Programming Assignment 2: Report

EP20B005: Aditya Kumar
EP20B025: Mohammed Mustafa

Github - https://github.com/Sylindril/PA2.git

Dueling DQNs:

To extend the codebase from tutorial 5 to dueling DQNs, some features were introduced:

Modifying the Q-Network:

```
class DuelingQNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed,fc1_units=128, fc2_units=64):
        super(DuelingQNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        # # Define shared layers, state value stream, and advantage stream
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc_value = nn.Linear(fc1_units, fc2_units)
        self.fc_adv = nn.Linear(fc1_units, fc2_units)
        self.value = nn.Linear(fc2_units, 1)
        self.adv = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        value = F.relu(self.fc_value(x))
        adv = F.relu(self.fc_adv(x))
        value = self.value(value)
        adv = self.adv(adv)
        adv_average = torch.mean(adv, dim=1, keepdim=True)
        #q = value + adv - adv_average
        q = value + adv - torch.max(adv, dim = 1, keepdim = True)[0]
        return q

QNetwork1 = DuelingQNetwork
```

We modify the initialisation to create two separate streams to estimate value and advantage and implement one of the two update rules. We have a common input layer, which is then connected to two dense layers (128, 64 nodes resp) before being sent to the output layer. The duelling DQNs share their input-to-first dense layer (128 nodes) weights.

- Update Rule 1 -

```
adv_average = torch.mean(adv, dim=1, keepdim=True)
q = value + adv - adv_average
```

- Update Rule 2 -

```
q = value + adv - torch.max(adv, dim = 1, keepdim = True)
```

Implementing a replay function on top of the replay buffer

```
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, history_size, fraction, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.initial_history = deque(maxlen = history_size)
        self.fraction = fraction
        self.batch_size = batch_size
        self.history_size = history_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)
        if len(self) <= self.history_size: self.initial_history.append(e)


    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences1 = random.sample(self.memory, k=int(self.batch_size*(1-self.fraction)))
        ''' Randomly sample a batch from initial history '''
        experiences2 = random.sample(self.initial_history, k=int(self.batch_size*self.fraction))
        experiences = experiences1 + experiences2

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)
```

A simple method that worked well to solve catastrophic forgetting was to maintain a history (HISTORY_SIZE = 1000) of initial experiences when the agent's exploration parameters were at their highest and regularly sample from this deque even into later episodes so that the agent doesn't forget the negative implications of wrong actions.

When a batch of experiences was sampled, 20% (FRACTION = 0.20) of the experiences were sampled from the initial history and the rest from the normal replay buffer. These hyper-parameters were optimized to the best of our ability.

This method may delay the agent's training later, but the FRACTION parameter may be annealed to offset this. This problem is well-documented in the literature, eg. *Implementing the Deep Q-Network*, Roderick et al.

DDQN - Cartpole

There were significant cases of catastrophic forgetting over here. We investigated several scenarios:

Average of 5 Successful attempts to solve the CartPole problem:

First, we introduced a variable in the dqn code to identify if the problem was solved in the run

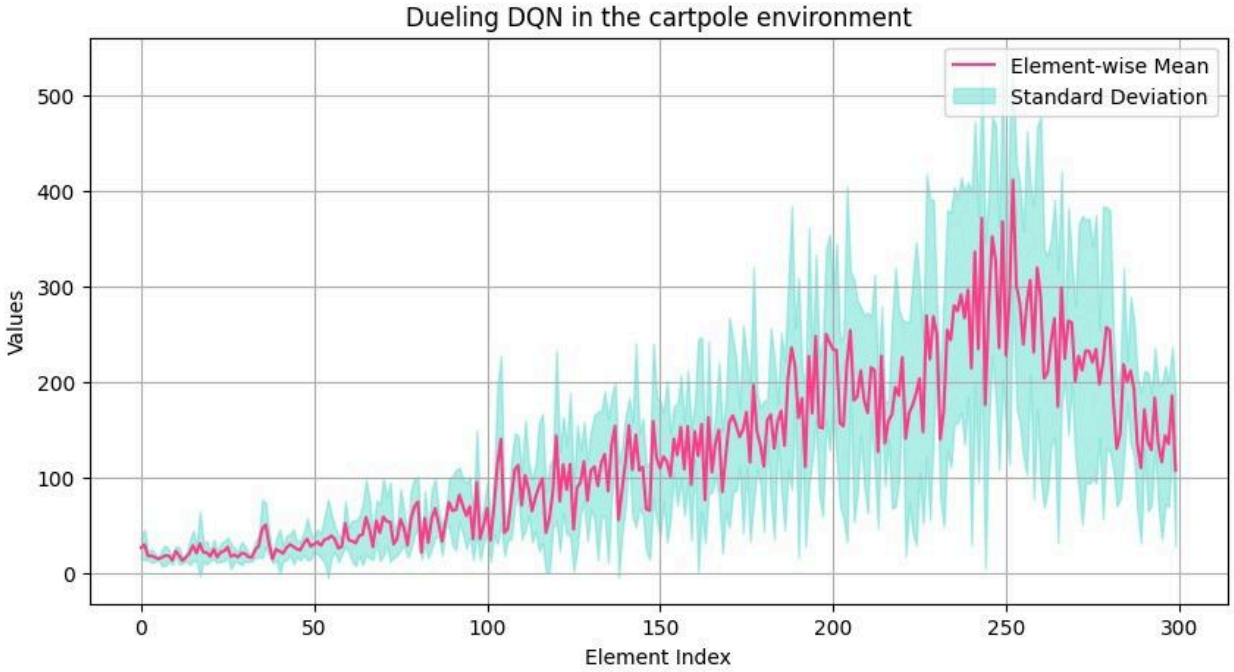
```
def dqn(n_episodes=10000, max_t=1000, policy="ep", eps_start=1.0, eps_end=0.01, eps_decay=0.995, \
                                                    tau_start=1.0, tau_end=0.05, tau_decay=0.995):

    scores_window = deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''
    solved = False

    eps_reward = []

    if policy=="ep":

      eps = eps_start
      for i_episode in range(1, n_episodes+1):
          # ipdb.set_trace(context=6)
          state = env.reset()
          score = 0
          for t in range(max_t):
              action = agent.act(state, eps)
              next_state, reward, done, _ = env.step(action)
              agent.step(state, action, reward, next_state, done)
              state = next_state
              score += reward
              if done:
                  break

          scores_window.append(score)
          eps_reward.append(score)

          eps = max(eps_end, eps_decay*eps)
          ''' decrease epsilon '''

          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")

          if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
          if np.mean(scores_window)>=-100.0:
            solved = True
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
            plot(eps_reward)
            return eps_reward, solved
          if i_episode % 500 == 0:
```

