HIRO Group Technical Challenge: Deep Q-Networks (DQN)

Krishna Sai Chemudupati

October 2020

1 Environment

The environment consists of a Lunar lander that needs to land smoothly within the goals at (0,0) to get maximum reward. We get the following information from the game:
State
1. x-coordinate lander
2. y-coordinate of the lander
3. horizontal velocity
4. vertical velocity
5. angle
6. angular velocity
7. left leg touched
8. right leg touched

Actions:
1. No operation (0)
2. Left thruster (1)
3. Main thruster (2)
4. Right thruster (3)

An episode is one complete game, where the episode starts when the lander starts and ends when the lander crashes, or comes to rest on the ground, or steps through a set number of time frames without ending.

2 DQN Algorithm

Initialize memory buffer
Initialize dqn model with random weights θ
Initialize target dqn model with same weights $\theta^{'} = \theta$
for (*e episodes*) {
  Reset state
  for (*t timesteps*) {
    Select random action with probability ϵ, otherwise $a = argmax_a Q(state, a)$
    Get next state, reward, done, x by stepping in environment with action a
    Append state, reward, next state, done to the memory
    Set state to next state
    if *t is a multiple of updateWeightEvery* **then**
      if *memory has more transitions than the size of minibatch* **then**
        Sample a minibatch from the memory
        Get states, actions, rewards, nextStates, and dones from minibatch.
        Get targets $y = rewards + gamma * max_a Q(nextStates, a, \theta^{'}) * (1 - dones)$
        Update weights θ using states and targets
    Every C timesteps update weights of target network $\theta^{'} = \theta$
  }
}

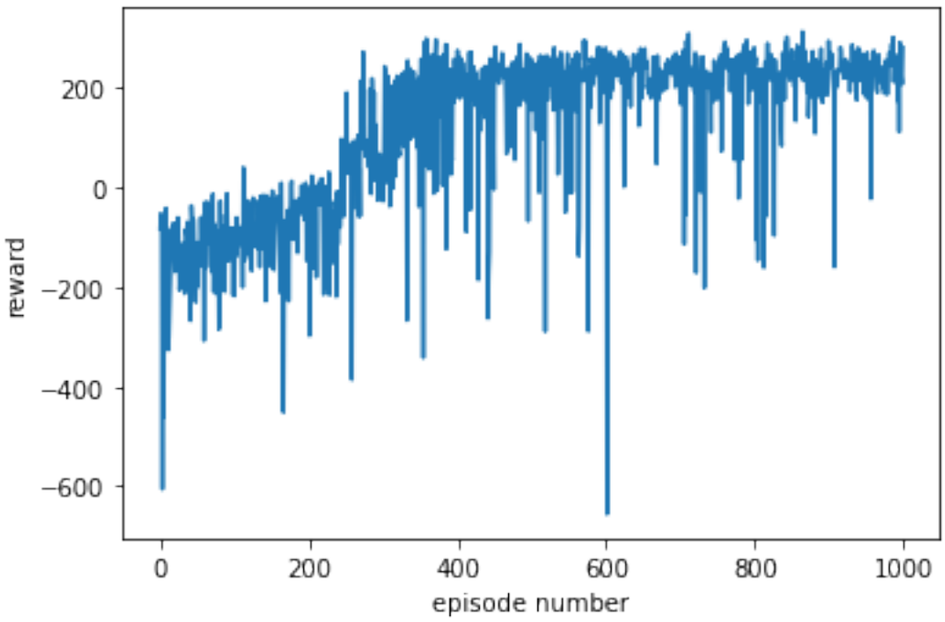
Algorithm 1: DQN with replay experience

3 Network Architecture & Hyperparameters

1. Input Layer (size 8) → Dense Layer (size 64, *relu*) → Dense Layer (size 64, *relu*) → Dense Layer (size 4, *linear*)
2. Episodes trained = 1000
3. Learning rate = 0.001
4. Gamma = 0.99
5. Epsilon = 1, with decay = 0.995 every episode
6. C (number of timesteps before the target model is updated) = 4
7. updateWeightsEvery = 4

