DA6400: Introduction to Reinforcement Learning
Assignment 2

Jayagowtham J ME21B078, Lalit Jayanti ME21B096

The code for this assignment can be found at:
`https://github.com/DA6400-RL-JanMay2025/programming-assignment-02-me21b078_me21b096`

Algorithms and Training

Agent

This is the common code block for both of our DDQN and MC-Reinforce Agents. It encapsulates initializing the agent and agent environment interaction parameters. The functions *update_* override the default parameters giving us more control over our environment.

```
# scripts/agents.py
class DDQNAgent:/class ReinforceMCwithoutBaselineAgent:/class ReinforceMCwithBaselineAgent:
    def __init__(self, state_space, action_space, seed):
        ''' Intialises the agent parameters (hyperparameters, neural networks)
        '''

    def reset(self):
        '''Resets all the hyperparameters and related objects
        '''

    def update_hyperparameters(self, **kwargs):
        '''Updates hyperparameters at the start of training
        '''

    def update_agent_parameters(self):
        '''Updates agent training parameters after every episode
        '''
```

Dueling Deep Q-Network (DDQN)

The Dueling deep Q-Network is an improvisation over deep Q-Network as it approximates both the value function and the action advantages. We obtain the corresponding Q-values by simple computation, and use it to update the network as we wish. It is necessary to note that the approximations truly represent their targets and not random approximations which somehow combine to give the correct Q-values (as explained in the paper).
We use a shared network to extract common features from the observations, and then branch out into 2 heads - value head and advantage head. The detailed network architecture is shown below:

```
class DDQNetwork(nn.Module):

    def __init__(self, state_size, action_size, network_type, seed):
        """
        Dueling Deep Q-Network (DDQN)

        Args:
            state_size (int): Dimension of the input state space.
            action_size (int): Total number of possible discrete actions.
            network_type (int): Type of dueling aggregation.
                - 1: Mean variant of DDQN.
                - 2: Max variant of DDQN.
            seed (int): Random seed for reproducibility.
```

```
        """

        super().__init__()

        self.seed = torch.manual_seed(seed)
        self.network_type = network_type

        hidden_size_1 = 64
        hidden_size_2 = 32
        self.shared_network = nn.Sequential(
            nn.Linear(in_features=state_size, out_features=hidden_size_1),
            nn.ReLU(),
            nn.Linear(in_features=hidden_size_1, out_features=hidden_size_1),
            nn.ReLU(),
            nn.Linear(in_features=hidden_size_1, out_features=hidden_size_1)
        )

        self.state_value_head = nn.Sequential(
            nn.Linear(in_features=hidden_size_1, out_features=hidden_size_2),
            nn.ReLU(),
            nn.Linear(in_features=hidden_size_2, out_features=1)
        )

        self.advantage_head = nn.Sequential(
            nn.Linear(in_features=hidden_size_1, out_features=hidden_size_2),
            nn.ReLU(),
            nn.Linear(in_features=hidden_size_2, out_features=action_size)
        )
```

We combine the value of the state and the advantage of the chosen action to yield the Q-function prediction of the observed state and chosen action as follows, with θ representing the network parameters:

(A) Type-1

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a';\theta) \right) \tag{1}$$

(B) Type-2

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a' \in \mathcal{A}} A(s,a';\theta) \right) \tag{2}$$

```
class DDQNetwork:
    def forward(self, observation):

        x = self.shared_network(observation)
        state_value = self.state_value_head(x)
        advantages = self.advantage_head(x)

        if self.network_type == 1:
            # Mean variant
            q_values = (state_value
                        + (advantages - torch.mean(advantages,
                                                   dim=1, keepdim=True)))
        elif self.network_type == 2:
            # Max variant
            q_values = (state_value
                        + (advantages - torch.max(advantages,
                                                  dim=1, keepdim=True)[0]))
        return q_values
```

We also maintain a replay buffer which helps our agent to sample relatively less correlated experiences from our memory. The implementation is as follows :

```python
class ReplayBuffer:

    def __init__(self, buffer_size, batch_size, seed, device='cpu'):
        ''' Replay buffer to store experience tuples. Initialises a deque memory of
    buffer_size and a batch_size to sample. Also instantiates a named tuple 'experience' to
    represent any observation.

        Args:
            buffer_size (int): Maximum number of experiences the buffer can hold.
            batch_size (int): Number of experiences to sample during training.
            seed (int): Random seed for reproducibility.
            device (str, optional): Device to which sampled tensors are moved
        '''

    def add(self, state, action, reward, next_state, done):

        experience = self.experience(state, action, reward, next_state, done)
        self.memory.append(experience)

    def sample(self):
        '''Sample experiences and  retrieve observations
        '''
        experiences = random.sample(self.memory, k=self.batch_size)
        # unpack experiences to retrieve the named elements
        return states, actions, rewards, next_states, dones
```

