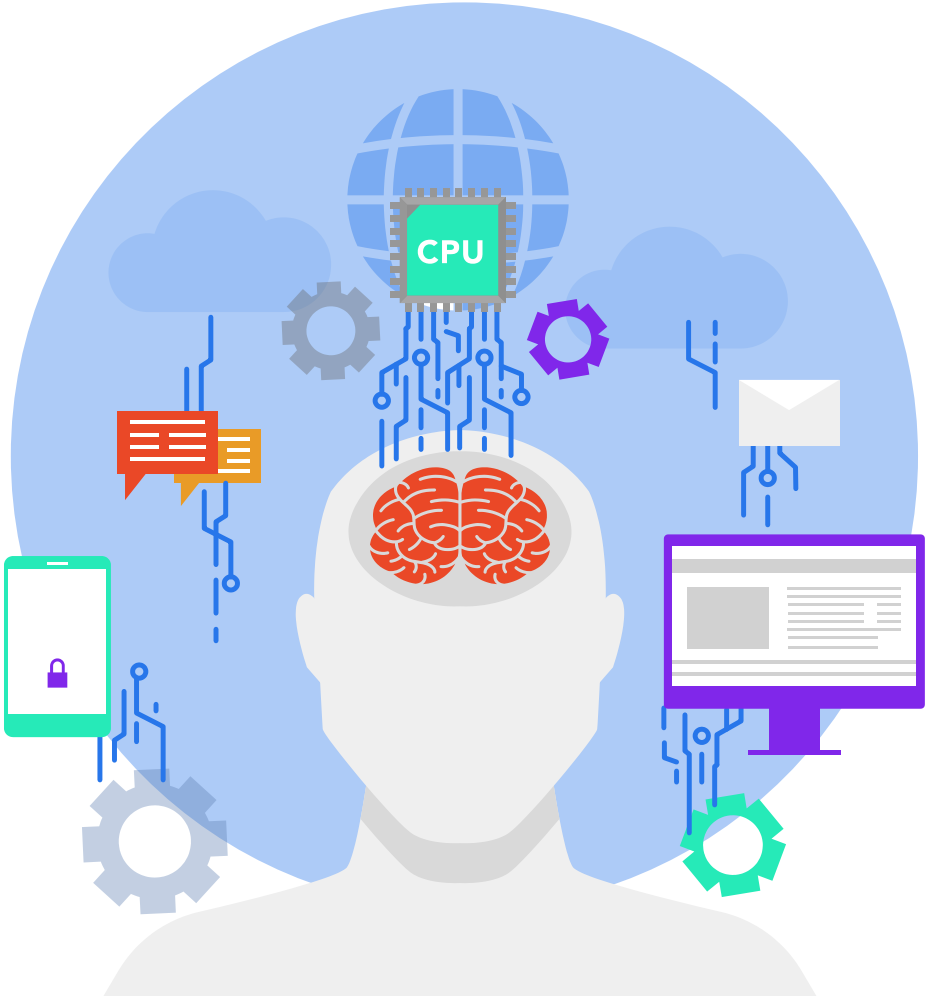
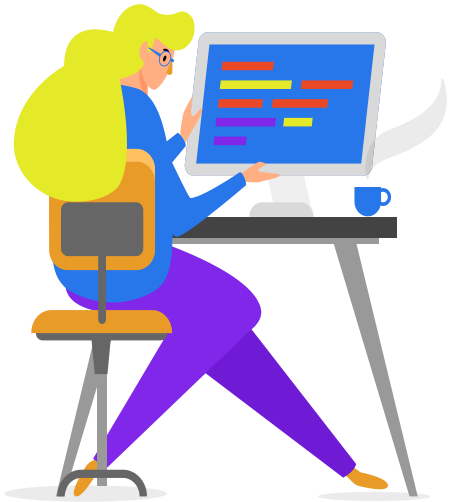


Deep Learning CA2 Part B

Jeyakumar Sriram (p2214618)
Shawn Lim Jun Jie (p2239745)

Contents

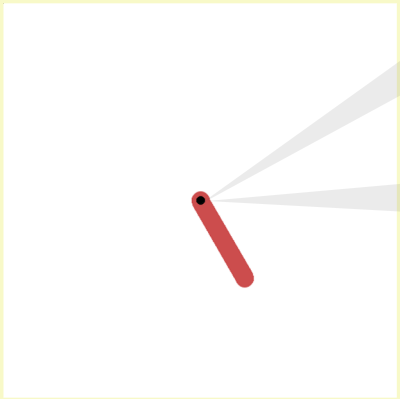


Objective

- Build and train a DQN capable of balancing a pendulum in an upright position.
- This is to be done in OpenAI’s Gym Pendulum Environment.
- We achieved this by building a simple DQN and explore and experiment ways we can improve the model performance.

