Playing Atari Pong with DQN
Assignment 3
Brijesh Vora

Part 1:
Q1 - (coding) Implement the “randomly sampling” function of replay memory/buffer (see line 89 of dqn.py for TODO and hints). Submit necessary code as sampling.zip.

Ans.

```python
def sample(self, batch_size):
    # TODO: Randomly sampling data with specific batch size from the buffer
    return zip(*random.sample(self.buffer, batch_size))
```

Q2. (written) Is randomly sampling strategy good? Should we treat each frame in the buffer equally?

Ans. Yes, the random sampling is a good strategy as it takes random samples from the buffer and not consecutive samples which after 1000’s of iterations smoothes out the loss function and minimizes it correctly. As the consecutive frames are almost similar, so there’s no point of using consecutive frames and so we want the frames to be different then random sampling makes much more sense.

Q3. (coding) Given a state, write code to calculate the Q value and the corresponding chosen action based on the neural network (see lines 50 ~ 55 in dqn.py). Submit necessary code as action.zip.

Ans.

```python
def act(self, state, epsilon):
    if random.uniform(0,1) <= epsilon:
        action = random.randrange(self.env.action_space.n)
    else:
        state =
Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0),requires_grad=True)
        # TODO: Given state, you should write code to get the Q value and chosen
action
        q_values = self.forward(state)
        action = torch.argmax(q_values).item()
    return action
```

Q4 (written) Given a state, what is the goal of line 48 and line 57 of dqn.py? Aren’t we calculating the Q value and the corresponding chosen state from the neural network?

Ans. Yes, we are. But if the epsilon is greater than or equal to random.uniform(0,1) then we are choosing random action. This is the exploration stage. It will explore through the search space and pick out the patterns that work and which doesn't work. Once it figures out the path where it is getting maximum reward from then it will go into the else part where it exploits the path. It chooses the action which has maximum q values associated with that state. So that's the goal of this function mentioned above.

Q5. (written and coding) Implement the appropriate objective function of DQN (see lines 69 ~ 73 of dqn.py), which is described in the Mitchell Q-learning text on the website. Write which update function did you use and why? Submit necessary code as TD.zip.

```python
def compute_td_loss(model, target_model, batch_size, gamma, replay_buffer):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state = np.concatenate(state)
    next_state = np.concatenate(next_state)

    state = torch.tensor(state, dtype=torch.float, device=DEVICE)
    next_state = torch.tensor(next_state, dtype=torch.float, device=DEVICE)
    action = torch.tensor(action, dtype=torch.long, device=DEVICE)
    reward = torch.tensor(reward, dtype=torch.float, device=DEVICE)
    done = torch.tensor(done, dtype=torch.float, device=DEVICE)

    # Make predictions
    indices = np.arange(batch_size)
    state_q_values = model(state)
    next_states_q_values = model(next_state)
    next_states_target_q_values = target_model(next_state)

    # Find selected action's q_value
    selected_q_value = model.forward(state)[indices, action]
    q_next = target_model.forward(next_state).max(dim=1)[0]

    expected_q_value = reward + gamma * q_next * (1 - done)
    loss = (selected_q_value - expected_q_value.detach()).pow(2).mean()

    return loss
```

As shown above the loss function is implemented and the MSE (Mean Square Error) is used as the weight update equation.

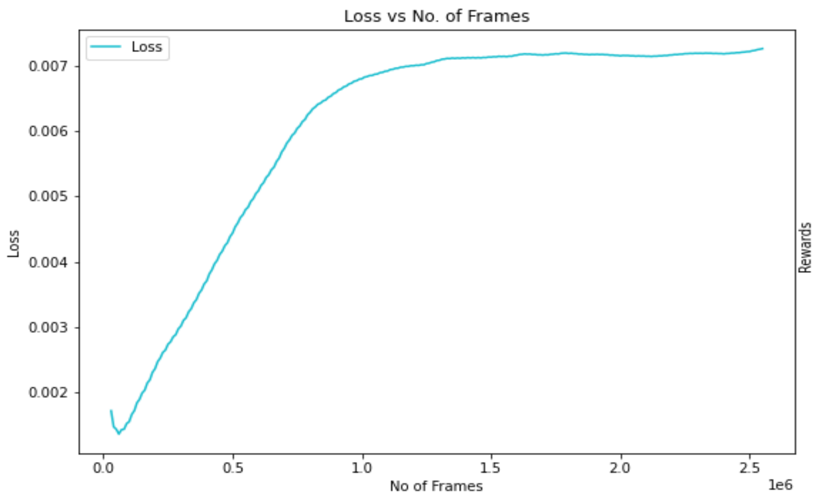
Q7. (written) After you make these changes, train DQN by running run_dqn_pong.py. To achieve relatively good performance, you need to train more than 1000000 frames. It takes ~ 10 hours on a

