1 Tuning the DQN

1.1 Hyperparameters

My final DQN has 1 hidden layer with 17 neurons inside. I apply ϵ-greedy policy with exponential decay. All hyperparameters are summarised in Table 1.

Hyperparameter	Value	Justification
No. of hidden layers	1	Using 1 layer is sufficient for this coursework. Neural network with >1 layers are harder to train and might risk overfitting.
No. of neurons per hidden layer	17	Kept increasing from 1 to 17 and it hit the return requirement
Memory size	200000	A large number to keep as many experiences as possible. Large memory size enables network to reuse its experiences thus more sample efficient.
Batch size	50	Tried 20, 30, ... to 70. >50 led to bad generalisation. <50 the sample was too stochastic for the network to learn. See Figure 1 for details
Frequency update target DNN	10	Increased from 5 until 20. <10 led to high variance in mean return and unstable learning. >10 the update was too slow for the network to learn
Learning rate	0.005	Tried 0.001, 0.005, 0.01, 0.1, 1. (see Figure 2). 0.005 worked the best. 0.001 the network learned too slowly. 0.01 the result sometimes did not pass the requirement. 0.1, 1 were too big for the network to converge.
Initial ϵ	0.4	Tried 0.5 and 0.4. 0.4 with exponential decay worked the best. See Section 1.2 for details.
ϵ decay factor	0.98	Tried 0.9995, 0.99, 0.9, 0.98. See Section 1.2 for details.

Table 1: Hyperparameters defined in DQN

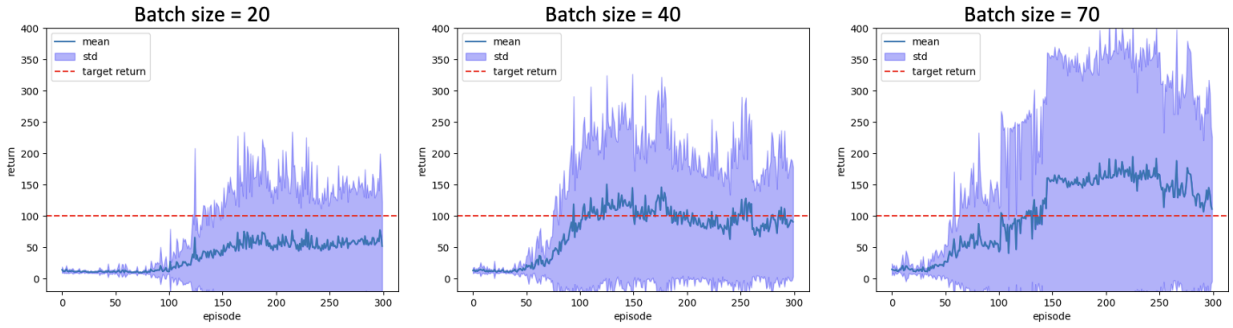

Figure 1: Learning curves with varying batch sizes (all other hyperparameters are the same in Table 1)

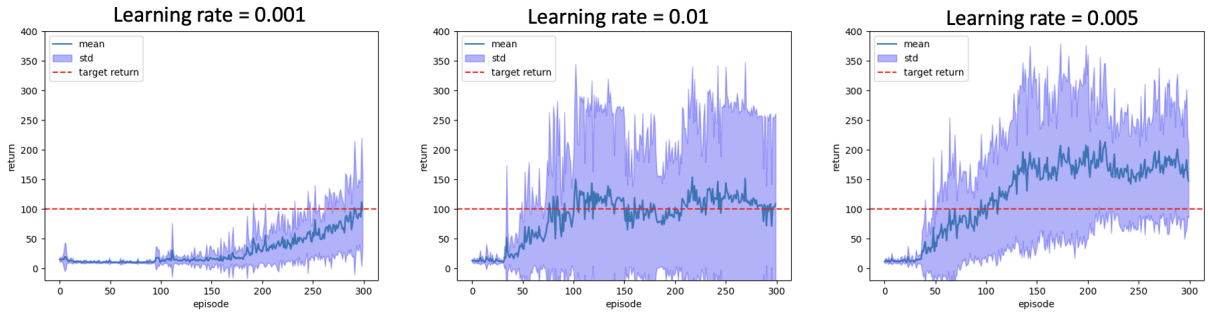

Figure 2: Learning curves with varying learning rates (all other hyperparameters are the same in Table 1)

1.2 Exploration & Exploitation

I use ϵ-greedy policy with an initial ϵ equal to 0.4 and apply an exponential decay rate equal to 0.98 in each episode to handle exploration and exploitation of the DQN. The idea is to let the network explore within the first few episodes then exploit the learnt experience (q-value) in later episodes with gradually decreasing ϵ.