And then, we obtained 5 such runs

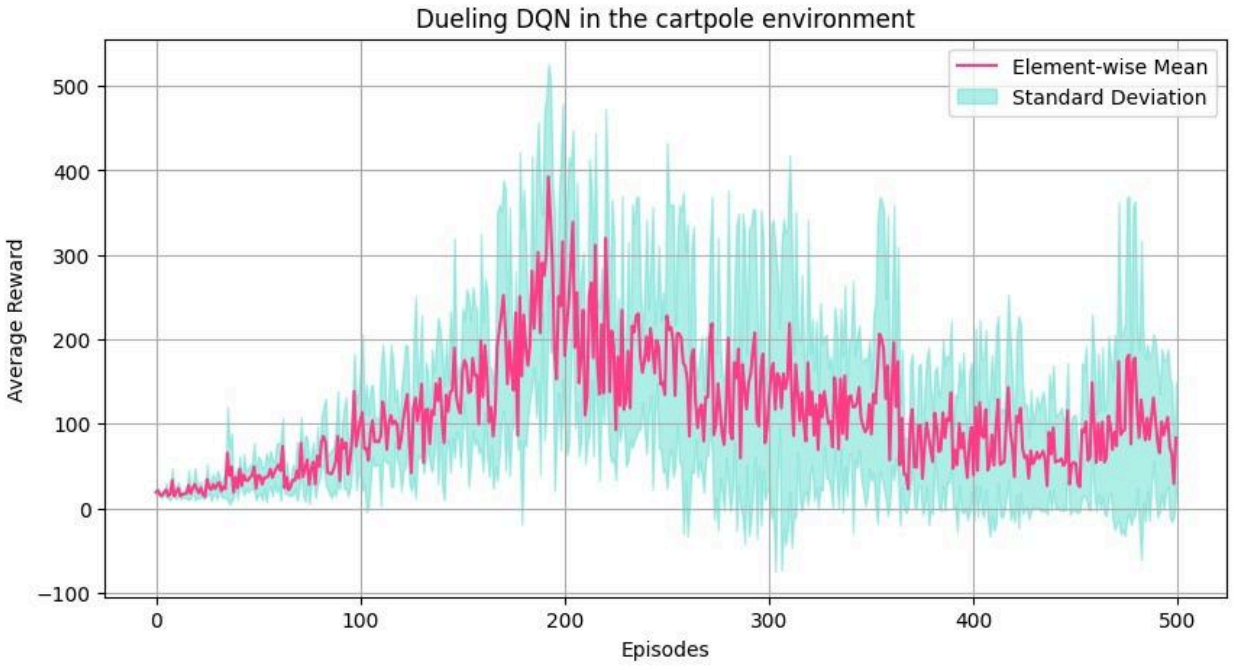
```
all_scores = []
seed = 0
counter = 0
while True:
  np.random.seed(seed)
  agent = TutorialAgent(state_shape, no_of_actions, seed)
  scores, solved = dqn(policy="ep", n_episodes = 300)
  if solved:
    all_scores.append(scores)
    counter += 1
  seed += 1
  if counter >= 5:
    break
```

- Here, we only took 5 cases where the cartpole problem was solved (score >195) within a fixed number of iterations.
- For **Type 1** Update - (Please note that in the labels, values = mean rewards over 5 iterations, and element index = episodes) is shown below. The following code snippet is the template for all graphs in the DDQN section -

```
arrays_avgUpdate = all_scores_avg.copy()
mean_values_avgUpdate = np.mean(arrays_avgUpdate, axis=0)
std_deviation_avgUpdate = np.std(arrays_avgUpdate, axis=0)

# Plotting
plt.figure(figsize=(10, 5))
plt.plot(mean_values_avgUpdate, label='Element-wise Mean', color='#FF3E87')
plt.fill_between(range(300), mean_values_avgUpdate - std_deviation_avgUpdate, mean_values_avgUpdate + std_deviation_avgUpdate, color='#64E0D5', alpha=0.5, label='Standard Deviation')
plt.title('Dueling DQN in the cartpole environment - Type 1 Update')
plt.xlabel('Episodes')
plt.ylabel('Mean Reward')
plt.legend()
plt.grid(True)
plt.show()
plt.savefig("DDQN_Cartpole_Type1.png")
```



- We do not observe any stable plateau of intelligence, but rather a consistent forgetting in the system as it continues to train. We repeated this experiment, but extended it for 500 iterations, and the graph of the mean of 5 successful runs appeared as follows in a **Type 2** update (note that the axes labeled should be interpreted as prior)



