# Final Report AI 6106 Mini Project Deep RL

**Group Name: AI2020**

Brandon Chua Shaojie
bran0026@e.ntu.edu.sg

Wu Yin
ywu030@e.ntu.edu.sg

Cheong Kok Yuet
kcheong012@e.ntu.edu.sg

## 1 Introduction

For this mini project, we have chosen the Atari game Space Invaders. We will be using deep neural networks for reinforcement learning. The algorithm chosen is Q-learning. Deep Q-learning Network (DQN) and Double Deep Q-learning Networks (DDQN) are the networks be trained to play Space Invaders.

Reinforcement learning is a method to train machines by rewards and punishment. This is done by giving rewards to the machine when it has performed what is considered the “right” action and punished when the “wrong” action is performed. The machine then aims to maximise its rewards while minimizing its punishments. The machine agent learns by interacting with environment. In our case, it is rewarded when it shoots down an enemy and punished when their own cannon is destroyed.

Q-learning is a type of off-policy reinforcement learning. It chooses actions according to their q-values. Q-values are calculated for action state pairs, which states the quality of taking that action in that state. It then tries to find the policy that returns maximum rewards.

DQN is basically q-learning with a neural network and DDQN uses two networks instead of one to estimate the q-values.

## 2 Problem statement

We have to train the network to be able to play the game Space Invaders. The aim of this game is to shoot down the enemy and to avoid getting your cannon shot down. You increase your score for shooting down the enemy aliens and you also get bonus points for shooting a special mystery ship that sometimes appear at the top of the screen. There are three defence bunkers to serve as protection for the player.

The aim of the agent is to maximise the score. Therefore, the aim of network is to teach the agent that shooting down enemies and the mystery ship is a good thing and getting your cannon shot down is a bad thing. The network thus has to learn that the q-values for actions that lead to increasing score should be higher. On the other hand, the q-values for actions that lead to getting your cannon destroyed (e.g. losing a life) should be low. The optimal policy here is to shoot enemies while avoiding getting killed.

Objectives:

1. Learn q-values for state-action pairs. Total of 6 actions are available in Space Invaders for each state. (No action, Move left, Move right, Shoot, Move left and shoot, Move right and shoot)

2. Maximise q-values for state-action pairs that lead to maximum total rewards.

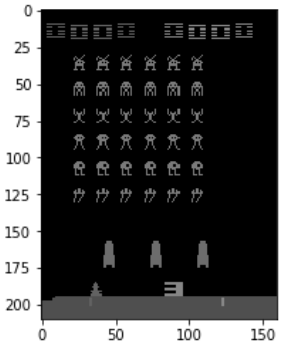
3. Train DQN and DDQN to perform the above 2 objectives.

# 3 Methodologies

For our group, we have decided upon using DQN and DDQN to train the agent to learn how to play Space Invaders in OpenAI Gym.

**Data Pre-processing**

For this experiment, images will be the input for our DQN and DDQN. Frames from the game Space Invaders that contains information about the environment will be the input. To facilitate faster training, we try to cut off unnecessary information from the frames so we have less data to process. Colour is one part we can cut off.




As seen from the picture, even when greyscaled, we can still identify where and what the objects are. Furthermore, the frames are shrinked to a size of 84 x 84 pixels.

Next, we stack several frames (we used 4 in this experiment) together and use it as the input for the network. The reason is that with just a single frame, the machine is not able to gather enough information about the environment. Our network is unable to tell from a single frame the velocity of enemies and their bullets. On the other hand, we are able to discern all this information with a stack of consecutive frames.

**Q-Learning**

Given the actions, states and rewards (***a, s, r***), we want to be able to find the optimal policy. The optimal policy is the one which will maximise the rewards in a game. This is done so using the Bellman Equation and an epsilon greedy strategy.