Tuning initial ϵ

Figure 3 shows the learning curves with varying initial ϵ and a fixed decay rate equal to 0.99. I began with $\epsilon = 0.5$. Although the learning curve with $\epsilon = 0.5$ attains the performance requirement, it only passes the target line a little. To improve the network performance, I use the fact that the learning curve has a stable upward trend. The upward trend suggests the network has enough time to explore properly with the current ϵ value. Lowering the ϵ may allow the network to better exploit its learnt experience thus leading to a steeper learning curve. As a result, I lowered ϵ to 0.4 and the result is decent, so I set initial $\epsilon = 0.4$ as my final choice.

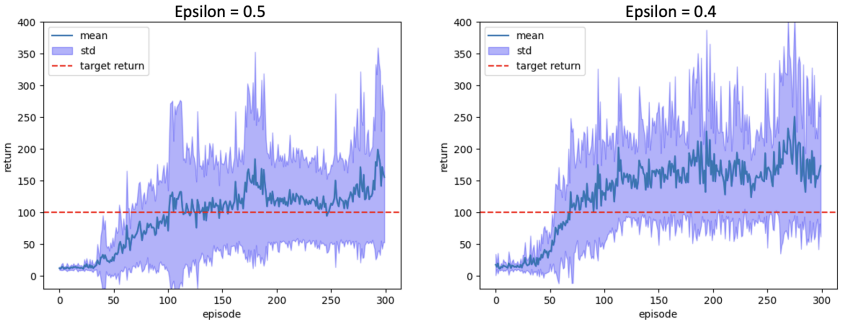

Figure 3: Learning curves with varying initial ϵ with fixed decay rate 0.99

Tuning ϵ decay rate

I fixed $\epsilon = 0.4$ to tune the decay factor. I tried 0.9995, 0.99, 0.9, and finally decided 0.98 as my final choice. Figure 4 shows the learning curves correspond to varying decay factors. For decay rate equal to 0.98, see Figure 5.

The curve with 0.9995 decay rate has a very noisy mean, implying the network does not act effectively. This might due to that the slow decaying rate prevents the network from choosing optimal action even if the network has leart the correct policy. Therefore I used a larger decay rate in the next trial.

The curve with 0.99 decay has a less noisy mean and passes the return requirement, but the model exceeds the red target line by just a little. Therefore I tried a lower decay rate to allow the model exploit more in the next trial.

The curve with 0.9 decay does not pass the return line, indicating the decay rate is too big for the model explore properly. Therefore I searched the range between 0.9 and 0.99. It turns out 0.98 works the best.

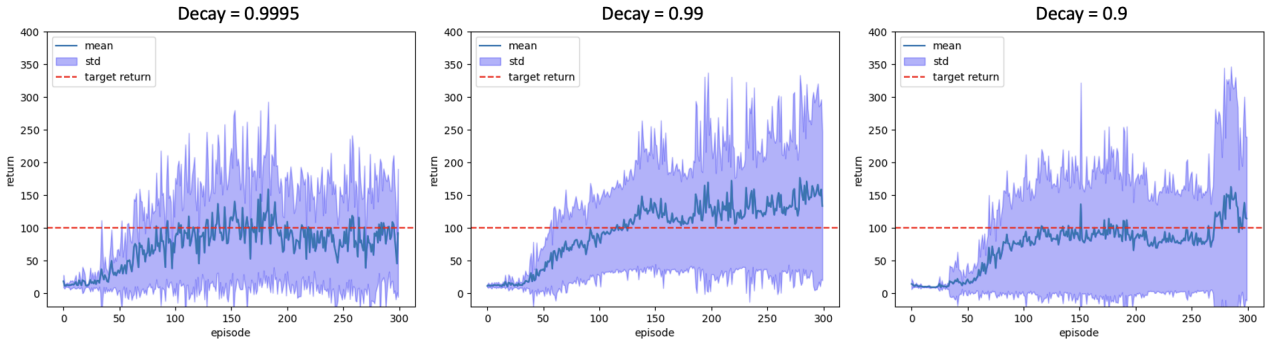

Figure 4: Varying ϵ decay rate with initial $\epsilon = 0.4$

1.3 Learning Curve

Figure 5 shows the learning curve with the hyperparameters summeriased in table 1.