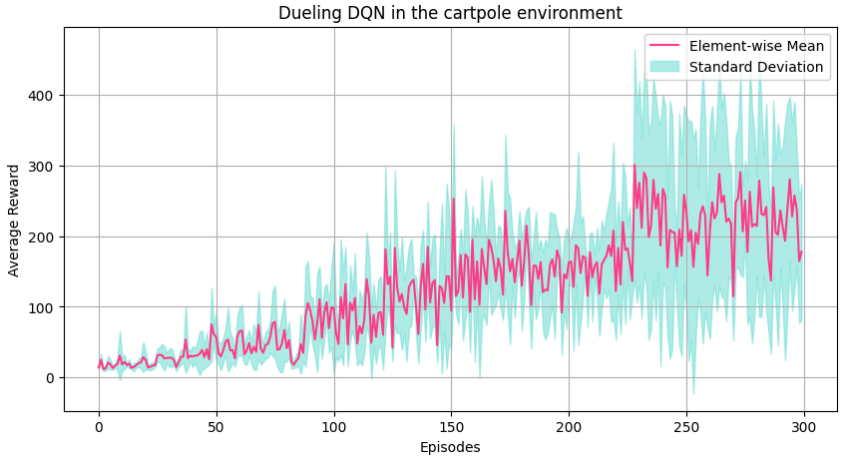
- A similar behaviour as type 1 was seen. However, the peak in the **Type 2 Update** was at ~200 episodes, which was before the **Type 1** update (~250 episodes)

Average of 5 Attempts Regardless of Success

We utilised straightforward looping to obtain this

```
all_scores = []
seed = 0
counter = 0
while True:
  np.random.seed(seed)
  agent = TutorialAgent(state_shape, no_of_actions, seed)
  scores, solved = dqn(policy="ep", n_episodes = 500)
  all_scores.append(scores)
  counter += 1
  seed += 1
  if counter >= 5:
    break
```

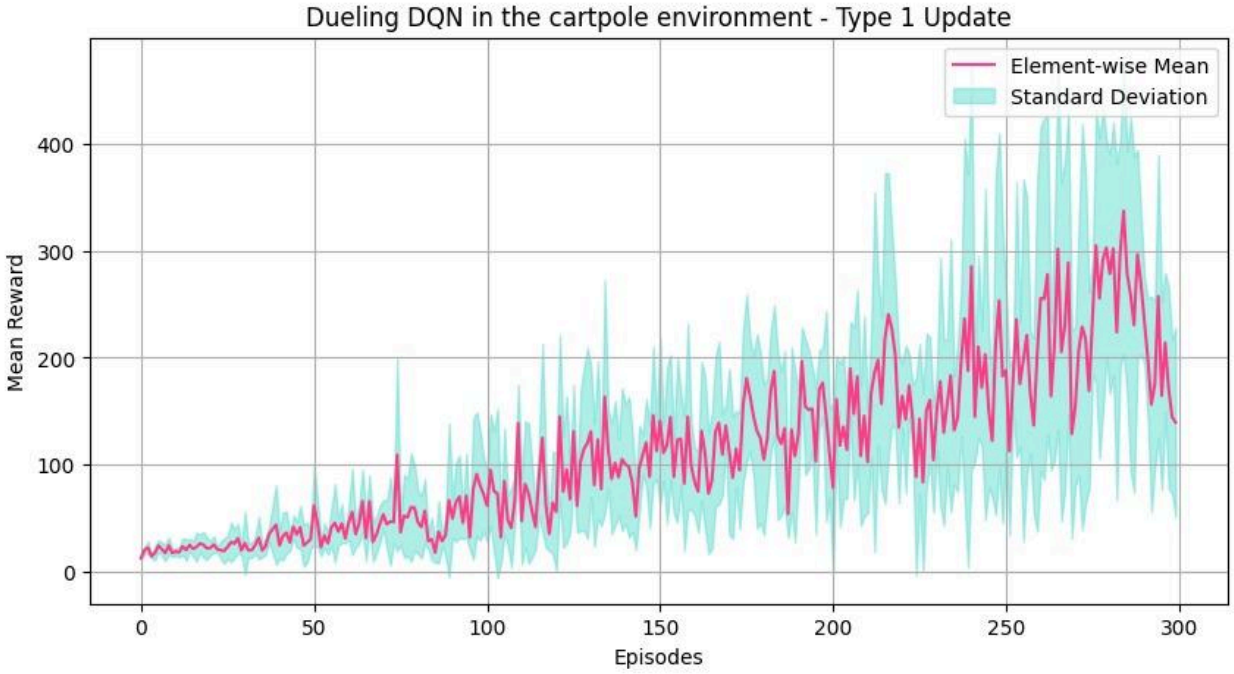
This is in alignment with our interpretation of the programming assignment's actual question and despite extensive attempts at hyperparameter optimisation (implementation of a history function, varying history sizes between 5 to 25 %, varying number of nodes in each layer between 32, 64 and 128, varying hyperparameters in the NN such as learning rate (0.0001,0.0005,0.001,0.01), batch size (16,32,64,128) and intervals between updating the frozen network (10,20), varying the epsilon decay), we were unable to obtain a stable answer; with every run, the variability and problems of catastrophic forgetting persisted. In some runs, we obtained several plateaus, as shown below -