Exploring the environment

OpenAI Pendulum Environment



Observation space

Provides the coordinates and angle of the pendulum in (x, y, θ).

Action space

Model inputs a value (torque) to influence the environment (the pendulum swing).

Theory behind DQN

Q-learning

- Model-free algorithm for finding optimal policies for decision making in an environment.
- The goal is to approximate the optimal action-value function.

Deep neural network

- Neural networks can be used as action-value function approximator.
- Inputs the environment's state and outputs Q-values for each action.

Experience replay

- DQN stores and randomly samples past experiences for learning.
- The purpose is to break temporal correlation, making more stable learning.

Loss function

- A simple DQN defines loss as mean square error between predicted Q-values and target Q-value.
- To minimise this loss value for more accurate Q-values prediction.

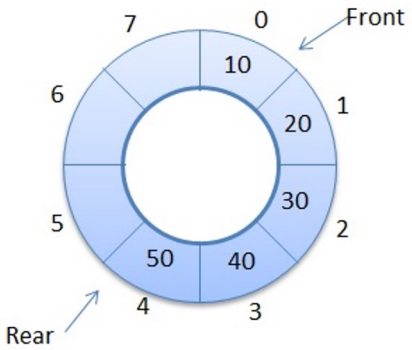
Simple DQN implementation

Built a simple Multi Layer Perceptron (MLP) which will be the “brain” of the agent.

```python
# Our Policy Network is a Simple Multi Level Perceptron

class BaseMLP(nn.Module):
    def __init__(self, inp, out, lr):
        super(BaseMLP, self).__init__()

        self.layer1 = nn.Linear(inp, 32)
        self.layer2 = nn.Linear(32, 64)
        self.layer3 = nn.Linear(64, 32)
        self.layer4 = nn.Linear(32, out)

        self.lr = lr
        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)

    def forward(self, x):
        x = self.layer1(x)
        x = F.relu(x)
        x = self.layer2(x)
        x = F.relu(x)
        x = self.layer3(x)
        x = F.relu(x)
        return self.layer4(x)
```

Simple DQN implementation

- Circular buffer is used to store past experience.
- Agent will sample random past experience to learn.
- Agent will not be using very old data to learn, hence a circular buffer where new data overwrites the old when the circular buffer is full.




```python
class Memory(object):

    def __init__(self, maxlen):
        self.buffer = deque([], maxlen=maxlen)
    def push(self, transition):
        self.buffer.append(transition)

    def sample(self, batch_size):
        mini_batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states = [], [], [], []
        # perform some manipulation to ensure dtypes match
        for transition in mini_batch:
            s, a, r, ns = transition
            states.append(s) # s is a tensor not a regular element
            actions.append([a])
            rewards.append([r])
            next_states.append(ns)

        states = torch.stack(states)
        actions = torch.tensor(actions, dtype=torch.float)
        rewards = torch.tensor(rewards, dtype=torch.float)
        next_states = torch.stack(next_states)
        return states, actions, rewards, next_states

    def __len__(self):
        return len(self.buffer)
```

Simple DQN implementation

- A simple DQN agent powered by the MLP we shown just now.
- It will output the an action based on the highest Q-value by inputting the action method with a state.
- Optimises itself by random sampling past experiences and calculate loss values and back propagate.

```python
class DQNAgent:
    def __init__(self):
        self.steps = 0
        self.memory = Memory(10000)

        self.batch_size = 256
        self.adamlr = 0.001
        self.gamma = 0.98

        # policy net, aka, Q function
        self.policy_net = BaseMLP(3, 9,self.adamlr)

    def action(self,state):

        with torch.no_grad():
            choice = float(torch.argmax(self.policy_net(state)).numpy())
            action = (choice-4)/2

        return choice, action

    def optimize(self):
        if len(self.memory) < self.batch_size:
            return

        batch = self.memory.sample(self.batch_size)
        state, action, reward, next_state = batch
        action = action.type(torch.int64)

        # target
        with torch.no_grad():   # parameters from previous iteration are held fixed
            yi = reward + self.gamma * self.policy_net(next_state).max(1)[0].unsqueeze(1)

        # Q(phi_i, a_i, theta)
        current_state_action = self.policy_net(state).gather(1,action)

        # Back_propagation
        loss = (yi - current_state_action) ** 2
        self.policy_net.optimizer.zero_grad()
        loss.mean().backward()
        self.policy_net.optimizer.step()
```

