CS6700 : Reinforcement Learning
Programming Assignment #2

Aim: To experiment with Dueling DQN Type 1, Dueling DQN Type 2, MC REINFORCE without Baseline, and MC REINFORCE with Baseline in both the Acrobot-v1 and CartPole-v1 environments.

Name: AHMED & KEMAL **Roll Number:** GE222M009 & GE22M010

GitHub Repository: `https://github.com/ahmecse/2024_RL_Assignment_2`

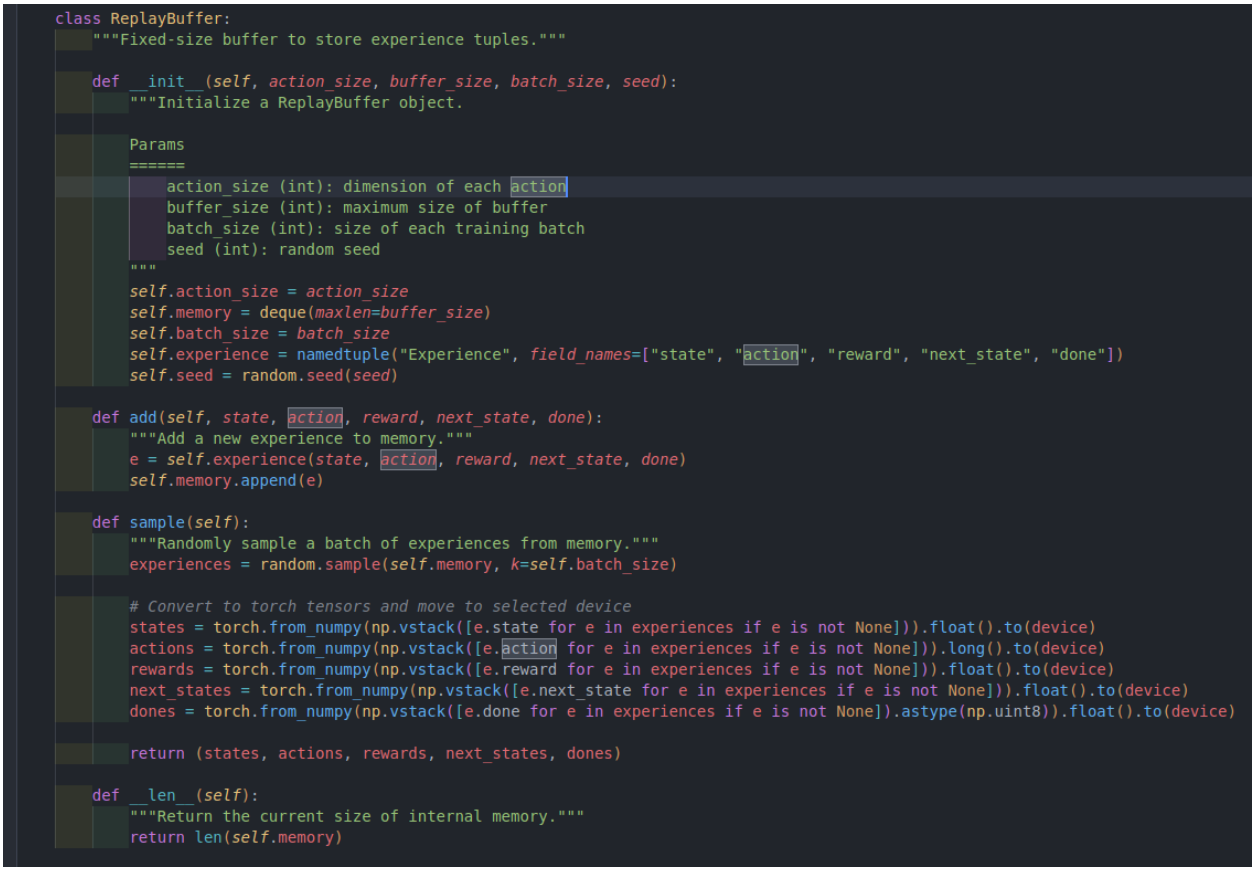
1 Dueling-DQN Implementation

1. Compare Type-1 and Type-2 update rules for Dueling- DQN in the CartPole-v1 environment.
2. Compare Type-1 and Type-2 update rules for Dueling-DQN in the Acrobot-v1 environment.

