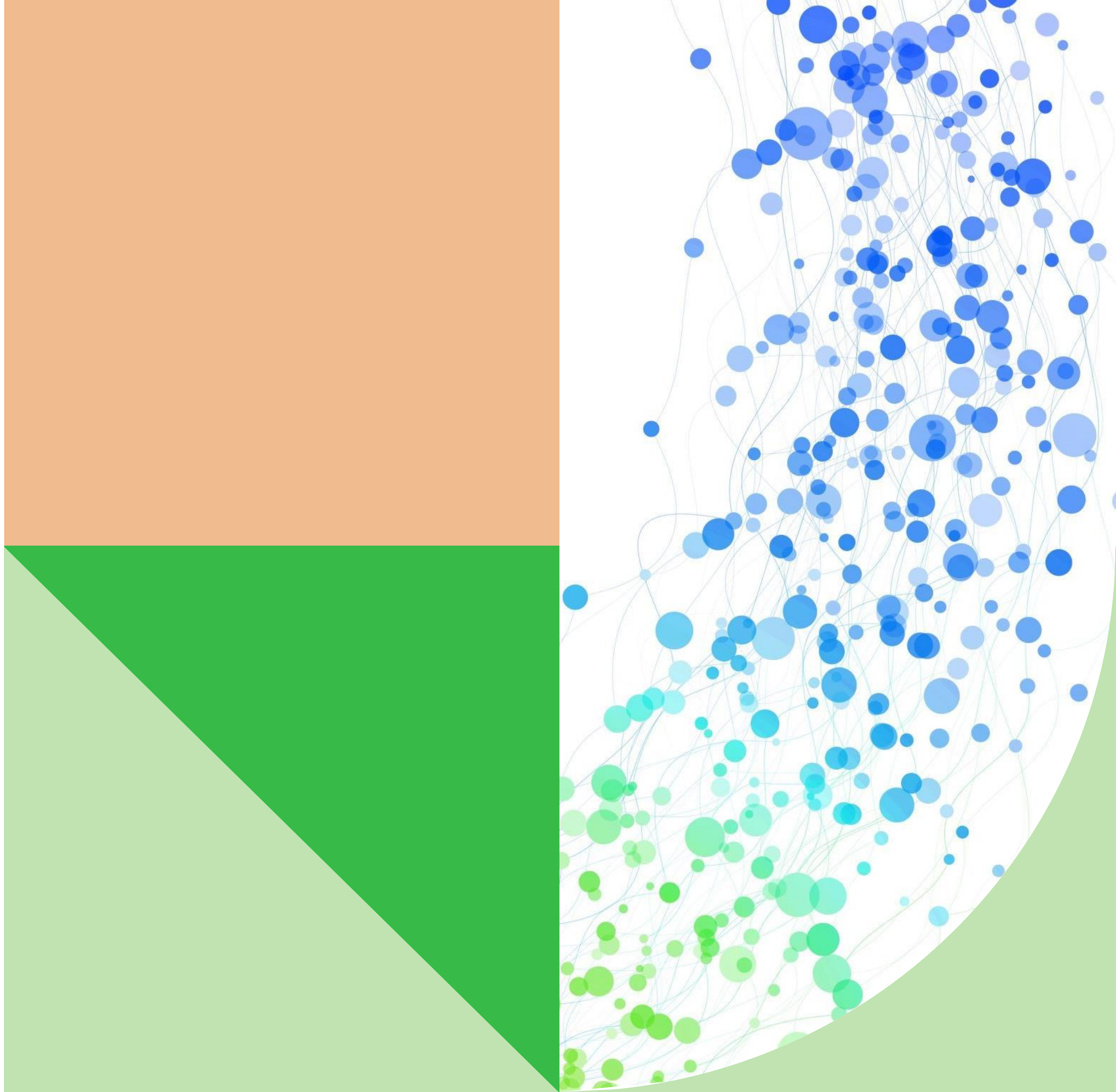
DELE CA2 Part B: Reinforcement Learning

By: Loh Kong Xuan Kieran (p2309053) &

Soon Jing Yi (p2308940)

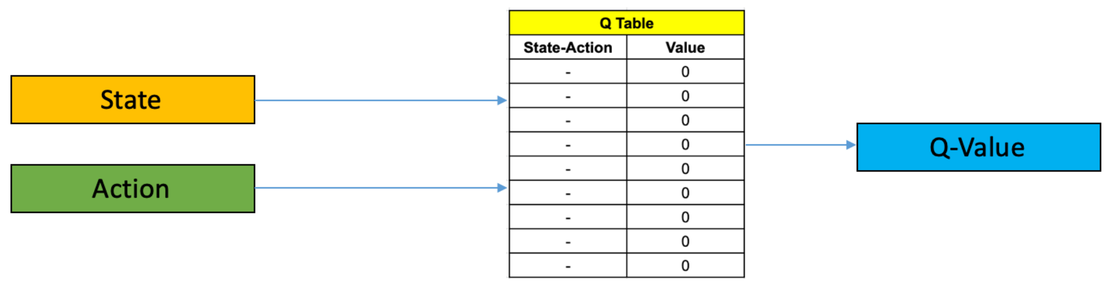


Overview

Reinforcement Learning (RL) VS Deep Reinforcement Learning

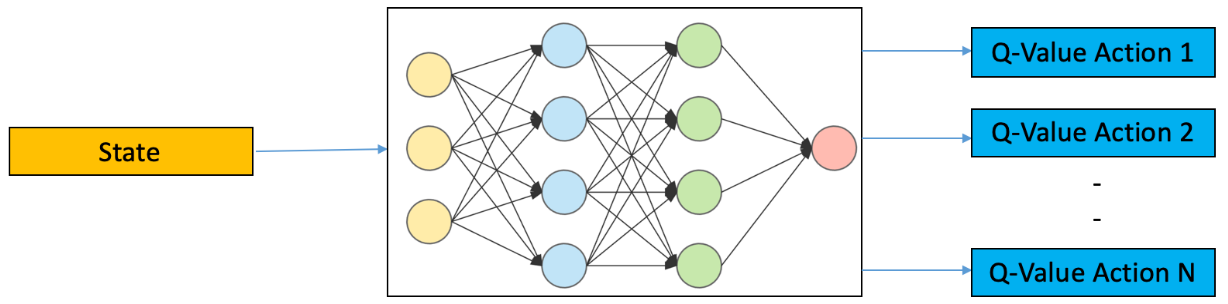
- *Reinforcement learning* (RL) is a machine learning technique that trains software to make decisions to achieve the most optimal results. It mimics the trial-and-error learning process that humans use to achieve their goals. Software actions that work towards your goal are reinforced, while actions that detract from the goal are ignored.
- *Deep Reinforcement Learning* combines reinforcement learning (RL) with deep learning techniques to enable agents to learn complex behaviors from high-dimensional sensory inputs, such as images. In Deep Reinforcement Learning, deep neural networks are used to approximate value functions, policies, or models of the environment. This allows the agent to make decisions based on large and unstructured data inputs, like raw pixels in video games.

