffer, in a nutshell, is to store a number of training pieces which consists of 'state', 'action', 'next state', and 'reward' and then sample from them to form a training set. In this case, states can be used for several times for each epoch of training. Moreover, as we have a large data set, two consecutive states are not likely to be sampled in a epoch so that the relation between states are also broken.[19] In our experiment, we used a data set with the size of 10,000 pieces and sampled 32 states for 64 epochs of training in total.

3.2 Results

3.2.1 Basic implementation

We trained our neural network with a discount factor of 0.99 with the basic implementation. The figure of average score of every 10 games is shown in Fig. 7. As we can see, our score did not get higher so we tried to look for the problems. We first tried to monitor the loss of our neural network and the result is shown in Fig. 8. This figure shows that the loss approached 0, which suggests the function converged and the training is sufficient.

Then we adapted the discount factor γ to a smaller value in order to provide a better result. We considered that the action which should be taken depends more on the current state than looking way further in the future. The average score of every 10 games is shown in Fig. 9. The score figure shows a similar result with $\gamma = 0.99$.

We also tried another discount factor $\gamma = 0.6$. The score for every 10 games is shown in Fig. 10. It shows the same result as the previous two experiments.

3.2.2 Replay buffer

We then experimented DQN with the improvement of replay buffer. Again, we sampled 32 pieces of the data with the size of 10,000 for 64 epochs. The average score is shown in Fig. 11, which is not increasing to 300 points, as the blogs results.

We also tried a smaller discount factor $\gamma = 0.6$. Fig. 12 shows the score that we achieved with replay buffer and the $\gamma = 0.6$.

3.3 Conclusion

We experimented with two approaches of DQN on the game 'BreakOut'. The results in our cases are not consistent with others' implements online, which are referred to earlier. Our thoughts on this is that the reward data are really sparse with '1's so that we might lack successful actions. For the existing '1's in our reward data, we found it hard to locate when the action was made to hit the ball and score the point.

[16] 13

[17] https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756

[18] Note that we have already tried to use consecutive states as 'state' and 'next state'. The score did not increase with training so we came up with this idea of delay. As the method of consecutive states is not reasonable, we will not show it in the result part

[19] 13

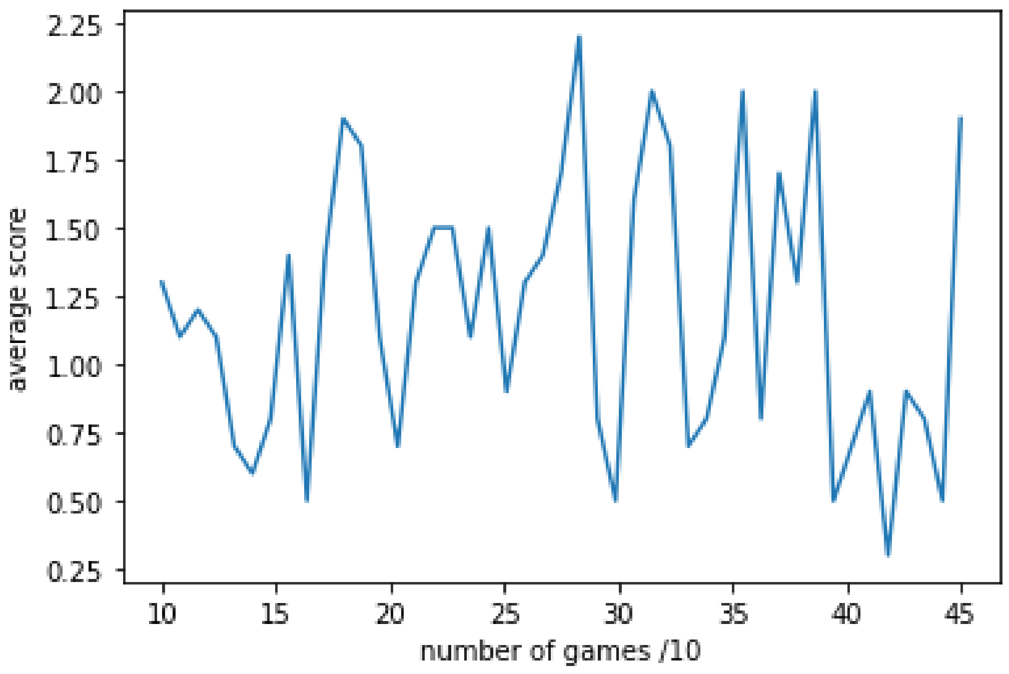


Figure 7: Average score of every 10 games from basic implementation with $\gamma = 0.99$

