Solving Control Problems using Reinforcement Learning

Mark Dingli
University of Malta
mark.dingli.21@um.edu.mt

1 Introduction

1.1 What is Reinforcement Learning?

Reinforcement learning (RL) is branch of machine learning that enables AI-driven agents with the capability to learn through trial and error by making use of feedback from their own actions. This method involves rewarding positive behaviors and penalizing negative ones. The agent engages with its environment, undertaking various actions with the primary objective of maximizing its total reward. Over time, the agent refines its behaviors based on environmental feedback, employing advanced techniques such as Q-learning. This iterative learning process allows the agent to enhance its performance and decision-making capabilities continuously [1].

1.2 How does Reinforcement Learning differ from other ML approaches?

Machine Learning (ML) includes a diverse range of approaches, each characterized by unique learning mechanisms. Reinforcement Learning (RL) emerges as a distinct subset of ML, where learning is influenced by the outcomes of actions, either positive or negative, rather than depending on a dataset. Unlike supervised learning, which utilizes labeled data to train models, RL operates without predefined labels, learning instead from direct interaction with the environment. This approach contrasts with supervised learning, where a 'teacher' provides correct outputs for training as can be seen in [2]. Another ML technique is unsupervised learning, which uses unlabeled data to detect patterns or structures, operating without any guidance on the expected output, as shown in [2]. In RL, the lack of direct input necessitates the development of an optimal policy or strategy to achieve specific goals. Essentially, while supervised and unsupervised learning aim at predictions or classifications based on data, RL trains agents to make a series of decisions within an environment, with the goal of optimizing total rewards.

1.3 Differences between Value Based, Policy Based and Actor Critic models.

The field of reinforcement learning encompasses various models, each with unique approaches to learning and decision-making. Among these, Value Based, Policy Based, and Actor Critic models stand out due to their distinct methodologies and applications.

1.3.1 Value Based Models.

As outlined in [3], value-Based models primarily focus on learning the value or action-value function to derive the optimal policy. In these models, an agent is employed to learn a value function, which estimates the long-term reward linked with each state or state-action pair. The core objective of value-based methods is to determine the value of states, or to estimate the value of states and actions. This is accomplished through the learning of value functions or Q-functions. The agent uses these value functions to select actions that maximise the expected reward.

1.3.2 Policy Based Models.

Policy-Based models in machine learning diverge from the value function approach, focusing instead on directly learning a policy. These methods entirely avoid the process of learning the values of states or actions. The primary aim of policy-based methods is to directly determine the optimal policy. This policy defines the strategy that an agent should adopt for selecting actions. Typically, the policy is parameterized, and these parameters are iteratively adjusted to optimize the expected return. This approach allows for a more direct route to determining the best action choices, without the intermediary step of value estimation [4].

1.3.3 Actor Critic Models.

Actor-Critic based models are a combination of value-based and policy-based methodologies. In these models, an agent simultaneously learns a value and a policy function. This combination leverages the strengths of both value-based and policy-based models. The *'actor'* component in these models is responsible for determining the course of action, functioning as a policy model. In contrast, the *'critic'* serves as a value model, appraising the actions taken by the actor. The feedback provided by the critic is instrumental in refining the strategies of both the actor and the critic itself. The learning process for the actor is grounded in the policy gradient method, which focuses on enhancing the policy directly. Meanwhile, the critic's role is to assess the actor's actions by calculating the value function as described in [5].

In summary, Value Based models prioritize learning the value of states and actions to make optimal decisions, while Policy Based models focus directly on learning the optimal policy. Actor Critic models blend these approaches, leveraging the strengths of both to create a more robust and dynamic learning process.

2 Background

2.1 Value Based Methods

Value Based Methods in Reinforcement Learning focus on estimating the potential long-term rewards for different states or actions in a given environment. These methods differ from other RL strategies since they do not directly learn or define a policy. A policy is the process through which an agent chooses actions based on the current state. Instead, Value Based Methods focus on evaluating the 'value' of each state or state-action pair. This 'value' represents the expected total reward that the agent can obtain, beginning from that state or after executing a specific action. The goal is to employ these value estimations to steer the agent's decisions, encouraging actions that optimize these expected rewards. This approach is essential in identifying the most advantageous actions an agent can take to achieve its goals efficiently [3].

An example of value-based methods is Q-Learning. This off-policy algorithm aims to learn the value of an action in a specific state, bypassing the need for a model of the environment. It achieves this by learning the optimal policy indirectly through the value of the best action in each state. An essential component of Q-Learning is its update equation, which iteratively refines the Q-values based on the agent's experiences.

The Q-Learning Update Equation is given by:

$$Q^{new}(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

In this equation, $Q(s,a)$ represents the current estimate of the value for taking action a in state s, with α as the learning rate, R as the immediate reward, γ as the discount factor, and $\max_{a'} Q(s',a')$ represents the estimated value of the best possible future action in the new state s'.

Another example is Value Iteration, used predominantly in Markov Decision Processes (MDPs). This algorithm iteratively updates the value of each state until convergence to the optimal value function is achieved. Value Iteration is especially effective in environments with a manageable state space and a known environmental model.

