COMPGI13: Advanced Topics in Machine Learning Assignment #3

Due on Tuesday, April 11, 2017

Thore Graepel, Koray Kavukcuoglu and Oriol Vinyals

Nitish Mutha: UCABMUT

Student No: 15113106

Github username: NitishMutha (https://github.com/NitishMutha)

April 11, 2017

Contents

Problem A

Cart-pole

Implemented the code for the cart-pole environment. The default environment has been modified as per the assignment requirements.

- Maximum episode length = 300
- Non-terminating reward = 0
- Terminating reward = -1
- Discount factor = 0.99

Performed various flavors of Q learning follows.

1. Random Policy: 3 episodes

Generated three trajectories under random uniform policy.

Episode lengths	Return(G)
15	-0.860058355
41	-0.662282041
25	-0.777821359

2. Random policy: Mean and Standard Deviation

Ran the Cart-pole for 100 episodes under random policy and calculated mean and standard deviation as shown in table.

Mean(Episode lengths) ± SD	Mean(Return(G)) ± SD
20.55 ± 10.57390656	-0.81778265 ± 0.080895649

3. Batch Q learning:

Implemented the batch Q learning with following parameters with learning rates as 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 0.5.

3.1 Parameters:

- Batch = 32
- Epochs = 100
- Damping Factor = 0.99
- Optimizer= RMSProp

Problem A continued on next page...

3.2 Algorithm:

Algorithm 1 Batch Q learning algorithm

```
1: Initialize Random episodes with random policy
2: Repeat (for each epoch)
3: initialize s
4: choose state s, action a, next state s' from the batch
5: Q ← function approximation with state
6: Q target ← function approximation with next state
7: Train network (linear/hidden layer) ← $loss = 0.5 * ((r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$
```

3.3 Linear Layer:

Constructed a linear layer which takes in 4 inputs of the state observations and outputs 2 actions. Initialized the weights vectors with truncated_normal. So during training, the network learns on minibatch with RMSProp Optimizer over the range of given learning rates. As it can be seen in the Figures below the network gets trained to reach up to 300 episodes length performance, but its not so stable. In order to enhances the stability of the linear layer added L2 regularization.

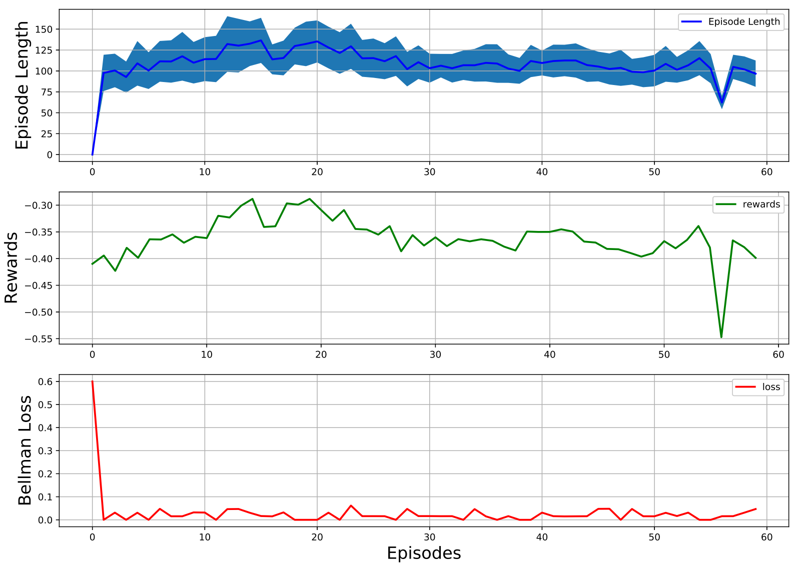

Figure 1: A3: Learning rate 0.00001

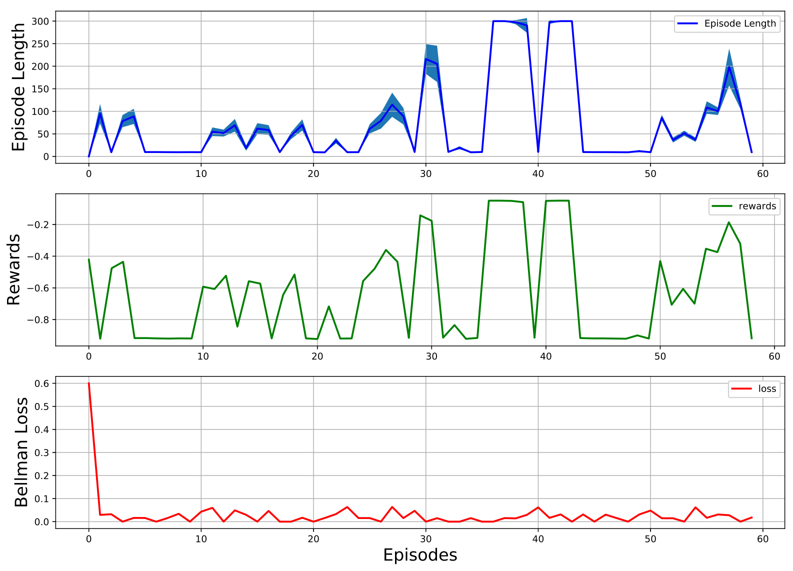

Figure 2: A3: Learning rate 0.0001

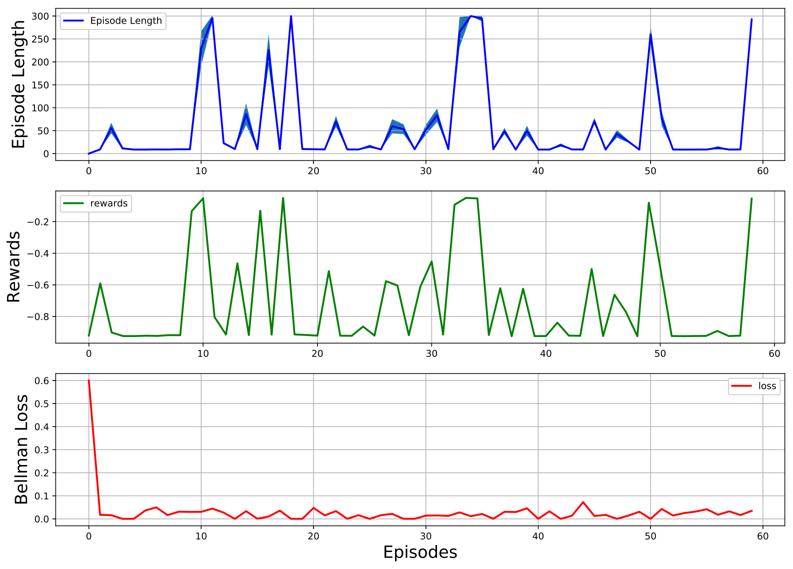

Figure 3: A3: Learning rate 0.001

Problem A continued on next page. . .