Q Learning VS Deep Q Learning



State-Action	Value
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

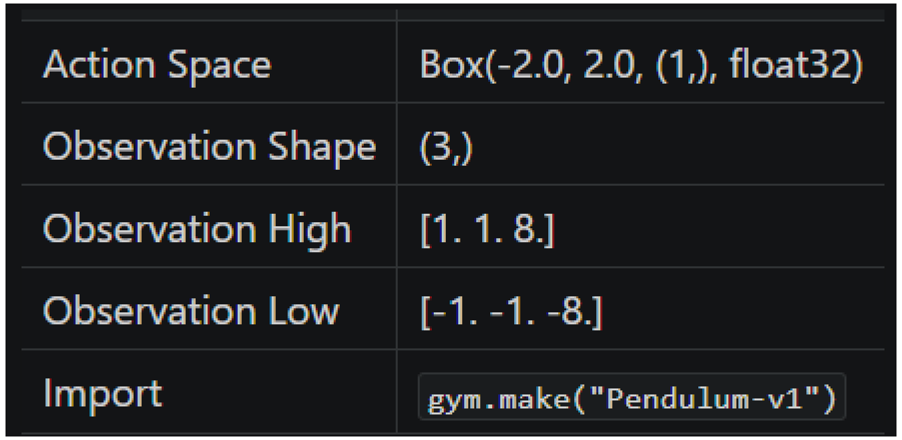
Q Learning



Deep Q Learning

https://www.v7labs.com/blog/deep-reinforcement-learning-guide

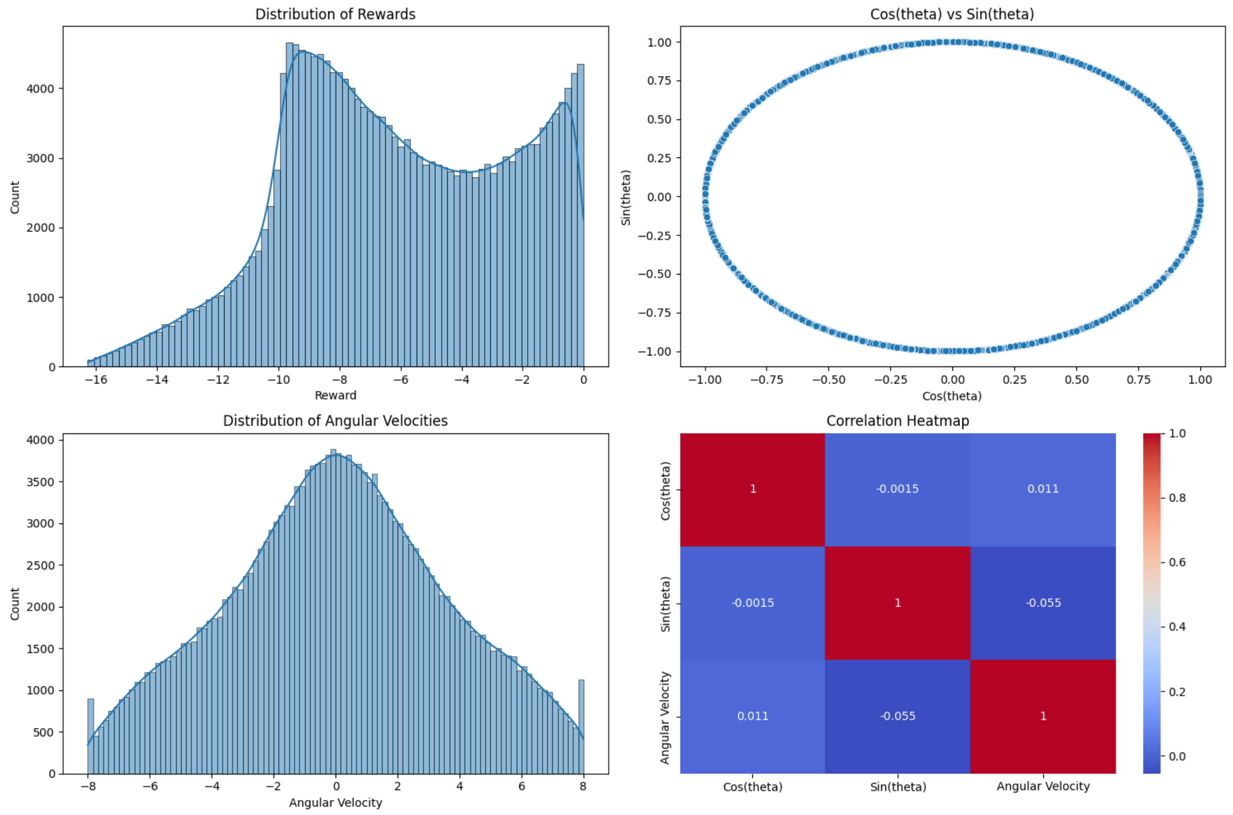
EDA – Pendulum v1



Action Space	Box(-2.0, 2.0, (1,), float32)
Observation Shape	(3,)
Observation High	[1. 1. 8.]
Observation Low	[-1. -1. -8.]
Import	`gym.make("Pendulum-v1")`

```
Summary Statistics:
Mean Reward: -6.1428
Std Reward: 3.6595
Min Reward: -16.2717
Max Reward: -0.0001

Observation Statistics:
Mean Cos(theta): -0.3595
Mean Sin(theta): -0.0005
Mean Angular Velocity: 0.0694
```



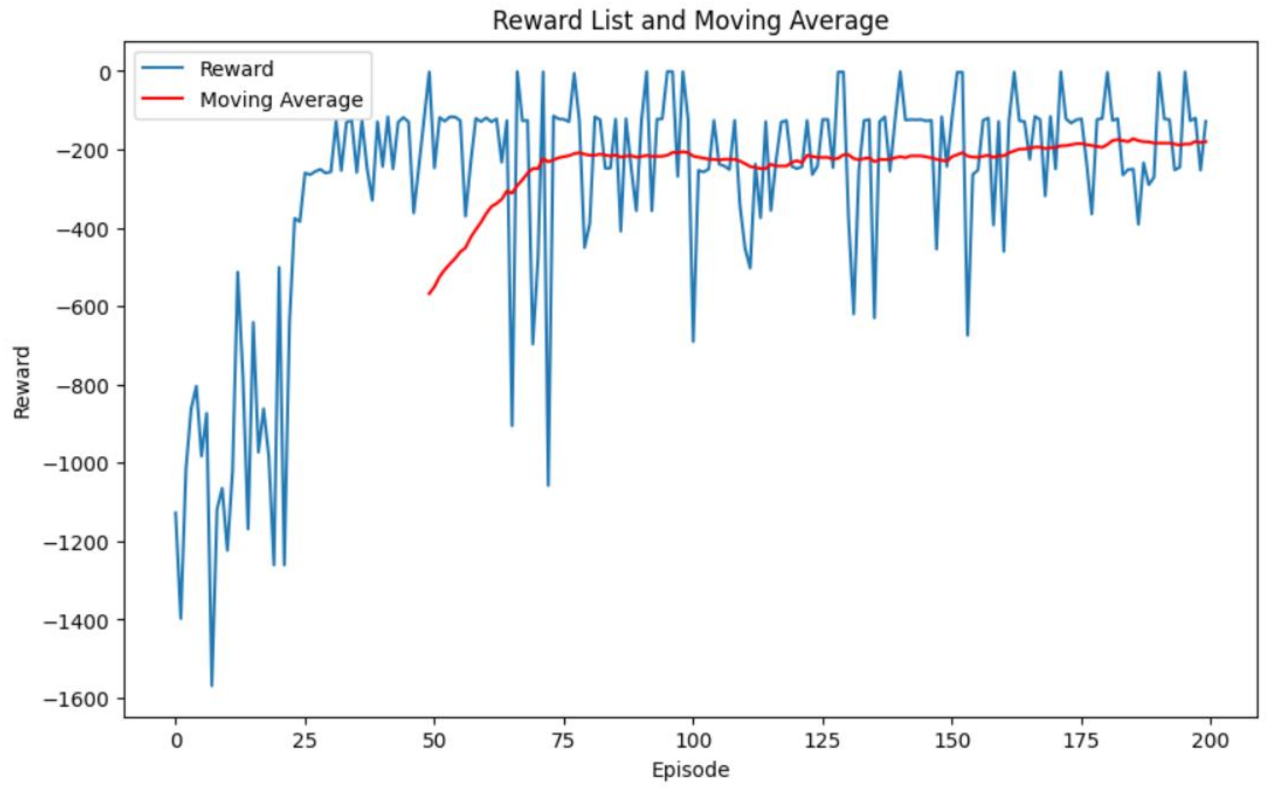
Deep Q-Network (DQN)