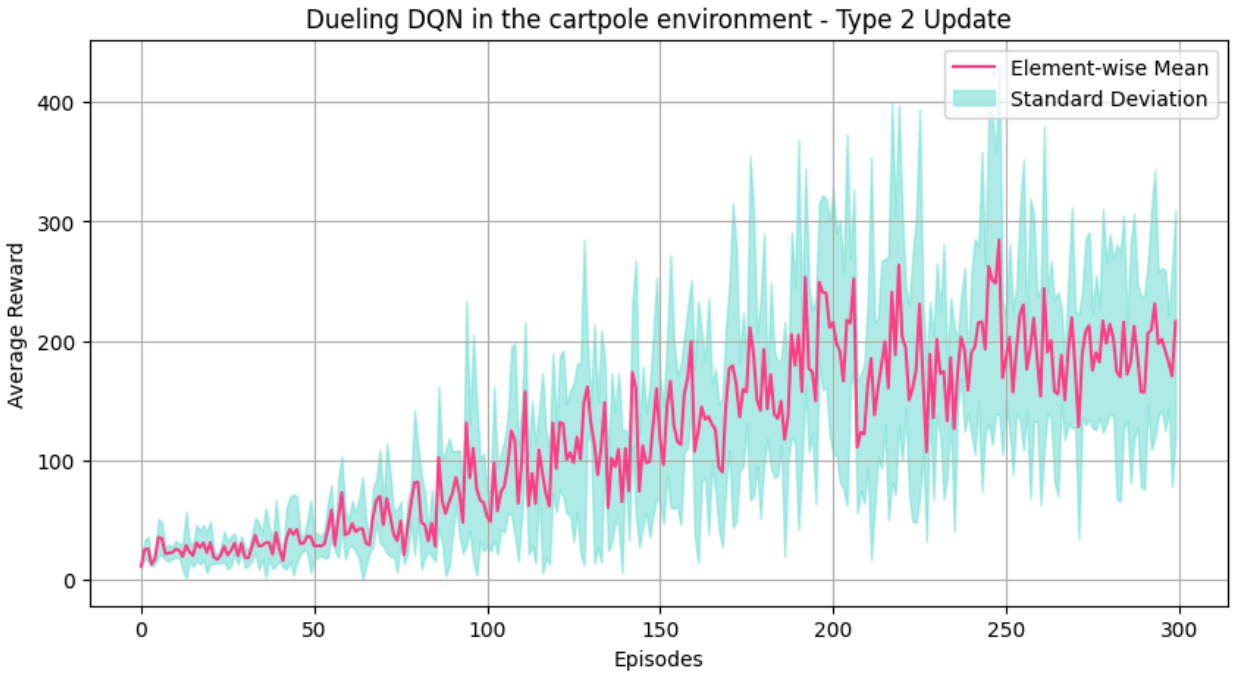
Ultimately, we arrived at the following:

```
'''
Bunch of Hyper parameters (Which you might have to tune later)
'''
BUFFER_SIZE = int(1e6)  # replay buffer size
HISTORY_SIZE = int(1000)
FRACTION = 0.2
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99          # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is present)
```

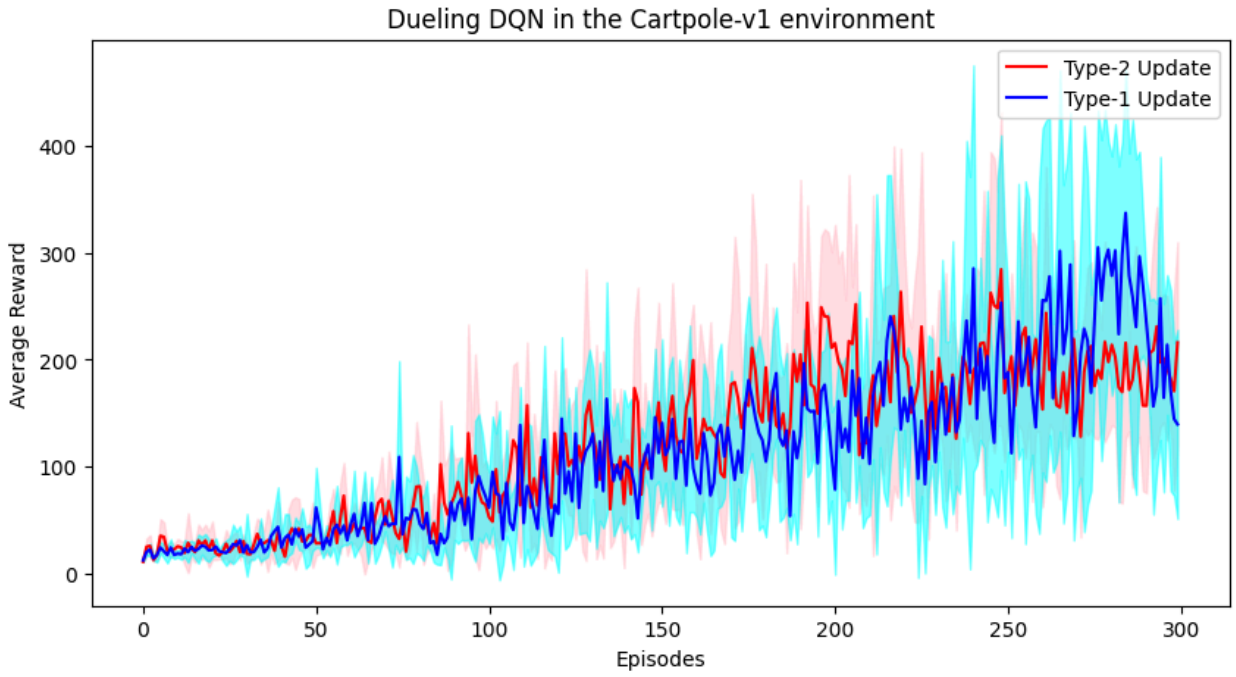
Type 1 Update:



Type 2 Update



Plotting them both on the same graph:



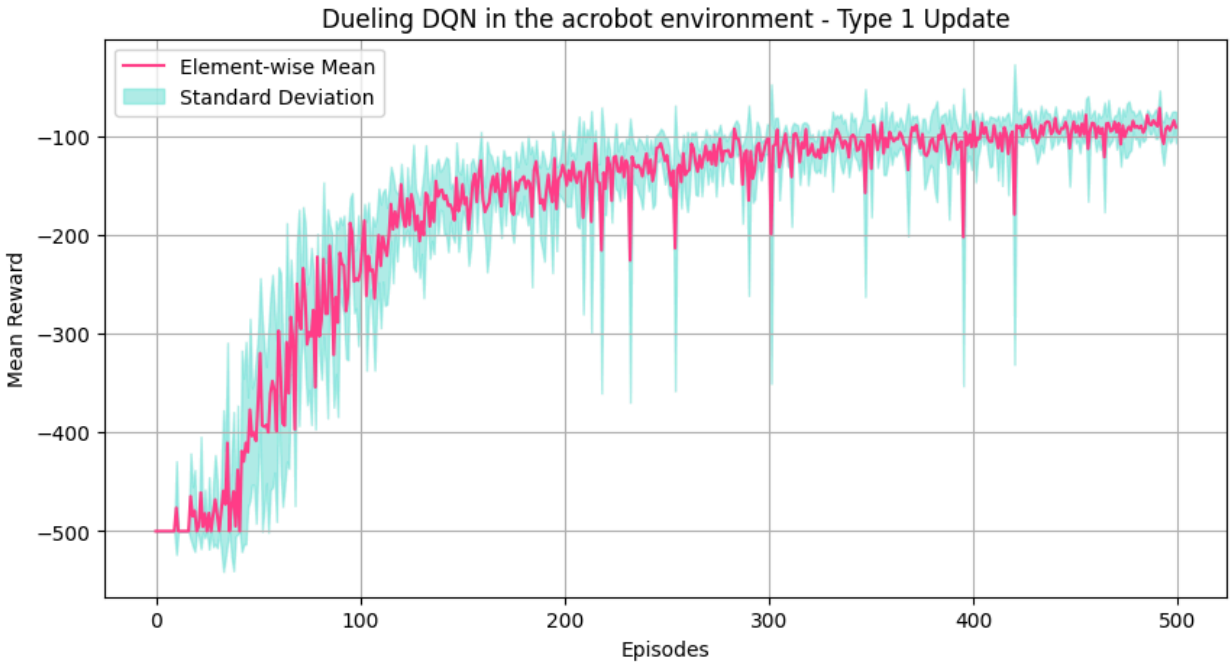
Here, the cyan shaders correspond to the variance of the type-2 update, and the pink shaders to the variance of the type-1 update.

DDQN - Acrobot:

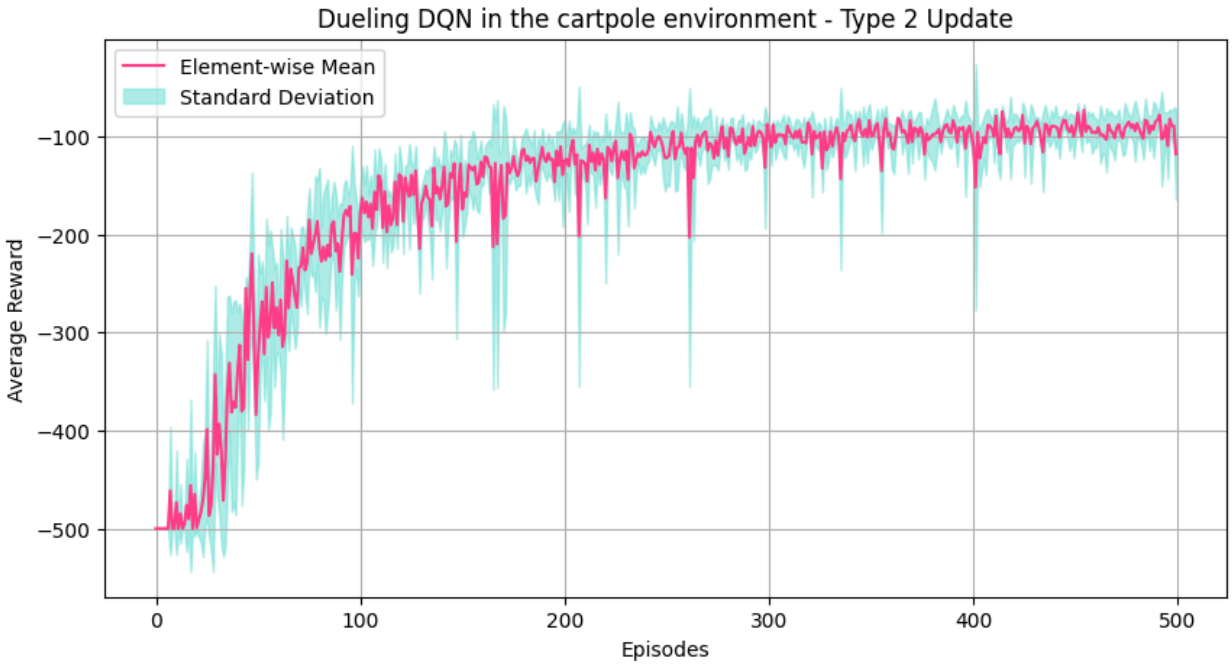
Here, our hyperparameter optimization was much more successful. Using the same common NN architecture and history functionality, we were able to obtain the following graphs:

```
'''
Bunch of Hyper parameters (Which you might have to tune later)
'''
BUFFER_SIZE = int(1e6)  # replay buffer size
HISTORY_SIZE = int(1000)
FRACTION = 0.2
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99          # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is present)
```