Bellman Equation is used to find the expected return for choosing action ***a*** in state ***s*.** This is shown in this equation:

$$q_*(s,a)=E[r+\gamma \max_{a'} q_*(s',a')]$$

where $q_*$ is the optimal policy and **γ** is the discounting factor. This equation states that the expected reward by following the optimal policy, $q_*$, to choose the action, ***a*,** in state, ***s*,** is the reward, ***r*,** (which we get from choosing action ***a*** in state **s**) plus the discounted return of the maximum reward we can get out of all possible actions, ***a'***, in the next state, ***s'***. Using this equation, we can update the q-values for each action-state pair according. After enough iterations, the q-values will converge to give use the optimal policy.

Just using the Bellman Equation is not enough. Given the situation that our agent finds a reward, it will continuously keep going back to it to exploit it. However, there might be other rewards that have higher values, but the agent will never find it since the q-values are updated in the way that the policy will always exploit the first reward it finds. In this case, we are stuck with a sub optimal value. To overcome this issue, we use an epsilon greedy strategy.

Epsilon greedy strategy is to give the agent a probability to explore instead of exploit. In the previous mentioned situation, instead of always exploiting a reward that the agent finds, we give the agent a probability that it might take a random action. This randomness will allow the agent to explore other states and find a better policy.

**Epsilon Greedy Strategy**

By utilizing this strategy, we hope that the network is able to explore all different state-action pairs to find the optimal policy. We want to explore the environment to avoid getting stuck in a local minimum. A random action (exploration) is chosen with probability ε and optimal action is chosen with probability 1- ε. As training progresses, the value of ε will decrease and eventually reach a value of 0.1.

**Replay Memory**

Replay memory is where we store our experience replays. Experience replays are observations (***s a, r, s'***). We randomly sample a number of experience replays to train our networks with. The reason for doing so is to remove correlations in the data. If we train our neural networks with back to back samples, it will be undesirable due to the correlation present in the samples. Thus, randomly choosing samples to train our network on can help to train a more accurate network.

**DQN**

DQN combines deep neural network and q-learning. As we are using images as input, we will be using convolutional neural network (CNN) to extract features from the images. After passing the image through the CNN layer, we flatten the output from the CNN and feed it into the linear layers to find the estimated q-values. Loss is calculated by sampling random batches of experience replays from the replay memory. Gradient descent is then carried out to update the weights in the network.

Q-learning in DQN is done by two passes into the network. The first pass is to find the q-values for all possible actions in the given state, ***s***. A second pass is needed to find the maximum reward we can get out of all possible actions, ***a'***, in the next state, ***s'***. This is then plugged into the Bellman Equation to calculate the target q-value for the first pass. This brings up an issue in DQN. When we update the weights for the network, the q-values will change and so will the targeted q-values. This causes instability in the network since the network is trying to update q-values to the target q-values, which keeps changing as the network gets updated.

In this case, we use another network for the second pass which we will refer to as the target network. The target network is basically a copy of the original network. The difference is that the target network does not get updated after every step. The target network is updated only after a certain number of steps. This means that the target q-value calculated using the target network does not change that often, and this allows for stability when training DQN.

**DDQN**

DQN has this issue called overestimation. DQN calculates the q-values by adding **r,** (which we get from choosing action ***a*** in state ***s*** and the discounted return of the maximum reward we can get out of all possible actions, ***a'***, in the next state, ***s'*****).** In the situation that a very high reward with low probability is encountered for a state-action pair (***s, a1***), it will lead to overestimation of that state-action pair due to how DQN calculated q-values. This means that during training, when given the same state with a different action, ***a2***, that leads to a more optimal policy, it is harder for the network to learn that this different action is better since the difference in values for the initial state-action pair (***s, a1***) and (***s, a2***) is too large.

DDQN tackles this problem by changing the way q-values are calculated. It uses two identical networks. The first network is for choosing the action to be taken in a given state. The second network is then used to calculate the q-value of the action selected by the first network. The second network is only updated after a certain number of steps. This helps to reduce the difference between state-action pairs since the second network essentially is the past version of the first network.

**Loss**

We use Mean Squared Error to calculate the loss for the networks.

### Optimizer

We chose RMSprop, a variant of stochastic gradient descent, as our optimizer.

### Activation Function

ReLU is used.