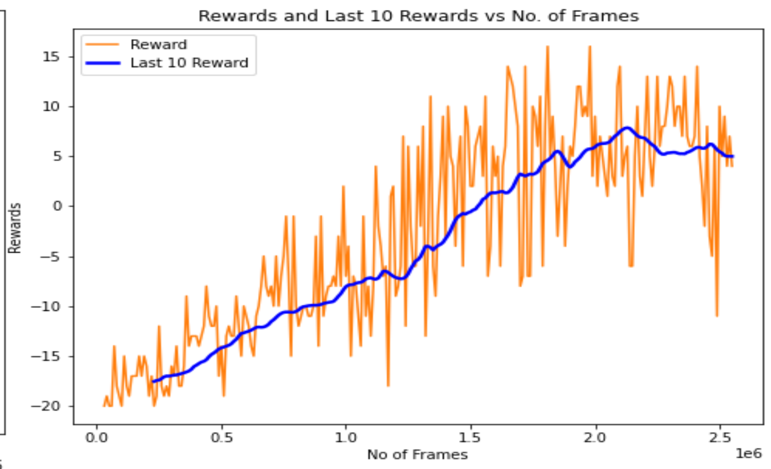
Google Cloud Virtual Machine with 2 vCPUs and 1 NVIDIA Tesla K80 GPU. Explain what parameter tuning you performed such as:

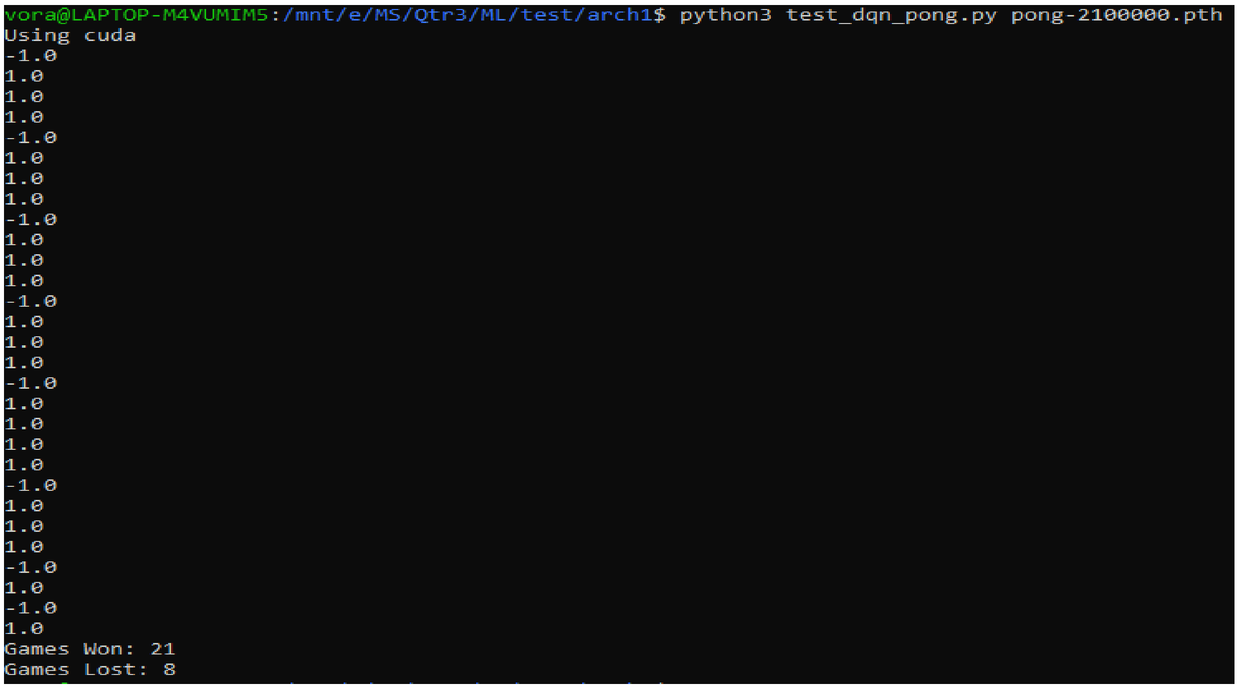
a. You might tune the hyper-parameters like γ and initial size and the size of the replay buffer to gain a better performance.
b. Modify run_dqn_pong.py to track and plot how loss and reward changes during the training process. Attach these figures in your report. Your grade will be based on the ranking of the final reward.