The Bellman Equation for Value Iteration is expressed as:

$$V^*(s) = \max_a \sum_{s',r} P(s',r|s,a)[r + \gamma V^*(s')]$$

Here, $V^*(s)$ is the optimal value function for state s, with $P(s',r|s,a)$ denoting the transition probability and γ the discount factor.

In short, value-based methods offer a structured approach for agents to make informed decisions. Through algorithms like Q-Learning and Value Iteration, value-based methods focus on maximizing expected cumulative rewards over time.

2.2 Deep Q-Networks Algorithm

Deep Q-Networks (DQNs) effectively merge the foundational principles of Q-Learning with the sophisticated capabilities of deep neural networks. This integration allows DQNs to excel in high-dimensional environments, where traditional Q-Learning methods fall short. In essence, DQNs utilize deep learning to approximate the Q-function, which estimates the total expected rewards for taking certain actions in given states. This approach enables the handling of vast and intricate input spaces with greater efficiency and effectiveness. By incorporating neural networks, DQNs can generalize across a range of states, learning to navigate and make decisions in intricate environments with an enhanced capacity for interpreting and responding to diverse scenarios as discussed in [6].

The core update equation of DQN builds upon the classical Q-Learning update rule, incorporating neural network parameters to enhance learning:

$$Q^{new}(s,a;\theta) \leftarrow Q(s,a;\theta) + \alpha[R + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta)]$$

In this formula, $Q(s,a;\theta)$ denotes the neural network's approximation of the action-value function, parameterized by θ. The term R represents the reward, while, γ is the discount factor. The expression $\max_{a'} Q(s',a';\theta^-)$ represents the estimated value of the best possible future action in the new state s', calculated using the target network parameters θ^-.

From my DQN implementation, the corresponding code snippet that implements this update rule is found in the *"optimize_model"* method of the *"Standard_DQNAgent"* class.

Specifically, the code lines that map to the above equation are:

```
# Calculate the current Q values
state_action_values = self.policy_net(state_batch).gather(1, action_batch)

# Calculate expected Q values
expected_state_action_values = (next_state_values * GAMMA) * (1 - done_batch) + reward_batch

# Calculate loss using Mean Squared Error between current and expected Q values
loss = F.mse_loss(state_action_values, expected_state_action_values.unsqueeze(1))
```

Figure 1. Code snippet showcasing the implementation of the key update rule in the DQN algorithm.

In these lines, *"state_action_values"* corresponds to $Q(s,a;\theta)$, and *"expected_state_action_values"* represents the term $R + \gamma \max_{a'} Q(s',a';\theta^-)$. The loss function *"mse_loss"* is used to compute the difference between the current and expected Q values, which is then minimized to update the network parameters θ.

To summarize, Deep Q-Networks (DQNs) leverage the power of neural networks to approximate action values across large state spaces and uses experience replay for efficient learning, making DQNs a highly effective solution for challenging reinforcement learning tasks.

2.3 Policy Based Methods

Policy Based Methods in Reinforcement Learning directly parameterize the policy, which assigns a probability to each action in a given state. Unlike Value Based Methods, which focus on learning a value function that is later used to determine a policy, Policy Based Methods aim to optimize the policy itself without the intermediate step of value estimation. This can be particularly advantageous in high-dimensional or continuous action spaces, where value functions are difficult to represent [4].

Policy Based Methods use a specific approach to update the policy, known as the policy gradient method. The equation for this is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)]$$

In this equation, $J(\theta)$ represents the objective function, typically the expected return. The term $\nabla_\theta J(\theta)$ is the policy gradient, indicating the direction in which the policy parameters θ should be adjusted to maximize $J(\theta)$. The policy, denoted as $\pi_\theta(s,a)$, gives the probability of selecting action a in state s, and is parameterized by θ. $Q^{\pi_\theta}(s,a)$ is the action-value function under the policy π_θ, reflecting the expected return for choosing action a in state s and following π_θ. This function is used to weight the gradient of the logarithm of the policy's probability for the taken action. The expectation $\mathbb{E}_{\pi_\theta}$ signifies that the average is taken over the distribution of states and actions as determined by the policy, thereby guiding the optimization of the policy towards greater expected returns.

In contrast to Value Based Methods, Policy Based Methods directly manipulate the probability of selecting actions, which allows them to effectively handle situations with stochastic policies and continuous action spaces. The main difference between value-based and policy-based methods is their approach to learning. Policy-based methods do not learn a Q-Value Function; instead, they learn a policy directly. Additionally, unlike value-based methods, policy-based methods can learn stochastic policies. This means they may take different actions given the same observation, as mentioned in [7]. This distinction allows policy-based methods to be more flexible in certain scenarios. Value Based Methods, often have to partition continuous action spaces and may struggle to represent the randomness inherent in some policies. Policy Based Methods often have better convergence properties and can learn stochastic policies, which is a significant advantage over Value Based Methods. However, they typically suffer from high variance in their estimates and often require more samples to achieve comparable results .

An example of a Policy Based Method is the *REINFORCE* algorithm, which utilizes Monte Carlo methods to estimate the policy gradient. By adjusting the policy parameters in the direction suggested by the gradient, the algorithm seeks to maximize the expected cumulative rewards.