We provide our agent two networks to learn the Q-function, a local network and a target network The target network updates slower compared to the local network controlled by a hyperparamter, addressing the issue of non-stationarity of targets. We clamp the gradients to avoid the issue of exploding gradients while back propagation.

```python
class DDQNAgent:
    def step(self, state, action, reward, next_state, done):

        # Store experiences, to be used for training the agent later
        self.replay_memory.add(state, action, reward, next_state, done)

        # If buffer has enough samples, update the Q-network
        if len(self.replay_memory) >= self.BATCH_SIZE:
            experiences = self.replay_memory.sample()
            self.learn(experiences)

        # Update the target network with values from local network
        self.time_step = (self.time_step + 1) % self.UPDATE_EVERY
        if self.time_step == 0:
            self.qnetwork_target.load_state_dict(
                self.qnetwork_local.state_dict())

    def act(self, state):
        """Epsilon-greedy action selection by the agent
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(
                state
            ).cpu().data.numpy().squeeze(0)
        self.qnetwork_local.train()

        action = epsilon_greedy(action_values, self.action_size, self.eps)
        return action, action_values

    def learn(self, experiences):
        """Update the Q-network using gathered experiences
        """

        # Unpack transitions
        states, actions, rewards, next_states, dones = experiences

```

```
        # Get target Q-Values of next states from the target network by maximizing over actions
        q_targets_next = self.qnetwork_target(
            next_states
        ).detach().max(1)[0].unsqueeze(1)

        # Get the target values based on the TD update rule
        q_targets = rewards + (self.GAMMA * q_targets_next * (1 - dones))
        q_expected = self.qnetwork_local(states).gather(1, actions)

        loss = F.mse_loss(q_expected, q_targets)

        self.optimizer.zero_grad()
        loss.backward()

        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)

        self.optimizer.step()
```

Monte Carlo REINFORCE (MC REINFORCE)

Monte-Carlo REINFORCE is a policy gradient algorithm, that directly optimizes a policy by learning from sampled trajectories. For this assignment we consider 2 variants of this algorithm, one without baseline (Type-1) and another with a baseline (Type-2).

(A) Type-1

The first important component of the Type-1 MC REINFORCE agent is the Policy Network that learns the probabilities of selecting actions based on the input state. Below is the implementation, which allows for a configurable number of layers and hidden sizes. A final Softmax layer ensures the output values lie in [0,1], representing valid probability distribution over the actions. For this assignment we set 2 hidden layers of size 64 and 32 respectively.

```
class PolicyNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed, hidden_layer_sizes=[64, 64, 64]):
        """A configurable policy network that maps the input state to output probabilites for choosing each action.

        Args:
            state_size (int): Dimension of the input state space.
            action_size (int): Total number of possible discrete actions.
            seed (int): Random seed for reproducibility.
            hidden_layer_sizes (list of int, optional): List defining the hidden layers.
                - The length of the list determines the number of hidden layers,
                - Each value specifies the number of units in that layer.
        """
        super().__init__()

        self.seed = torch.manual_seed(seed)

        layers = []
        input_size = state_size

        for hidden_size in hidden_layer_sizes:
            layers.append(nn.Linear(input_size, hidden_size))
            layers.append(nn.ReLU())
            input_size = hidden_size

        layers.append(nn.Linear(input_size, action_size))
        layers.append(nn.Softmax(dim=1))

        self.network = nn.Sequential(*layers)

    def forward(self, observation):
```

```
        probs = self.network(observation)
        return probs
```

The next component of this agent is the implementation of the update equation, that allow the agent to learn from the sampled trajectories. For this implementation we chose to accumulate the gradients per episode and then apply them in one go for computational efficiency. Line 28 of this implementation is the update equation for the Policy Network. We have used the update equation for the discounted case (which contains an additional term of γ^t).

$$\theta = \theta + \alpha\gamma^t G_t \frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{3}$$

The final component of the agent is the action selection, which is provided from Line 48 onward. The actions are selected based on the probabilities from the Policy Network.

```

class ReinforceMCwithoutBaselineAgent:

    def __init__(self, state_space, action_space, seed, device='cpu'):
        ...
    def reset(self, seed=0):
        ...
    def update_hyperparameters(self, **kwargs):
        ...
    def update_agent_parameters(self):
        """ Updates the Policy Network with a sampled trajectory, after every episode.
        """

        # Computing the discounted returns
        ...
        # Unpack the experiences and Repack as torch.tensors
        ...

        # Computing the log-probabilites for the chosen actions (to compute loss)
        action_probs = self.policy_network(states)
        dist = Categorical(action_probs)
        log_probs = dist.log_prob(actions)
        discounts = self.GAMMA**torch.arange(0, len(self.episode_history),
                                             dtype=torch.float32,
                                             device=self.device)

        # Updating the Policy Network by accumulating the gradients
        policy_loss = -(discounts * returns * log_probs).sum()

        self.policy_optimizer.zero_grad()
        policy_loss.backward()
        for param in self.policy_network.parameters():
            if param.grad is not None:
                param.grad.data.clamp_(-5, 5)
        self.policy_optimizer.step()

        # Clear the history (to store the next trajectory)
        self.episode_history.clear()

    def step(self, state, action, reward, next_state, done):
        """Stores the state-transition for each step to
        episode_history (to be used after the episode to update the Policy Network).
        """
        e = self.experience(state, action, reward, next_state, done)
        self.episode_history.append(e)

    def act(self, state):
        """Selects an action based on the probabilites output by the Policy Network.
        """
        state = torch.tensor(state, dtype=torch.float32,
                             device=self.device).unsqueeze(0)

```

```
        self.policy_network.eval()
        with torch.no_grad():
            action_probs = self.policy_network(state)
        self.policy_network.train()

        m = Categorical(action_probs)

        action = m.sample().item()
        return action, None
        return action, None
```

(B) Type-2

The Type-2 MC REINFORCE agent implements the baseline version of the algorithm. The first important component of this agent is the Policy Network. Here we use the same implementation as described in the previous section, however for this agent we set 3 hidden layers each of size 64.