Ans.

Architecture 1;- below images - is the one where I used the number of frames and the same CNN provided in starter code. As we can see, the reward increases very slowly and doesn't give us a very good model. Even after training 2.5M frames, it wins 21 pts and the opponents wins 8pts. After 2.5M frames, the reward saturates and the loss also doesn't go down as the training goes. It takes around 5-6 hrs to train. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

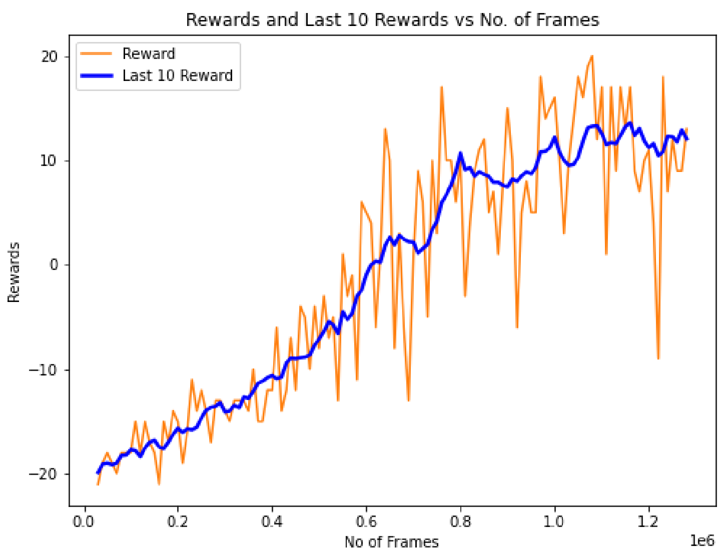


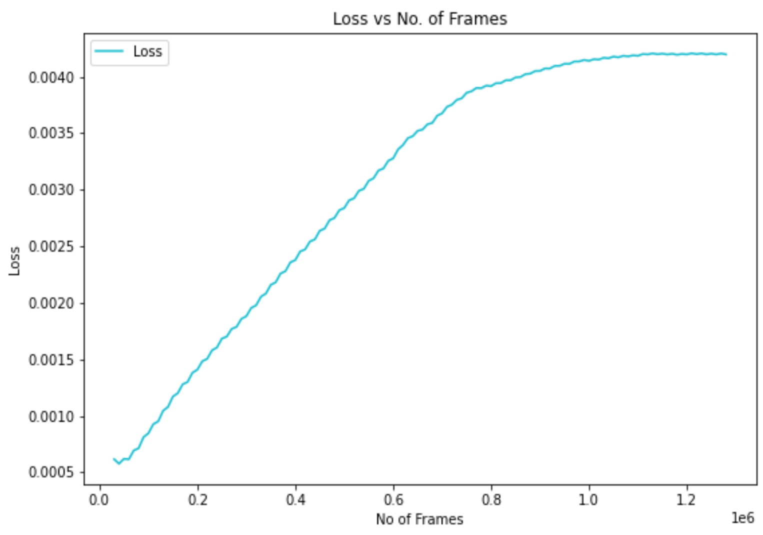


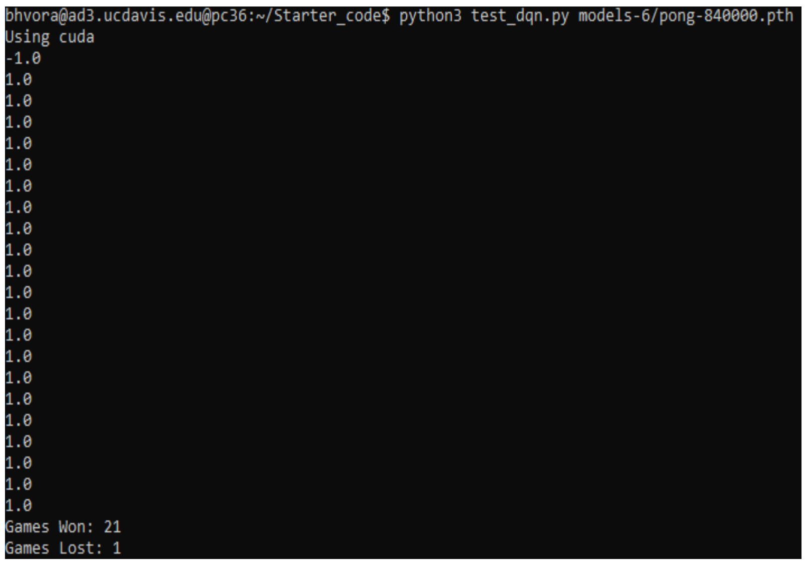


```
vora@LAPTOP-M4VUMIM5:/mnt/e/MS/Qtr3/ML/test/arch1$ python3 test_dqn_pong.py pong-2100000.pth
Using cuda
-1.0
1.0
1.0
1.0
-1.0
1.0
1.0
1.0
-1.0
1.0
1.0
1.0
-1.0
1.0
1.0
1.0
-1.0
1.0
1.0
1.0
1.0
-1.0
1.0
1.0
1.0
-1.0
1.0
-1.0
1.0
Games Won:  21
Games Lost:  8
```