In essence, Policy Based Methods provide a different approach to Reinforcement Learning by optimizing the policy directly. This can lead to more natural learning in environments with complex action spaces and can introduce stochasticity in the policy, which can be beneficial in certain contexts. Despite their differences, both Policy Based and Value Based Methods contribute uniquely to RL, and the choice between them depends on the specific requirements of the problem at hand.

2.4 Actor-Critic Methods

Actor-Critic Methods combine the concepts of value-based and policy-based methods. Utilizing two distinct models, the Actor, which suggests actions based on the current state, and the Critic, which evaluates these actions using a value function, Actor-Critic methods effectively leverage the strengths of both policy optimization and value function estimation [5].

The Actor-Critic architecture can be represented with the following equations:

$$\textbf{Actor: } \pi(a|s,\theta) = P[A = a|S = s,\theta]$$

$$\textbf{Critic: } V^\pi(s,\omega) = \mathbb{E}[R|S = s,\omega]$$

Here, the Actor, defined by the policy $\pi(a|s,\theta)$, dictates the likelihood of selecting each possible action a in a given state s. In contrast, the Critic estimates the value function $V^\pi(s,\omega)$, reflecting the expected return from state s under the current policy π. The Actor is responsible for action selection, while the Critic assesses the quality of actions taken, guiding the Actor towards better decisions.

2.4.1 Stochastic Actor-Critic Algorithms.

Stochastic Actor-Critic Algorithms are a subset of Actor-Critic methods where the policy (Actor) is stochastic in nature. These algorithms model the policy as a probability distribution over actions, enabling the agent to explore a range of actions with varying probabilities. This approach is particularly beneficial in environments where the optimal action is continuous or when exploration is crucial for learning.

In these methods, the Actor outputs probabilities for each action in a given state, and actions are selected based on these probabilities. The Critic evaluates the actions by estimating the value function, providing feedback to the Actor for policy improvement. The stochastic nature of the policy in these methods allows for effective exploration of the action space, making it possible to discover and learn optimal strategies even in complex environments.

One key advantage of Stochastic Actor-Critic Algorithms is their ability to capture multiple potentially good actions in a given state, allowing the agent to learn more robust and generalizable policies.

***PPO-GAE*: An Example of a Stochastic Actor-Critic Algorithm**

The Proximal Policy Optimization with Generalized Advantage Estimation (PPO-GAE) algorithm [8] is a good example of a Stochastic Actor-Critic method. PPO-GAE optimizes a stochastic policy in a way that balances between exploration and exploitation, using a clipped surrogate objective function and Generalized Advantage Estimation.

The core update equations for PPO-GAE are:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

Here, $L^{CLIP}(\theta)$ is the objective function optimized during training, incorporating the probability ratio $r_t(\theta)$ and the advantage estimation $\hat{A}_t$. The advantage estimation $\hat{A}_t$ is calculated using Generalized Advantage Estimation (GAE), which combines information across multiple time steps for more efficient learning.

Code Implementation and Explanation from the notebook *"stochastic_policy_approach"*:

```
def forward(self, x):
    x = self.model(x)
    # Get mean of action distribution
    loc = torch.tanh(self.mu_head(x)) * 2
    # Get standard deviation of action distribution
    scale = torch.exp(self.logstd_head(x))
    return loc, scale
```

Figure 2. Code snippet illustrating the *forward* pass of the Actor Network in the PPO-GAE algorithm.

In this snippet from the *"ActorNet"* class, the forward pass of the Actor Network is designed to calculate the probabilities for various actions. The network computes the mean (*'loc'*) and standard deviation (*'scale'*) for the action distribution, which are essential components in defining the stochastic policy $\pi(a|s, \theta)$. The *'loc'* represents the expected value of the actions in a given state, while *'scale'* denotes the degree of exploration or uncertainty around these actions. This approach allows the Actor to propose actions that are not just optimal but also incorporate an element of randomness, facilitating effective exploration in diverse environmental scenarios.

```
    # Compute the surrogate losses
    surrgt1 = rho * gae
    surrgt2 = rho.clamp(1 - epsilon, 1 + epsilon) * gae

    # Calculate the policy loss
    policy_loss = -torch.minimum(surrgt1, surrgt2).mean()
```

Figure 3. Code snippet demonstrating the computation of surrogate losses in the PPO-GAE algorithm.

Here, the surrogate loss for PPO's objective $L^{CLIP}(\theta)$ is computed. The probability ratio (*'rho'*) is used along with the advantage estimation (*'gae'*) to calculate the clipped objective, which aims to limit the policy update's deviation from the previous policy. This snippet can be found in the *"train"* method.

```
# Calculate Generalized Advantage Estimation (GAE)
gaes = []
# Initialize GAE
gae = T(np.zeros(nenvs)).view(nenvs, -1)
for i in range(len(b_obs) - 1, -1, -1):
    delta = b_rewards[i] + gamma * b_values_[i] * (1 - b_dones[i]) - b_values[i]
    gae = delta + gamma * lmbda * (1 - b_dones[i]) * gae
    gaes.insert(0, gae)
```

Figure 4. Code snippet demonstrating the calculation of Generalized Advantage Estimation (GAE) in the PPO-GAE algorithm.