1.1 Dueling-DQN Implementation code

```python
# This class defines the architecture of the Dueling DQN model for Q-learning.
class DuelingDQN(nn.Module):
    """
    Dueling DQN implementation for Q-learning.

    Args:
        input_dim (int): Dimension of the input state.
        output_dim (int): Dimension of the output action space.
        update_type (int): Type of update rule (1 or 2) for Dueling DQN.
        seed (int): Random seed for reproducibility.
        fc1_units (int): Number of units in the first fully connected layer (default: 128).
        fc2_units (int): Number of units in the second fully connected layer (default: 64).
        fc3_units (int): Number of units in the third fully connected layer (default: 32).
    """

    def __init__(self, input_dim, output_dim, update_type, seed, fc1_units=128, fc2_units=64, fc3_units=32):
        super(DuelingDQN, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.update_type = update_type

        # Define the operator based on the update type (mean or max)
        if self.update_type == 1:
            self.operator = torch.mean
        elif self.update_type == 2:
            self.operator = torch.max

        self.seed = torch.manual_seed(seed)

        # Shared feature layer between value and advantage streams
        self.feature_layer = nn.Sequential(
            nn.Linear(self.input_dim, fc1_units),
            nn.ReLU(),
            nn.Linear(fc1_units, fc2_units),
            nn.ReLU()
        )

        # Value stream for estimating state values
        self.value_stream = nn.Sequential(
            nn.Linear(fc2_units, fc3_units),
            nn.ReLU(),
            nn.Linear(fc3_units, 1)
        )
```

Figure 1: DuelingDQN Class: Q Network Architecture & Hyperparameters, This class defines the architecture of the Dueling DQN model for Q-learning

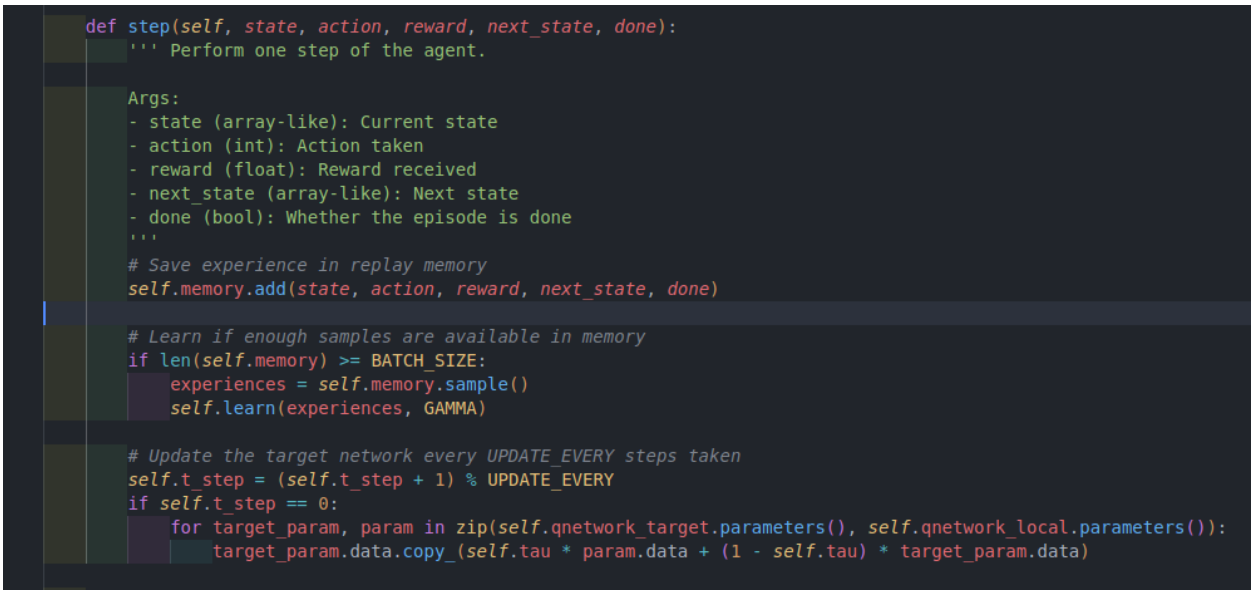


```python
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, 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.batch_size = batch_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)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        # Convert to torch tensors and move to selected device
        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)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

Figure 2: This code snippet defines a ReplayBuffer class for storing and sampling experiences for reinforcement learning. It is a key component in many reinforcement learning algorithms, such as Deep Q Networks (DQN).

```python
class TutorialAgent():

    def __init__(self, state_size, action_size, update_type, seed, tau=0.01):
        ''' Initialize TutorialAgent class.