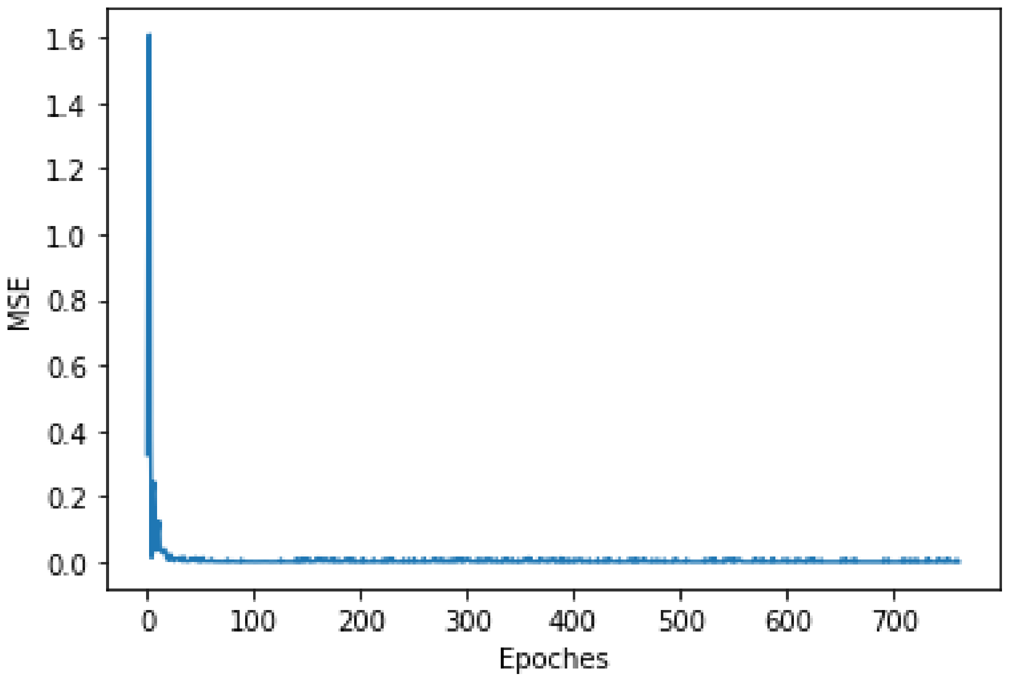


Figure 8: Mean square error of of neural network with $\gamma = 0.99$

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, art. arXiv:1312.5602, December 2013.

A Moore. Efficient memory-based learning for robot control. 1990.

Arthur; Silver David van Hasselt, Hado; Guez. Deep reinforcement learning with double q-learning. 2015.