The second component of this agent is the Value Network, which provides the baseline values for the agent. Below is the implementation, a special feature of this network is that the output size is 1, which represents the state-value.

```
class ValueNetwork(nn.Module):

    def __init__(self, state_size, seed, hidden_layer_sizes=[64, 64, 64]):
        """A configurable value network that maps the input state to a state-value.

        Args:
            state_size (int): Dimension of the input state space.
            seed (int): Random seed for reproducibility.
            hidden_layer_sizes (list of int, optional): List defining the hidden layers.
                - The length of the list determines the number of hidden layers,
                - Each value specifies the number of units in that layer.
        """
        super().__init__()

        self.seed = torch.manual_seed(seed)

        layers = []
        input_size = state_size

        for hidden_size in hidden_layer_sizes:
            layers.append(nn.Linear(input_size, hidden_size))
            layers.append(nn.ReLU())
            input_size = hidden_size

        layers.append(nn.Linear(input_size, 1))

        self.network = nn.Sequential(*layers)

    def forward(self, observation):

        state_value = self.network(observation)
        return state_value
```

The next component of this agent is the implementation of the update equations, that allow the agent to learn from the sampled trajectories. For the Policy Network, as before we chose to accumulate gradients per episode and then apply them in one go for computational efficiency. Whereas for the Value Network, we use the TD(0) update rule and hence update it at every step of every episode. Further, to address the non-stationarity of targets we use a target and local network and periodically update the target Value Network with parameters from the local Value Network.

In the following implementation, Line 31 computes the advantages based on the baseline. Line 35 implements the Policy update. We have used the update equation for the discounted case (which contains an additional term of γ^t).

$$\theta = \theta + \alpha^{\theta}\gamma^{t}\left(G_t - V(S_t;\mathbf{\Phi})\right)\frac{\nabla\pi(A_t \mid S_t,\theta)}{\pi(A_t \mid S_t,\theta)} \tag{4}$$

Line 64 and 67 implement the TD(0) updates for the Value Network.

$$\mathbf{\Phi} \leftarrow \mathbf{\Phi} + \alpha^{\Phi}\left[R + \gamma V(S_{t+1}; \mathbf{\Phi}^{'}) - V(S_t; \mathbf{\Phi})\right] \nabla V(S_t; \mathbf{\Phi}) \tag{5}$$

Where α^{θ} and α^{Φ} represent the different learning rates of the corresponding networks. And, $\mathbf{\Phi}$ represents the parameters of the local Value Network and $\mathbf{\Phi}^{'}$ represents the parameters of the target Value Network.

```
class ReinforceMCwithBaselineAgent:

    def __init__(self, state_space, action_space, seed, device='cpu'):
        ....

    def reset(self, seed=0):
        ...

    def update_hyperparameters(self, **kwargs):
        ...

    def update_agent_parameters(self):
        """ Updates the Policy Network with a sampled trajectory, after every episode.
        """

        # Computing the discounted returns
        ...
        # Unpack the experiences and Repack as torch.tensors
        ...
        with torch.no_grad():
            state_values = self.value_network_local(states).detach().squeeze()

        # Computing the log-probabilites for the chosen actions (to compute loss)
        action_probs = self.policy_network(states)
        dist = Categorical(action_probs)
        log_probs = dist.log_prob(actions)
        discounts = self.GAMMA**torch.arange(0, len(self.episode_history),
                                             dtype=torch.float32,
                                             device=self.device)

        # Using state_values as a baseline
        advantages = returns - state_values

        # Updating the Policy Network by accumulating the gradients
        policy_loss = -(discounts * advantages * log_probs).mean()

        self.policy_optimizer.zero_grad()
        policy_loss.backward()
        for param in self.policy_network.parameters():
            if param.grad is not None:
                param.grad.data.clamp_(-1, 1)
        self.policy_optimizer.step()

        # Clear the history (to store the next trajectory)
        self.episode_history.clear()


    def step(self, state, action, reward, next_state, done):
        """Stores the state-transition for each step to
        episode_history (to be used after the episode to update the Policy Network).

        Also updates the Value Network using the transition by applying a TD(0) update.
        """
        e = self.experience(state, action, reward, next_state, done)

        self.episode_history.append(e)

        state = torch.tensor(np.array([state]), dtype=torch.float32, device=self.device)
        next_state = torch.tensor(np.array([next_state]), dtype=torch.float32, device=self.
    device)
```

```
60
61          state_value = self.value_network_local(state).squeeze()
62          next_state_value = self.value_network_target(next_state).squeeze()
63
64          # Get the target values based on the TD update rule
65          td_target = reward + self.GAMMA*next_state_value*(1-done)
66
67          # Update the Value Network
68          value_loss = F.mse_loss(state_value, td_target)
69
70          self.value_optimizer.zero_grad()
71          value_loss.backward()
72          for param in self.value_network_local.parameters():
73              if param.grad is not None:
74                  param.grad.data.clamp_(-1, 1)
75          self.value_optimizer.step()
76
77          # Update the target network with values from local network
78          self.time_step = (self.time_step + 1) % self.UPDATE_EVERY
79          if self.time_step == 0:
80              self.value_network_target.load_state_dict(
81                  self.value_network_local.state_dict()
82              )
83
84      def act(self, state):
85          """Selects an action based on the probabilites output by the Policy Network.
86          """
87          state = torch.tensor(state, dtype=torch.float32,
88                               device=self.device).unsqueeze(0)
89
90          self.policy_network.eval()
91          with torch.no_grad():
92              action_probs = self.policy_network(state)
93          self.policy_network.train()
94
95          m = Categorical(action_probs)
96
97          action = m.sample().item()
98          return action, None
```

The actions are selected based on the probabilities from the Policy Network, as shown from Line 84 onward.