### Hyperparameters for Training

N_games = 1000 (Number of games)
Gamma = 0.99, 0.999 (Discounting Factor)
Epsilon = 1.0 (Starting value of ε)
epsilon_min=0.1 (Final value of ε)
lr = 0.0001, 0.00025 (Learning rate)
batch_size=32 (Batch learning size)
replace_count=5000 (Number of steps taken before target network is updated)
eps_decay=5*1e-5, 5*1e-6 (ε decay rate)
memory_size=50000 (Number of experienced replays in Replay Memory)

Models are trained using PyTorch.

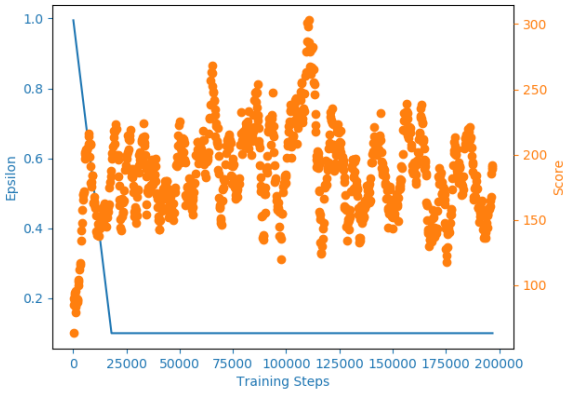
### Overall Process

1. Pre-process data
2. Use epsilon greedy to choose our action.
3. Perform that action and store observation in Replay Memory
4. Sample randomly from Replay Memory to calculate loss
5. Use RMSprop to update weights on original network
6. Update the other networks after 5000 steps.
7. Repeat till 1000 games are done.

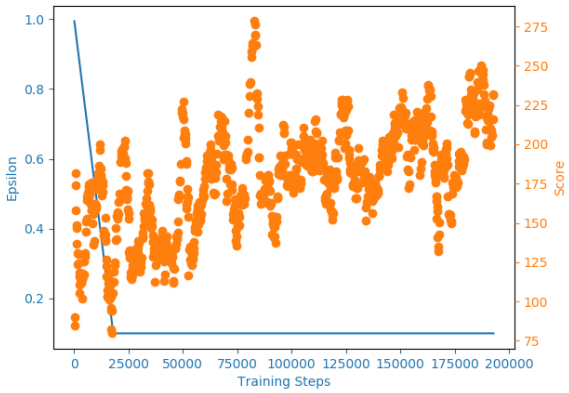
## 4 Experiment result

### Learning Rate

We experimented with different learning rates of 0.0001 and 0.00025 for DQN and DDQN.

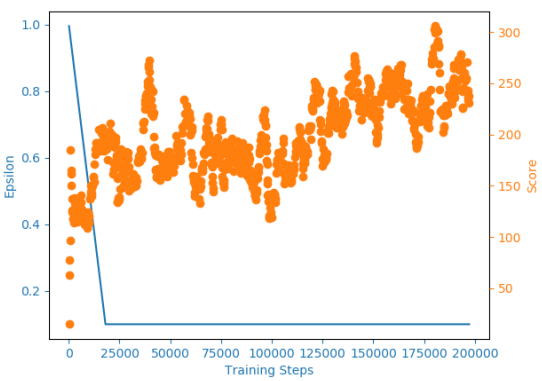



Plot 1) DQN, Learning rate = 0.0001

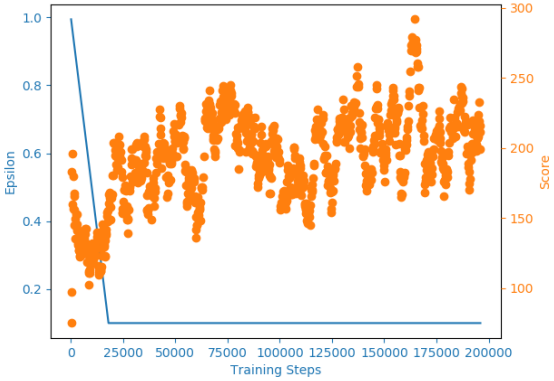



Plot 2) DQN, Learning rate = 0.00025

A learning rate of 0.00025 is better than 0.0001. Plot 1 shows more instability than plot 2. Around 125000 training steps, plot 1 slightly decreases in scores. Plot 2 scores increases as more training steps are taken. At the end, plot 1 hovers around score 175 while plot 2 hovers around score 200.