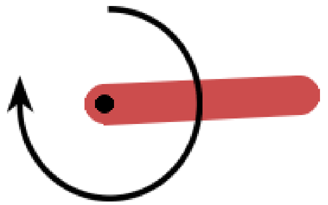
Deep Q-Network (DQN) is an algorithm that merges deep learning with the Q-learning method to optimize action-value functions in reinforcement learning tasks. It involves representing the current state of an environment numerically, often using raw pixel values or preprocessed features. This representation is then fed into a deep neural network, typically a convolutional neural network (CNN), which outputs Q-values for each possible action the agent can take.

There are 3 main components in a DQN:

1. Replay Buffer (Memory)
2. Q-Network
3. Target Network

Base DQN


Reward List and Moving Average
Reward
Moving Average
0
−200
−400
−600
−800
−1000
−1200
−1400
−1600
Reward
0
25
50
75
100
125
150
175
200
Episode



Deep Deterministic Policy Gradient (DDPG)

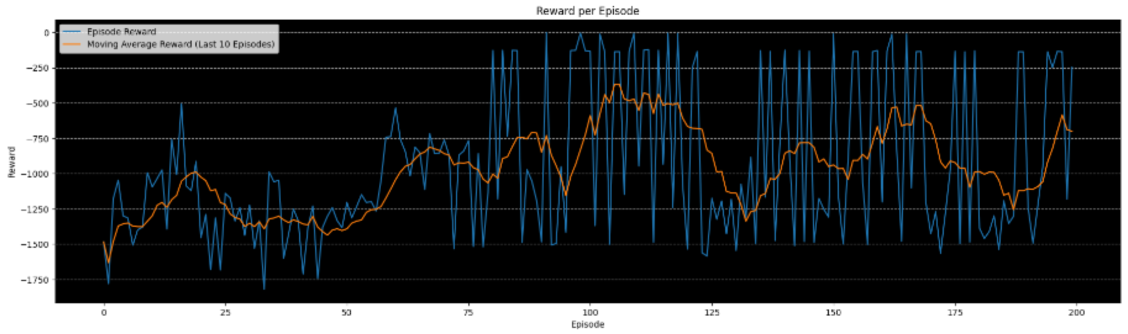
Deep Deterministic Policy Gradient (DDPG) is an off-policy reinforcement learning algorithm designed for environments with continuous action spaces, like the pendulum environment. It is an actor-critic method, which means it consists of two neural networks: an actor and a critic.

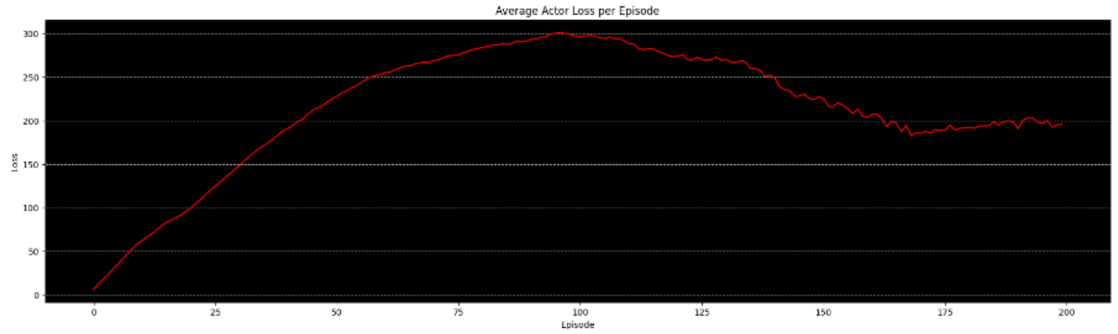
Instead of learning the Q function and using it to derive the policy by selecting the action with the highest Q-value like in DQN, DDPG learns the policy directly using an actor network and uses a critic network to provide feedback on the quality of actions to improve the policy. This allows the DDPG actor to output continuous actions, which gives more precision as actions can be varied smoothly, giving finer control.

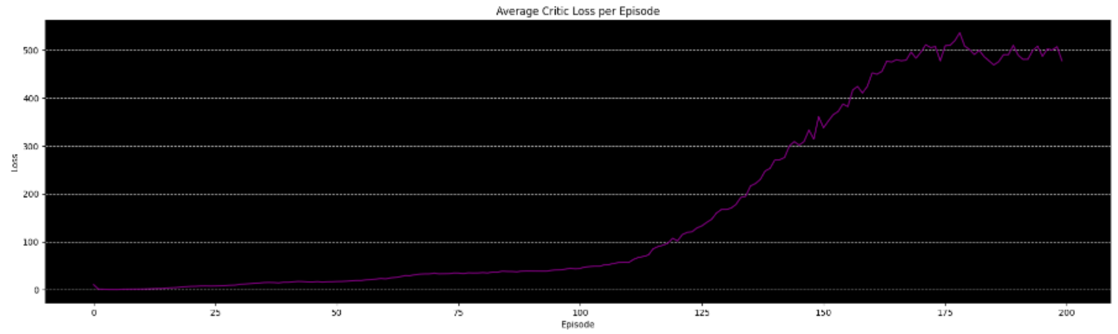
There are 5 main components in a DDPG:

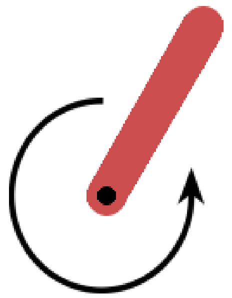
1. Replay Buffer (Memory)
2. Actor Network: learns the policy function that maps states to actions.
3. Critic Network: learns the Q-function using experiences sampled from the replay buffer, uses this to evaluate and guide the actor network.
4. Target Networks – Target Actor & Target Critic
5. Exploration Noise