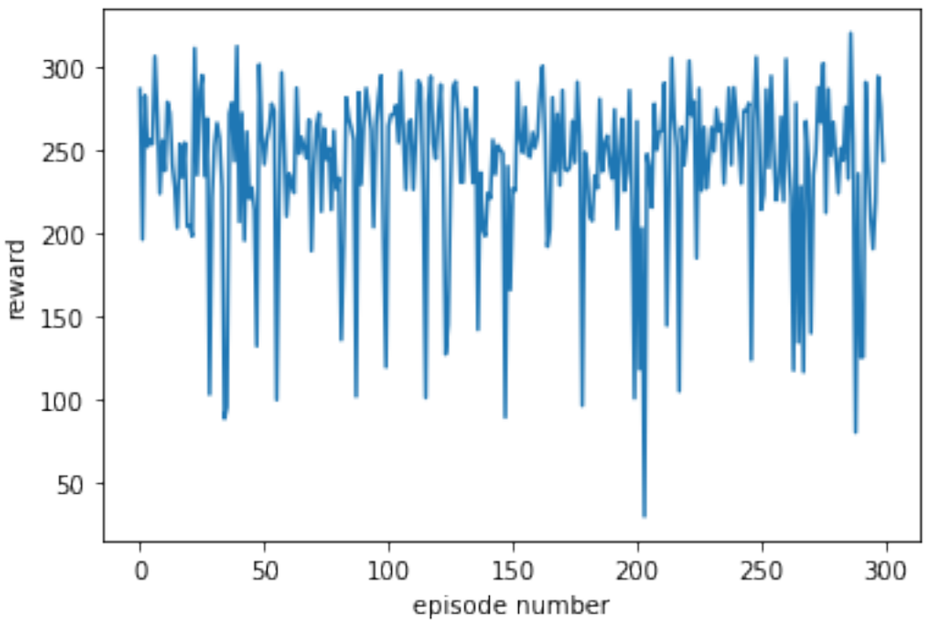
4 Experiments

4.1 Rewards vs Episodes when training



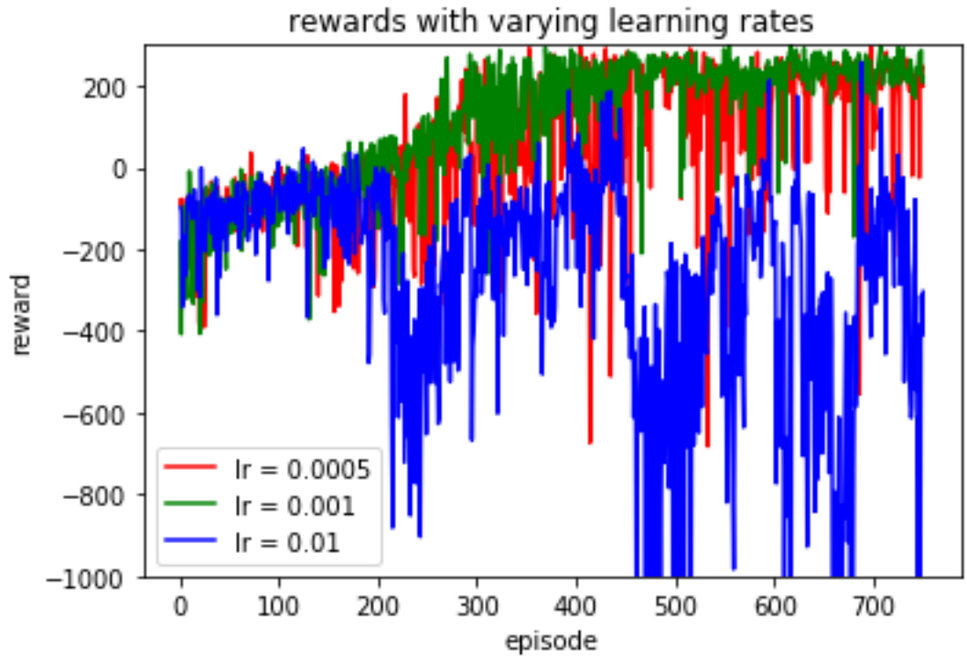
The model converges well to rewards consistently greater than 200. But as you can see there are still a few cases where the rewards go below 200. If I let the model train for more episodes it could fix that, but unfortunately Colab keeps timing out and I would need to run this again from the beginning.