The code for Generalized Advantage Estimation (GAE) calculates the advantage estimation $\hat{A}_t$ using the temporal difference error (*'delta'*). This part of the code aligns with the GAE equation and is crucial for efficiently estimating the advantages over multiple time steps. This snippet can be found in the *"RunningMem"* class. The PPO-GAE algorithm showcases the effective use of stochastic policies in complex environments. The algorithm's balance between exploring diverse strategies and exploiting actions makes it an optimal choice for continuous action spaces.

2.4.2 Deterministic Actor-Critic Algorithms.

Deterministic policies, in contrast to stochastic ones, provide a direct mapping from states to actions. This approach is exemplified in algorithms like *Deep Deterministic Policy Gradient (DDPG)* [9]. Such methods excel in scenarios with

continuous action spaces, offering greater sample efficiency by reducing the variance inherent in action selection. Deterministic policies streamline the learning process, particularly in environments where a clear and consistent response is preferable.

***DDPG*: An Example of a Deterministic Actor-Critic Algorithm**

The core update equations for DDPG are:

$$Q(s,a) = R(s,a) + \gamma Q'(s', \mu'(s'))$$
$$\nabla_{\theta^\mu} J \approx \mathbb{E}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_t}]$$

Here, $Q(s,a)$ is the action-value function estimated by the Critic. The Critic is updated using a Bellman-like equation, which includes a reward $R(s,a)$, a discount factor γ, and the action-value function Q' of the next state-action pair, with the action determined by the target policy μ'. The Actor (policy function $\mu(s)$) is updated using policy gradients, aiming to maximize the expected return as estimated by the Critic. The gradients are calculated with respect to the parameters of the Actor network θ^μ and are based on the gradients of the action-value function Q provided by the Critic. The algorithm uses target networks Q' and μ' to improve stability.

Code Implementation and Explanation from the notebook *"deterministic_policy_approach"*:

```
def update():
    # Sample a batch of experiences from the replay memory
    states, actions, rewards, dones, next_states = replay_memory.sample_batch()

    # Reset the Ornstein-Uhlenbeck noise for exploration
    ou_noise.reset()

    # Calculate the critic loss
    Qvals = critic(states, actions)  # Predict the current Q-values
    with torch.no_grad():
        # Predict next actions using the target actor network
        actions_ = actor_target(next_states)
        # Predict next Q-values using the target critic network
        Qvals_ = critic_target(next_states, actions_)
        # Set Q-values to 0 for terminal states
        Qvals_[dones] = 0.0
        # Compute the target Q-values
        target = rewards + GAMMA * Qvals_
    # Compute the loss between target and current Q-values
    critic_loss = F.smooth_l1_loss(target, Qvals)

    # Calculate the actor loss (negative mean of the critic's Q-values)
    actor_loss = -critic(states, actor(states)).mean()

    # Update the actor network
    # Zero the gradients
    actor_optimizer.zero_grad()
    # Backpropagate the loss
    actor_loss.backward()
    # Perform a step of optimization
    actor_optimizer.step()

    # Update the critic network
    # Zero the gradients
    critic_optimizer.zero_grad()
    # Backpropagate the loss
    critic_loss.backward()
    # Perform a step of optimization
    critic_optimizer.step()
```

Figure 5. Code snippet highlighting the *update* function in the DDPG algorithm.

This part of the code demonstrates the optimization of both the Actor and Critic networks. The reset of the Ornstein-Uhlenbeck noise process (‘*ou_noise.reset()*‘) is crucial for introducing exploration in the deterministic policy, ensuring that the agent can effectively explore different strategies. The line *‘Qvals = critic(states, actions)*‘ aligns with the computation of $Q(s,a)$, while the calculation of ‘actor_loss‘ implements the gradient ascent on the Actor network, reflecting the policy gradient update ($\nabla_{\theta^\mu} J$).

```
# Forward pass through the network
def forward(self, x):
    # Scale the output by the maximum action value
    x = self.model(x) * self.pmax
    # Clip the output to be within the specified range
    torch.clip_(x, self.pmin, self.pmax)
    return x
```

Figure 6. Code snippet demonstrating the *forward* pass of the Actor Network in the DDPG algorithm.

This code snippet from the *"PolicyNet"* class corresponds to the forward pass of the Actor Network, which is critical for implementing the policy function $\mu(s)$. The function scales and clips the output to ensure that the actions remain within feasible bounds, directly reflecting the deterministic policy approach in DDPG.

This shows how DDPG is a great example of a deterministic Actor-Critic framework, offering a stable and efficient approach for problems in continuous action spaces. Its structure allows for direct action determination while maintaining the benefits of Actor-Critic methods.

2.4.3 Conclusion (Policy Based Methods).

In conclusion, Actor-Critic methods combine policy-based and value-based approaches, offering flexible and effective solutions for various scenarios. Both stochastic methods, like PPO-GAE, and deterministic approaches, such as DDPG, are well-suited for environments with continuous action spaces. Stochastic methods are particularly advantageous for tasks requiring extensive exploration and adaptability, while deterministic methods provide more direct and consistent action mapping, with exploration facilitated through specific techniques like the Ornstein-Uhlenbeck process. The choice between these types depends on the environment’s specific demands and the task’s goals.