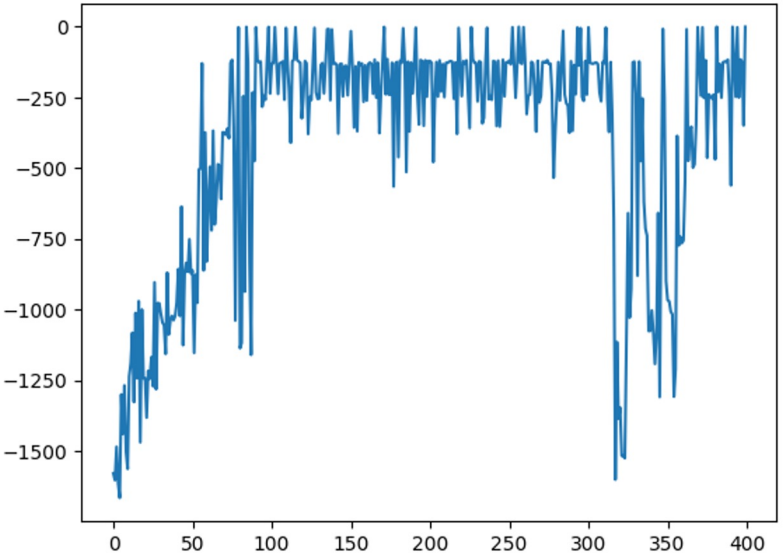
Simple DQN training

- For training, the agent will output an action for the environment, and the environment outputs the next state and reward.
- The state, agent's choice, reward, and next state are appended into the memory.
- After the memory reaches a certain size, the agent starts optimising.

```python
def train_model(DQNAgentX, episodes):
    agent = DQNAgentX()
    env = gym.make('Pendulum-v1')
    score_list = []
    for episode in range(episodes):
        print(f"Episode: {episode}",end="")
        state, _ = env.reset()
        state = torch.tensor(state)
        done = False
        score = 0
        while not done:
            choice, action = agent.action(state)

            next_state, reward, terminated, truncated, _ = env.step([action])

            score+=reward
            done = terminated or truncated

            if terminated:
                next_state = None
            else:
                next_state = torch.tensor(next_state,dtype=torch.float32)

            agent.memory.push((state, choice, reward, next_state ))
            state = next_state

            if len(agent.memory) > 1000:
                agent.optimize()

        print(f", Score: {score}")
        score_list.append(score)

    env.close()

    plt.plot(score_list)
    plt.show()

    return agent, score_list
```

Simple DQN training

- The graph on the right shows the score of the agent performance.
- The score is calculated by the sum of all the rewards in each episode.
- The model took around a 100 episodes to be good.
- The model suffers from “Catastrophic Forgetting” after 300 episodes.



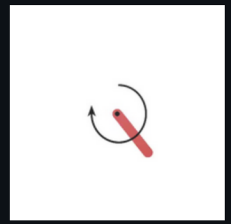
Simple DQN score comparison

When computing for the average 100 episodes score, we got -215. And when comparing to the best performant models, we are not far from the leaderboard, even for a basic model.

Pendulum-v0

The inverted pendulum swingup problem is a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright.

- Environment details
- *Pendulum-v0 is an unsolved environment, which means it does not have a specified reward threshold at which it's considered solved.*



User	Best 100-episode performance	Write-up	Video
KanishkNavale	-106.9528	MultiAgent Policy	
msinto93	-123.11 ± 6.86	D4PG	
msinto93	-123.79 ± 6.90	DDPG	
heerad	-134.48 ± 9.07	writeup	
BS Haney	-135	Write-up	YouTube
ThyrixYang	-136.16 ± 11.97	writeup	
MaelFrancesc	-146.4 (mean 900 ep)	writeup	
lirnli	-152.24 ± 10.87	writeup	

Simple DQN visualisation observation

Observations

- Model is able to balance the pendulum well.
- Suggesting that a simple model is capable of learning effectively.

Issues

- The model tends to apply huge amount of torque to maintain the position of the pendulum after reaching the top.
- Due to the model not getting enough discrete actions and a lack of action that produces near 0 torque.

Potential solutions

Increase action array size

- Increase the number of discrete actions.
- Provide the model more unique options to apply force.