4.2 Rewards vs Episodes after training



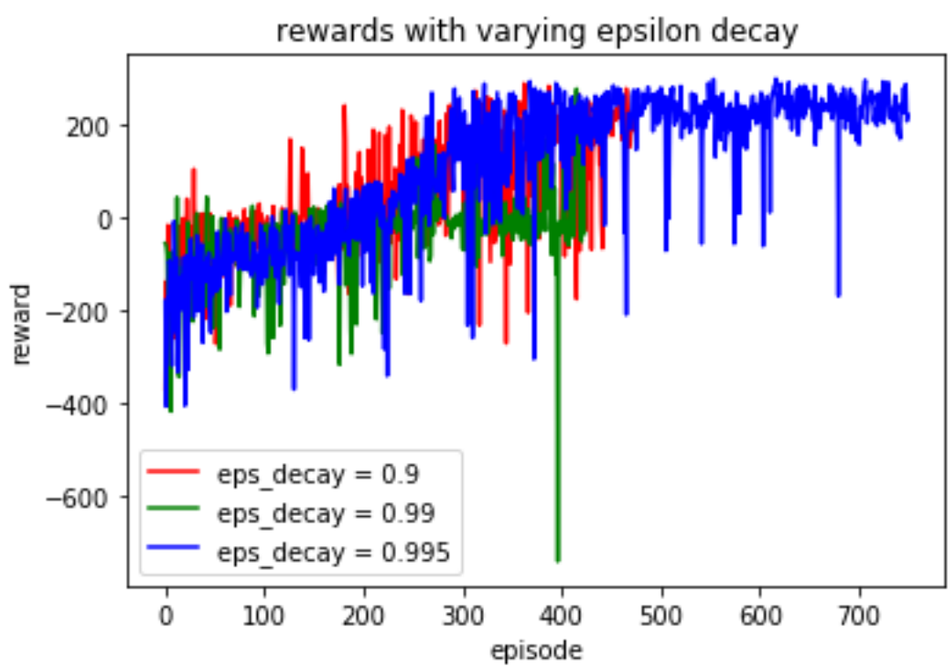
This plot shows rewards of running the trained model for 300 episodes. The average reward per episode is around 250, but we do see occasional drop in the rewards like mentioned before.