3 Methodology

Main Libraries Used: Pytorch, Gym, Numpy, Matplotlib, Collections, and OS.

3.1 Experiment 1

Problem Addressed: OpenAI Gym LunarLander (Discrete Actions) - "LunarLander-v2"

The "LunarLander-v2" environment in OpenAI Gym simulates the challenge of safely landing a spacecraft on the lunar surface. The agent controls the spacecraft's thrusters to safely land it between two flags. The state of the environment includes:

- Lander's position
- Velocity
- Angle
- Angular velocity
- Leg contact with the ground

The agent must learn to balance fuel efficiency with the safety and accuracy of the landing.
The actions in this environment are discrete, comprising of:

- Do nothing
- Fire the left orientation engine
- Fire the main engine
- Fire the right orientation engine

The reward structure is designed to encourage a safe and fuel-efficient landing. Positive rewards are given for decreasing distance to the landing pad and making a successful landing, while penalties are also given for moving away from the landing pad, excessive fuel usage, and crashing.

Validation Measure: The implementation aimed to develop an agent capable of achieving an average score of 195 or above over the last 50 episodes.

3.1.1 Implementing Standard DQN.

Model Architecture: The DQN architecture consists of 3 fully connected layers with ReLU activation functions, designed to approximate the optimal Q-value function:

```
class STANDARD_DQN(nn.Module):
    # The constructor of the STANDARD_DQN class, inheriting from nn.Module
    def __init__(self, state_size, action_size):
        # Initialize the superclass
        super(STANDARD_DQN, self).__init__()

        # Define the first fully connected layer
        # It takes the state size as input and outputs 64 features
        self.fc1 = nn.Linear(state_size, 64)

        # Define the second fully connected layer
        # It takes the 64 features from the previous layer and outputs 64 features
        self.fc2 = nn.Linear(64, 64)

        # Define the third fully connected layer
        # It takes the 64 features from the previous layer and outputs the number of actions
        self.fc3 = nn.Linear(64, action_size)

    # The forward pass of the network
    def forward(self, x):
        # Apply a ReLU activation function after the first fully connected layer
        x = F.relu(self.fc1(x))
        # Apply a ReLU activation function after the second fully connected layer
        x = F.relu(self.fc2(x))
        # The final layer outputs the Q-values for each action
        return self.fc3(x)
```

Figure 7. Code snippet of Standard DQN Architecture

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	128
BUFFER_SIZE	10000
LEARNING_RATE	0.0005
EPS_START	1.0
EPS_END	0.01
EPS_DECAY	0.995

Table 1. Standard DQN Hyperparameters

Replay Memory: The Replay Memory stores and retrieves experiences (*state, action, reward, next state, and done*) as the agent interacts with the environment. This memory buffer is implemented as a deque (double-ended queue) with a fixed maximum size, enabling efficient sampling of uncorrelated experiences. In this implementation, the *'ReplayMemory'* class initializes a deque with a specified capacity. The *'push'* method is used to store new experiences, while the *'sample'* method retrieves a random batch of experiences for training, crucial for effective learning by breaking sequential correlations:

```
class ReplayMemory:
    # Constructor for ReplayMemory class
    def __init__(self, capacity):
        # Initialize a double-ended queue with a fixed maximum size (capacity)
        # This deque will store the transitions
        self.memory = deque([], maxlen=capacity)

    # Method to push a new transition into the memory
    def push(self, *args):
        self.memory.append(Transition(*args))

    # Method to randomly sample a batch of transitions from the memory
    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    # Method to return the length of the memory
    def __len__(self):
        return len(self.memory)
```

Figure 8. Code snippet of Standard DQN *ReplayMemory*

Optimize Model: This snippet involves sampling from the replay memory, calculating the loss using Mean Squared Error, and updating network weights through backpropagation:

```
# Method to optimize the model (update the policy network)
def optimize_model(self):
    # If not enough samples, return without training
    if len(self.memory) < BATCH_SIZE:
        return

    # Sample a batch of transitions from memory
    transitions = self.memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))
     .
     .
     .
    # Calculate loss using Mean Squared Error between current and expected Q values
    loss = F.mse_loss(state_action_values, expected_state_action_values.unsqueeze(1))
    # Backpropagate the loss to update network weights
    self.optimizer.zero_grad()  # Reset gradients to zero
    loss.backward()  # Compute gradient of loss
    self.optimizer.step()  # Update network weights
```

Figure 9. Code snippet of the *Optimize_model* method in the Standard DQN

3.1.2 Implementing Double DQN (First Improvement).

The implementation of Double DQN aimed to improve the stability and performance of the Standard DQN by addressing the overestimation of Q-values.