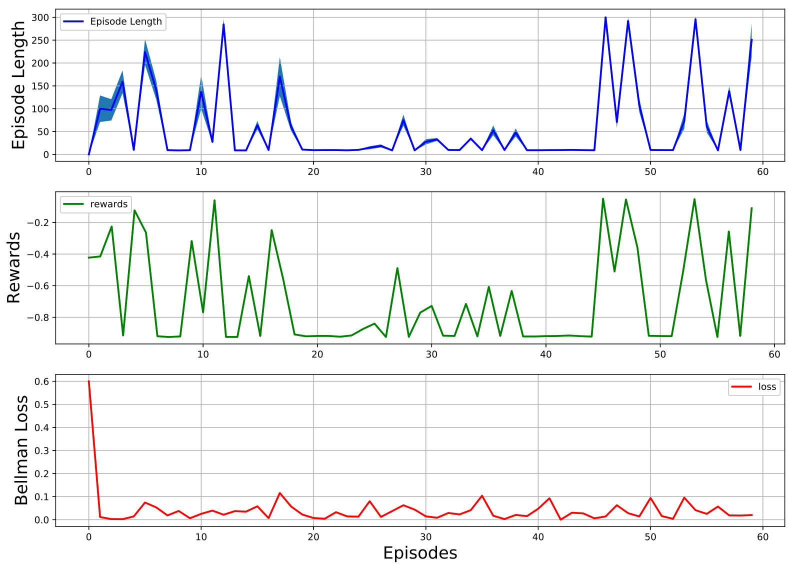

Figure 4: A3: Learning rate 0.01

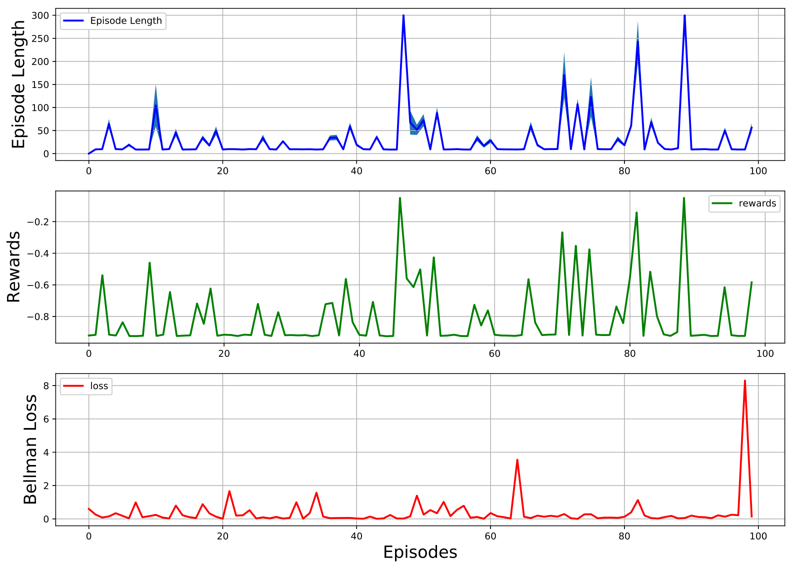

Figure 5: A3: Learning rate 0.1

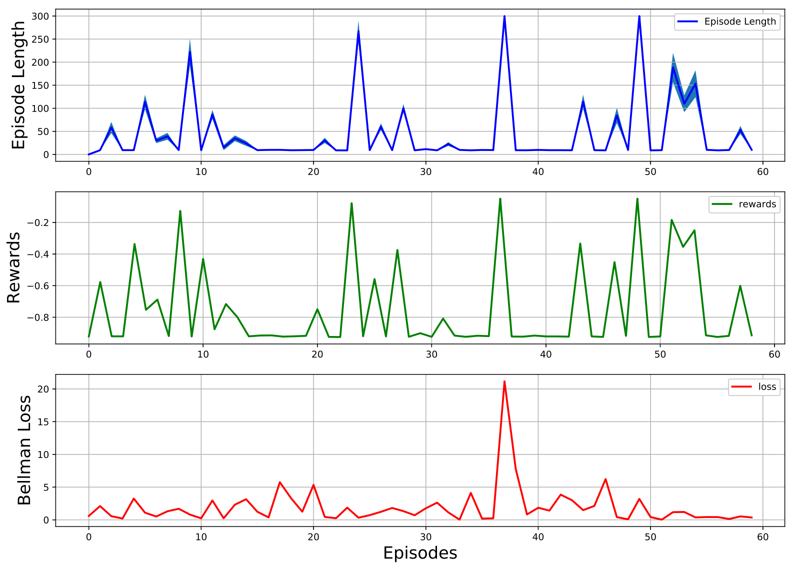

Figure 6: A3: Learning rate 0.5

Test Performance:
After completing the training, ran the agent on the trained environment for 100 episodes based on the best trained weights obtained in training. Results seen in Table.

Learning rate	Mean(Episode lengths)	Mean(Return(G))
0.00001	232.16	-0.167811527
0.0001	300	-0.049536257
0.001	299.97	-0.049536257
0.01	300	-0.049536257
0.1	300	-0.049536257
0.5	300	-0.049536257

3.4 100 Hidden Layer:

Constructed a 100 units hidden layer network with 4 state inputs and 2 action outputs. Used RMSPropOptimizer to train the network over the given range of learning rates. Better performance is seen for 10^{-5}, 10^{-4}, 10^{-3}. Higher learning rates has poor stability.

Problem A continued on next page...