Training

This code block consists of the main training loop for the agent, it handles the agent's actions, stepping the environment forward and also updating the agent's parameters for learning. Further this code block also takes care of logging the results which are subsequently used to perform analysis and also hyperparameter tuning.

```
1  # scripts/training.py
2  class Trainer:
3
4      def training(self, env, agent, n_episodes=10000, process_training_info=lambda *args, **
       kwargs: (False, {})):
5          """
6          To train an agent in the given environment.
7
8          Args:
9              - env: The environment for training.
10             - agent: An agent with `.step()`, `.act()`, and `.update_agent_parameters()`.
11             - n_episodes (int, optional): Number of training episodes. Defaults to 10000.
12             - process_training_info (function, optional): Runs after each episode.
13                 - First return value must be a `bool` for early stopping.
14                 - Second return value must be a `dict` to update the progress bar's postfix.
15
16         Returns:
17             - dict: Summary of the training process.
18         """
```

```
        # Initialize variables for logging here

        progress_bar = tqdm(range(1, n_episodes+1), desc="Training")

        # Training loop
        for i_episode in progress_bar:
            state, _ = env.reset()
            score = 0
            total_reward = 0
            terminated, truncated = False, False
            episode_history = []

            # Running an episode
            while not (terminated or truncated):
                action, action_vals = agent.act(state)
                next_state, reward, terminated, truncated, _ = env.step(action)
                episode_history.append((state, action, reward, next_state))
                agent.step(state, action, reward, next_state, terminated)
                state = next_state
                score += self.compute_score(reward)
                total_reward += reward

            agent.update_agent_parameters()

            # Code for logging here

        return {
            "computation_time": end_time - begin_time,
            "scores": np.array(history_scores),
            "total_rewards": np.array(history_total_rewards)
        }

    def compute_score(self, reward):
        return reward
```

Metrics and Hyperparameters

To quantitatively compare the performance of the algorithms in an environment we have used the following two metrics:

- **Cumulative regret** : The sum of difference between return assuming optimal policy and the 5 run average return at each step. For this assignment we have tuned the hyperparameters to minimize the regret in all experiments.
- **Maximum rolling average score** : The maximum of the rolling average return over the past 100 episodes averaged over 5 runs

Further, based on the description in section 1, the agents are characterized by a few hyperparameters:
DDQN Hyperparameters (Type-1 and Type-2)

- **BUFFER_SIZE** : The maximum number of experiences stored in the replay buffer.
- **BATCH_SIZE** : The number of experiences sampled from the buffer for each learning step.
- **UPDATE_EVERY** : Frequency (in steps) at which the Dueling Q-network is updated.
- **LR** : The learning rate used for updating the Dueling Q-network.
- **eps_start** : The starting value of ϵ in the ϵ-greedy policy.
- **eps_end** : The minimum value ϵ can take while training.
- **decay_type** : Determines whether ϵ decays linearly or exponentially over time.
- **frac_episodes_to_decay** : The fraction of total episodes over which ϵ is decayed before being fixed at its minimum value.

MC REINFORCE Hyperparameters

- **LR_POLICY** : The learning rate used for updating the policy network. (available for both Type-1 and Type-2)
- **LR_VALUE** : The learning rate used for updating the value network. (available only for Type-2)
- **UPDATE_EVERY** : Frequency (in steps) at which the value network is updated. (available only for Type-2)

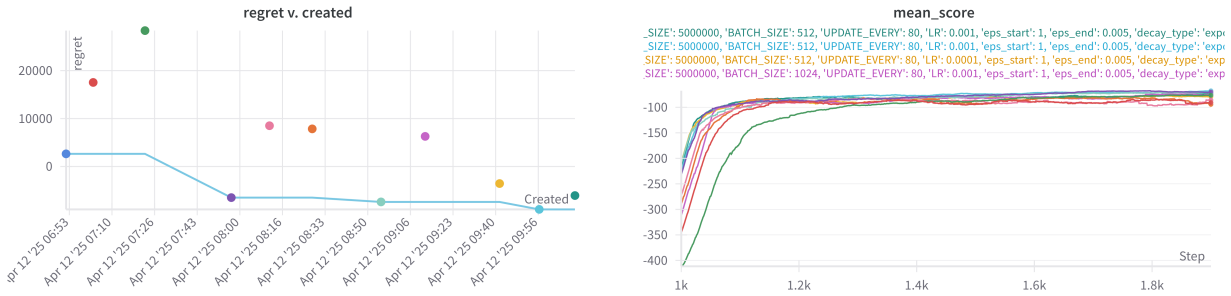
We have tuned the hyperparameters using `wandb` to minimize regret in all experiments, by performing a `Bayesian serach` over the search-space of hyperparameters. The exact configurations of the serach-space for the hyperparameters can be found in the `configs/` folder.

For all experiments involving Dueling DQN, we trained the agent for 1000 episodes for both Type-1 and Type-2 variants, evaluating 10 different hyperparameter configurations selected through Bayesian search. Similarly, for all Monte Carlo REINFORCE experiments, we trained the agent for 2000 episodes, again sampling 10 hyperparameter configurations through Bayesian search.