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	128
BUFFER_SIZE	10000
LEARNING_RATE	0.0005
EPS_START	1.0
EPS_END	0.01
EPS_DECAY	0.995

Table 2. Double DQN Hyperparameters

Model Architecture: The neural network in Double DQN also consists of three fully connected layers, with ReLU activation functions between layers:

```
class DOUBLE_DQN(nn.Module):
    # The constructor of the DOUBLE_DQN class, inheriting from nn.Module
    def __init__(self, state_size, action_size):
        # Initialize the superclass
        super(DOUBLE_DQN, self).__init__()

        # Define the first fully connected layer
        # It takes the state size as input and outputs 64 features
        self.fc1 = nn.Linear(state_size, 64)

        # Define the second fully connected layer
        # It takes the 64 features from the previous layer and outputs 64 features
        self.fc2 = nn.Linear(64, 64)

        # Define the third fully connected layer
        # It takes the 64 features from the previous layer and outputs the number of actions
        self.fc3 = nn.Linear(64, action_size)

    # The forward pass of the network
    def forward(self, x):
        # Apply a ReLU activation function after the first fully connected layer
        x = F.relu(self.fc1(x))
        # Apply a ReLU activation function after the second fully connected layer
        x = F.relu(self.fc2(x))
        # The final layer outputs the Q-values for each action
        return self.fc3(x)
```

Figure 10. Code snippet of Double DQN Architecture

Action Selection Strategy: Double DQN uses an epsilon-greedy approach for action selection, balancing exploration and exploitation:

```
def select_action(self, state):
    # Random sample for epsilon-greedy strategy
    sample = random.random()
    # Current value of epsilon
    eps_threshold = self.epsilon
    # Epsilon-greedy policy: choose best action or random action
    if sample > eps_threshold:
        # Disable gradient calculation
        with torch.no_grad():
            # Choose the action with the highest Q-value for the current state
            return self.policy_net(state).max(1)[1].view(1, 1)
    else:
        # Choose a random action
        return torch.tensor([[random.randrange(self.action_size)]], dtype=torch.long)
```

Figure 11. Code snippet of *select_action* function in Double DQN

Optimization Process: The optimization process in Double DQN differs from Standard DQN in the way it calculates the Q-values for next states. While Standard DQN estimates the future Q-values directly from the target network, Double DQN decouples the selection of actions and the evaluation of those actions. Specifically, it uses the policy network to select the best action for the next state and then uses the target network to evaluate the Q-value of this selected action. This approach helps to mitigate the overestimation bias often observed in Standard DQN, leading to more accurate and stable value estimations:

```
def optimize_model(self):
    .
    .
    .
    # Double DQN Logic:
    # Use the policy net to select the best action for next states
    next_state_actions = self.policy_net(non_final_next_states).max(1)[1].unsqueeze(1)
    # Use the target net to calculate the Q value of these actions
    next_state_values = torch.zeros(BATCH_SIZE)
    next_state_values[non_final_mask] = self.target_net(non_final_next_states).gather(1, next_state_actions).squeeze(1)
    .
    .
    .
    # Compute the loss between current Q values and expected Q values
    loss = F.mse_loss(state_action_values, expected_state_action_values.unsqueeze(1))
    # Zero gradients before backward pass
    self.optimizer.zero_grad()
    # Backward pass to compute gradients
    loss.backward()
    # Update the weights of the policy network
    self.optimizer.step()
```

Figure 12. Code snippet of *optimize_model* function in Double DQN

Updating the Target Network: This snippet demonstrates how Double DQN reduces overestimation by using the policy net to choose actions and the target net to evaluate their value. Regularly updating the target network with weights from the policy network is a critical part of the Double DQN algorithm for maintaining stability:

```
def update_target_net(self):
    # Update the target network with the policy network's weights
    self.target_net.load_state_dict(self.policy_net.state_dict())
```

Figure 13. Code snippet of *update_target_net* function in Double DQN

In summary, Double DQN enhances the stability of learning by addressing the overestimation bias present in Standard DQN, making it more reliable for complex environments like "LunarLander-v2."

3.1.3 Implementing Noisy DQN (Second Improvement).

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	128
BUFFER_SIZE	10000
LEARNING_RATE	0.0005
EPS_START	1.0
EPS_END	0.01
EPS_DECAY	0.995

Table 3. Noisy DQN Hyperparameters

The Noisy DQN introduces randomness directly into the network's architecture to enhance exploration. This approach eliminates the need for external exploration mechanisms like epsilon-greedy strategies.

NoisyLinear Layer: The 'NoisyLinear' layer is a modified linear layer with noise added to its weights and biases. It uses random noise during training, while reverting to deterministic behaviour during evaluation:

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_size, out_size):
        super(NoisyLinear, self).__init__()

        # Initialize the mean (mu) and standard deviation (sigma) for the weights
        self.w_mu = nn.Parameter(torch.empty((out_size, in_size)))
        self.w_sigma = nn.Parameter(torch.empty((out_size, in_size)))

        # Initialize the mean (mu) and standard deviation (sigma) for the biases
        self.b_mu = nn.Parameter(torch.empty((out_size)))
        self.b_sigma = nn.Parameter(torch.empty((out_size)))

        # Uniformly initialize the mean parameters
        uniform_(self.w_mu, -math.sqrt(3 / in_size), math.sqrt(3 / in_size))
        uniform_(self.b_mu, -math.sqrt(3 / in_size), math.sqrt(3 / in_size))