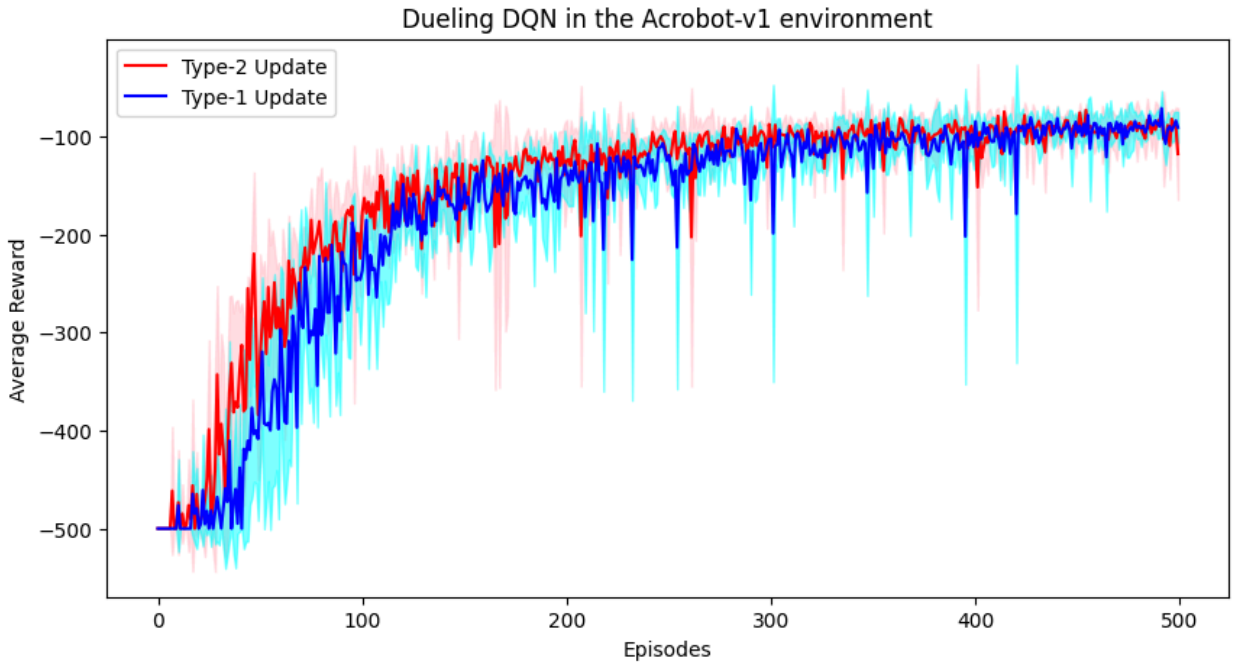
Type 1 Update:



Type 2 Update:



Combined Graph:



Here, the cyan shaders correspond to the variance of the type-2 update, and the pink shaders to the variance of the type-1 update.

DDQN: Inferences

Cartpole-v1 Environment

- The performance of the two DDQNs is quite similar, and achieves relatively high rewards, spiking upto 400 (individual) or 300 (mean). To achieve the best performance, we could save the network parameters at the spike, as suggested in Roderick et al.s' paper
- The Type-1 update rule explicitly learns the state value and advantage streams, while the Type-2 update rule subtracts the maximum advantage from the state value.
- Type-2 update rule could potentially be more stable during training, as it directly optimizes the actual Q-value instead of the decomposed components. Indeed, we observe Type 2 Learning to have overall slightly less variance than Type 1, suggesting more stable learning.
- As Cartpole is not very complex, this lack of complexity could be the reason why the difference between the two update rules is not significant.
- As the relative importance of different actions isn't as crucial, the explicit decomposition of the Q-value function in Type-1 Learning doesn't seem to confer any advantages over the slight stability offered by Type-2

Acrobot-v1 Environment

- The performance of the two Dueling-DQN implementations diverges more in the Acrobot-v1 environment compared to the Cartpole-v1 environment, in that there are more spikes in Type-2, and more variance

- The Type-1 Update Rule appears to outperform the Type-2 Update Rule, achieving higher average rewards and exhibiting lower variance.
- The performance of the Type-2 Update Rule is slightly more volatile, with larger fluctuations in the average rewards.
- As Acrobot is known to be more complex than Cartpole, the explicit decomposition of the Q-value function in the Type-1 Update Rule seems to be more beneficial. It allows the model to better capture the relative importance of different actions.
- On the other hand, the formulation of the Type-2 Update Rule, where the maximum advantage is subtracted from the state value, might be more sensitive to the complexity of the Acrobot task, leading to slower convergence and higher variance
- Perhaps learning the state value and advantage streams separately confers a more efficient exploration and learning in this environment by the Type 1 update.

MC REINFORCE:

```python
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 256)
        self.dropout = nn.Dropout(p=0.5)
        self.affine2 = nn.Linear(256, 2)

        self.saved_log_probs = []
        self.rewards = []

    def forward(self, x):
        x = self.affine1(x)
        x = self.dropout(x)
        x = F.relu(x)
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=1)
```

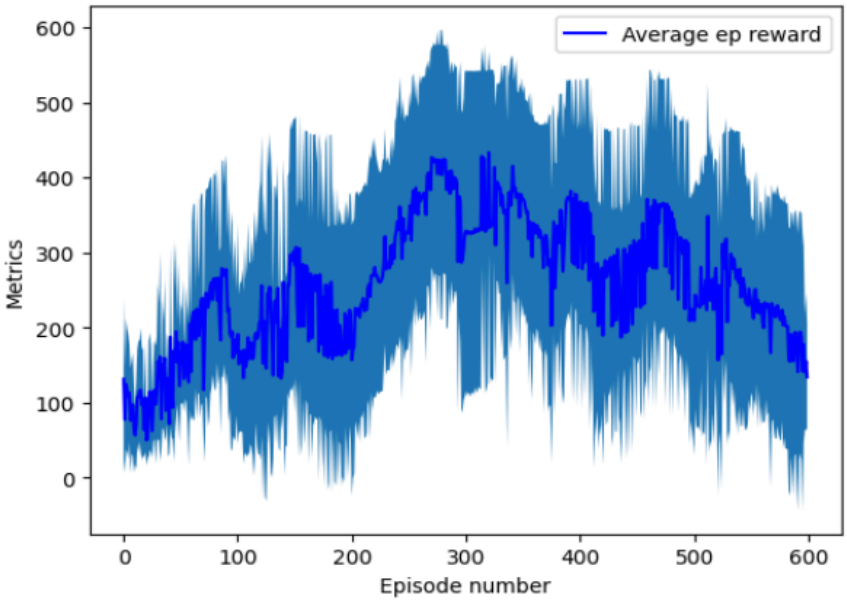
A deep neural network with 2 dense layers, 1 dropout layer and a softmax activation layer was found to be best suited to model the actor policy. The hidden layer size was settled on 256 since it produced a better running reward than 128,64.