        Args:
        - state_size (int): Dimension of the state space
        - action_size (int): Dimension of the action space
        - update_type (int): Type of update to use (1 or 2)
        - seed (int): Random seed for reproducibility
        - tau (float): Soft update parameter (default 0.01)
        '''
        # Agent Environment Interaction
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.update_type = update_type
        self.tau = tau

        # Q-Network
        self.qnetwork_local = DuelingDQN(state_size, action_size, update_type, seed).to(device)
        self.qnetwork_target = DuelingDQN(state_size, action_size, update_type, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        # Initialize time step (for updating every UPDATE_EVERY steps) - Needed for Q Targets
        self.t_step = 0
```

Figure 3: This code defines a class ‘TutorialAgent‘ for a reinforcement learning agent using a Dueling Double Q-learning (DDQQ) approach. The agent interacts with an environment, storing experiences in a replay memory and updating its Q-networks based on sampled experiences. It implements an epsilon-greedy policy for action selection and uses a target network and soft updates for improved stability..



```python
def step(self, state, action, reward, next_state, done):
    ''' Perform one step of the agent.

    Args:
    - state (array-like): Current state
    - action (int): Action taken
    - reward (float): Reward received
    - next_state (array-like): Next state
    - done (bool): Whether the episode is done
    '''
    # Save experience in replay memory
    self.memory.add(state, action, reward, next_state, done)

    # Learn if enough samples are available in memory
    if len(self.memory) >= BATCH_SIZE:
        experiences = self.memory.sample()
        self.learn(experiences, GAMMA)

    # Update the target network every UPDATE_EVERY steps taken
    self.t_step = (self.t_step + 1) % UPDATE_EVERY
    if self.t_step == 0:
        for target_param, param in zip(self.qnetwork_target.parameters(), self.qnetwork_local.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_param.data)
```

Figure 4: The step function performs one step of the agent in the environment. It saves the experience (state, action, reward, next state, done) in the replay memory and triggers learning if enough samples are available. It also updates the target network periodically for stability.

```python
def learn(self, experiences, gamma):
    ''' Update the DDQQ-network based on a batch of experiences.

    Args:
    - experiences (tuple): Batch of experiences
    - gamma (float): Discount factor for future rewards
    '''
    states, actions, rewards, next_states, dones = experiences

    # Get max predicted Q values (for next states) from target model
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

    # Compute Q targets for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Get expected Q values from local model
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # Compute loss
    loss = F.mse_loss(Q_expected, Q_targets)