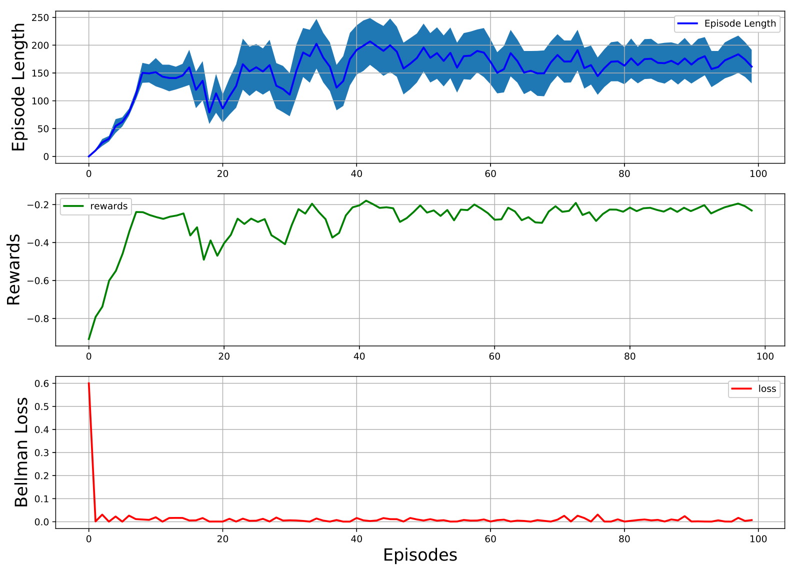

Figure 7: A3 Hidden: Learning rate 0.00001

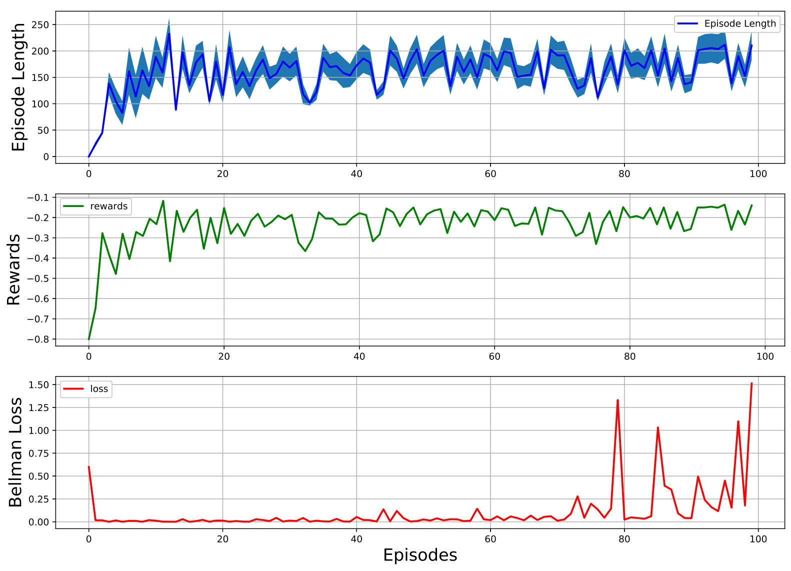

Figure 8: A3 Hidden: Learning rate 0.0001

Problem A continued on next page. . .

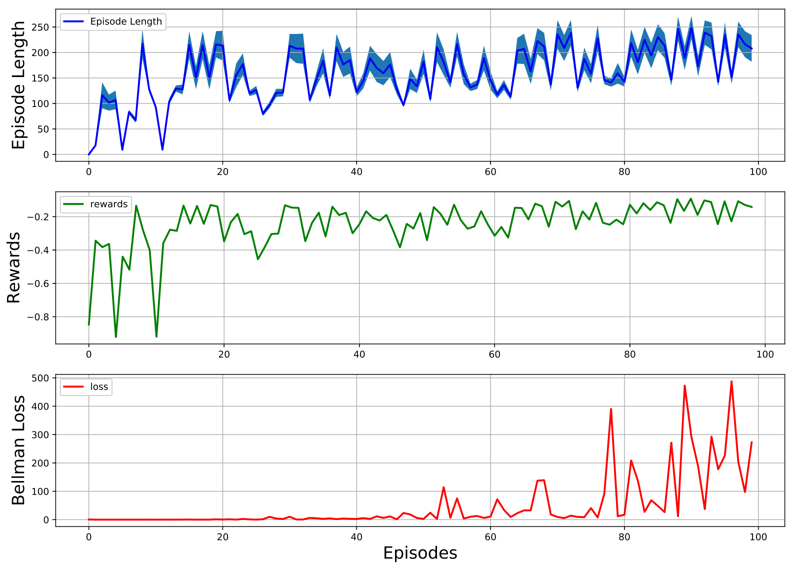

Figure 9: A3 Hidden: Learning rate 0.001

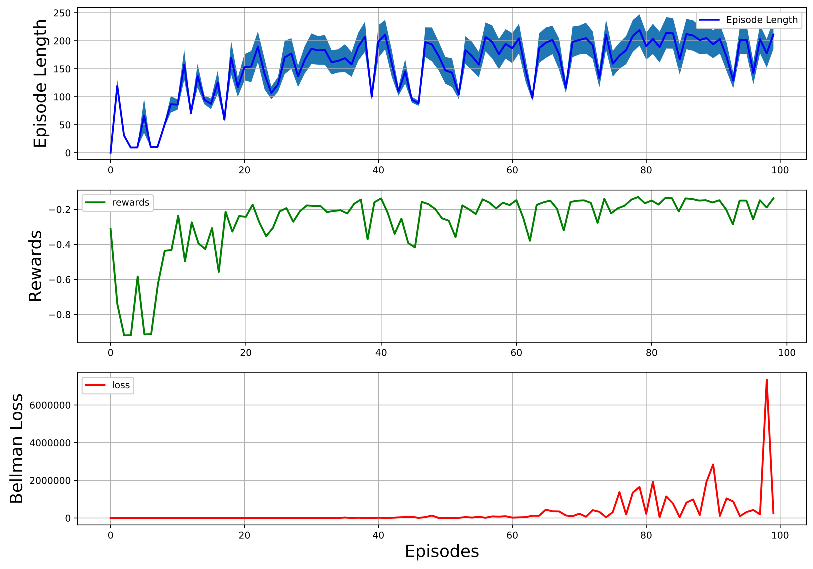

Figure 10: A3 Hidden: Learning rate 0.01

Problem A continued on next page. . .