WITHOUT BASELINE:

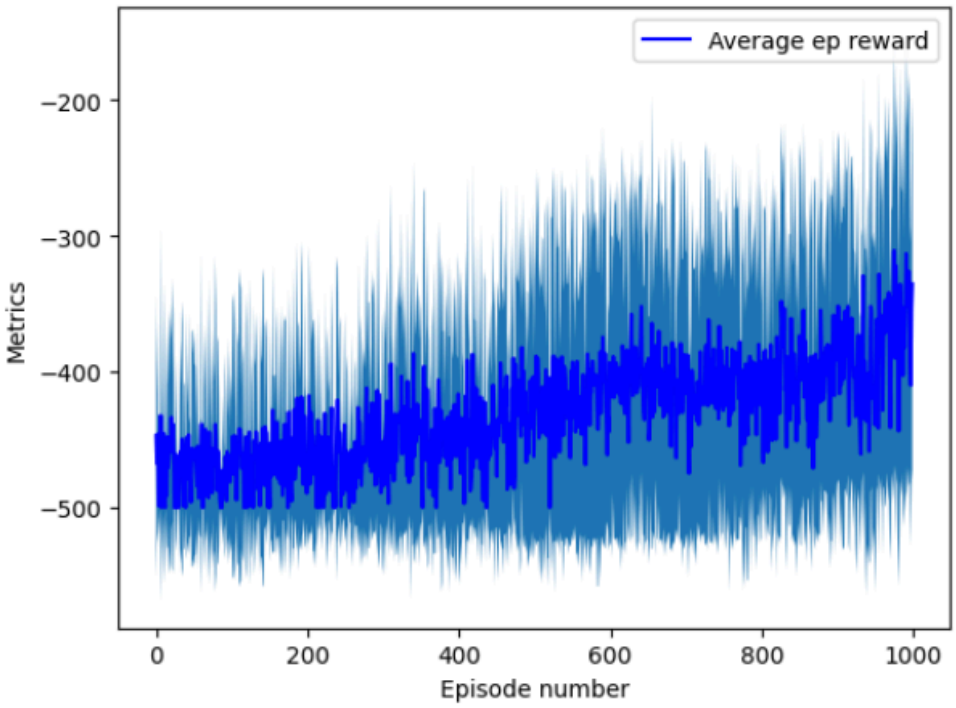
The MC REINFORCE (without baseline) method uses the accumulated G_t return in order to update the policy's parameters and this was implemented by calculating and storing the returns backwards into a deque. Finally, gradient ascent was used on log_probability loss * return.

```python
def finish_episode():
    gamma = 0.99
    R = 0
    policy_loss = []
    returns = deque()
    for r in policy.rewards[::-1]:
        R = r + gamma * R
        returns.appendleft(R)

    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + eps)

    for log_prob, R in zip(policy.saved_log_probs, returns):
        policy_loss.append(-log_prob * R)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()
    del policy.rewards[:]
    del policy.saved_log_probs[:]
```

CARTPOLE:



ACROBOT:



The figure above plots the average reward received per episode on the cartpole environment. The catastrophic forgetfulness problem leads to decreasing performance after 300 episodes and could not be corrected by any combination of hyper-params.

WITH BASELINE:

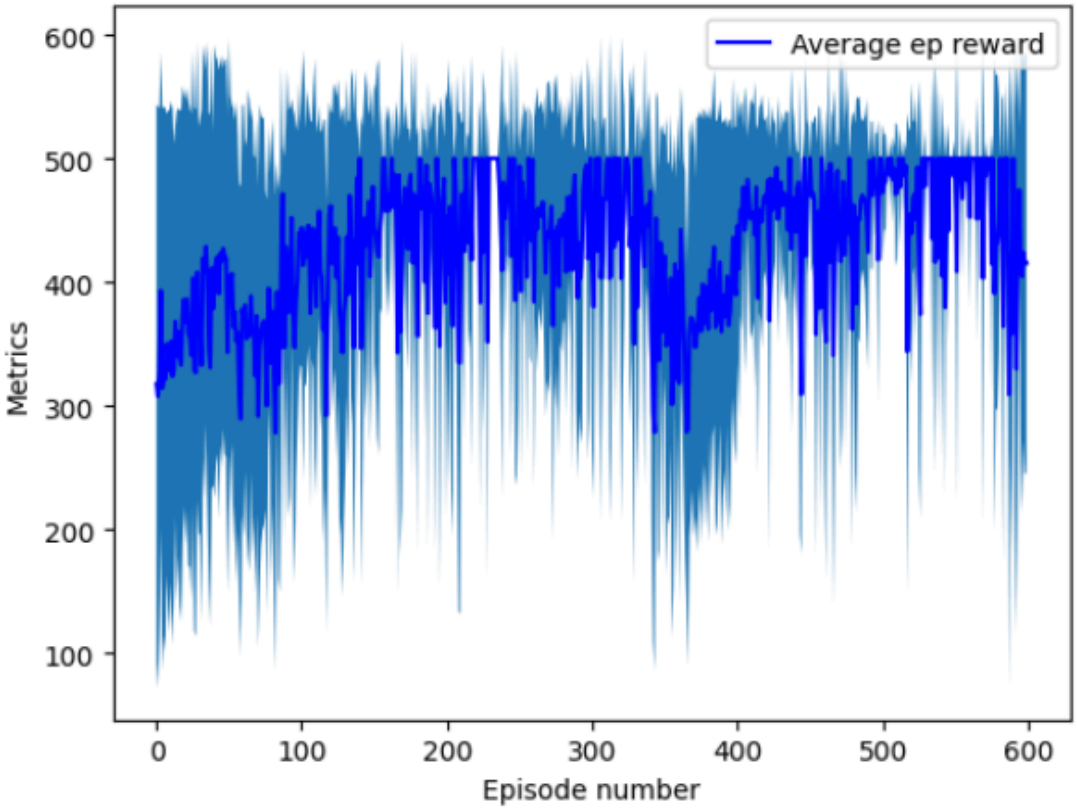
The MC REINFORCE (with baseline) method uses a value function estimate as the baseline to reduce variance of the policy gradient update. A shared network was used to generate the policy's action probabilities and the value estimate. The TD loss and the MC policy loss were combined to train the common network with the action_head layer parameters receiving the MC loss update and the value_head parameters receiving the TD loss update.

```python
class BaselinePolicy(nn.Module):
    def __init__(self):
        super(BaselinePolicy, self).__init__()
        # shared network used to train
        self.affine1 = nn.Linear(4, 128)
        self.dropout = nn.Dropout(p=0.5)

        self.action_head = nn.Linear(128,2)
        self.value_head = nn.Linear(128, 1)

        self.saved_actions = []
        self.rewards = []

    def forward(self, x):
        x = self.affine1(x)
        x = self.dropout(x)
        x = F.relu(x)
        action_prob = F.softmax(self.action_head(x))
        state_values = self.value_head(x)
        return action_prob, state_values
```