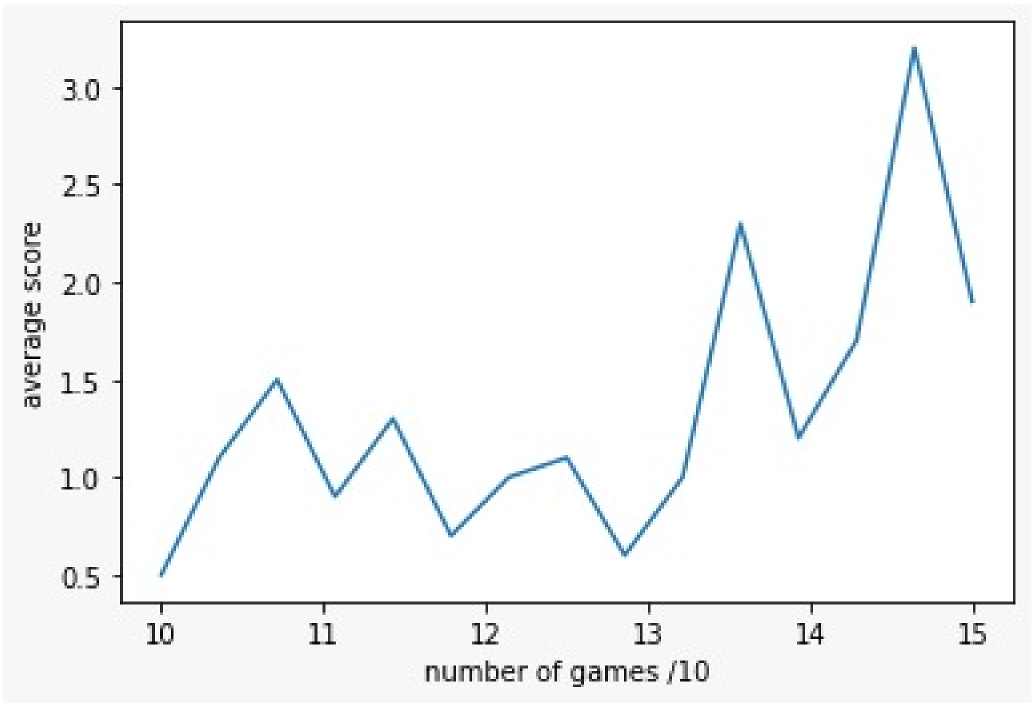


Figure 9: Average score for every 10 games with $\gamma = 0.8$

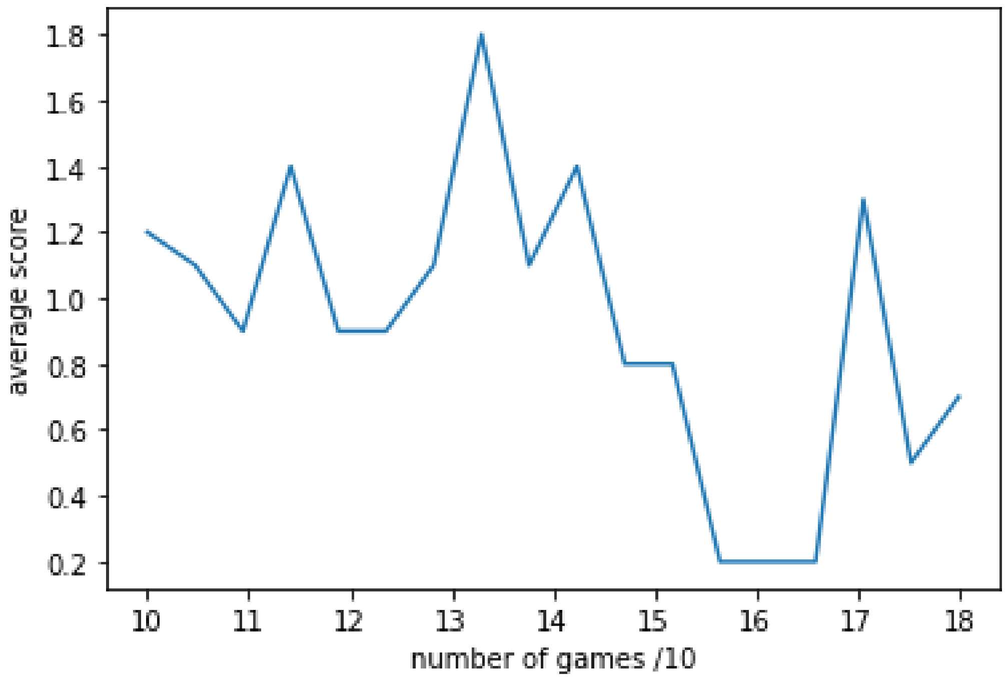


Figure 10: Average score for every 10 games with $\gamma = 0.6$