Base DDPG

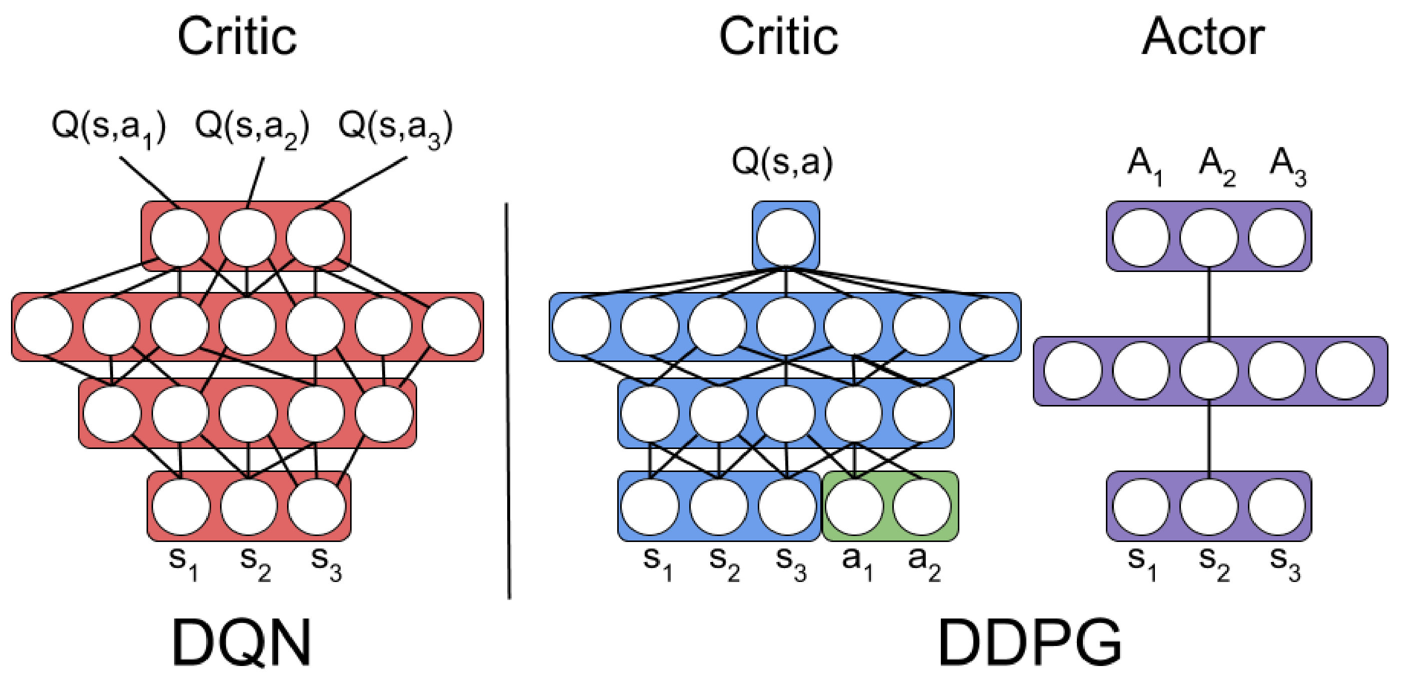

Reward per Episode
Episode Reward
Moving Average Reward (Last 10 Episodes)
Reward
0
−250
−500
−750
−1000
−1250
−1500
−1750
0
25
50
75
100
125
150
175
200
Episode


Average Actor Loss per Episode
Loss
300
250
200
150
100
50
0
0
25
50
75
100
125
150
175
200
Episode


Average Critic Loss per Episode
Loss
500
400
300
200
100
0
0
25
50
75
100
125
150
175
200
Episode



DQN vs DDPG Architecture:



https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learning-algorithm-38ca8698131b

Model Improvement (DQN)

1. Finding the optimum number of actions in the action space
2. Finding the optimum memory buffer size
3. Finding the optimum target network update frequency
4. Finding the optimum epsilon
5. Finding the optimum gamma
6. Finding the optimum batch size
7. Finding the optimum learning rate

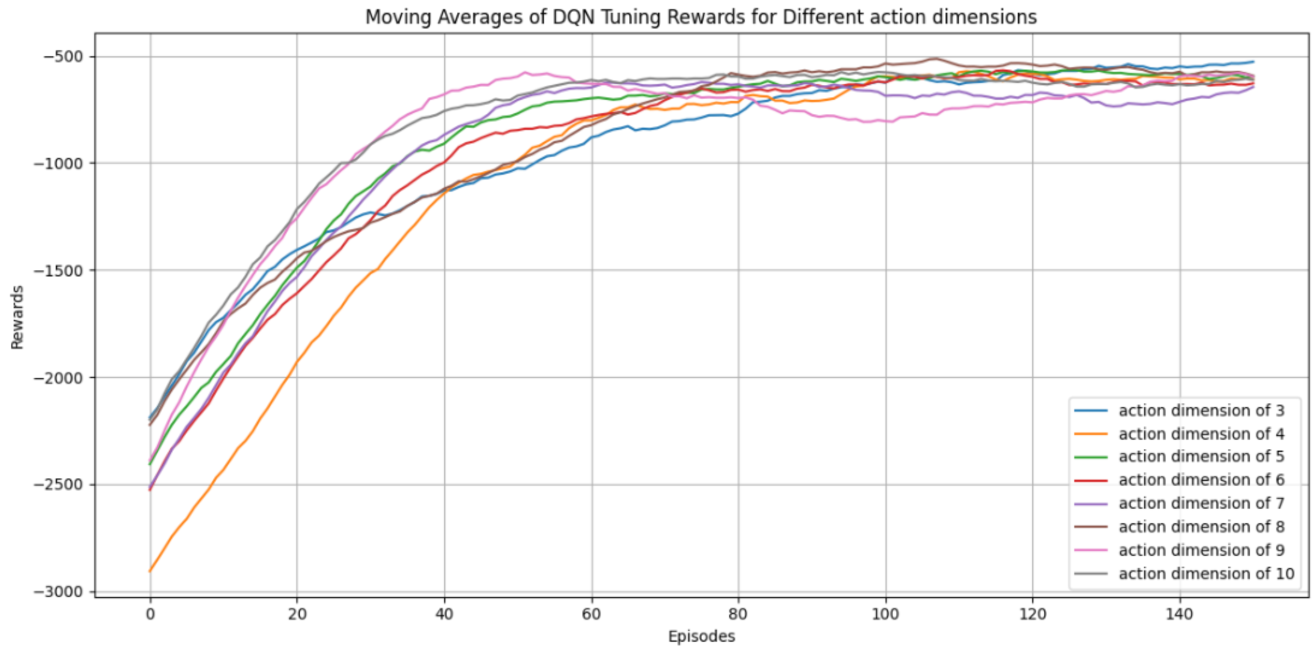
During the tuning process, is one trial enough?

In Pendulum-v1, a single trial typically isn't enough to determine the performance of a model due to the inherent variability and stochasticity in the environment and agent behavior. It's essential to conduct multiple trials to get a more reliable measure of the agent's performance and to account for randomness in the environment and initialization of the agent's parameters.

Therefore, we will run 3 trials for each hyperparameter and then take the average rewards of the 3 trials to get the best hyperparameter.