Plot 3) DDQN, Learning rate = 0.0001



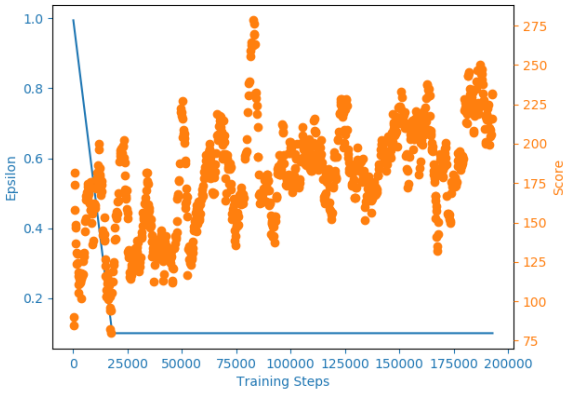

Plot 4) DDQN, Learning rate = 0.00025

As seen from the above plots, a learning rate of 0.0001 is better than 0.00025. The scores for plot 4 fluctuate more than plot 3. This shows that plot 3 is more stable. At the end, plot 3 has scores hovering around 250, which is 50 more than plot 4, which is around 200.

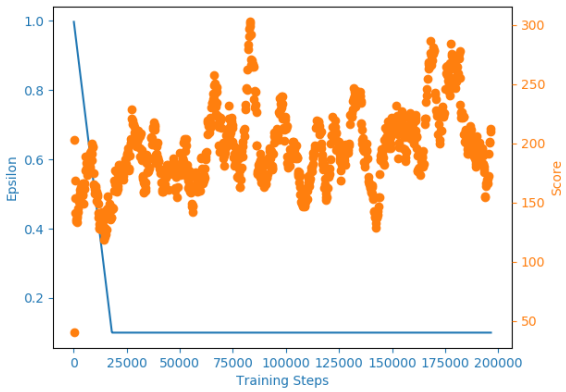
For DQN, a learning rate of 0.00025 is better while for DDQN a learning rate of 0.0001 better. Comparing plot 2 and plot 3, we can see that DDQN performs better. DDQN is more stable with less fluctuations. At the end of training, DDQN hovers around score 250, which is 50 more than DQN, which hovers around score 200.

**Gamma**

We experimented with different gamma of 0.99 and 0.999 for DQN and DDQN.

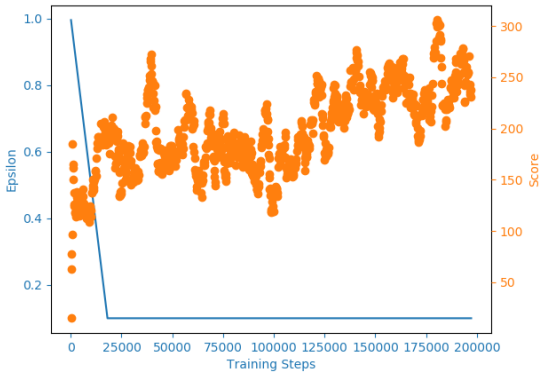



Plot 2) DQN, Gamma = 0.99

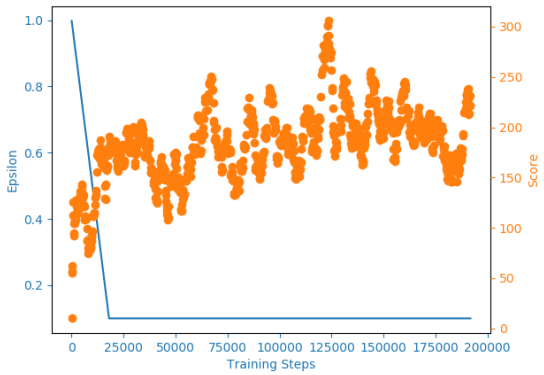



Plot 5) DQN, Gamma = 0.999

A gamma of 0.99 is better than 0.999 for DQN. Plot 5 is more stable during training but plot 2 converges at a higher score at the end.