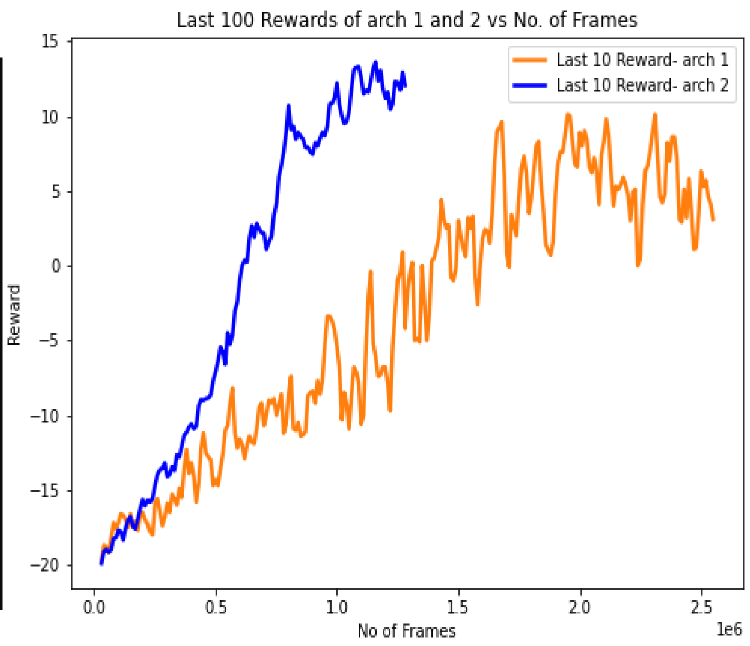
Architecture 2:-







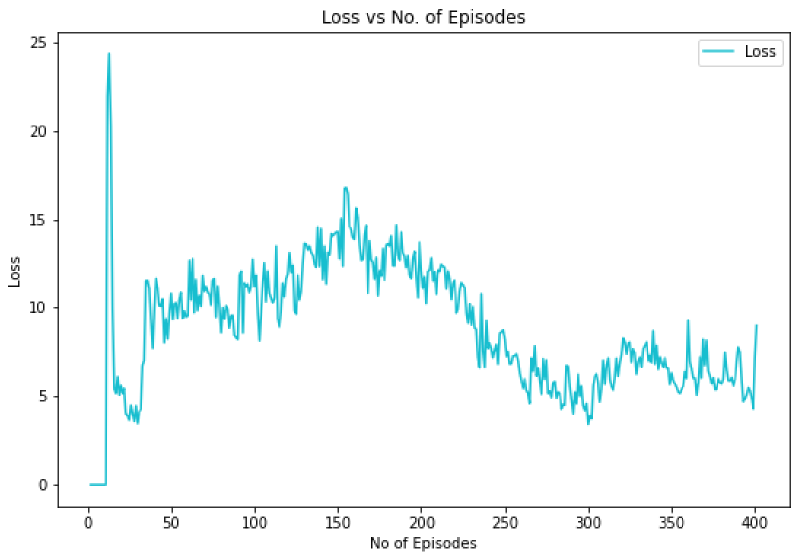
```
bhvora@ad3.ucdavis.edu@pc36:~/Starter_code$ python3 test_dqn.py models-6/pong-840000.pth
Using cuda
-1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
Games Won: 21
Games Lost: 1
```