        # Initialize the sigma parameters with a constant value
        nn.init.constant_(self.w_sigma, 0.017)
        nn.init.constant_(self.b_sigma, 0.017)

    def forward(self, x, sigma=1):
        # During training, add noise to the weights and biases
        if self.training:
            # Generate random noise for weights and biases
            w_noise = torch.normal(0, sigma, size=self.w_mu.size())
            b_noise = torch.normal(0, sigma, size=self.b_mu.size())

            # Apply noisy linear transformation
            return F.linear(x, self.w_mu + self.w_sigma * w_noise, self.b_mu + self.b_sigma * b_noise)
        else:
            # During evaluation, use the mean values without noise
            return F.linear(x, self.w_mu, self.b_mu)
```

Figure 14. Code snippet of the *NoisyLayer*

Model Architecture: The Noisy DQN model utilizes the 'NoisyLinear' layers to induce exploration through internal noise. Each layer adds stochasticity to the learning process, which is crucial for exploring the action space effectively:

```python
class NOISY_DQN(nn.Module):
    # The constructor of the NOISY_DQN class, inheriting from nn.Module
    def __init__(self, state_size, action_size):
        super(NOISY_DQN, self).__init__()
        # Initialize the first fully connected layer with noise
        self.fc1 = NoisyLinear(state_size, 64)
        # Initialize the second fully connected layer with noise
        self.fc2 = NoisyLinear(64, 64)
        # Initialize the third fully connected layer with noise
        # This layer outputs Q-values for each action
        self.fc3 = NoisyLinear(64, action_size)

    def forward(self, x):
        # Pass input through the first noisy linear layer and apply ReLU activation
        x = F.relu(self.fc1(x))
        # Pass through the second noisy linear layer and apply ReLU activation
        x = F.relu(self.fc2(x))
        # Pass through the final noisy linear layer to get Q-values for each action
        return self.fc3(x)
```

Figure 15. Code snippet of Noisy DQN Architecture

Optimization Process: The optimization process in Noisy DQN resembles standard DQN but leverages the noise-infused network for learning. The loss calculation and backpropagation methods remain consistent with the standard DQN approach.

These aspects demonstrate the unique strategy of Noisy DQNs in embedding exploration directly within the network's architecture, thus removing the dependence on external exploration strategies. This approach offers a distinct method for addressing challenges in reinforcement learning.

3.1.4 Implementing the combination of the two Improvements (Double & Noisy) DQN.

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	128
BUFFER_SIZE	10000
LEARNING_RATE	0.0005
EPS_START	1.0
EPS_END	0.01
EPS_DECAY	0.995

Table 4. Combined DQN (Double & Noisy) Hyperparameters

Model Architecture: The Combined DQN architecture blends NoisyLinear layers with Double DQN principles:

```python
# Combined DQN Model
class COMBINED_DQN(nn.Module):
    # The constructor of the COMBINED_DQN class, inheriting from nn.Module
    def __init__(self, state_size, action_size):
        super(COMBINED_DQN, self).__init__()
        # Initialize the first fully connected layer with noise
        self.fc1 = NoisyLinear(state_size, 64)
        # Initialize the second fully connected layer with noise
        self.fc2 = NoisyLinear(64, 64)
        # Initialize the third fully connected layer with noise
        # This layer outputs Q-values for each action
        self.fc3 = NoisyLinear(64, action_size)