Plot 3) DDQN, Gamma = 0.99



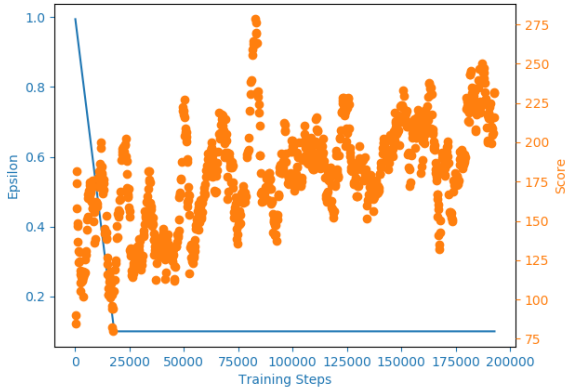

Plot 6) DDQN, Gamma = 0.999

A gamma of 0.99 is better than 0.999 for DDQN. Plot 1 is more stable and has a higher score average at the end of training.

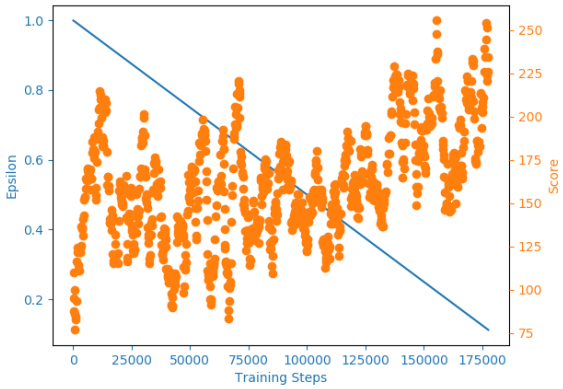
For both DQN and DDQN, a gamma of 0.99 is better.

**Epsilon Decay**

We experimented with different decay rates of 5e-5 and 5e-6 for DQN and DDQN.

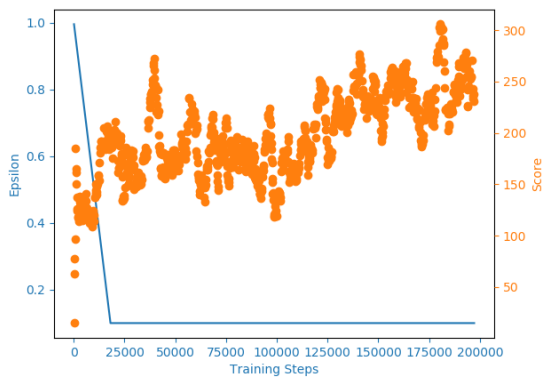



Plot 2) DQN, Decay rate = 5e-5

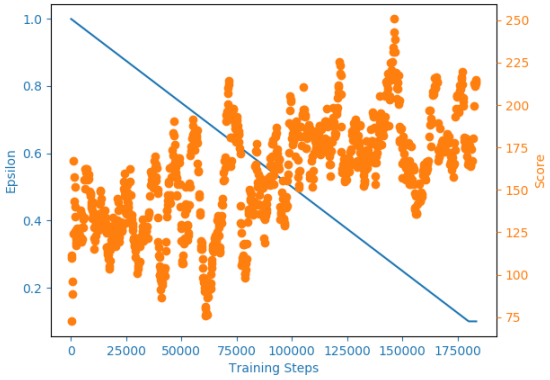



Plot 7) DQN, Decay rate = 5e-6

A decay rate of 5e-5 is better for DQN. Plot 2 is more stable than plot 7.