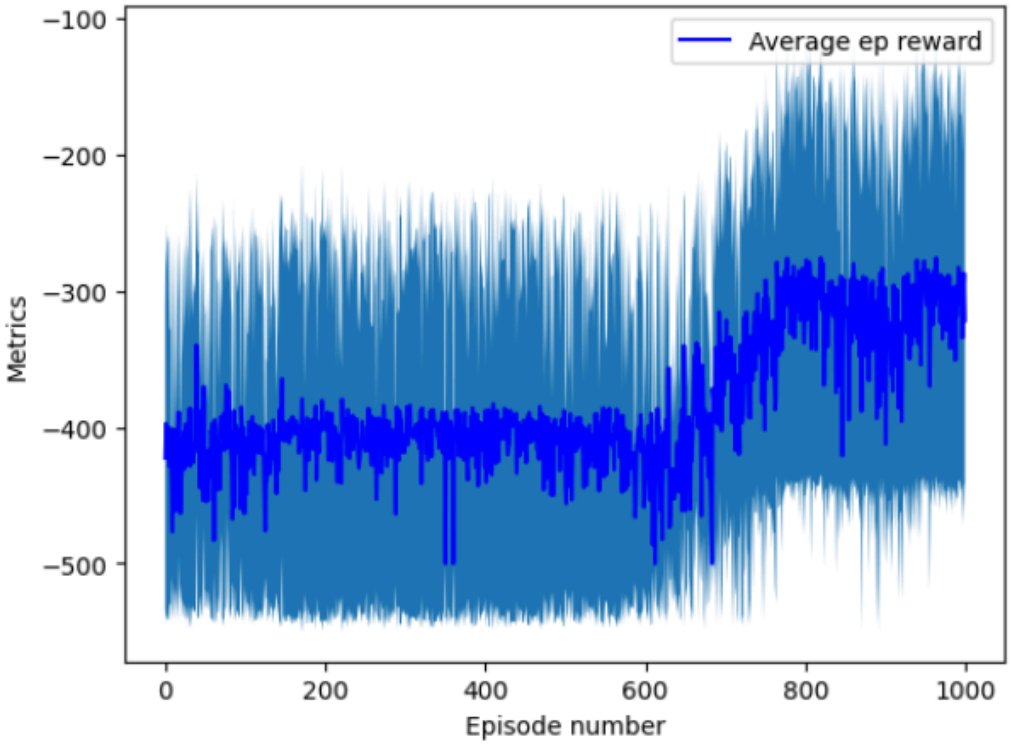
```python
for r in model.rewards[::-1]:
    R = r + gamma * R
    returns.insert(0, R)
# returns will contain [G_t,......G_t+n] in order

returns = torch.tensor(returns)
returns = (returns - returns.mean()) / (returns.std() + eps)

for (log_prob, value), R in zip(saved_actions, returns):

    with torch.no_grad(): advantage = R - value.item()
    policy_losses.append(-log_prob * advantage)
    value_losses.append(nn.MSELoss()(value, torch.tensor([R])))
```

Advantage was appropriately computed as (return - baseline) and used to form the loss function.

CARTPOLE:


Average ep reward
Metrics
Episode number
600
500
400
300
200
100
0
100
200
300
400
500
600

ACROBOT:



It can be seen that the variance in the received reward decreases as the number of episodes proceeds beyond 200 and the average episode reward archives the highest 500 mark consistently. 5 runs were averaged to produce the plot above and only 600 episodes are illustrated due to the catastrophic forgetting later.

MC REINFORCE: Inferences

- Reduced Variance: The baseline helps to reduce the variance of the gradient estimates in the REINFORCE algorithm. Without a baseline, the gradients can have high variance, leading to unstable learning and slow convergence. By subtracting a baseline value from the returns, the algorithm can reduce the variance of the gradients, resulting in more stable and efficient learning.
- Credit Assignment: In environments like CartPole and Acrobot, where rewards are sparse (i.e., non-zero rewards are received only at the end of an episode), it becomes challenging for the algorithm to assign credit to the earlier actions that contributed to the final outcome. The baseline acts as a reference point, allowing the algorithm to better attribute credit to the actions that led to better-than-expected returns, and vice versa for actions that led to worse-than-expected returns.
- Reward Scale: The CartPole and Acrobot environments typically have relatively small reward values (e.g., +1 for success, 0 for every other step). Without a baseline, the small reward values can lead to gradients with small magnitudes, slowing down the learning process. By subtracting a baseline value, the gradient updates can have larger magnitudes, leading to faster learning.
- Exploration-Exploitation Trade-off: The baseline can help strike a better balance between exploration and exploitation. Without a baseline, the algorithm may converge to a suboptimal

policy too quickly, as it only considers the total returns without considering the relative performance of actions. With a baseline, the algorithm can better identify actions that lead to better-than-expected returns, encouraging exploration of promising regions of the state-action space.

- Generalization: In environments like CartPole and Acrobot, where the state space can be relatively large, the baseline can help the algorithm generalize better to unseen states. By considering the relative performance of actions compared to a baseline, the algorithm can learn more robust value estimates that are less sensitive to specific state-action combinations.

Github Repo Link:

https://github.com/Sylindril/PA2.git