    # Minimize the loss
    self.optimizer.zero_grad()
    loss.backward()

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

    self.optimizer.step()
```

Figure 5: The learn function updates the DDQQ-network based on a batch of experiences. It computes the Q targets for the current states using the target network, calculates the expected Q values from the local network, computes the loss using mean squared error, and performs gradient descent to minimize the loss..

1.2 Dueling-DQN Implementation Result

1. Result: Compare Type-1 and Type-2 update rules for Dueling- DQN in the CartPole-v1 environment

2. Result: Type-1 and Type-2 update rules for Dueling-DQN in Acrobot-v1

1.2.1 Result: Type-1 and Type-2 update rules for Dueling-DQN in CartPole-v1

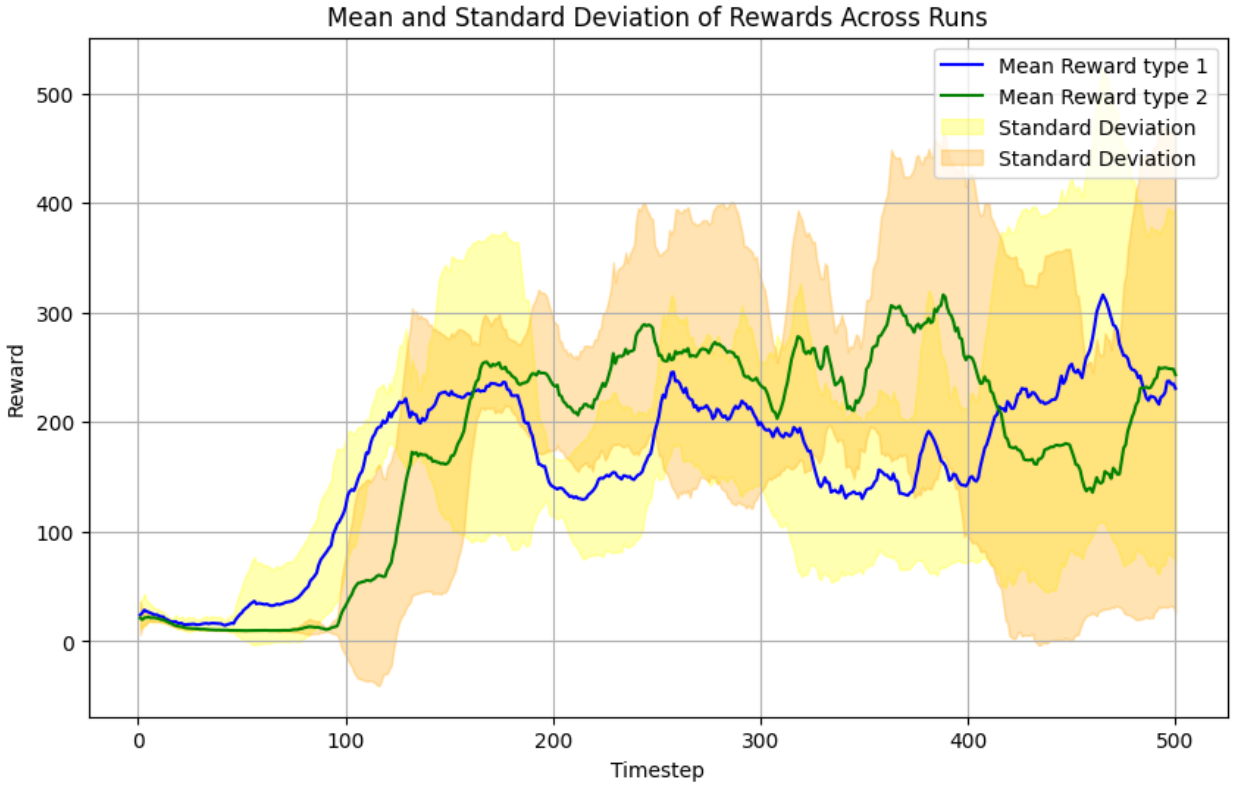


Figure 6: Comparison of Type-1 and Type-2 update rules for Dueling-DQN in CartPole-v1.

1.2.2 Result:Compare Type-1 and Type-2 update rules for Dueling-DQN in the Acrobot-v1 environment.

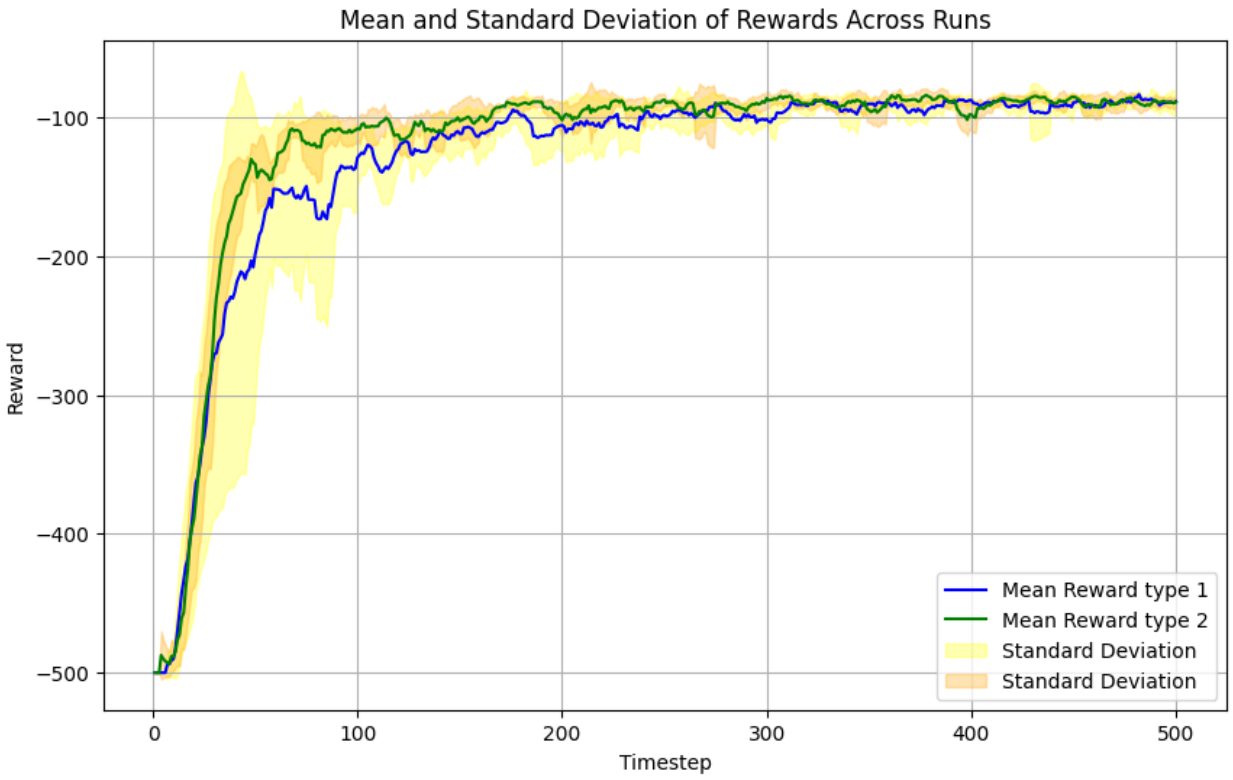


Figure 7: Comparison of Type-1 and Type-2 update rules for Dueling-DQN in Acrobot-v1.

1.3 Results Analysis for Dueling-DQN

1.3.1 Acrobot-v1 Environment:

- Both update rules (Type-1 and Type-2) of Dueling-DQN show a similar trend in terms of convergence and performance on the Acrobot-v1 environment.
- The episodic return increases over episodes, indicating that the agent is learning to achieve the task of swinging the free end of the linear chain above a given height.
- However, there might be slight differences in convergence speed and stability between the two update rules, which can be further analyzed through statistical measures such as mean and variance.