1. Finding the optimum number of actions in the action space

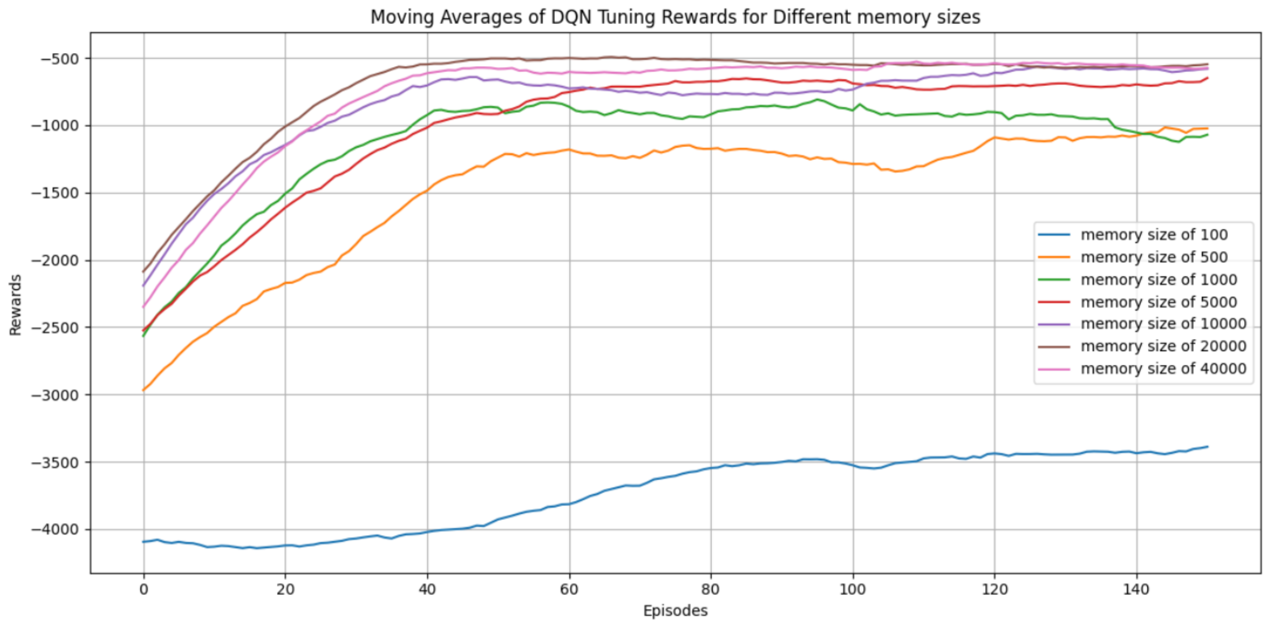
- Tuning the action space is important because it directly affects the agent's learning efficiency and effectiveness. The action space defines all the possible actions the agent can take, so its size influences how well the agent can explore and exploit the environment. If the action space is too limited, the agent might not have enough options to learn effectively, while a space that's too large can make the learning process overly complex and slow.



From the graph: We will choose an action dimension of 3 as it has the most stable curve with the highest reward at the end

2. Finding the optimum memory buffer size

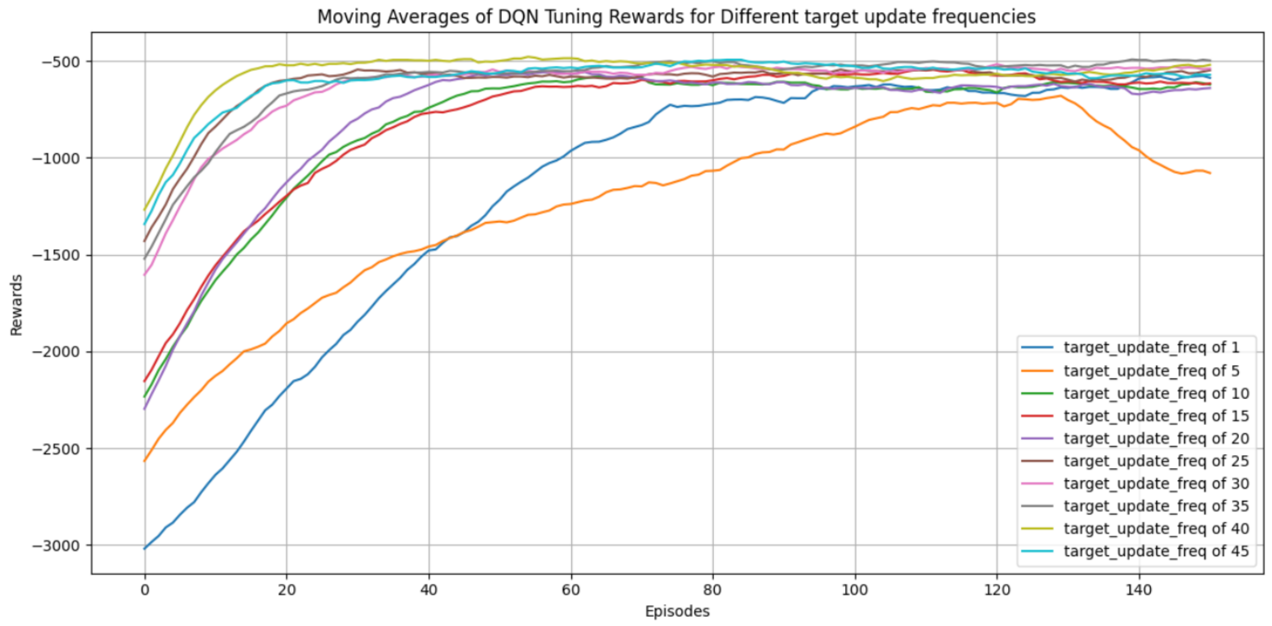
- The memory buffer size is important to tune because it allows the agent to learn from a variety of experiences rather than just the most recent ones. If the memory buffer is too small, the agent might only learn from recent experiences and miss out on valuable information from earlier interactions, which can make learning less stable and less effective. Conversely, if the buffer is too large, it might slow down the training process and use more computational resources than necessary. Tuning the memory buffer size helps balance these factors, ensuring that the agent learns from a diverse set of experiences while keeping the training process efficient.



From the graph: We will choose a memory buffer size of <u>20,000</u> as it has the most stable curve with the highest reward at the end

3. Finding the optimum target network update frequency

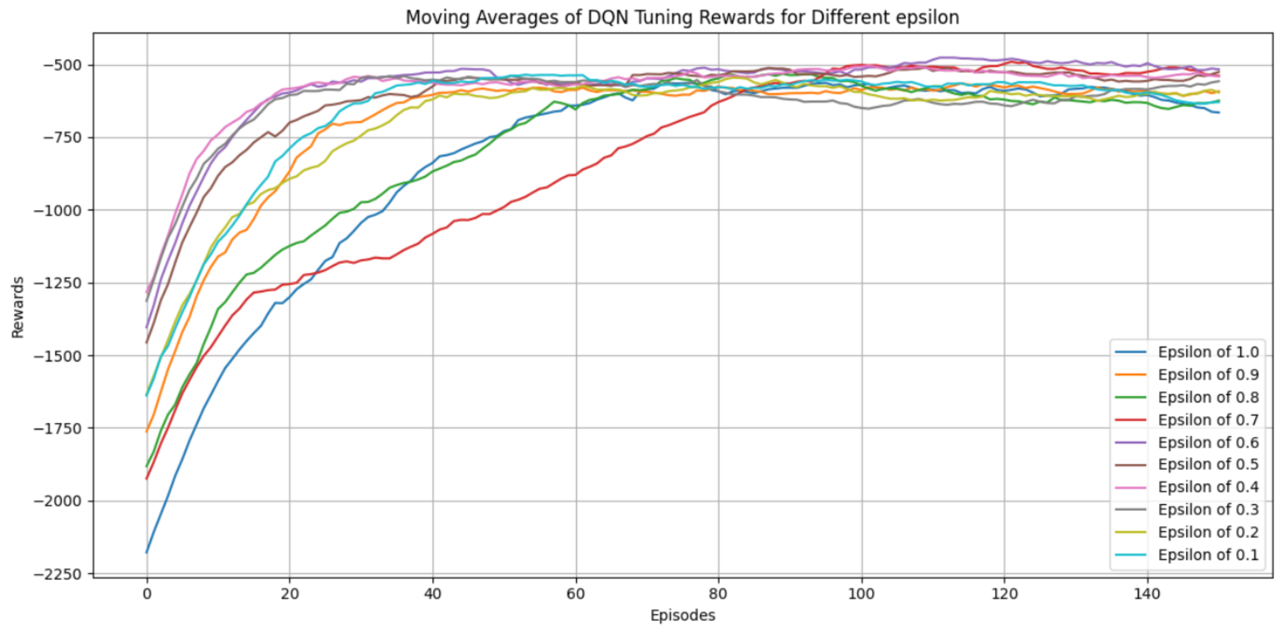
- The target update frequency refers to how often the target network, which helps stabilize training, is updated with the weights of the main Q-network. This frequency is important because if the target network is updated too frequently, it can lead to instability and noisy learning since the target values are changing too quickly. On the other hand, if updates are too infrequent, the target network might not adequately reflect the latest state of the main Q-network, which can slow down learning and make the training less effective. Tuning this frequency helps strike a balance, ensuring stable and efficient learning by keeping the target network in sync with the main network at a reasonable pace.