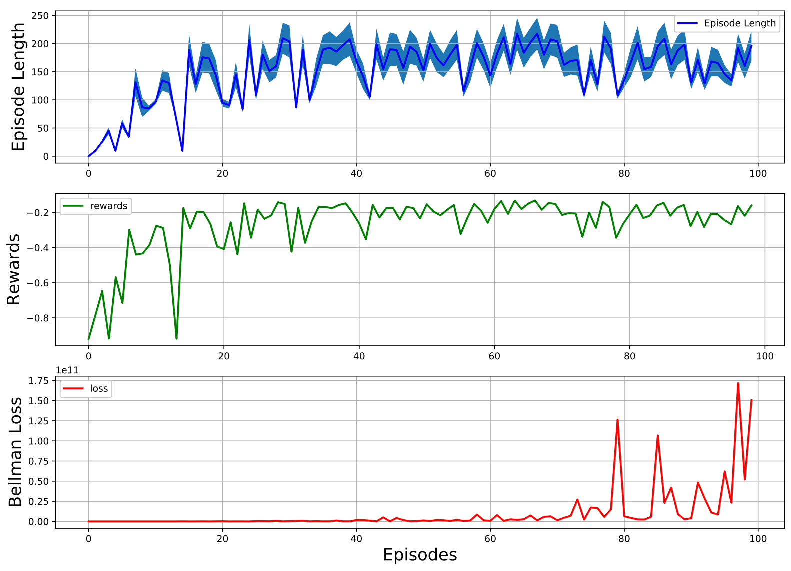

Figure 11: A3 Hidden: Learning rate 0.1

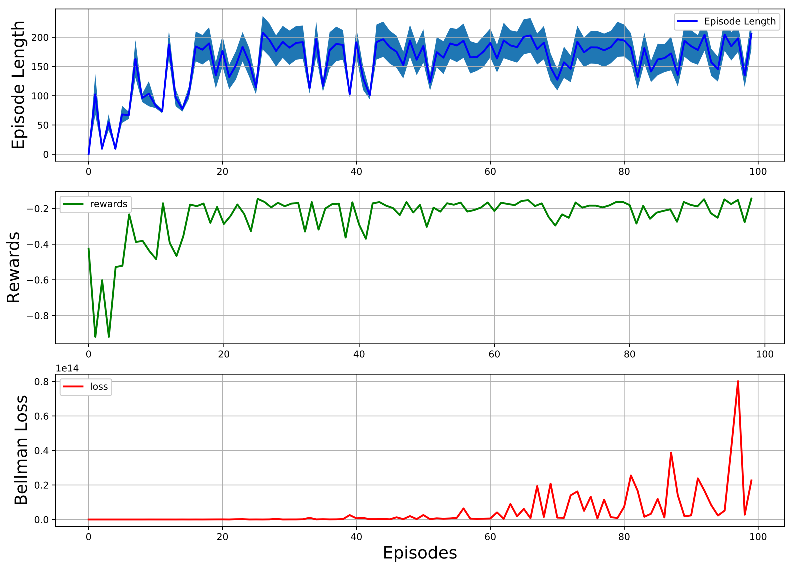

Figure 12: A3 Hidden: Learning rate 0.5

Test Performance:
Hidden layer results are much stable than linear layers, but has low episode lengths in test as compared to linear ones. After completing the training, ran the agent on the trained environment for 100 episodes based on the best trained weights obtained in training. Results seen in Table.

Learning rate	Mean(Episode lengths)	Mean(Return(G))
0.00001	200	-0.207136046455
0.0001	245	-0.111169754794
0.001	245	-0.0946820926081
0.01	217	-0.130710916888
0.1	221.24	-0.130736975817
0.5	196.3	-0.167426517219

4: Online Q learning

Implemented the online Q learning with 100 layers hidden units followed by ReLU. During training the model does converges to 300 episode length for long time, but on multiple experiments to varies a lot due to which the 100 average episode lengths comes down. I have Provided 2 plots demonstrating both 100 average and single experiment runs.

4.1 Parameters:

- Damping Factor = 0.99
- Exploration $\epsilon = 0.05$
- Number of episodes = 2000
- Learning rate= 0.000005
- Optimizer= RMSProp

4.2 Algorithm:

Algorithm 2 Online Q learning algorithm

```
1: Repeat (for each episode)
2: initialize s
3: Choose action a ($\epsilon$ greedy) and state s using policy derived from Q
4: Take action a and observe reward r and next state s'
5: Q target ← function approximation with next state s'
6: Train network ← $loss = 0.5 * ((r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$
7: state s ← next state s'
8: until terminal state
```

Problem A continued on next page...

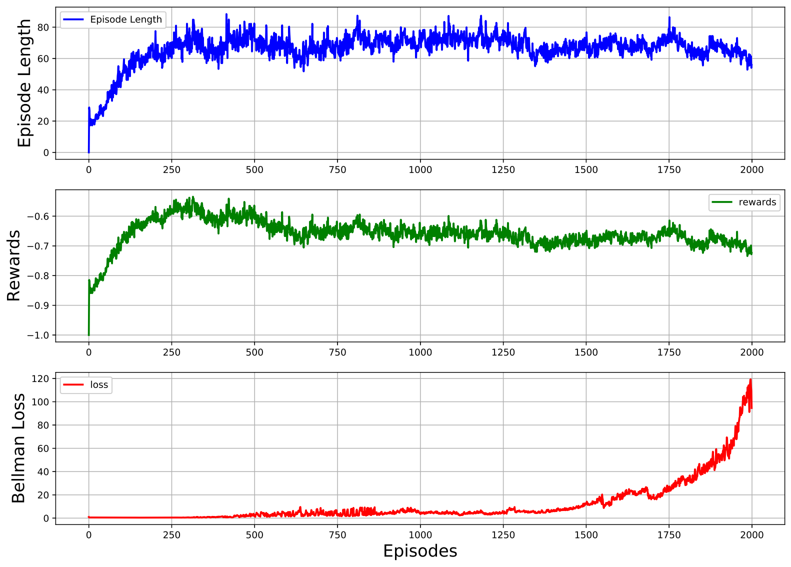

Figure 13: A4: Training plot for average of 100 experiments

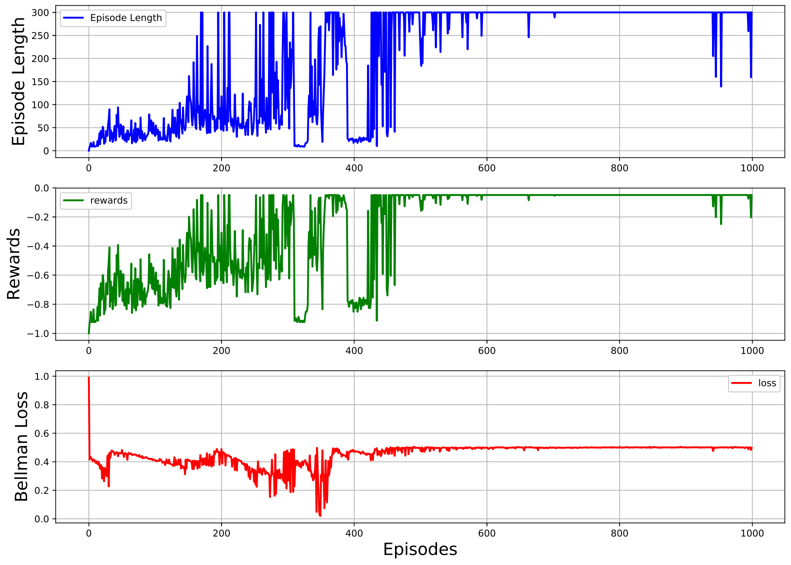

Figure 14: A4: Training plots for single run

Test Performance:
After completing the training, ran the agent on the trained environment for 100 episodes after storing the best trained weights. The model with 100 unit of hidden layer with online learning does converge to 300 steps and stays at that level. But in multiple runs of the same experiment, some iterations agent did badly due to which the overall average of episode lengths came down in the graph shown below. In the graph for

single run of the agent, it can be seen that the agent converges after 600 episode. Average test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.0495362566377