After tuning, we selected the best configuration for each algorithm and environment. We then conducted 5 independent runs using different environment seeds and plotted the mean and standard deviation of episodic returns across episodes. (The plots are labeled Scores vs Episodes and Rolling Means of Scores vs Episodes).

Acrobot-v1

Part (i): Dueling DQN - Type-1 - Hyperparameter tuning



(a) Tuning to minimize Regret

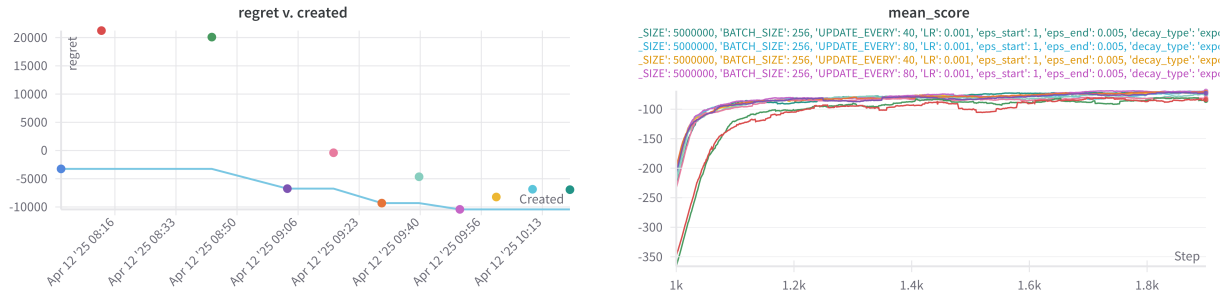
(b) Rolling average score

Figure 1: Tuning for Acrobot Dueling DQN - Type-1

DDQN Type 1 Hyperparameters	
BUFFER_SIZE	5×10^6
BATCH_SIZE	512
UPDATE_EVERY	80
LR	1×10^{-3}
eps_start	1
eps_end	0.005
decay_type	exponential
frac_episodes_to_decay	0.3

Table 1: Top Hyperparameter configuration for Dueling DQN - Type-1

Part (ii): Dueling DQN - Type-2 - Hyperparameter tuning



(a) Tuning to minimize Regret

(b) Rolling average score

Figure 2: Tuning for Acrobot Dueling DQN - Type-2

DDQN Type 2 Hyperparameters	
BUFFER_SIZE	5×10^6
BATCH_SIZE	256
UPDATE_EVERY	80
LR	1×10^{-1}
eps_start	1
eps_end	0.001
decay_type	exponential
frac_episodes_to_decay	0.5

Table 2: Top Hyperparameter configuration for Dueling DQN - Type-2

Part (iii): Acrobot-v1 - Dueling DQN - Comparison and Inferences

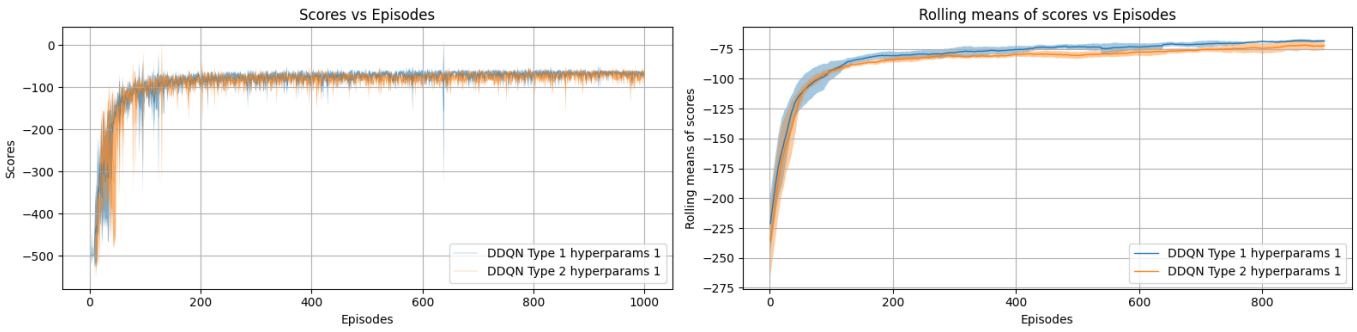
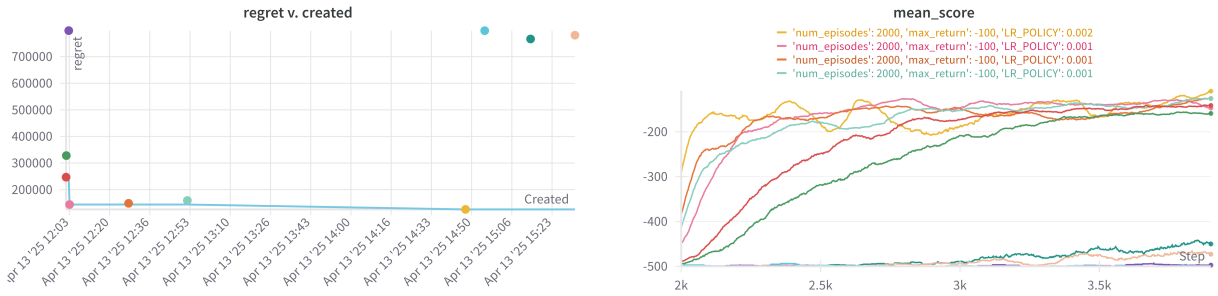


Figure 3: Comparing Dueling DQN Type-1 and Type-2

Both the time series plots look similar. But on a smaller scale, we observe that Type-1 pips Type-2 and after a point, consistently remains above Type-2. Considering our small action space and the environment's dynamics, we don't want to aggresively stick to one action while disregarding others. This might lead to an overoptimistic bias and lead us to converge to a sub-optimal policy or worse, not converge at all. Hence, the average operator (which stabilizes the learning) unlike the max operator (which aggresses the learning) leads to more consistent performance.