From the graph: We will choose a target network update frequency of <u>30</u> as it has the most stable curve with one of the highest rewards at the end

4. Finding the optimum Epsilon

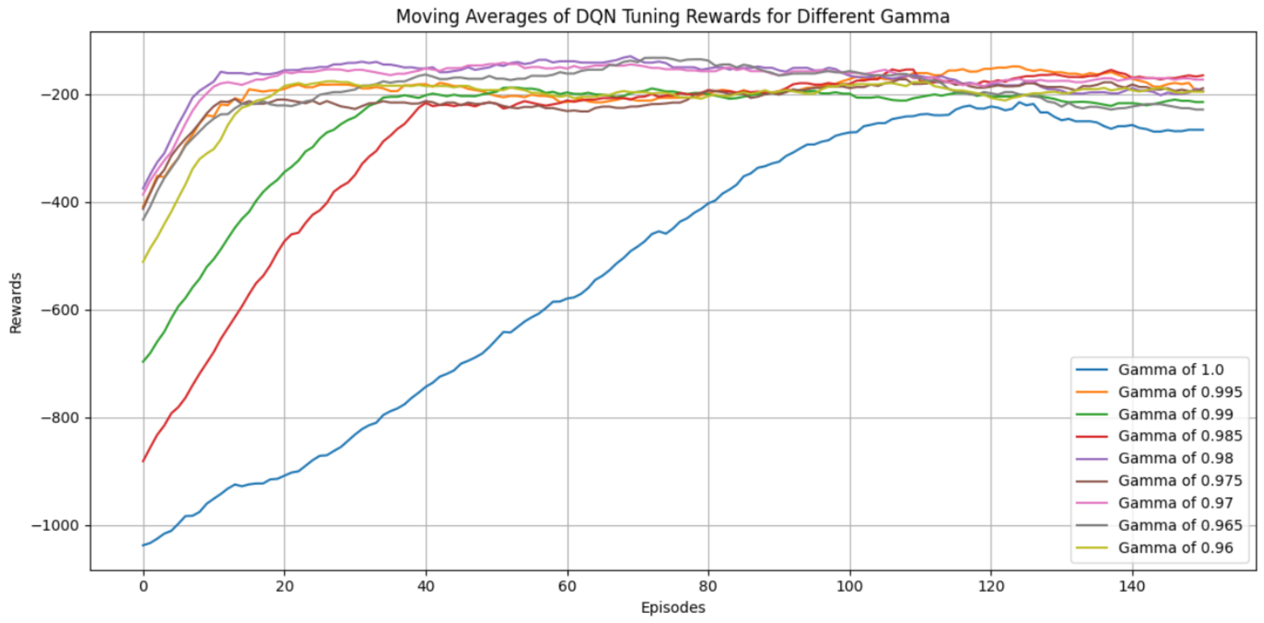
- Epsilon controls the exploration-exploitation trade-off by determining how often the agent chooses a random action rather than the best-known action. A high epsilon encourages more exploration, allowing the agent to try out different actions and discover potentially better strategies. However, too much exploration can make learning inefficient. Conversely, a low epsilon favors exploitation, where the agent relies mostly on known actions, which can lead to missing out on discovering better strategies. Tuning epsilon helps balance exploration and exploitation, ensuring that the agent learns effectively while also exploring enough to improve its performance.



From the graph: We will choose an epsilon of 0.2 as it has the most stable curve

5. Finding the optimum Gamma

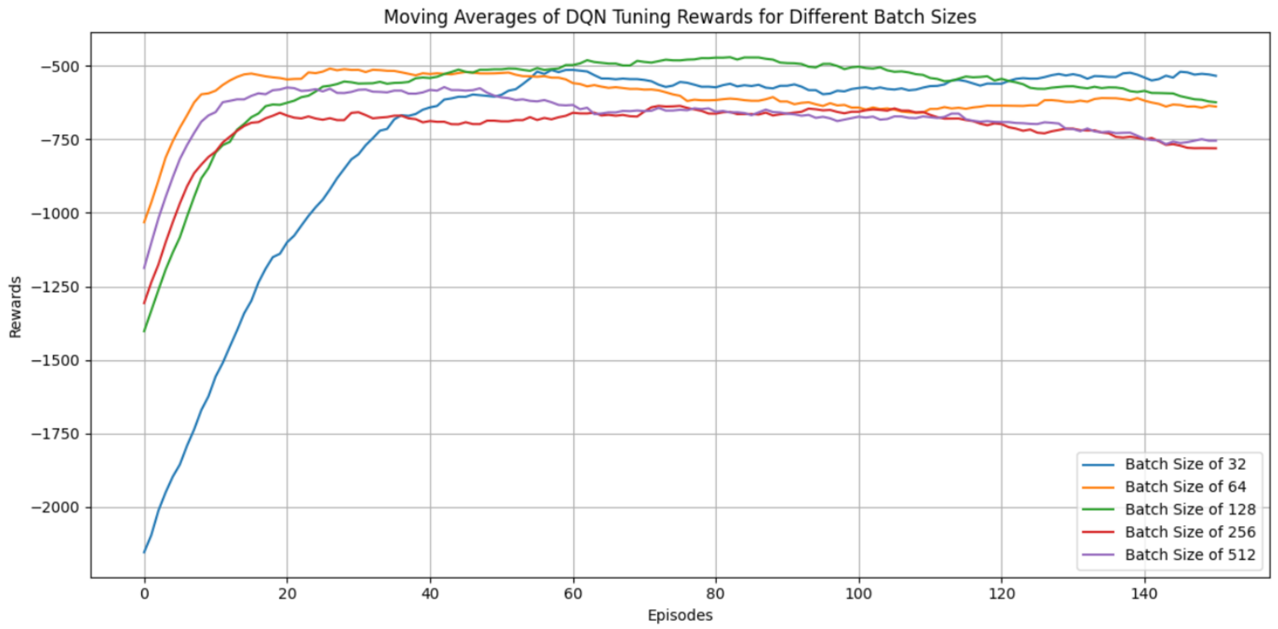
- Gamma represents the discount factor, which determines how much importance the agent gives to future rewards compared to immediate rewards. A high gamma means the agent values future rewards more, leading to a focus on long-term goals. Conversely, a low gamma makes the agent prioritize immediate rewards, focusing more on short-term gains. Tuning gamma is crucial because it affects how the agent balances between immediate and future rewards, influencing its overall learning strategy and performance in reaching long-term objectives.