4.3 Rewards with varying learning rates



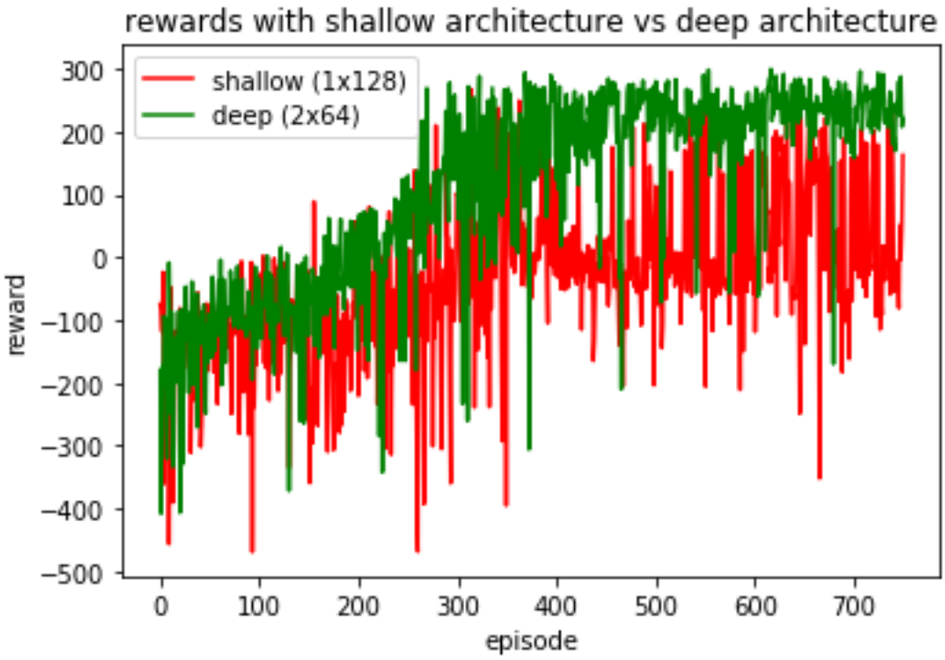
The model is very sensitive to the learning rate. When the learning rate is 0.01, the model doesn't converge at all. The reward per episode increase and then decreases periodically. This could be because the learning rate is too high for the given problem and it crosses over the minimum from either direction, and never approaches it.
The learning rate of 0.0005 seems to have more randomness or variability in terms of the rewards received per episode compared to 0.001. This maybe because it is more sensitive to every small local minimum and maximum of the loss function.
Overall, the learning rate was a very important hyperparameter in this task, and lr of 0.001 performed the best.

4.4 Rewards with varying epsilon decay



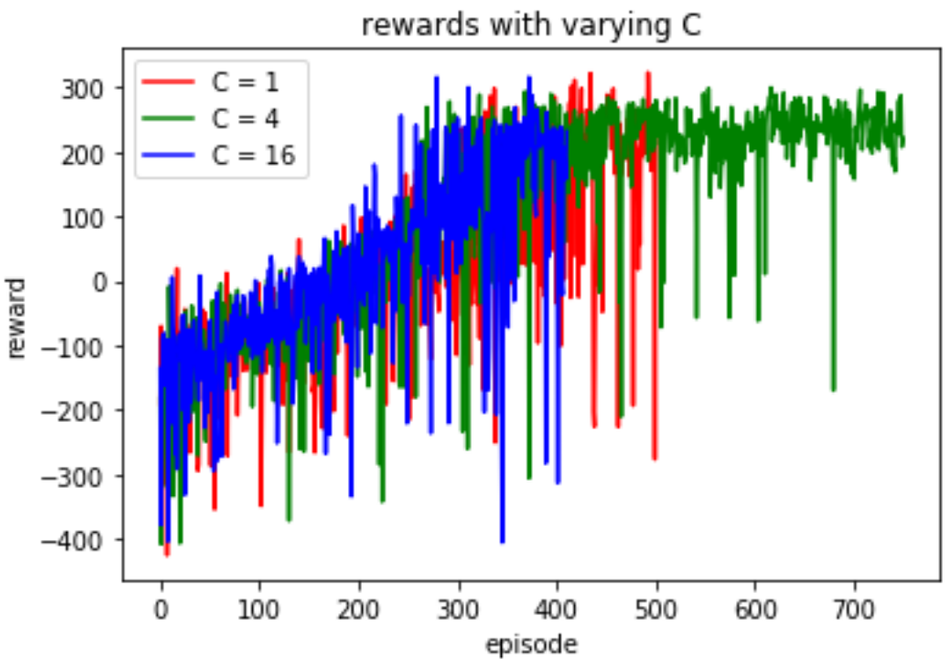
Note: eps_decay = 0.9 and 0.995 have only ∼500 data points due to timeout, and lack of time to run the experiment again.
With a higher epsilon_decay the model has more episodes to the explore random states. When the epsilon_decay is 0.9, the model has not explored enough and we see it in the variability in the red line where the model does well in some cases and very bad in the others. We also see models with eps_decay of 0.99 and 0.995 do much better in terms of the variability of rewards in each episode. With eps_decay of 0.99 we see the model converging slower compared to 0.995 maybe because the model has not explored enough till the 400th episode where thereafter we see a sudden rise in the rewards.