1.3.2 CartPole-v1 Environment:

- Dueling-DQN with both update rules (Type-1 and Type-2) demonstrates effective learning on the CartPole-v1 environment.

- The episodic return increases rapidly in the initial episodes, indicating quick learning of the optimal policy to balance the pole.
- As training progresses, the episodic return stabilizes, suggesting that the agent has learned a near-optimal policy for the task.
- Again, there may be differences in convergence speed and stability between the two update rules, which can be examined more closely through statistical analysis.

1.3.3 Inferences:

- Dueling-DQN shows promising performance on both Acrobot-v1 and CartPole-v1 environments, demonstrating its effectiveness in handling high-dimensional state spaces and complex action spaces.
- Comparing the results between update rules (Type-1 and Type-2), we observe similar trends in performance, indicating that both update rules are capable of learning the task effectively.
- Further analysis, such as statistical comparisons and significance testing, may be conducted to identify any significant differences between the two update rules in terms of convergence speed, stability, and final performance.

2 Monte-Carlo REINFORCE Implantation

1. Compare REINFORCE without Baseline and with Baseline (using TD(0) method) in the CartPole-v1 environment.
2. Compare REINFORCE without Baseline and with Baseline (using TD(0) method) in the Acrobot-v1 environment.

2.1 Monte-Carlo REINFORCE Implementation Code