5: Online Q learning with different hidden layer units

Following the online Q learning developed in the problem A4, we now run the same model with different hidden layers. We find out that will less hiddent layers, model does reach the optimal solution of 300 steps per episode but it fails to stay consistent. While the 1000 layer network takes time to reach peak but does quite well and converges for quite some time then spikes down for a step and again recovers. Same parameters were used as in problem A4.

5.1 30 units hidden layer

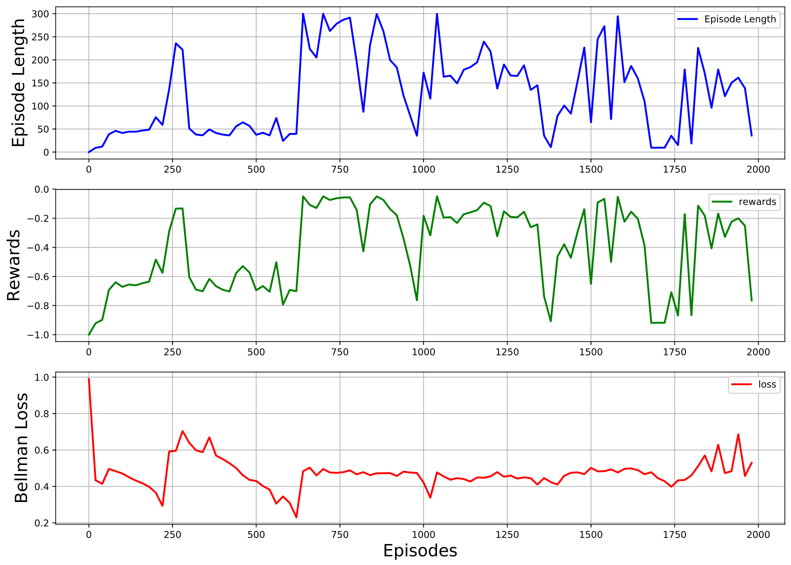

Figure 15: A5: Training plots 30 hidden layer

Test Performance:
After completing the training, ran the agent on the trained environment for 100 episodes, with the best trained weights during training. As compared to 100 unit hidden layers, 30 units does not converge in 2000 episodes and is quite unstable. It does touches 300 steps mark but falls quite often. Average test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.049536257

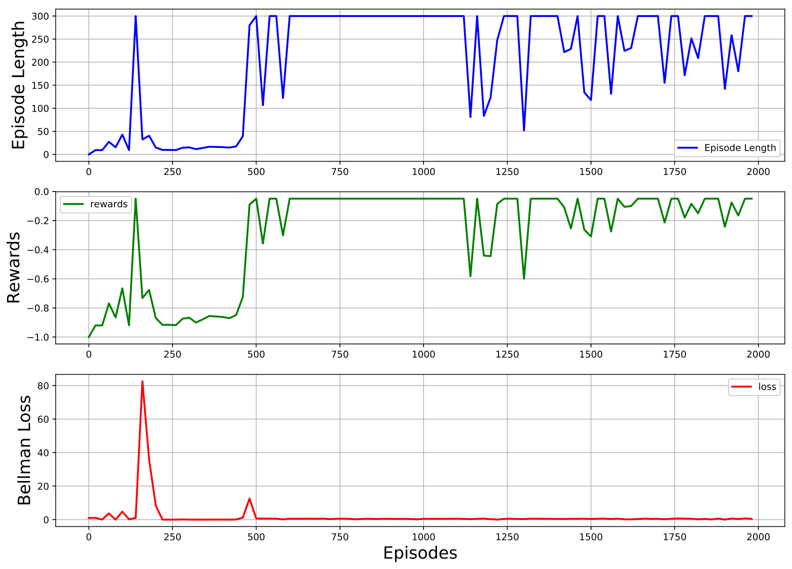

Figure 16: A5: Training plots 1000 hidden layer

5.2 1000 units hidden layer

Test Performance:
After completing the training, ran the agent on the trained environment for 100 episodes after storing the best trained weights. As compared to 100 unit hidden layers, 1000 units seems to converge pretty quick, but after about 1000 steps it starts getting somewhat unstable. Average test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.049536257

6: Online Q learning with Experience Replay

6.1 Parameters:

- Experience Buffer = 500
- Batch = 20
- Damping Factor = 0.99
- Exploration $\epsilon = 0.05$
- Number of episodes = 2000
- Learning rate = 0.00001
- Optimizer= RMSProp

6.2 Algorithm:

Algorithm 3 Online Q learning with Experience Replay

```
1: Initialize Experience buffer
2: Repeat (for each episode)
3: initialize s
4: Choose action a ($\epsilon$ greedy) and state s using policy derived from Q
5: Take action a and observe reward r and next state s'
6: Save the new experience <s,a,r,s'> into the experience buffer and remove the oldest.
7: Take minibatch of experience replay buffer
8: Q $\leftarrow$ function approximation with batch state s
9: Q target $\leftarrow$ function approximation with batch next state s'
10: Train network using mini batch of experience buffer $\leftarrow loss = 0.5 * ((r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$
11: state s $\leftarrow$ next state s'
12: until terminal state
```

Problem A continued on next page. . .