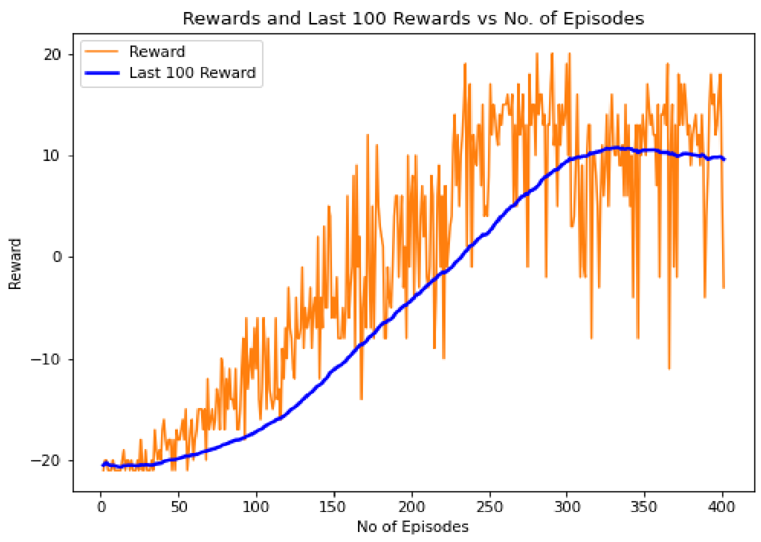


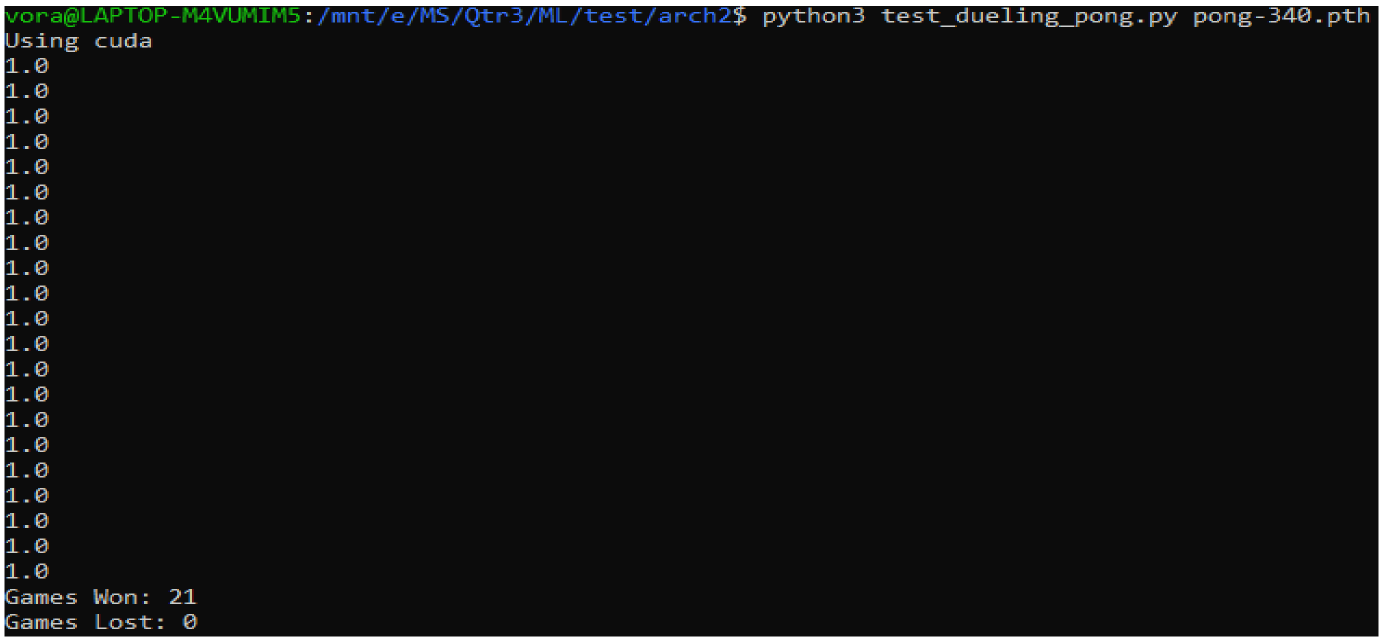
No of Frames	My agent	Game Agent
600000	21	13
740000	21	8
840000	21	1

The models are all in the respective architecture folder and so is the code. All the images are in the images folder.

Part 2:-
Architecture 3:-







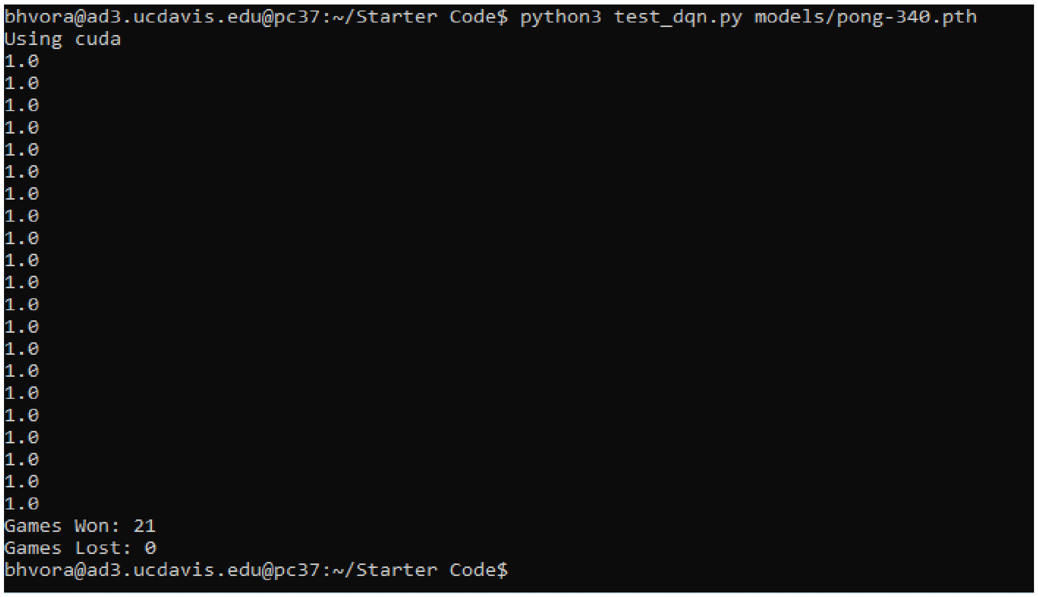
```
vora@LAPTOP-M4VUMIM5:/mnt/e/MS/Qtr3/ML/test/arch2$ python3 test_dueling_pong.py pong-340.pth
Using cuda
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
Games Won: 21
Games Lost: 0
```

Architecture 3 uses the Dueling Dqn discusses here: [1511.06581] Dueling Network Architectures for Deep Reinforcement Learning (arxiv.org)
Just changing the NN architecture and loss function a little bit and instead of training on frames I trained on a number of episodes, I was able to get **best performance in just 340 episodes**. Also it takes just 2 hrs to train.