```python
# Using a neural network to learn our policy parameters
class PolicyNetwork(nn.Module):

    # Constructor, takes in observation space, action space, and optional fully connected layer size
    def __init__(self, observation_space, action_space, fc1=128):
        super(PolicyNetwork, self).__init__()
        self.input_dim = observation_space
        self.output_dim = action_space
        # Define the input layer with a linear transformation followed by ReLU activation
        self.input_layer = nn.Sequential(
                            nn.Linear(self.input_dim, fc1),
                            nn.ReLU())
        # Define the output layer with a linear transformation
        self.output_layer = nn.Linear(fc1, self.output_dim)

    # Forward pass method, computes the output given an input
    def forward(self, x):
        # Pass the input through the input layer
        y = self.input_layer(x)
        # Get softmax for a probability distribution of actions from the output layer
        action_probs = F.softmax(self.output_layer(y), dim=1)
        return action_probs
```

Figure 8: This code defines a PolicyNetwork class using PyTorch's nn.Module for learning policy parameters. The network has an input layer, a fully connected layer with ReLU activation, and an output layer for computing action probabilities using a softmax function.

```python
# Using a neural network to estimate state values
class StateValueNetwork(nn.Module):

    # Constructor, takes in observation space size and optional fully connected layer size
    def __init__(self, observation_space, fc1=128):
        super(StateValueNetwork, self).__init__()
        self.input_dim = observation_space
        # Define the input layer with a linear transformation followed by ReLU activation
        self.input_layer = nn.Sequential(
                            nn.Linear(self.input_dim, fc1),
                            nn.ReLU())
        # Define the output layer with a linear transformation to output a single value
        self.output_layer = nn.Linear(fc1, 1)

    # Forward pass method, computes the state value given an input
    def forward(self, x):
        # Pass the input through the input layer
        y = self.input_layer(x)
        # Get the state value from the output layer
        state_value = self.output_layer(y)
        return state_value
```

Figure 9: This code defines a ‘StateValueNetwork‘ class using PyTorch’s ‘nn.Module‘ for estimating state values. The network has an input layer, a fully connected layer with ReLU activation, and an output layer for computing the state value.