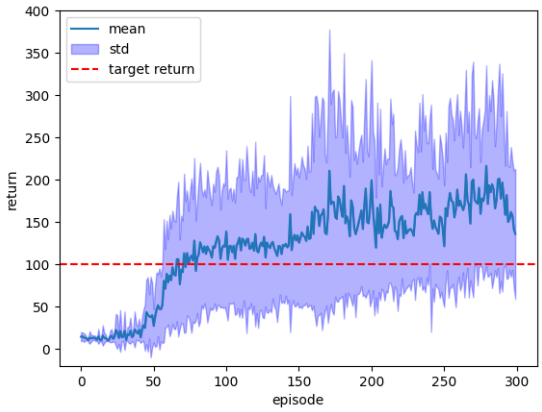

Figure 5: Learning curve of tuned DQN

2 Visualise DQN policy

2.1 Slices of the greedy policy action

Figure 6 shows the learnt policy of a network with different cart velocities and the cart position fixed at 0.0. Colours represent actions to take. Yellow indicates pushing cart to the left. Blue indicates pushing cart to the left.

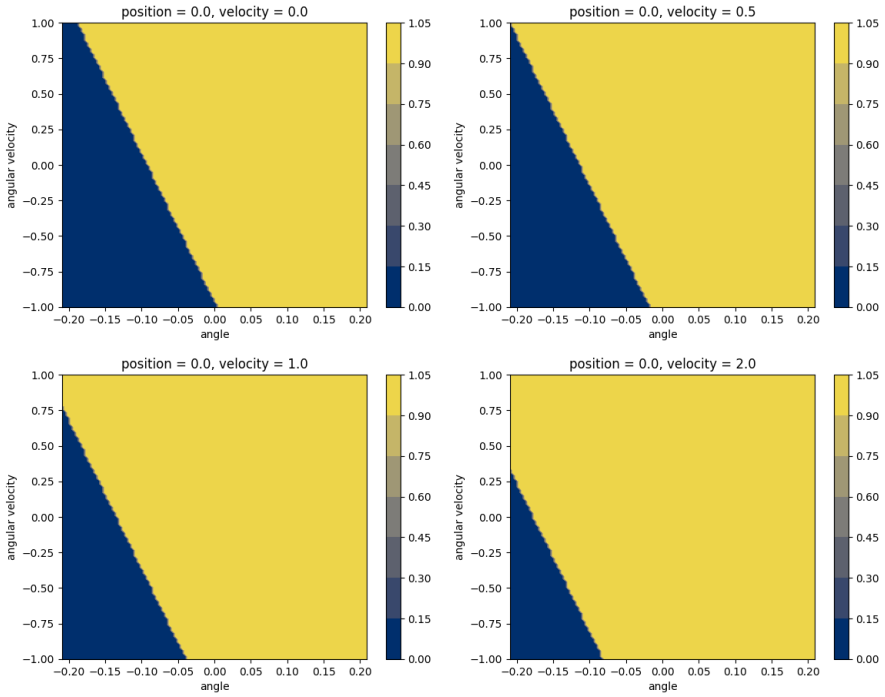

Figure 6: Greedy policy action with different velocities. Yellow indicates pushing cart to the left. Blue indicates pushing cart to the left.

The policy makes intuitive sense. It says the cart should be pushed left if both the angle and angular velocity of the pole are negative (the pole tilts to the left and falls left) and vice versa. If the pole has a negative angle and a positive angular velocity, the policy depends on the velocity of the cart. If the cart velocity is 0 (the cart is not moving), the cart should be pushed left to catch the rectifying pole. However, if the cart also has positive velocity (moving right), the cart should also be pushed right to accompany with the rectifying pole's inertia. Notice that the blue triangle area decreases as the cart velocity increases. This obeys Newton's first law. If the whole system (both the cart and the pole) is moving right, the cart should be pushed right to comply with the system's inertia.

2.2 Slices of the Q function

Figure 7 shows the learnt q-values with different velocities and the cart position fixed at 0.0. Colours indicate the magnitude of estimated q-values. Yellow and blue represent high and low q-values respectively.

In general the q-value is higher if the pole is pointing upright (angle close to 0) and not falling (angular velocity close to 0). The q-value becomes lower if the pole is falling in the same direction as the pole's angle. The two qualities above hold true for all four cart velocities.

However, as velocity of the cart increases, the yellow area decreases since the cart and pole system becomes unstable. The top-right corner of the graph gets affected the most by the velocity of the cart. If the cart is moving fast towards right (velocity=2) but the pole is falling right as well, the cart has to be pushed further to the right to catch the falling pole. However, this increases the cart velocity even more and leads to the violation of cart position requirement (cart distance from centre greater than ± 2.4), leading to low q-values. On the other hand, given a positive cart velocity, the pole with both negative angle and angular velocity has higher q-values than positive ones because once the cart is pushed left, it is less likely to exceeds the given position range.