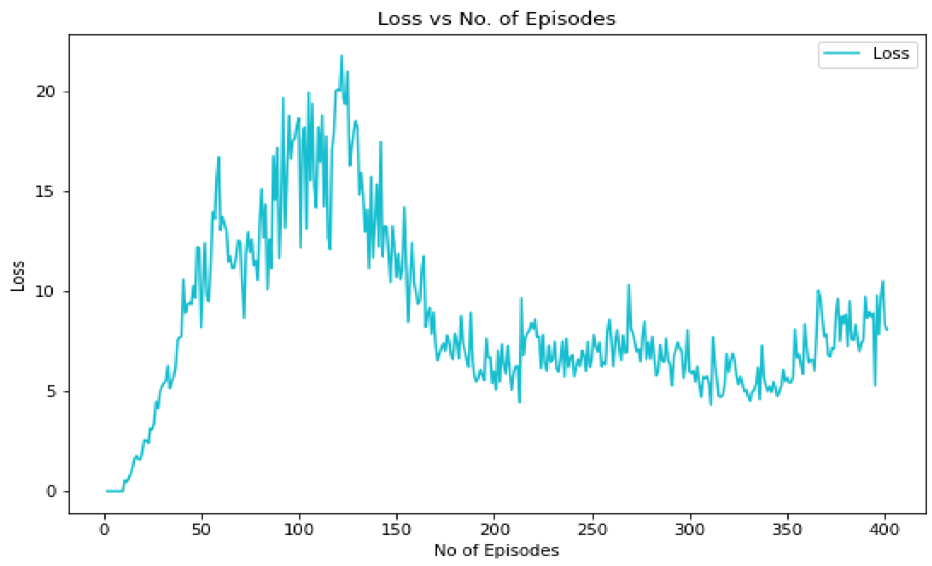
Architecture 4:-

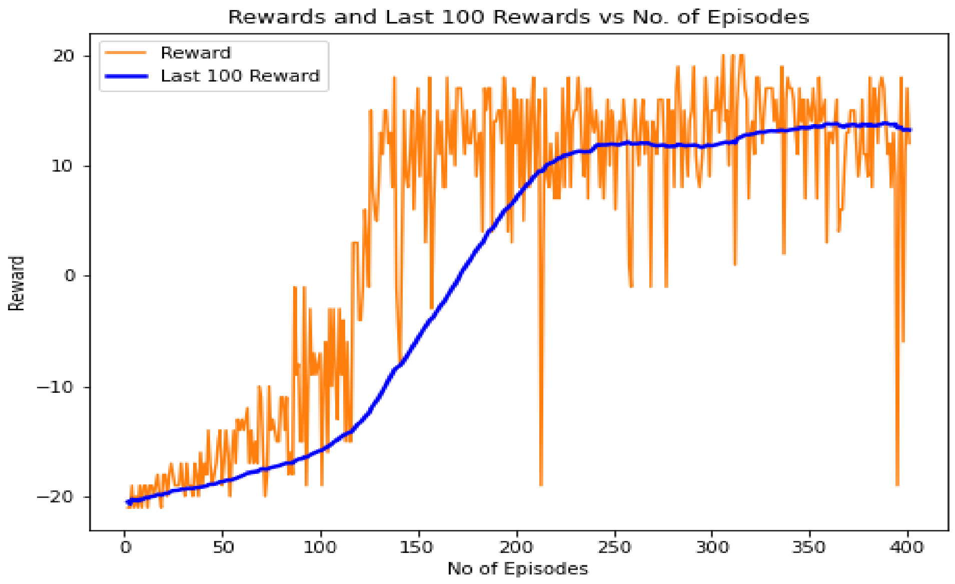
1. Replay Buffer size = 50000
2. Learning rate = 0.001
3. Initial Replay =10000
4. Gamma = 0.99
5. Batch_size = 64
6. No of Episodes = 100000
7. No of steps/episode = 100000
8. Epsilon final = 0.05
9. Epsilon_decay = 0.99

```
if epsilon > epsilon_final:
    epsilon *=epsilon_decay
```