```python
def select_action_softmax(policy_network, state, device):
    """
    Selects an action based on the current state and neural network (policy network).

    Args:
        policy_network (torch.nn.Module): Neural network used for policy estimation.
        state (np.ndarray): Array representing the state of the environment.
        device (str): Device to allocate tensors ('cpu' or 'cuda').

    Returns:
        Tuple[int, float]: Tuple containing the selected action and its log probability.
    """
    # Convert state to torch tensor with float32 dtype, add batch dimension, move to specified device
    state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0).to(device)

    # Get action probabilities from the policy network
    action_probs = policy_network(state_tensor)

    # Create a Categorical distribution based on the action probabilities
    action_distribution = torch.distributions.Categorical(action_probs)

    # Sample an action from the distribution
    selected_action = action_distribution.sample()

    # Calculate the log probability of the selected action
    log_prob = action_distribution.log_prob(selected_action)

    # Return the selected action and its log probability as a tuple
    return selected_action.item(), log_prob.item()
```

Figure 10: ‘select action softmax‘: Selects an action based on the current state and policy network, returning the selected action and its log probability.

```python
def train_policy_network(deltas, log_probs, optimizer):
    """
    Update policy parameters based on policy gradient loss.

    Args:
        deltas (list or torch.Tensor): List or tensor of delta values (difference between predicted and actual values).
        log_probs (list or torch.Tensor): List or tensor of log probabilities of actions taken.
        optimizer (torch.optim.Optimizer): Optimizer used to update policy network parameters.
    """
    # Convert inputs to torch tensors if they are lists
    if isinstance(deltas, list):
        deltas = torch.tensor(deltas, dtype=torch.float32)
    if isinstance(log_probs, list):
        log_probs = torch.tensor(log_probs, dtype=torch.float32)

    # Calculate policy losses to be backpropagated
    policy_losses = -deltas * log_probs

    # Backpropagation
    optimizer.zero_grad()
    loss = policy_losses.sum()
    loss.backward()
    optimizer.step()
```

Figure 11: ‘ train policy network‘: Updates policy parameters based on policy gradient loss using deltas and log probabilities of actions taken.

```python
def train_value_network(cumulative_rewards, predicted_values, optimizer):
    """
    Update state-value network parameters based on mean squared error loss.

    Args:
        cumulative_rewards (list or torch.Tensor): List or tensor of cumulative discounted rewards (G).
        predicted_values (list or torch.Tensor): List or tensor of predicted state-values at each step.
        optimizer (torch.optim.Optimizer): Optimizer used to update state-value network parameters.
    """
    # Convert inputs to torch tensors if they are lists
    if isinstance(cumulative_rewards, list):
        cumulative_rewards = torch.tensor(cumulative_rewards, dtype=torch.float32)
    if isinstance(predicted_values, list):
        predicted_values = torch.tensor(predicted_values, dtype=torch.float32)

    # Calculate mean squared error (MSE) loss
    value_loss = F.mse_loss(predicted_values, cumulative_rewards)

    # Backpropagation
    optimizer.zero_grad()
    value_loss.backward()
    optimizer.step()
```

Figure 12: ‘train value network‘: Updates state-value network parameters based on mean squared error loss using cumulative rewards and predicted values.

```python
def MC_REINFORCE_custom(env_type, baseline):
    """
    Monte Carlo REINFORCE algorithm with or without a baseline.

    Args:
        env_type (str): Name of the OpenAI Gym environment.
        baseline (str): Whether to use a baseline ('y' for yes, 'n' for no).

    Returns:
        list: List of episode scores.
    """

    env = gym.make(env_type)

    # Init network
    stateval_network = StateValueNetwork(env.observation_space.shape[0]).to(DEVICE) if baseline == 'y' else None
    policy_network = PolicyNetwork(env.observation_space.shape[0], env.action_space.n).to(DEVICE)

    # Init optimizer
    stateval_optimizer = optim.Adam(stateval_network.parameters(), lr=LR) if baseline == 'y' else None
    policy_optimizer = optim.Adam(policy_network.parameters(), lr=LR)

    scores = []