4.5 Rewards from shallow architecture vs. deep architecture



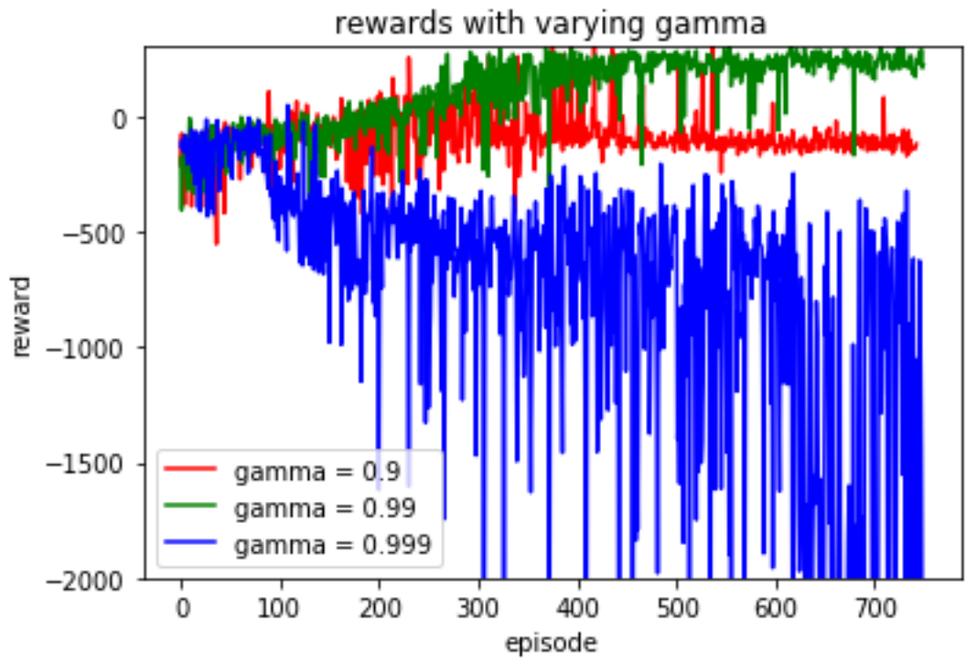
The deeper model converges and the shallow model doesn't. This may be because one hidden layer is not enough for a model to learn game well, and having the multiple layers helps the model learn more abstract things where maybe, each layer learns a specific aspect of playing the game well. I could have added more neurons to the shallow model, and I tried training a model with a single hidden layer of 256 neurons, but it takes longer to train the model, and the performance does not get any better. Unfortunately, I didn't record the rewards for that experiment to show it on this plot.

4.6 Rewards with varying C



Note: C = 1 and 16 have only ∼450 data points due to timeout, and lack of time to run the experiment again.
There doesn't seem to be much difference between performance of models with varying values of C. When C = 1 the dqn model and the target dqn model are always the same, and this seems to induce more variability in the rewards per episode. We see this trend again between C=4 and C=16, where C=16 seems to produce the least variable performance with each episode. I still chose C=4 to be my final hyperparameter as I have more information on it, and the model performs well.

4.7 Rewards with varying gamma (γ)



Gamma basically lets you set an importance to rewards you get in the future. As we increase γ, we give greater importance to rewards in the future. Here we see that the models with γ values of 0.9 and 0.999 completely fail to even get positive rewards over 750 episodes. Whereas, $\gamma = 0.99$ lets the model converge to rewards over 200. What is interesting is that there is no specific trend with rising γ, and 0.99 is the best γ value for my model.

5 Challenges

The model is very sensitive to hyperparameters. Tuning the hyperparameters to even make the model converge took a long time. Colab was a very useful tool and a free computing resource, but it has a problem of timing out after some time of inactivity, and requires the browser to be running.
There is also a learning curve to deep reinforcement learning where it is not immediately apparent as to why the model even works.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[2] TensorFlow. Basic regression: Predict fuel efficiency.