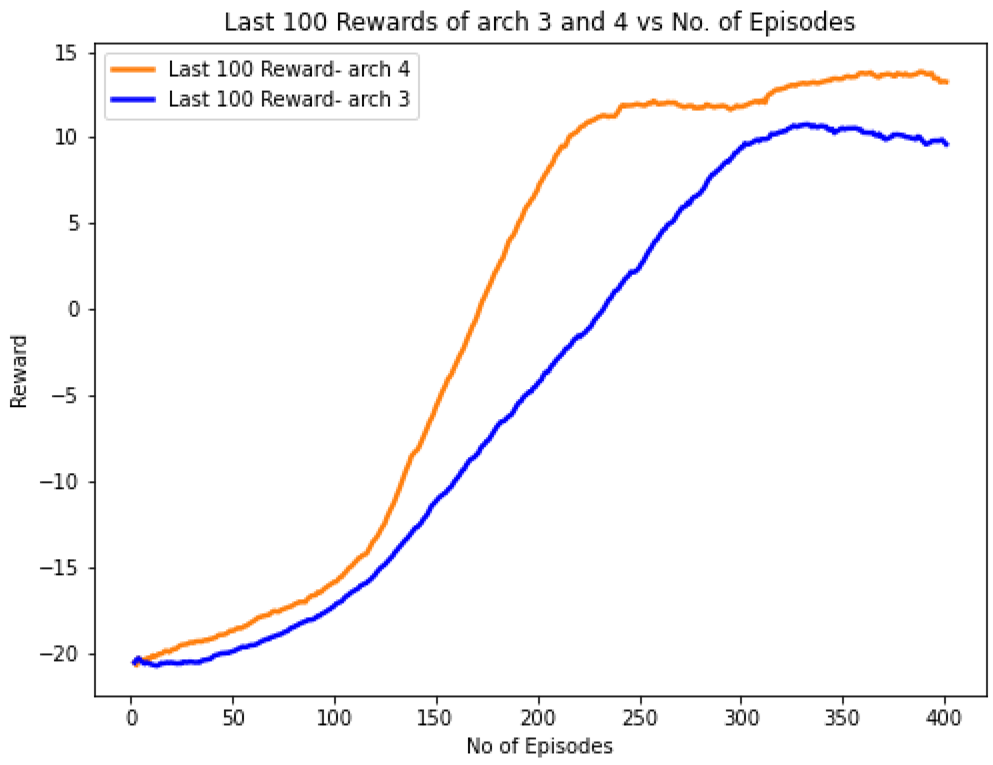
```
bhvora@ad3.ucdavis.edu@pc37:~/Starter Code$ python3 test_dqn.py models/pong-340.pth
Using cuda
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
Games Won: 21
Games Lost: 0
bhvora@ad3.ucdavis.edu@pc37:~/Starter Code$
```





Architecture 4 is just a simple change in starter code in that it has the same **architecture** given in starter code but only the loss function is a bit modified and training on number of episodes instead of frames.

No of Episodes	**My agent**	**Game Agent**
130	21	8
230	21	3
290	21	1
340	21	0



Last 100 Rewards of architecture 3 and architecture 4. As we can see architecture 4 performs **better** than architecture 3.
Surprisingly, Both the architectures trained the model in **340 episodes** - i.e both the model beat the game by **21 - 0** in 340 episodes as you can see in the above pictures.

Also from the above table we can see that episode 130 (trained around 45 mins)performs the same as architecture 1(trained around 5 hrs).