Part (iv): Monte Carlo REINFORCE - Type-1 - Hyperparameter tuning



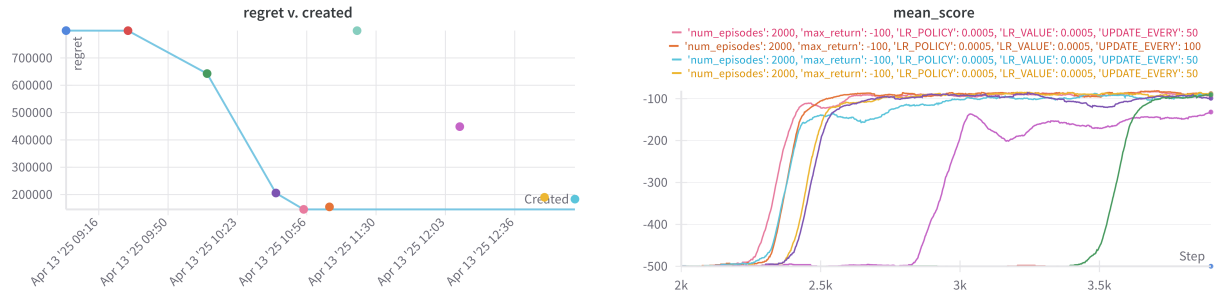
(a) Tuning to minimize Regret (b) Rolling average score

Figure 4: Tuning for Acrobot MC REINFORCE - Type-1

REINFORCE Type 1 Hyperparameters	
LR_POLICY	1×10^{-3}

Table 3: Top Hyperparameter configuration for Monte Carlo REINFORCE - Type-1

Part (v): Monte Carlo REINFORCE - Type-2 - Hyperparameter tuning



(a) Tuning to minimize Regret (b) Rolling average score

Figure 5: Tuning for Acrobot MC REINFORCE - Type-2

REINFORCE Type 2 Hyperparameters	
LR_POLICY	5×10^{-4}
LR_VALUE	5×10^{-4}
UPDATE_EVERY	50

Table 4: Top Hyperparameter configuration for Monte Carlo REINFORCE - Type-2

Part (vi): Acrobot-v1 - Monte Carlo REINFORCE - Comparison and Inferences

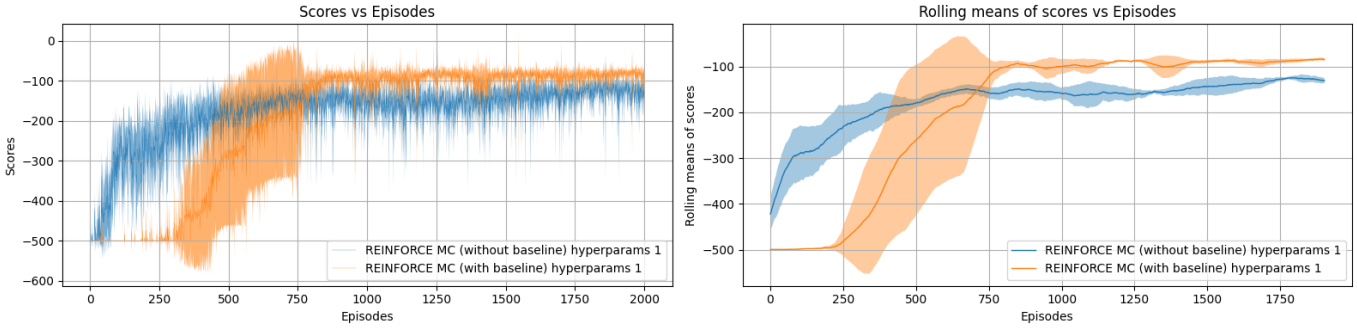
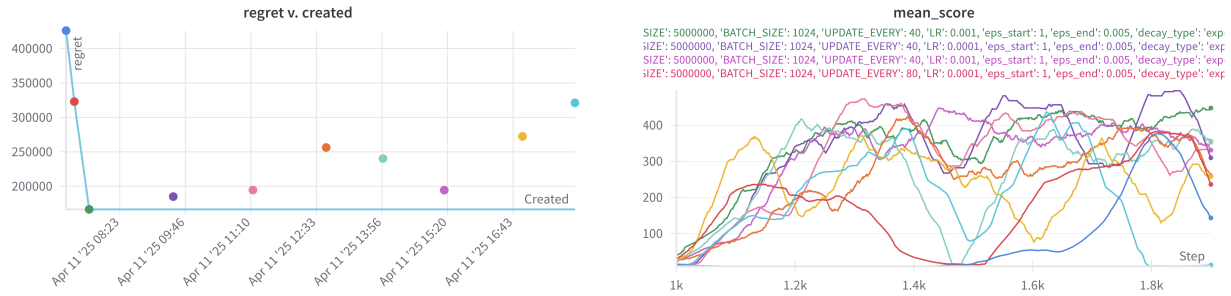


Figure 6: Comparing MC REINFORCE Type-1 and Type-2

We observe that, MC REINFORCE Type-2 (with baseline) outperforms Type-1 after about 750 episodes. Although, theoretically Type-2 agent is expected to learn faster, for this set of hyperparameters and network configuration it is not practically observed for about the first 250 episodes. This may be attributed to the fact that the Value Network is not trained yet and thus may be skewing the estimates. However, as both the Policy and Value Networks improve, the Type-2 agent demonstrates its advantages of having less variance in probability estimates and learns faster than Type-1 agent and reaches a higher value of score earlier.