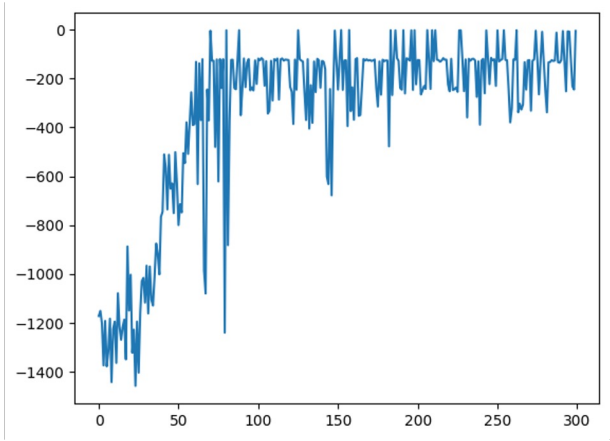
Quadratic spread

- Make the action array quadratic in spread
- Allows for more nuance control near zero torque.

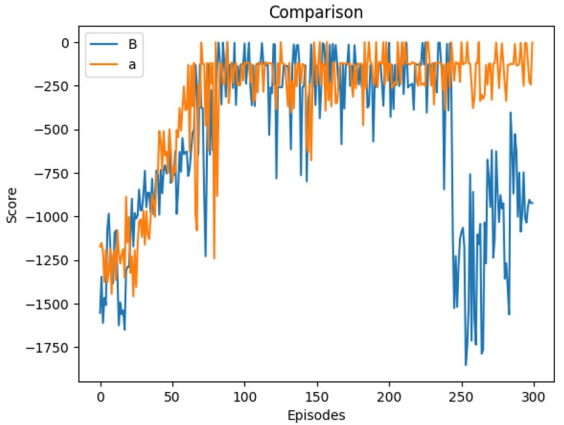
Model improvement (Best practices)

- Use Leaky ReLU instead of the normal ReLU
- Allows for non-zero gradient for negative inputs
- To address the dying ReLU problem

Score for improved model



Score for original vs improved model

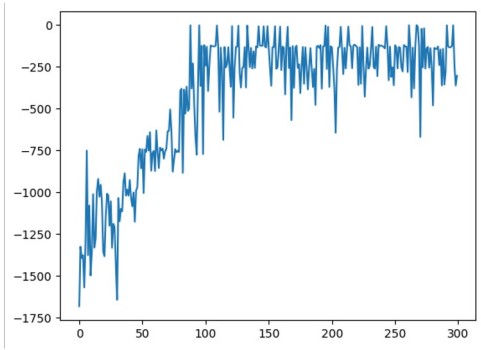


Model improvement (Training stabilization)

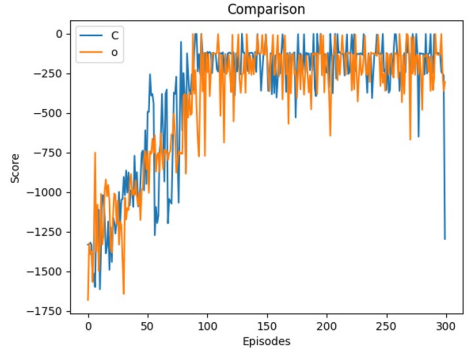
- Greedy Epsilon
  - Gives model the ability to exploit the best known action, or explore with random action.
  - To prevent model from converging a suboptimal policies, or “local minimum”.

- Gradient Clipping
  - Capping the minimum or maximum gradients to prevent extreme parameters updates.
  - To mitigate instability issues such as exploding gradients.

Score for improved model



Score for Leaky ReLU vs current improvement

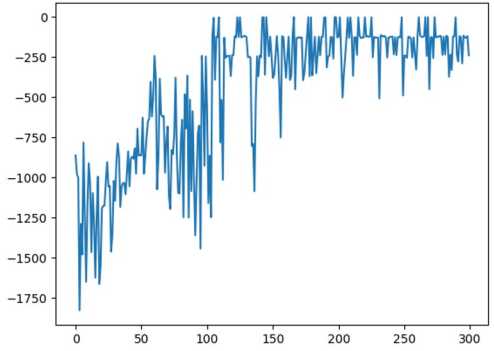


Model improvement (Enhanced action space)

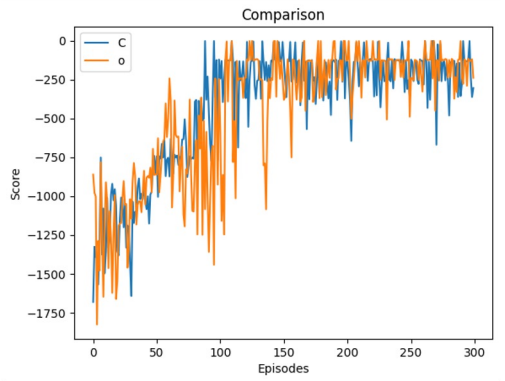
- Increase action array size
  - Gives the model more discretise input.
  - Can potentially provide a better baseline.