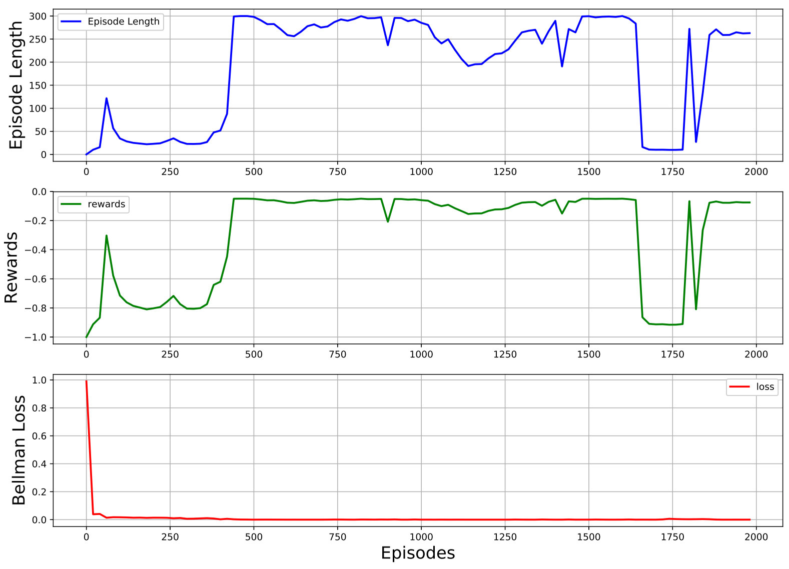

Figure 17: A6: Training plots experience reply

6.3 Test Performance:

Adding the experience replay buffer to the online Q learning improved stability and convergence. After completing the training, ran the agent on the trained environment for 100 episodes after storing the best trained weights. Averaged test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.049536257

7: Q learning with target network

7.1 Parameters:

- Experience Buffer = 10000
- Batch = 500
- Damping Factor = 0.99
- Exploration $\epsilon = 0.05$
- Number of episodes = 2000
- Learning rate = 0.0001
- Target Network update steps = 5
- Optimizer = RMSProp

7.2 Algorithm:

Algorithm 4 Online Q learning with Target Network

```
1: Initialize Experience buffer
2: Repeat (for each episode)
3: initialize s
4: Choose action a ($\epsilon$ greedy) and state s using policy derived from Q
5: Take action a and observe reward r and next state s'
6: Save the new experience <s,a,r,s'> into the experience buffer and remove the oldest.
7: Take minibatch of experience replay buffer
8: Q ← function approximation with batch state s
9: Q target ← function approximation with batch next state s'
10: Train network using mini batch of experience buffer ← $loss = 0.5 * ((r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$
11: state s ← next state s'
12: Every 5 steps update target network
13: until terminal state
```

Problem A continued on next page...

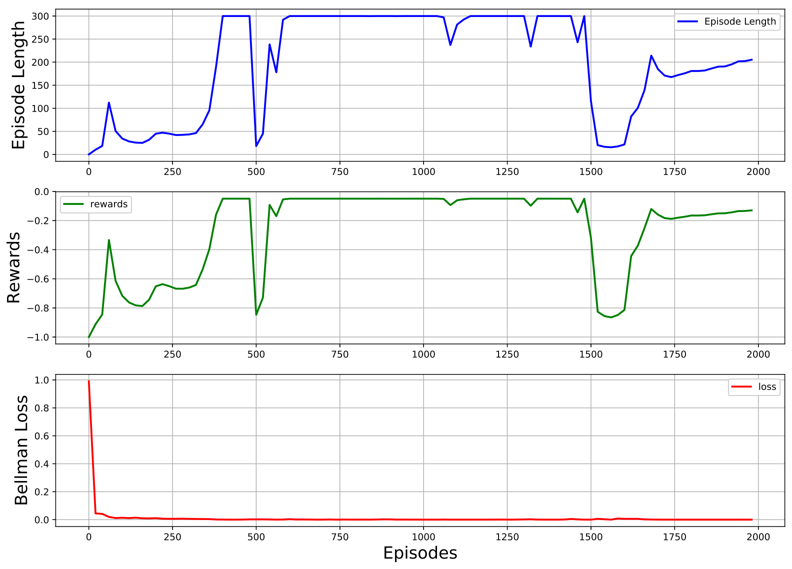

Figure 18: A7: Training plots target networks

7.3 Test Performance:

The target network gets converged and stable as compared to the previous implemented algorithms, until at one point where it falls down and recovers again. After completing the training, ran the agent on the trained environment for 100 episodes after storing the best trained weights. Averaged test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.049536257

8: Modification: Sarsa

Implemented the sarsa as the modified version.

8.1 Parameters:

- Experience Buffer = 10000
- Batch = 500
- Damping Factor = 0.99
- Exploration $\epsilon = 0.05$
- Number of episodes = 2000
- Learning rate = 0.0001
- Optimizer = RMSProp

8.2 Algorithm:

Algorithm 5 Sarsa [2]

```
1: Initialize buffer
2: Repeat (for each episode)
3: initialize s
4: Choose action a ($\epsilon$ greedy) and state s using policy derived from Q
5: Take action a and observe reward r and next state s'
6: Repeat for each step in episode
7: Choose next action a' ($\epsilon$ greedy) and next state s using policy derived from Q'
8: Q $\leftarrow$ function approximation with batch state s
9: Q target $\leftarrow$ function approximation with batch next state s'
10: Train network $\leftarrow loss = 0.5 * ((r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$
11: state s $\leftarrow$ next state s', action a $\leftarrow$ next action a'
12: until terminal state
```

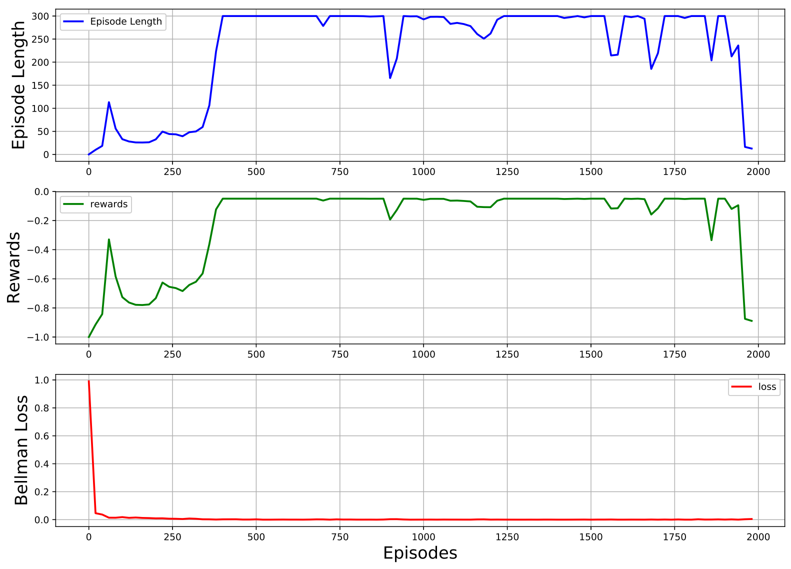

Figure 19: A8: Training plots SARSA

8.3 Test Performance:

As compared to the previous versions of the Q learning algorithms, SARSA was observed to be the most stable of all. It converges very quickly and comparatively stable performance. After completing the training, ran the agent on the trained environment for 100 episodes after storing the best trained weights. Averaged test results seen in Table below.