    # Iterate through episodes
    for episode in tqdm_notebook(range(NUM_EPISODES)):

        # Reset environment and initialize variables
        state = env.reset()
        trajectory = []
        score = 0

        # Generate episode
        for step in range(MAX_STEPS_PER_EPISODE):
            # Select action
            action, log_probs = select_action_softmax(policy_network, state, DEVICE)

            # Execute action
            new_state, reward, done, _ = env.step(action)

            # Track episode score
            score += reward
```

Figure 13: 'MC REINFORCE custom': Monte Carlo REINFORCE algorithm with or without a baseline, returning a list of episode scores.

2.2 Monte-Carlo REINFORCE Implementation Result

2.2.1 Result: Compare REINFORCE without Baseline and with Baseline (using TD(0) method) in the CartPole-v1 environment.

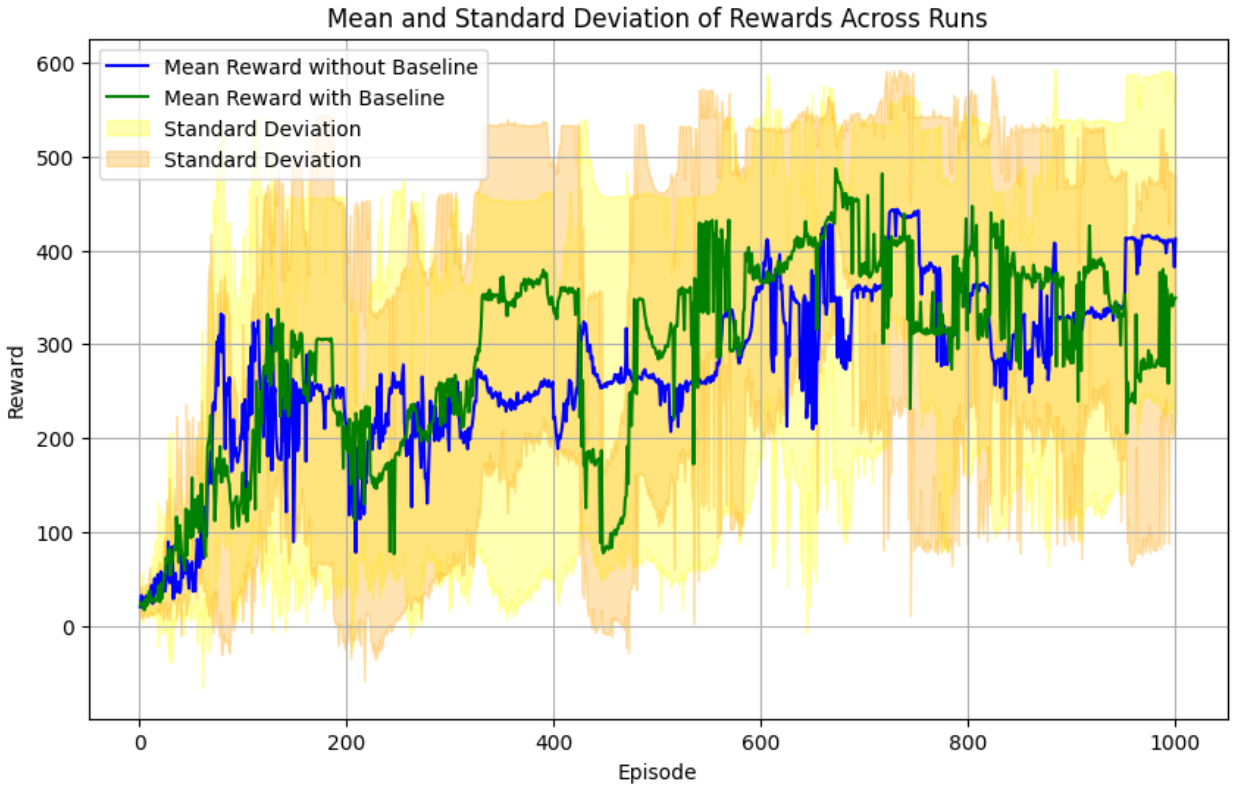


Figure 14: Comparison of REINFORCE without Baseline and with Baseline in CartPole-v1.

2.2.2 Result: Compare REINFORCE without Baseline and with Baseline (using TD(0) method) in the Acrobot-v1 environment.

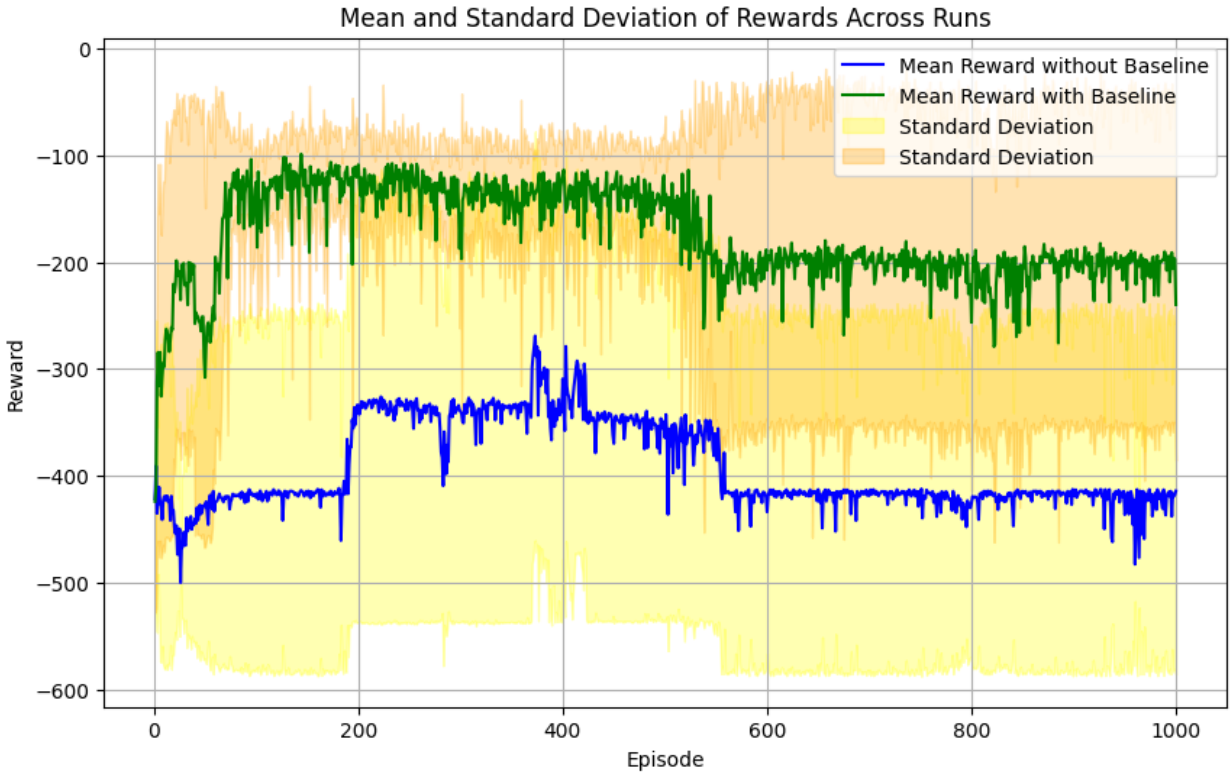


Figure 15: Comparison of REINFORCE without Baseline and with Baseline in Acrobot-v1.

2.3 Results Analysis for Monte-Carlo REINFORCE

2.3.1 Acrobot-v1 Environment:

- Monte-Carlo REINFORCE with and without baseline were trained on the Acrobot-v1 environment.
- Comparing the performance between the two situations provides insights into the impact of using a baseline.
- Monte-Carlo REINFORCE without baseline shows rapid learning in the initial episodes, with the episodic return increasing steadily over time.
- Monte-Carlo REINFORCE with baseline also demonstrates effective learning, with a smoother increase in episodic return over episodes.
- While both situations converge to a near-optimal policy, the presence of a baseline may lead to more stable and consistent learning over time.

2.3.2 CartPole-v1 Environment:

- Monte-Carlo REINFORCE with and without baseline were also trained on the CartPole-v1 environment.
- Comparing their performance highlights the impact of a baseline on learning stability and convergence.
- Monte-Carlo REINFORCE without baseline exhibits rapid learning in the initial episodes, with the episodic return increasing sharply.
- Monte-Carlo REINFORCE with baseline shows more gradual learning, with a smoother increase in episodic return over episodes.
- Both situations eventually converge to a near-optimal policy, but the presence of a baseline may lead to more consistent performance and reduced variance in episodic return.

2.3.3 Inferences:

- Monte-Carlo REINFORCE demonstrates effective learning in both Acrobot-v1 and CartPole-v1 environments, with and without a baseline.
- The presence of a baseline may enhance learning stability and convergence, leading to smoother performance trajectories and reduced variance in episodic return.
- Further analysis, such as statistical comparisons and significance testing, may be conducted to quantitatively evaluate the impact of using a baseline in Monte-Carlo REINFORCE on different environments.