- More concentrated discrete values around 0
  - By applying quadratic or exponential distribution of action values.
  - Gives the model more nuanced options around 0.

Score for improved model



Score for previous improvements vs current improvement



Double DQN

The problem

- Simple DQN uses the same network for both selection and evaluation of actions.
- Can potentially lead to overestimation of Q-values.

The solution

- Implement Double DQN by having two separate networks.
- One of them for selecting best action, another for evaluating that action's Q-value.

The action

- A separate MLP for evaluating Q-values of selected actions.
- Parameters for both MLPs will be the same at first, and will be synchronised periodically.

Implementing Double DQN

Modifying the agent to have two MLPs: policy and target network

```
class DoubleDQNAgent:
    def __init__(self):
        self.steps = 0
        self.memory = Memory(10000)

        # Greedy Episilon Hyperparams
        self.epsilon       = 1.0
        self.epsilon_decay = 0.98
        self.epsilon_min   = 0.001

        self.batch_size = 256
        self.adamlr = 0.001
        self.targetlr = 0.01
        self.gamma = 0.98

        self.numOfAction = 20
        self.action_array = np.concatenate(((-1 *( np.exp(np.linspace(0

        self.policy_net = MLP(3, self.numOfAction,self.adamlr)
        self.target_net = MLP(3, self.numOfAction,self.adamlr)
        self.target_net.load_state_dict(self.policy_net.state_dict())
```

Modifying the optimiser to for the target network to evaluate.

```
def optimize(self):
    if len(self.memory) < self.batch_size:
        return

    batch = self.memory.sample(self.batch_size)
    state, action, reward, next_state = batch
    action = action.type(torch.int64)

    with torch.no_grad():
        expected_q = self.target_net(next_state).max(1)[0].unsqueeze(1)
        expected = reward + self.gamma * expected_q

    # Actual values - Policy Net
    actual = self.policy_net(state).gather(1,action)

    # Back_propagation
    loss = F.smooth_l1_loss(actual, expected)
    self.policy_net.optimizer.zero_grad()
    loss.mean().backward()
    nn.utils.clip_grad_norm_(self.policy_net.parameters(), 0.5)
    self.policy_net.optimizer.step()

    for target_param,param in zip(self.target_net.parameters(),self.policy_net.parameters()):
        target_param.data.copy_(target_param.data * (1.0 - self.targetlr) + param.data * self.targetlr)
```

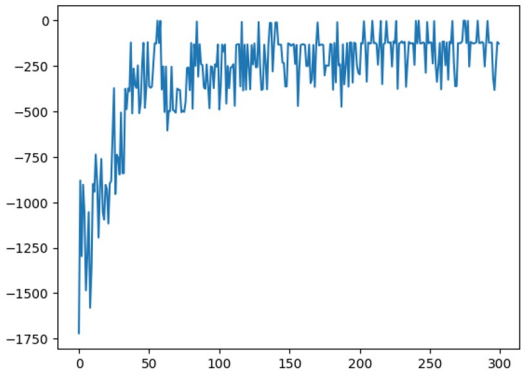
Double DQN result

The result of Double DQN shows that:

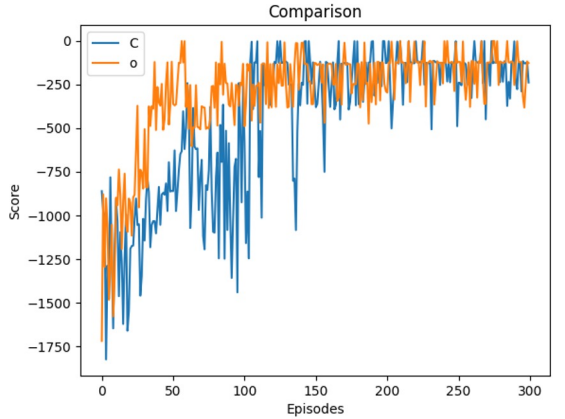
1. The model learn very fast, reaching around -250 score in 50 episodes.
2. The score graph looks generally stable.
3. With the evaluation score (100 episodes average) of -150, it's right on the leaderboard of the Pendulum Problem.

Conclusion: Double DQN is far better in our scenario.