Plot 3) DDQN, Decay rate = 5e-5



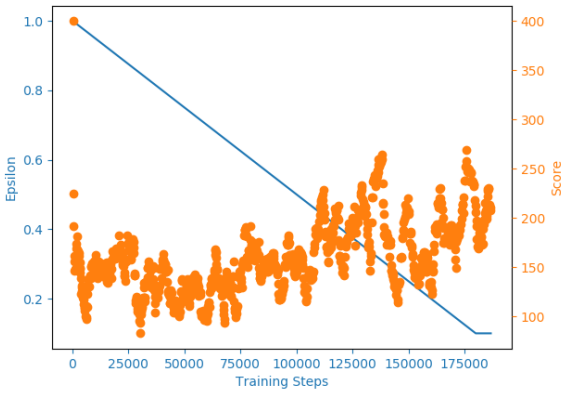

Plot 8) DDQN, Decay rate = 5e-6

A decay rate of 5e-5 is better for DDQN. Plot 3 converges towards a higher score compared to plot 8. Plot 3 is also more stable during training.

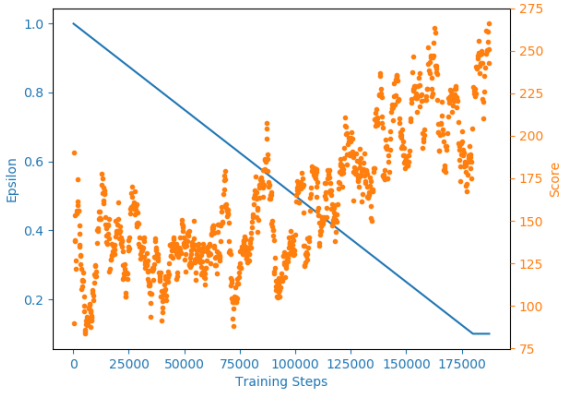
For DQN and DDQN, a decay rate of 5e-5 is better.

**Batch Normalization**

We experimented with batch normalization in the network for DQN and DDQN.

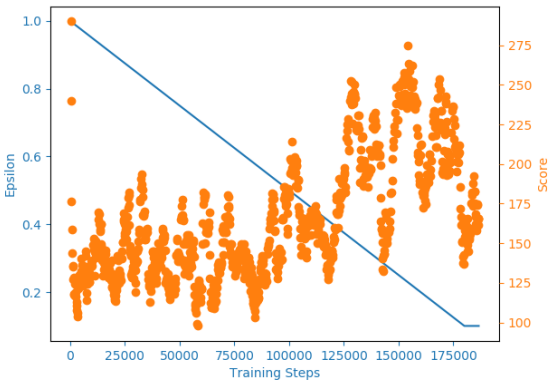



Plot 9) DQN, without batch normalization

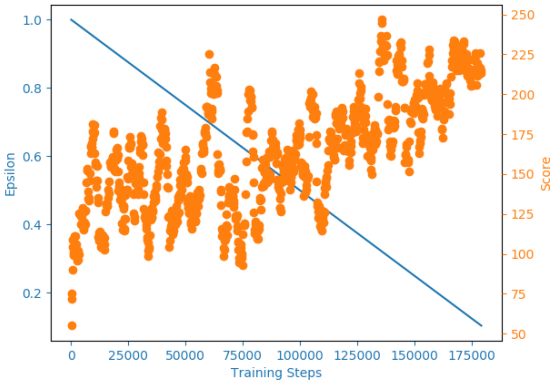



Plot 10) DQN, with batch normalization

Using batch normalization is better for DQN. From plot 10, we can see that it converges at a higher score.




Plot 11) DDQN, without batch normalization



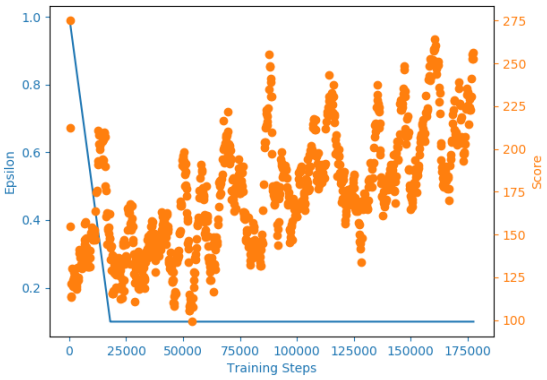

Plot 12) DDQN, with batch normalization

Using batch normalization is better for DDQN. From plot 12, it can be seen it makes the training more stable and converges at a higher score.