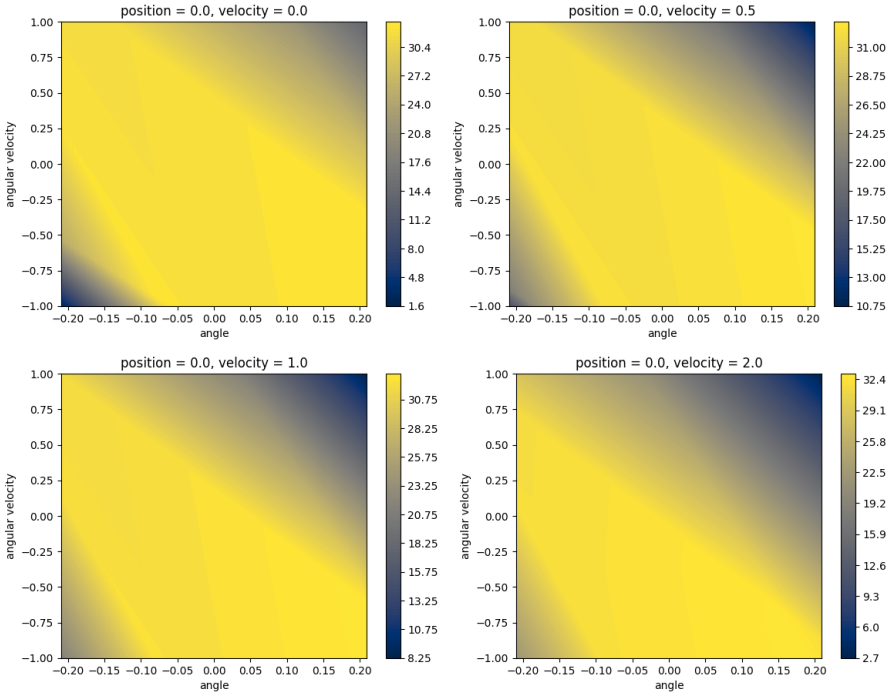

Figure 7: Learnt q-values with different velocities and the cart position fixed at 0.0. Yellow and blue indicate high and low q-values respectively.

3 Transform DQN into a DDQN

Figure 8 shows the learning curves between normal DQN and DDQN. Although the network performance is unstable in each run, DDQN achieves higher mean return than DQN in general.

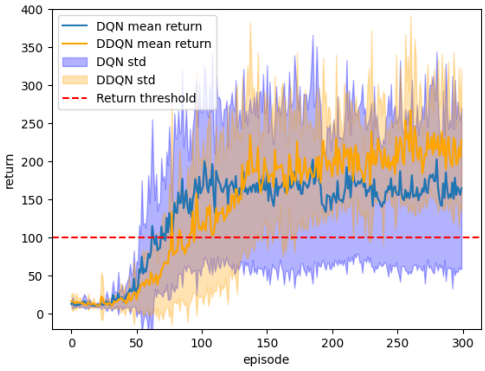

Figure 8: Learning curves between normal DQN and DDQN.

I modified the loss() function in file utils.py. I added one additional argument is_DDQN to the function. If the network is DDQN, the function first selects the action from the policy network then evaluate it using the target network. Therefore, the bellman target is updated using two networks. Else, if the network is DQN, the bellman target is only updated with the target network.

```
# if the network is DDQN, select action from policy network
# and gather action value from the target network
if is_DDQN:
    policy_dqn_actions = policy_dqn(next_states).max(1).indices.reshape([-1,1])
    bellman_targets = (~dones).reshape(-1)*(target_dqn(next_states))\
    .gather(1,policy_dqn_actions).reshape(-1)+rewards.reshape(-1)
# else if the network is DQN, select and evaluate bellman targets using target network only
else:
    bellman_targets = (~dones).reshape(-1)*(target_dqn(next_states)).max(1).values + \
    rewards.reshape(-1)
```

The lower mean return received by DQN attributes to overestimation of action value. DQN uses the same action value in policy network to both select and evaluate an action, which prompts the network to produce overestimated values and select overly optimistic actions. In contrast, DDQN uses one network to select optimal action and the other network to compute action value for the optimal action selected. This breaks the dependence between action chosen and the action value computed in DQN thus leading to a better mean return.