Score for Double DQN

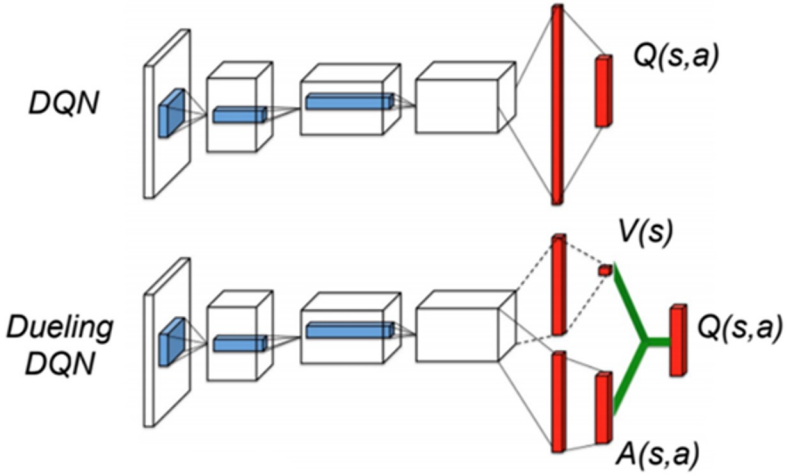


Score for previous simple DQN vs Double DQN model



Dueling Double DQN

- By having two separate sequential linear layers as shown on the right, where one will output the value, while the other output a series of advantages, then we have a dueling DQN.
- Couple with our Double DQN, we have the Dueling Double DQN.
- We want to see if having two separate layers can improve the performance by letting the model know the value of each state.



Implementing Dueling Double DQN

- Simply create two sequential layers, one will output one feature, the value stream, and another will output the size of the action array provided, the advantage stream.
- When combining the values and advantages, we can't just simply add them, we add them and minus the mean of the advantage, according to the formula on the paper for Dueling DQN.

```python
class DDQN_MLP(nn.Module):
    def __init__(self, inp, out, lr):
        super(DDQN_MLP, self).__init__()

        self.layer1 = nn.Linear(inp, 32)
        self.layer2 = nn.Linear(32, 64)

        self.value_stream = nn.Sequential(
            nn.Linear(64, 64),
            nn.LeakyReLU(inplace=True),
            nn.Linear(64, 1)
        )

        self.advantage_stream = nn.Sequential(
            nn.Linear(64, 64),
            nn.LeakyReLU(inplace=True),
            nn.Linear(64, out)
        )

        self.lr = lr
        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)

    def forward(self, x):
        x = self.layer1(x)
        x = F.leaky_relu(x)
        x = self.layer2(x)
        x = F.leaky_relu(x)
        # x = self.layer3(x)
        # x = F.leaky_relu(x)

        values = self.value_stream(x)
        advantages = self.advantage_stream(x)
        q_values = values + advantages - advantages.mean()

        return q_values
```

Formula for combining value and advantage:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta,\alpha) \right). \quad (9)$$

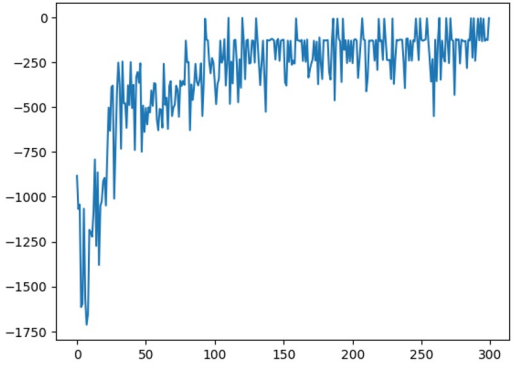
Dueling Double DQN result

The result of Dueling Double DQN shows that:

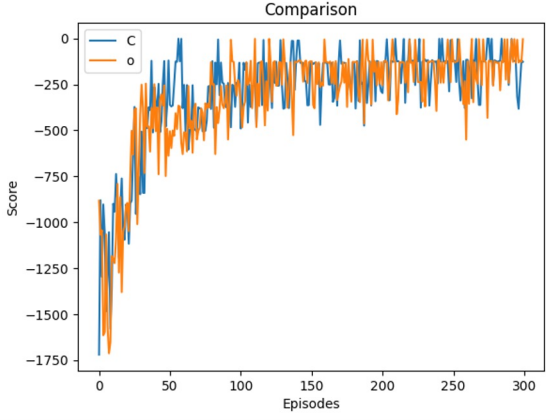
1. For this pendulum problem, it is not an improvement compared to Double DQN, based on the score comparison.
2. Having an advantage and value streams does not seem to provide an advantage for this simple environment.