CartPole-v1

Part (i): Dueling DQN - Type-1 - Hyperparameter tuning



(a) Tuning to minimize Regret

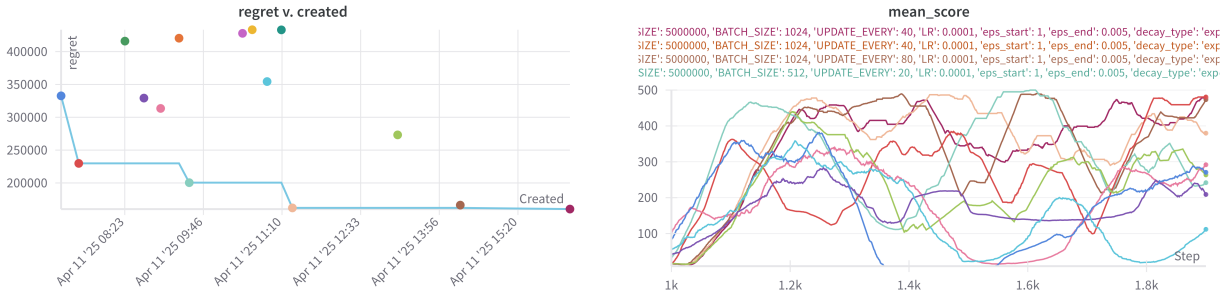
(b) Rolling average score

Figure 7: Tuning for CartPole Dueling DQN - Type-1

DDQN Type 1 Hyperparameters	
BUFFER_SIZE	5×10^6
BATCH_SIZE	1024
UPDATE_EVERY	40
LR	1×10^{-3}
eps_start	1
eps_end	0.005
decay_type	exponential
frac_episodes_to_decay	0.3

Table 5: Top Hyperparameter configuration for Dueling DQN - Type-1

Part (ii): Dueling DQN - Type-2 - Hyperparameter tuning



(a) Tuning to minimize Regret

(b) Rolling average score

Figure 8: Tuning for CartPole Dueling DQN - Type-2

DDQN Type 2 Hyperparameters	
BUFFER_SIZE	5×10^6
BATCH_SIZE	1024
UPDATE_EVERY	40
LR	1×10^{-4}
eps_start	1
eps_end	0.005
decay_type	exponential
frac_episodes_to_decay	0.5

Table 6: Top Hyperparameter configuration for Dueling DQN - Type-2

Part (iii): CartPole-v1 - Dueling DQN - Comparison and Inferences

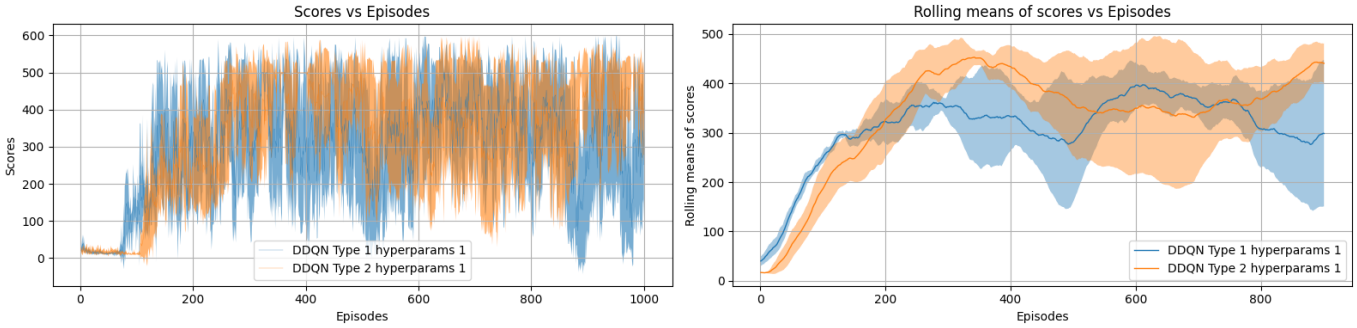
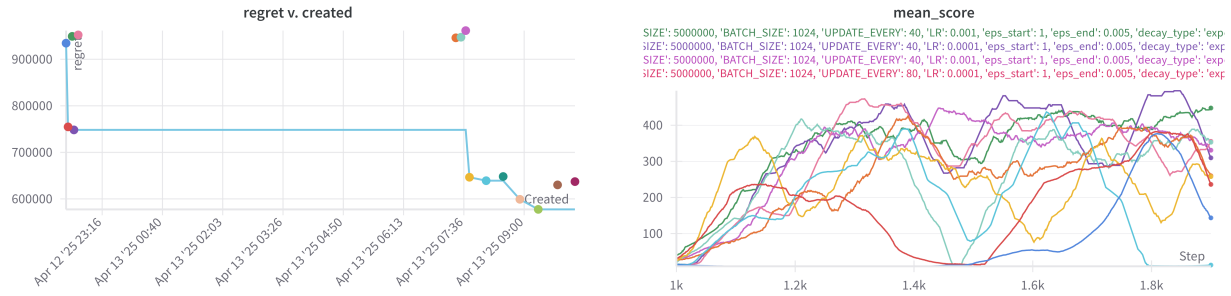