Mean(Episode lengths)	Mean(Return(G))
300	-0.049536257

Problem B:

Atari Game Env

Prepossessing Observation space:

The observation space of all three Atari games are 210x160x3 in dimensions. So in order to make it computationally light weight we perform certain pre-processing tasks on the observations of these environments. Also as the aspect ratio of the games were not symmetric hence cropped out the unwanted regions of the game to make it a sqaure of 160x160 and then perform the resize which will restore the game aspect ratio. In the following section, I will describe each one of them in detail.

1. Pong-v3

While resizing the frame of Pong environment to 28x28x3, I realized that the ball gets disappeared due to interpolation applied while straight resizing form 210x160 to 28x28. Instead we can tackle the issue of the missing ball, by performing step resize so that we can at each stage due to interpolation we retain the ball. Along with this since the source frame is not a square, it will result in distortion hence will lead to poor learning of the actual environment. In-order to overcome these issues, I performed the following enhance version of the pre-processing. And after testing performance with 28x28 which was not good, I decided to go with 60x60 resolution. [3]

Steps:

1. Crop frame 210x160 → 160x160 (cropping out scores and just keeping active game area)
2. Remove background colors
3. Gamma correction
4. Step resize 160x160 → 120x120
5. Step resize 120x120 → 90x90
6. Step resize 90x90 → 60x60
7. RGB to Gray

Resulting frame after pre-processing

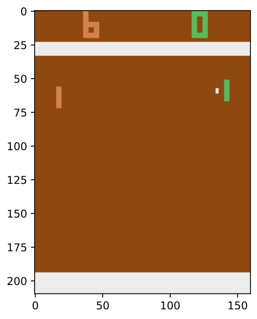

Figure 20: Original frame

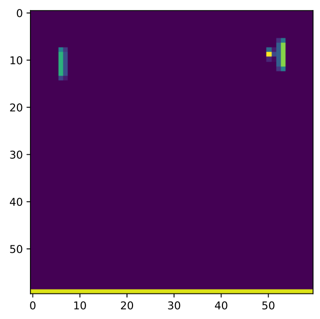

Figure 21: Preprocessed frame

2. Boxing-v3

Similarly for the Boxing game, cropped out the unwanted regions of the observation space and resize along with gray scaling the frame. Preprocessing for this game was fairly straightforward. After gray scaling the distinction between the black player and the background was not so clear, hence performed gamma correction, which improved the contrast between them.

Steps:

1. Crop frame 210x160 → 143x143 (cropping out scores and just keeping active game area)
2. Remove background colors
3. Step resize 143x143 → 60x60
4. RGB to Gray

Resulting frame after pre-processing

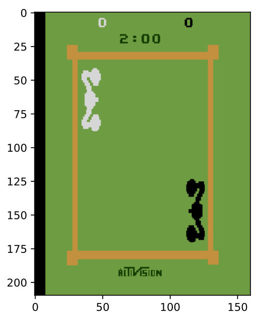

Figure 22: Original frame

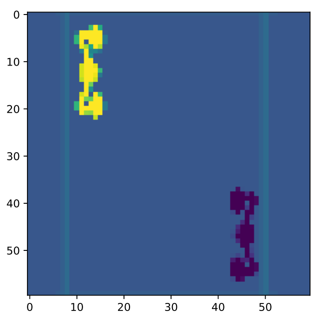

Figure 23: Preprocessed frame

3. MsPacman-v3

In MsPacman also since there is a lot of information, while resizing and grey scaling the frame, I ensure that even if the objects in the environment gets pixelated, any objects shall not disappear. More attention was put into preserving the chips which pacman feed as that would be one of the important signal for the agent to maximize its reward and strategy.

Steps:

1. Crop frame 210x160 → 160x160 (cropping out scores and just keeping active game area)
2. Remove background colors
3. Step resize 160x160 → 128x128
4. Step resize 128x128 → 60x60
5. RGB to Gray

Problem B: continued on next page...

Resulting frame after pre-processing

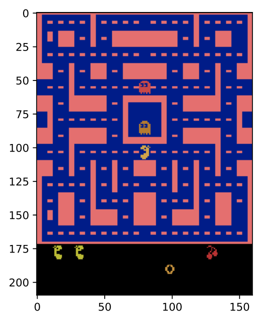
Figure 24: Original frame

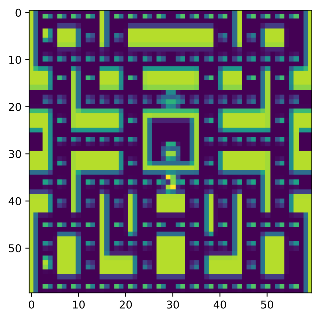
Figure 25: Preprocessed frame

Function Approximation (Baseline version)

Since the observation space is the frame of pixels, we use the convolution neural networks to approximate Q value function. Implemented Convolution Neural Network with following specifications.

- First Layer
  - Filter : 6x6
  - Stride : 4 (since the dimension chosen were higher, increased the stride to reduce computational complexities.)
  - Channels : 16
  - Nonlinear function: ReLU
- Second Layer
  - Filter : 4x4
  - Stride : 2
  - Channels : 32
  - Nonlinear function: ReLU
- Flatten Layer
  - Fully connected : 256
- Linear layer
  - Output size : number of actions for that environment

Parameters:

- Batch size = 32
- Experience Buffer = 100000
- Clipped rewards = -1, 0 or 1

Problem B: continued on next page. . .

- Damping Factor = 0.99
- Exploration $\epsilon = 0.1$
- Learning rate = 0.001
- Training steps = 1 Million
- Target network update = 5000 steps
- Agent evaluation = 50000 steps

Function Approximation (DeepMind's DQN paper version) [1]

After training with the baseline version of the model, I implemented the DeepMind's DQN version of the model in order to improve training with 48x48 frame size. Final submitted coded has this specs.

- First Layer
  - Filter : 8x8
  - Stride : 4 (since the dimension chosen were higher, increased the stride to reduce computational complexities.)
  - Channels : 32
  - Nonlinear function: ReLU
- Second Layer
  - Filter : 4x4
  - Stride : 2
  - Channels : 64
  - Nonlinear function: ReLU
- Third Layer
  - Filter : 3x3
  - Stride : 1
  - Channels : 64
  - Nonlinear function: ReLU
- Flatten Layer
  - Fully connected : 256
- Linear layer
  - Output size : number of actions for that environment

Problem B: continued on next page. . .