From the graph: We will choose a gamma of 0.99 as it has the most stable curve

6. Finding the optimum Batch Size

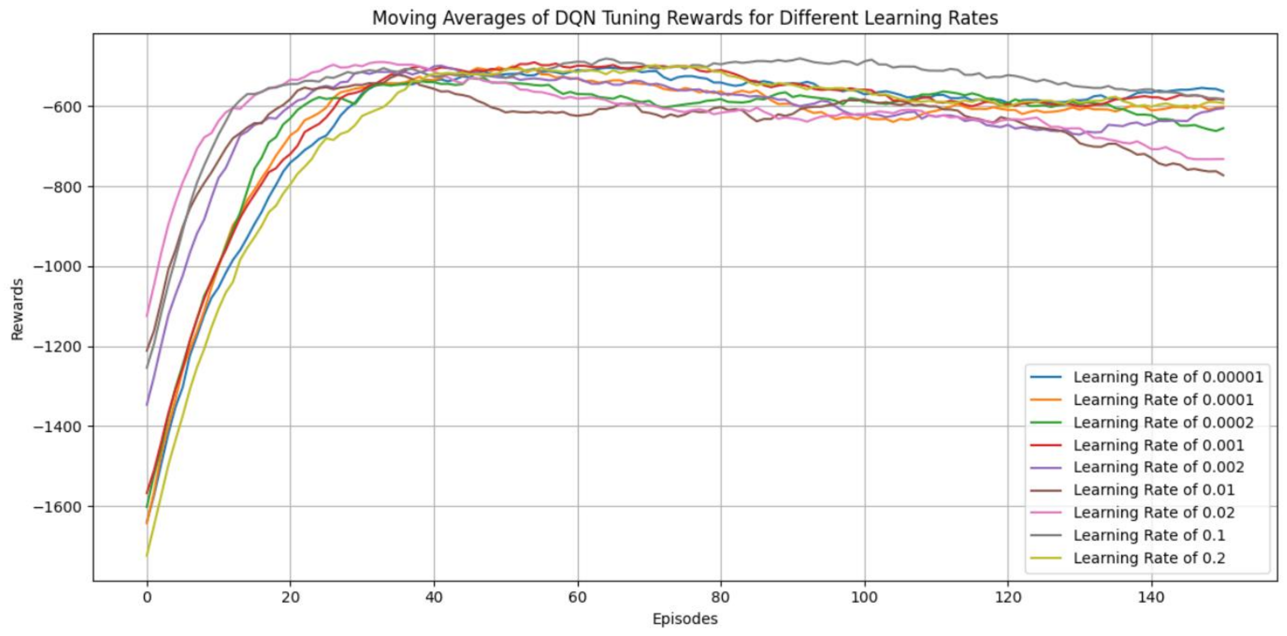
- The batch size determines how many experiences from the memory buffer are used to update the Q-network at each training step. A larger batch size provides a more stable estimate of the gradients because it averages out the noise from individual experiences. However, it requires more memory and computation. A smaller batch size updates the network more frequently and can lead to faster training, but it might be less stable and more susceptible to noisy updates. Tuning the batch size helps balance between stable learning and computational efficiency, ensuring that the training process is both effective and manageable.



From the graph: We will choose a batch size of 128 as it has the most stable curve

7. Finding the optimum Learning Rate

- The learning rate controls how much the Q-network's weights are adjusted during each update. A high learning rate can make the training process faster but might lead to unstable updates and overshooting optimal values. On the other hand, a low learning rate results in more stable updates but can slow down the learning process and make it harder to converge to a good policy. Tuning the learning rate helps ensure that the Q-network learns efficiently while maintaining stability and avoiding issues like oscillations or slow convergence.



From the graph: We will choose a learning rate of <u>0.001</u> as it has the most stable curve

Model Improvement (DDPG)

1. Increasing the complexity of actor and critic networks
   - By increasing the number of layers and neurons, the networks gain the ability to capture more intricate patterns and relationships in the data.
   - For the actor network, this means it can better model complex policy functions and generate more refined actions.
   - For the critic network, the increased complexity allows for more accurate approximation of the Q-value function, improving the quality of feedback given to the actor.
   - Overall, these changes lead to a more capable agent that can learn more effectively from its interactions with the environment, potentially resulting in better performance and more accurate policies.
2. Finding the best noise function to use (OU Noise or Parameter Space Noise)
   - OU Noise: Adds correlated noise to the actions, smoothing exploration in continuous action spaces by generating correlated noise values. It helps prevent abrupt changes and enables stable exploration.
   - PSN: Introduces noise directly into the policy network's parameters, leading to diverse exploration by modifying the policy itself. This method broadens exploration by altering the underlying policy rather than just the actions.
   - We found that PSN worked better in our case as its ability to modify the policy parameters directly allows for more effective and diverse exploration.
3. Evaluating the effects of PSN Desired Distance Parameter on Exploration and Exploitation
   - The desired distance represents the target amount of deviation between the noisy and original actions, guiding the adaptation of noise levels to ensure effective exploration without excessive deviation.
   - A smaller desired distance will result in less noise and more exploitation, as the model will be less likely to explore new actions.
   - A larger desired distance will result in more noise and more exploration, as the model will be more likely to try new actions.
   - Overall, desired distance of 0.1 showed the fastest convergence and highest reward at the end, giving the best balance between exploration and exploitation.

Model Improvement (DDPG)