Figure 9: Comparing Dueling DQN Type-1 and Type-2

The learning as we can see is highly noisy, with even the mean score across 5 experiments not stable and oscillating like a sine wave. We can observe, amidst this wave of uncertainity, that Type-2, as learning progresses pips Type-1 by a considerable amount. Initially, as with the case of Acrobot, Type-1 leads Type-2 by stabilized learning. But over time, aggressive learning is the right thing to do. The key difference between the environments is that in Acrobot, the agent needs to build momentum, so the optimal action at each state is not obvious and comparison among actions is definitely helpful in all stages of learning. While, in CartPole after a good amount of learning, we are almost certain of our action given a state. Say, for example, the pole is tilted left. The obvious action is to accelarate in the opposite direction. Hence, we observe the difference in the winner in both these environments.

Part (iv): Monte Carlo Reinforce - Type-1 - Hyperparameter tuning



(a) Tuning to minimize Regret

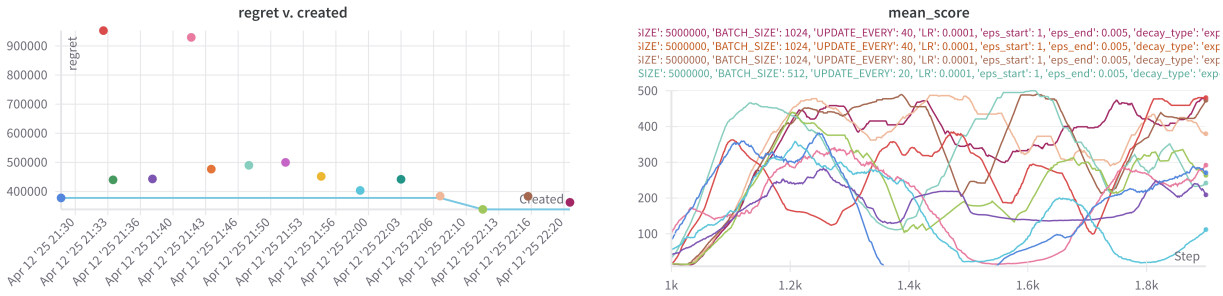
(b) Rolling average score

Figure 10: Tuning for CartPole MC REINFORCE - Type-1

REINFORCE Type 1 Hyperparameters	
LR_POLICY	1×10^{-3}

Table 7: Top Hyperparameter configuration for Monte Carlo REINFORCE - Type-1

Part (v): Monte Carlo Reinforce - Type-2 - Hyperparameter tuning



(a) Tuning to minimize Regret

(b) Rolling average score

Figure 11: Tuning for CartPole MC REINFORCE - Type-2

REINFORCE Type 2 Hyperparameters	
LR_POLICY	5×10^{-4}
LR_VALUE	2.5×10^{-3}
UPDATE_EVERY	20

Table 8: Top Hyperparameter configuration for Monte Carlo REINFORCE - Type-2

Part (vi): CartPole-v1 - Monte Carlo REINFORCE - Comparison and Inferences

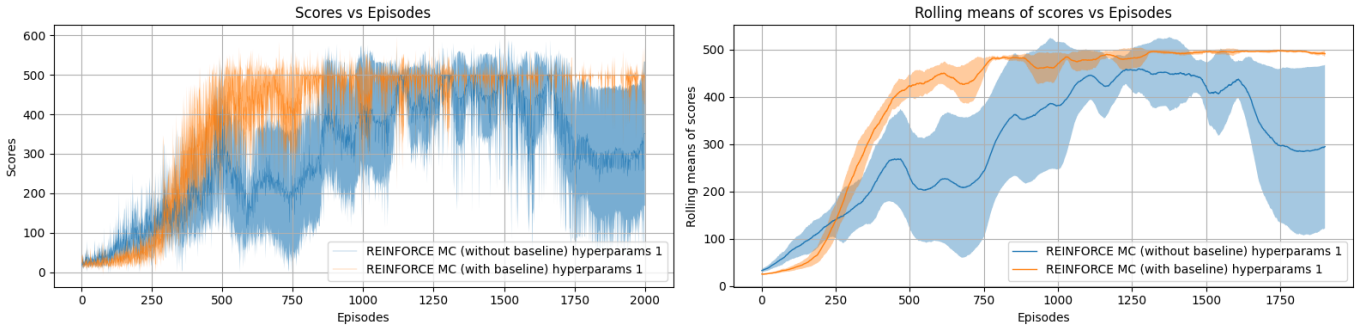


Figure 12: Comparing MC REINFORCE Type-1 and Type-2

In the CartPole environment, we observe that Type-2 agent (with baseline) outperforms Type-1 both in terms of stability, speed of convergence and maximum score. This experiment verifies that having a baseline reduces variance and allows for faster learning. In the very beginning for about 250 episodes, the Type-1 agent out performs the Type-2 agent, leading us to conjecture that without a baseline, the algorithm reacts more directly to returns, which may lead to faster early learning, however, in the long run it may lead to unstable learning.