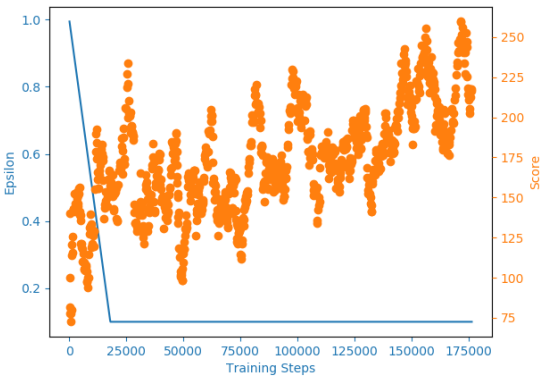
Thus, for both DQN and DDQN, using batch normalization is better. Comparing plot 10 and plot 12, we can see that DQN and DDQN comes very close when comparing performance. DQN converges at a higher score but DDQN is more stable during training and has less variance in the scores. Overall, DDQN is better.

**DQN vs DDQN (Using best parameters for each)**

| Parameters | N | DDQN |
|---|---|---|
| Learning Rate | 0.00025 | 0.0001 |
| Gamma | 0.99 | 0.99 |
| Epsilon Decay Rate | 5e-5 | 5e-5 |
| Batch Normalization | Yes | Yes |




Plot 13) DQN




Plot 14) DDQN

As seen from the above graph, DDQN is more stable, and therefore is a better solution than DQN for playing Space Invaders.

## Conclusion

Overall, we can conclude that using DDQN is better than using DQN. DDQN addresses the major issue with using DQN, which is the overestimation problem. DDQN tackles this by using two networks to calculate the q-value.

An interesting thing to note is that although we chose the better parameters for DDQN for plot 14, it does not perform as well as the DDQN in plot 3. The DDQN in plot 3 is the same as the one in plot 14 except that it does not use batch normalization. This is different from what we concluded before, which is that batch normalization is better for training.

One reason is that although some tests have shown batch normalization improving training results, it's an optimization to help train faster, so we shouldn't think of it as a way to make our network better. On the other hand, since it lets you train networks faster, it means we can iterate over more designs in a shorter amount of time. It also lets you build deeper networks, which are usually better. So when you factor in everything, there is still a chance that we are going to end up with better results if we build our networks with batch normalization.

Another possible reason might be that different hyper parameters can affect each other and its effect on training efficiency is not independent of one another.

## Suggestion for Further Improvement

We suggest using policy gradient methods. The policy implied by Q-Learning is deterministic. This means that Q-Learning can't learn stochastic policies, which can be useful in some environments. It also means that we need to create our own exploration strategy since following the policy will not perform any exploration. We usually do this with $\epsilon$-greedy exploration, which can be quite inefficient.

There is no straightforward way to handle continuous actions in Q-Learning. In policy gradient, handling continuous actions is relatively easy. As its name implies, in policy gradient we are following gradients with respect to the policy itself, which means we are constantly improving the policy. By contrast, in Q-Learning we are improving our estimates of the values of different actions, which only implicitly improves the policy. You would think that improving the policy directly would be more efficient, and indeed it very often is.

In general, policy gradient methods have very often beaten value-based methods such as DQNs on modern tasks such as playing Atari games.

# 5 References

Buchholz, M. (March 17, 2019), Deep Reinforcement Learning. Introduction. Deep Q Network (DQN) algorithm. Retrieved from https://medium.com/@markus.x.buchholz/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862

deeplizard, Reinforcement Learning - Goal Oriented Intelligence. Retrieved from https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv

Moghadam, P.H. (July 22, 2019), Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay. Retrieved from https://medium.com/@parsa_h_m/deep-reinforcement-learning-dqn-double-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9823

Tabor, P. (November 25, 2019), Deep-Q-Learning-Paper-To-Code. Retrieved from https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code

Chablani, M. (June 28, 2017), Batch Normalization. Retrieved from https://towardsdatascience.com/batch-normalization-8a2e585775c9