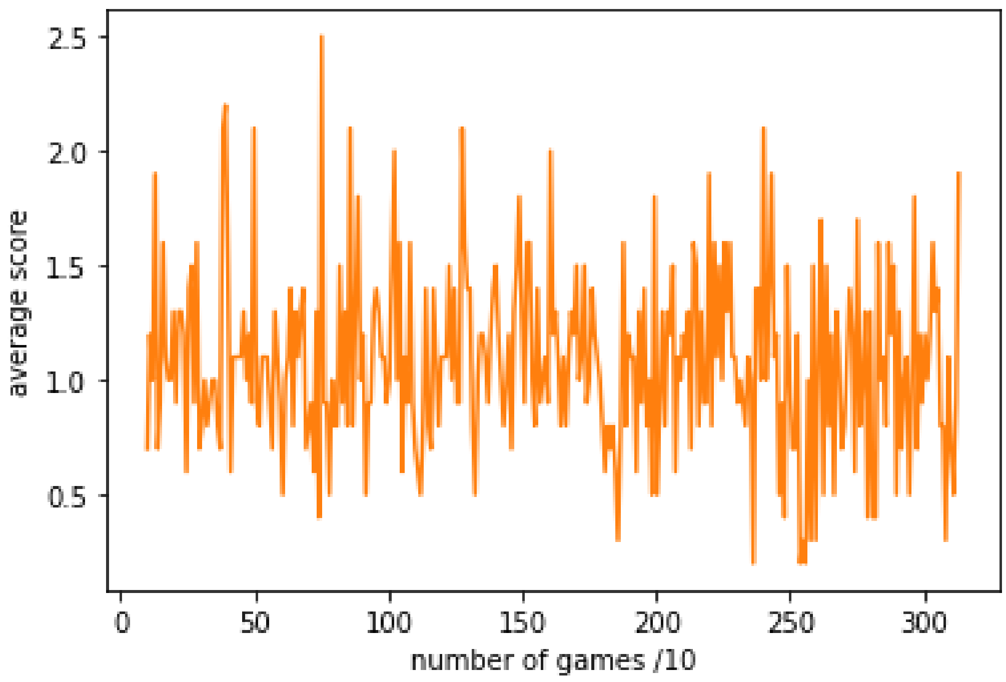


Figure 11: Average score of every 10 game using replay buffer

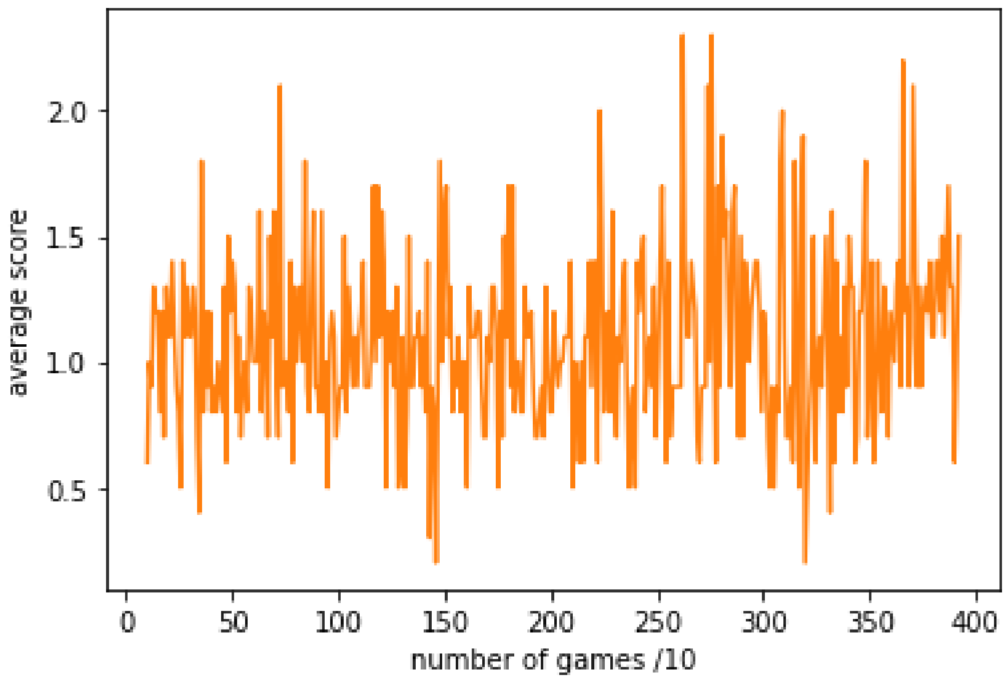


Figure 12: Average score for every 10 games using replay buffer with $\gamma = 0.6$