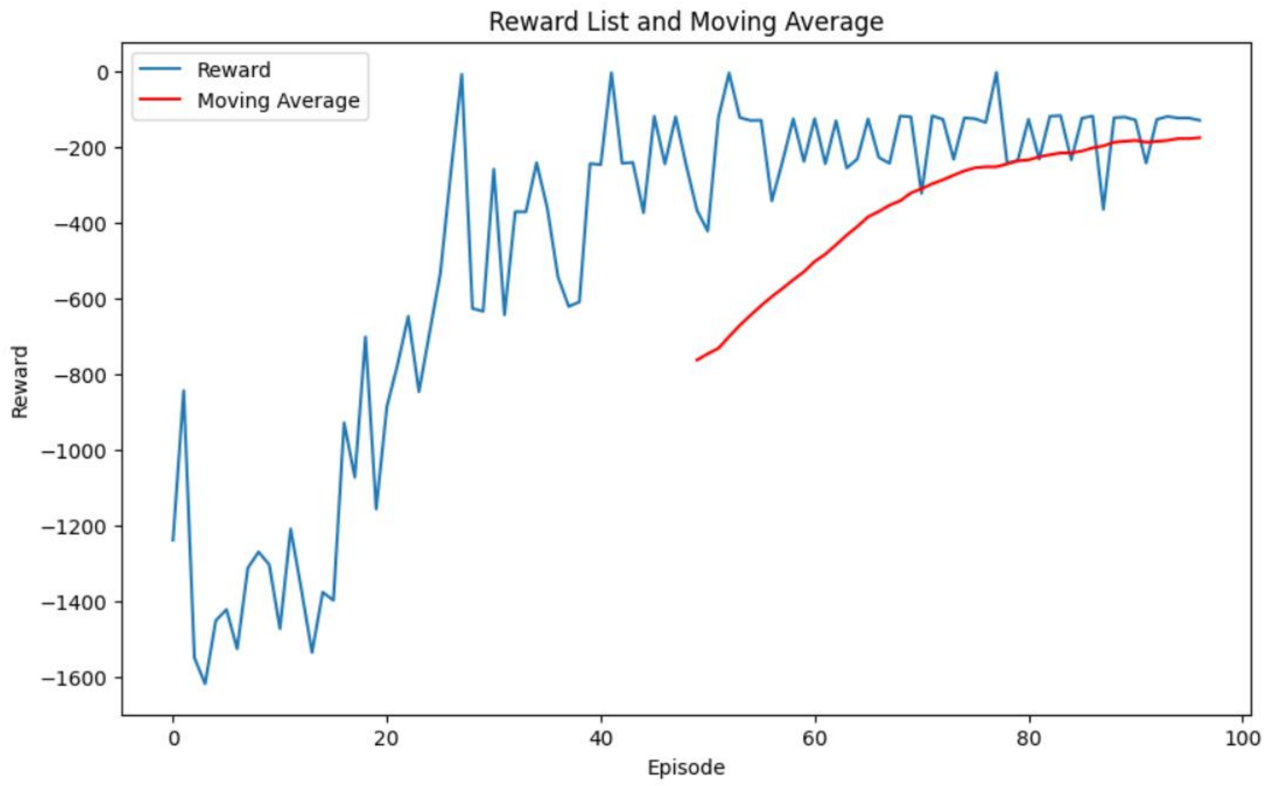
4. Using prioritized experience replay

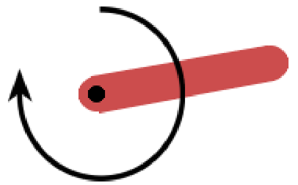
- PER focuses the learning process on experiences that are likely to provide more valuable learning by assigning a priority value.
- After replaying experiences, the priorities are updated based on the new Temporal Difference (TD) errors, which measures the difference between the current value estimate and the value derived from taking an action and moving to a new state. This ensures that experiences with larger errors, which are more informative, receive higher priorities in future sampling.
- By replaying these important experiences more frequently, the agent can learn more efficiently and converge faster.
- PER can help in reducing overfitting to specific experiences by ensuring that a diverse set of important experiences are sampled.

5. Choosing the best remaining parameters

- Gamma and Tau values were adjusted.
  - Gamma: the discount factor, it determines how much future rewards are worth compared to immediate rewards. Closer to 0, the agent will prioritize immediate rewards more heavily and will be more shortsighted. Closer to 1, the agent will consider future rewards more significantly, making it more farsighted.
  - Tau: the soft update factor for updating the target networks. A small tau results in very slow updates, which makes the target networks more stable but less responsive to changes. A larger tau results in faster updates, which makes the target networks more responsive but can lead to instability.
  - Increased Gamma to 0.999 to make the agent more farsighted, placing greater emphasis on future rewards rather than immediate ones. This adjustment helps the agent consider long-term outcomes, leading to more strategic decision-making.
  - Decreased Tau to 0.004 to slow down the updates of the target networks. This makes the target networks more stable and less susceptible to noise, although it reduces their responsiveness to changes in the policy.
- Actor and Critic learning rates.
  - Increased the actor learning rate to accelerate the convergence of the actor network, enabling it to adapt more quickly to the optimal policy.
  - Decreased the critic learning rate to enhance stability, preventing excessive fluctuations in the value estimates and improving overall training stability.

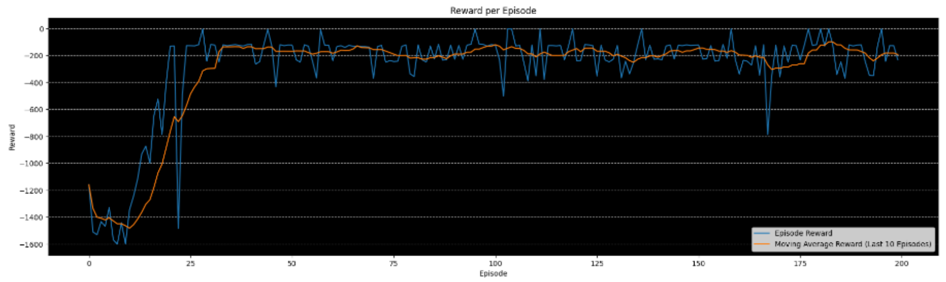
Final Model Evaluation (DQN)

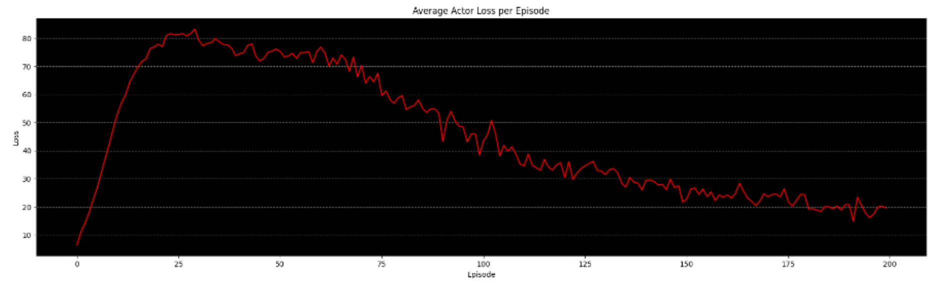


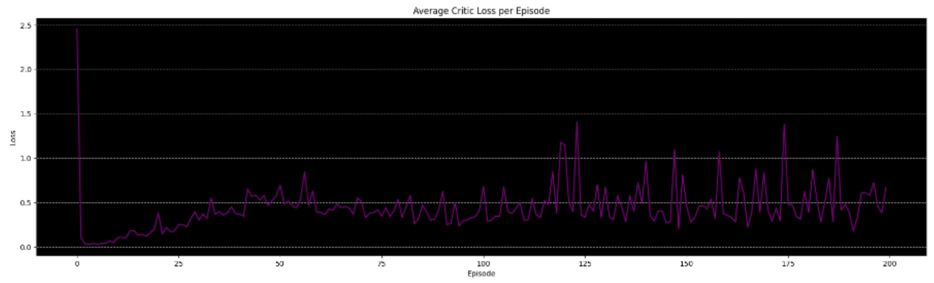


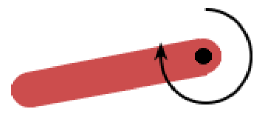
The moving average curve is much more stable as compared to the baseline model while the pendulum gif is stabilising upright extremely well. There are also much less spikes in the rewards.

Final Model Evaluation (DDPG)









The moving average curve is much more stable and the Actor & Critic networks' losses are decreasing better as compared to the baseline model while the pendulum gif is stabilising upright decently well. There are also much less spikes in the rewards.

Conclusion

To summarise, we tried experimenting with 2 different models, DQN and DDPG, *with more focus on the DQN*, to balance the pendulum in the Pendulum-v1 environment from gym. We also tuned both the models, with a heavy focus on making the moving average curve of the reward list smoother and more stable. Our main focus for determining the 'best setup' was to get the most stable learning.

Our final model evaluation for both models show a slightly better outcome for the DQN as compared to the DDPG. We find this surprising as we expected the DDPG to be better suited for this problem as it has a continuous action space rather than a discrete one. Then again, a possible reason for that is because we spent more time and effort on the DQN instead of the DDPG. Furthermore, it DDPG suffers from overestimation bias, causing the algorithm to overstate the Q-value of certain actions (https://arxiv.org/abs/2403.05732). It is also sensitive to hyperparameters and is unstable.

Finally, focusing solely on the DQN, our final model has a smooth and stable moving average curve for the rewards, as well as a stable balancing pendulum. Additionally, it also learns relatively fast, taking only 100 episodes to reach high stability.