Conclusion: Continue using the Double DQN for our scenario

Score for Dueling Double DQN



Score for Double DQN vs Dueling Double DQN

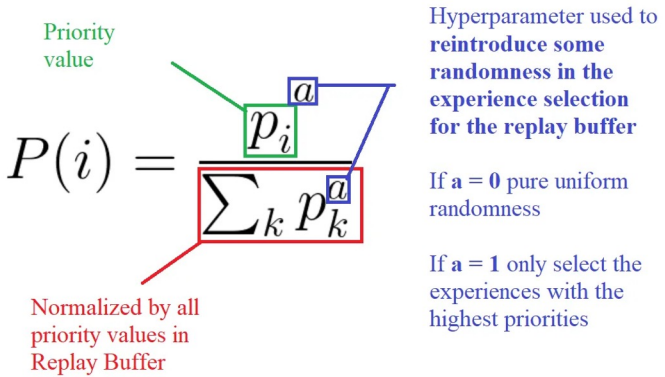


Further model improvement (Enhanced memory sampling)

By using Prioritised Experience Replay:

- We assign priority values to each experience.
- Experiences with higher learning potential (higher TD error), are assigned higher priority values.
- When sampling for experiences, those with higher priority values will be more likely to be sampled for learning.

However, experiences with lower priority values will be ignored. Hence, there will be an element of probability to ensure higher priority values will get higher chance while smaller priority values small chance, giving all experiences the chance to be used. This also comes with a hyperparameter a to adjust the randomness, as shown on the right.



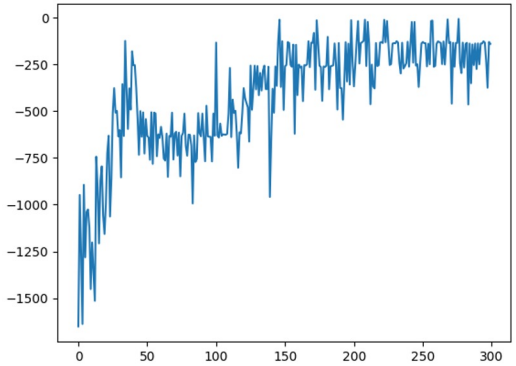
Model improvement result (Enhanced memory sampling)

The results for this improvement shows that:

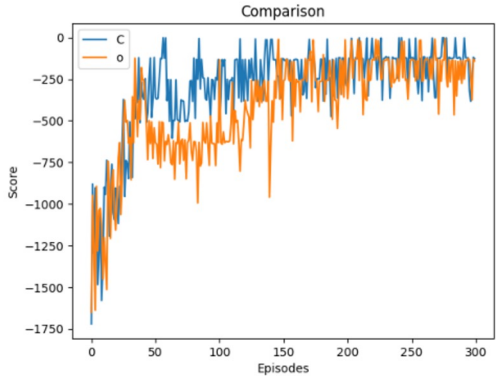
1. It took more epochs for the model to converge.
2. No increase in performance compare to using random sampling.
3. Given the simplicity of the pendulum environment, this improvement does not provide any benefits.

Conclusion: Regular random sampling simply works well for this scenario.

Score for this improvement



Score for before vs after



Further model improvement (Noisy net)

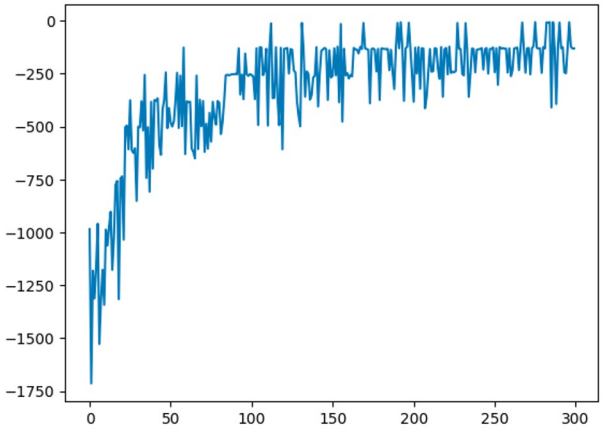
Lastly, we will also try out another way to let the model balance between exploration and exploitation.

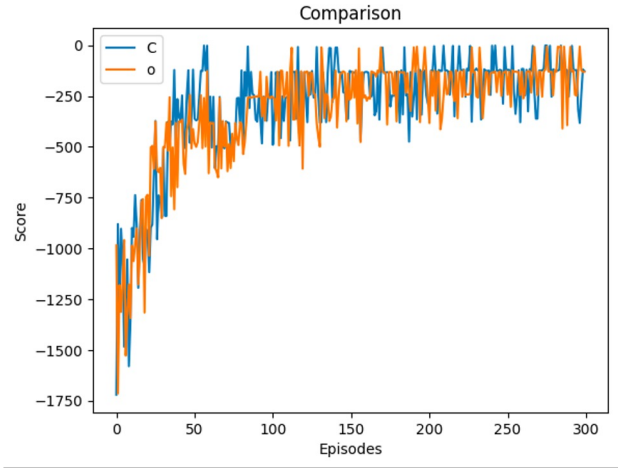
By introducing a noisy layer on the output of the MLP, we can take advantage of the follow benefits:

1. Epsilon-greedy strategy might not be sufficient enough in complex environment.
2. As this is a parameterisable layer, it allows the model to adjust the noise for appropriate situations.
3. The noise is able to introduce randomness on the action selection, making it less prone to overestimation bias.