Parameters:

- Batch size = 32
- Experience Buffer = 200000
- Clipped rewards = -1, 0 or 1
- Damping Factor = 0.99
- Exploration ϵ = 1 to 0.1 decaying
- Learning rate = 0.001
- Training steps = 2 Million
- Target network update = 5000 steps
- Agent evaluation = 50000 steps

1. Random Policy: 100 episodes:

Evaluated the agents performance on the three Atari games under random uniform policy.

Game	Mean(Frame count) ± SD	Mean(Reward) ± SD	Mean(Score)
Pong-v3	1225 ± 130.252524	-0.92844566 ± 0.263646154	-21 ± 0.848528137424
Boxing-v3	2381.06 ± 15.46856167	-0.335109517 ± 1.140889224	1.42 ± 3.904353
MsPacman-v3	632.44 ± 83.5600765916	2.40859862079 ± 0.594039004181	17.43 ± 5.43

2. Random policy: Initialized but untrained Q-network

Evaluated the agents performance on the three Atari games with initialized but untrained Q-network.

Game	Mean(Frame count) ± SD	Mean(Score) ± SD	Mean(Score)
Pong-v3	1019.97 ± 8.903319606	-1.130669386 ± 0.034373787	-21 *pm* 0.0
Boxing-v3	2380.1 ± 12.8526261908	-0.576541221318 ± 0.0255066898858	-27 ± 2.9336529
MsPacman-v3	432.97 ± 5.71218872237	2.1947320605 ± 0.0660771004106	6 ± 0.0

3. Training agents

The models for all three games have been trained with following parameter along with the aforementioned preprocessing steps.

The shapes of the curves differs from that we observe in Supervised Learning as in the Reinforcement learning environment the data in not iid. It is quite random and depends on the previous frames in the game. Hence due to this the learning curves are not as stable as we would expect in Supervised Learning. Model does converges but there are few spikes in between which reduces the rewards. The training loss curve decays exponentially.

MsPacman-v3

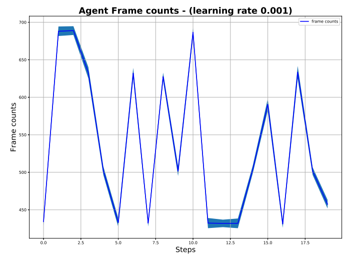

Figure 26: MsPacman Frames

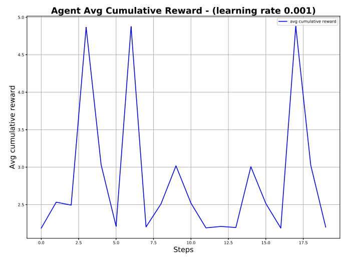

Figure 27: MsPacman Rewards

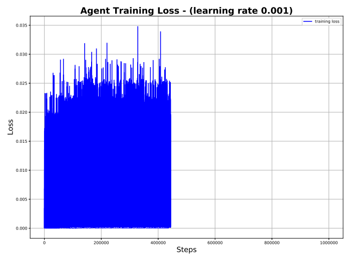

Figure 28: MsPacman Loss

Pong-V3

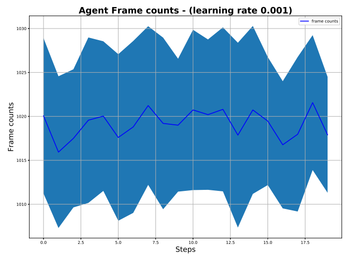

Figure 29: Pong Frames

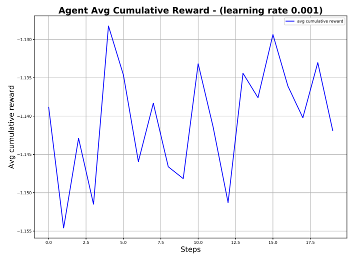

Figure 30: Pong Rewards

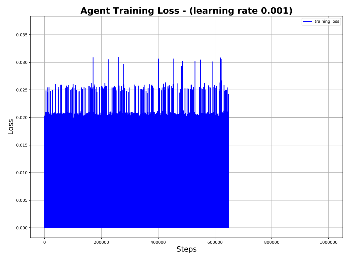

Figure 31: Pong Loss

Boxing-v3

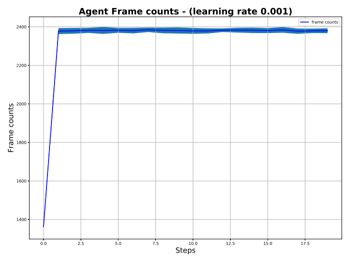

Figure 32: Boxing Frames

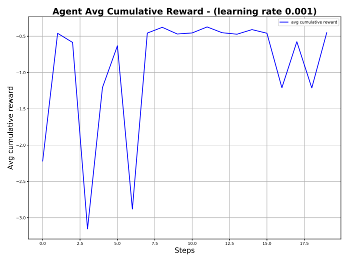

Figure 33: Boxing Rewards

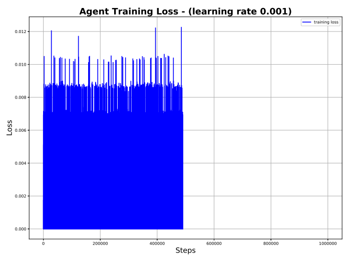

Figure 34: Boxing Loss

4. Test performance

Final test performance after completing the training, ran the agent on the trained environment for 100 episodes. It has been observed that given the limited computational resources and time after training for 2 million time-steps the agent we can see a learning trend in the agents performance. Averages results seen in Table below.

Agent	Mean(Frame lengths)	Mean(Cumulative Discounted Return)	Mean(Score)
Pong-v3	1329.23 ± 2.993	-0.89763 ± 0.3295	19 ± 0.7653252
Boxing-v3	2501.93 ± 2.349	0.459837 ± 0.0231539	3 ± 1.284
MsPacman-v3	783.07 ± 7.09753	4.89759639104 ± 0.396523	26.0 ± 4.89

I have tried implementing the baseline provided in the assignment sheet with 28x28 image resolution but it didn't seem to train the agent for me well. So later I decided to implement the DeepMind's DQN paper and created a 3 layer ConvNet which takes input of the size 40x40. And trained the model on them. But due to time limitations I was not able to provide graphs for the same. [1]

References

[1] V. Mnih, “Playing atari with deep reinforcement learning.”

[2] “Reinforcement learning.”

[3] A. Karpathy, “Deep reinforcement learning: Pong from pixels,” 2016.