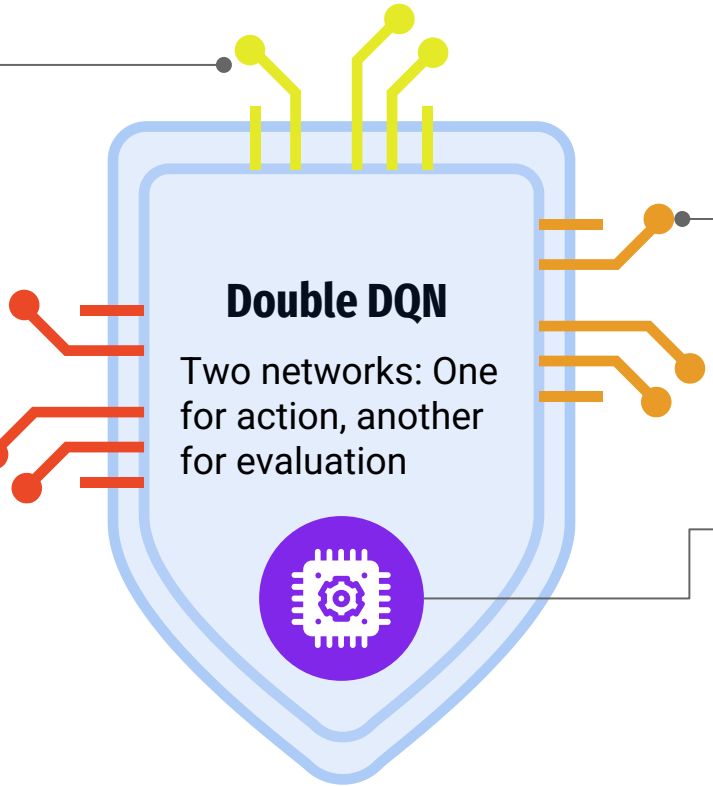
Model improvement result (Noisy net)

Improvements: Although the graphs look very similar for both, the Noisy DQN has a slightly higher 100-epoch average. It consistently ranks higher than the previous model





Our final model



01

Dueling

Fix over Estimation of Q Value

02

Gradient clipping

Mitigate instability issues

03

Exponential action array

Allows for more nuanced action near zero torque.

04

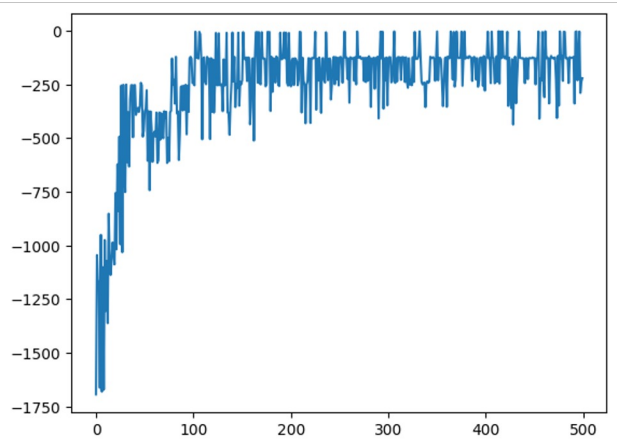
Noisy net

More efficient way to give exploration or exploitation for agent

Double DQN

Two networks: One for action, another for evaluation

Our final model evaluation



- Consists of mostly basic model improvement.
- Converge at around 100 episodes.
- On evaluation, it achieved a 100 epoch average score of -170.

This score is very close to the global highscore of -100

This model can be even better if we use more advanced architectures like A3C. This is because DQNs are primarily designed for discrete problem where you have a set number of possibilities. That is what a Q function is about. When we use it for continuous value prediction problems like this, the model will definitely face limitations.

Conclusion

- We have learnt how to develop DQN in reinforcement learning.
- The simple DQN was initially unstable, but basic model improvements help to mitigate the issues.
- After the introduction of the Double DQN, any further model improvements were unnecessary due to the simplicity of the pendulum environment.
- Hence, we learn that the techniques used for reinforcement learning should align to the complexity of the problem we are solving. A simple technique is enough for a simple problem.