    def forward(self, x):
        # Pass input through the first noisy linear layer and apply ReLU activation
        x = F.relu(self.fc1(x))
        # Pass through the second noisy linear layer and apply ReLU activation
        x = F.relu(self.fc2(x))
        # Pass through the final noisy linear layer to get Q-values for each action
        return self.fc3(x)
```

Figure 16. Code snippet of Combined DQN (Double & Noisy) Architecture

The Combined DQN model represents a combination of the Double and Noisy DQN approaches. This architecture integrates the *'NoisyLinear'* layers from the Noisy DQN, which introduce stochasticity into the network, enhancing exploration. Simultaneously, it adopts the *'optimize_model'* function from the Double DQN, utilizing separate networks for action selection and value evaluation to mitigate overestimation in Q-values. Additionally, the combined model leverages the epsilon-greedy strategy for action selection and the efficient experience replay mechanism from both approaches, ensuring balanced exploration and exploitation. This combination provides a solution that leverages both intrinsic exploration through network noise, Q-value estimation, and effective learning from diverse experiences. It should be noted that the code for the *'NoisyLinear'* layers, the *'optimize_model'* function, and other components like the *'select_action'* strategy and the *'replay_memory'* management remains consistent with those outlined in the respective Noisy and Double DQN sections as previously mentioned.

3.2 Experiment 2

Problem Addressed: OpenAI Gym LunarLander (Continuous Actions) - "LunarLanderContinuous-v2"

In the "LunarLanderContinuous-v2" environment, OpenAI Gym presents the task of landing a spacecraft on the lunar surface with continuous action control. This version requires more precise and variable control of the spacecraft's thrusters, adding complexity to the landing maneuver. The agent needs to manage continuous thrust levels for both orientation and descent, aiming for a safe and fuel-efficient landing. The environment's state includes:

- Lander's position
- Velocity
- Angle
- Angular velocity
- Leg contact with the ground

The agent's objective remains to achieve a safe, fuel-efficient landing, but with a continuous range of possible thruster outputs.

The continuous action space in this environment involves:

- Adjusting the main engine's thrust
- Controlling the directional thrusters for orientation

Validation Measure: Just like Experiment 1, the implementation aimed to develop an agent capable of achieving an average score of 195 or above over the last 50 episodes.

3.2.1 Implementing a Stochastic policy approach (PPO-GAE).

Hyperparameter	Value
GAMMA	0.99
LAMBDA	0.95
EPSILON	0.2
BATCH_SIZE	64
EPOCH_REPEAT	10
LEARNING_RATE (Actor)	0.0003
LEARNING_RATE (Critic)	0.0003

Table 5. PPO-GAE Hyperparameters

In the PPO-GAE implementation for "LunarLanderContinuous-v2", the Actor Network, as partially shown in Figure 2, plays a crucial role in proposing actions based on the current state, outputting a probability distribution characterized by mean (*loc*) and standard deviation (*scale*). This network is constructed with fully connected layers followed by *Tanh* activation functions, resulting in the final layer that defines the action distribution. Complementing the Actor, the Critic Network evaluates the proposed actions as can be seen in Figure 17 below. It is responsible for predicting the value function, a measure of the expected return from a given state under the policy in action. The Critic's architecture mirrors the Actor's, consisting of fully connected layers with *Tanh* activations, outputting a single value indicative of the state's value.

Optimization and Loss Computation in PPO-GAE, as detailed in Figure 3, involve a surrogate objective function to update the Actor network, where the ratio of new and old probabilities of actions is calculated, and a clipping mechanism is employed to moderate policy updates. The Generalized Advantage Estimation (GAE), illustrated in Figure 4, is a key component of the PPO-GAE approach. GAE calculates the advantage estimation, effectively balancing bias and variance in the computation. The Critic Network undergoes updates by minimizing the temporal difference error, which aligns the predicted state values with the computed target values. Training comprises alternating phases of environment interaction for data sampling and network optimization. The Training Loop iteratively performs actions in the environment and stores the resulting experiences. Subsequently, during training epochs, these experiences are processed to compute advantages and update both networks. Reaching or surpassing the validation measure indicates that the agent has successfully solved the environment.

```python
class CriticNet(nn.Module):
    def __init__(self, input_size, hidden_units=64, output_size=2):
        super(CriticNet, self).__init__()
        # Neural network layers
        self.model = nn.Sequential(
            # First linear layer
            nn.Linear(input_size, hidden_units),
            # Tanh activation function
            nn.Tanh(),
            # Second linear layer
            nn.Linear(hidden_units, int(hidden_units/2)),
            # Tanh activation function
            nn.Tanh()
        )
        # Output layer for value function
        self.value_head = nn.Linear(int(hidden_units/2), 1)

    def forward(self, x):
        x = self.model(x)
        # Get value function output
        value = self.value_head(x)
        return value

    def __call__(self, x):
        # Call forward method
        out = self.forward(x)
        return out
```

Figure 17. Code snippet of *CriticNet* in PPO-GAE

3.2.2 Implementing a Deterministic policy approach (DDPG).

The update function (Figure 5), a pivotal aspect of DDPG, involves calculating the loss between predicted and target Q-values and updating both the Actor and Critic networks. The Actor is updated using policy gradients to maximize the expected return, as partially shown in Figure 6, which illustrates a snippet of the forward pass through the network. Meanwhile, the Critic is trained to minimize the temporal difference error, ensuring the convergence of the Actor's policy towards optimality. The Replay Memory, integral to the training, stores and samples diverse, uncorrelated experiences, enhancing the learning process. Additionally, the Ornstein-Uhlenbeck process is used for action exploration, introducing time-correlated noise to the actions, aiding in effective exploration of the action space. The training loop involves

iteratively applying actions, accumulating rewards, and updating the networks, continuing until the agent achieves the predefined validation measure.

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10000
TAU	0.01
LEARNING_RATE (Actor)	0.0003
LEARNING_RATE (Critic)	0.0003

Table 6. DDPG Hyperparameters

4 Results and Discussion

4.1 Experiment 1

In the "LunarLander-v2" environment, the Noisy DQN emerged as the most efficient model, solving the challenge in 422 episodes, which is 22.7% more quickly than the Standard DQN at 546 episodes. The Double DQN model also displayed notable improvement, reaching the solution in 462 episodes, approximately 15.4% faster than the Standard DQN. The Combined DQN (Double & Noisy DQN) achieved its goal in 431 episodes, standing 21.1% faster than the Standard DQN. These figures underscore the substantial improvements that noise injection and overestimation bias correction contribute to the efficacy of DQN models, as showcased in Figure 18.

Model	Episodes to Solve
Standard DQN	546
Double DQN	462
Noisy DQN	422
Combined DQN	431

Table 7. Number of episodes required for each model to solve the LunarLander-v2 environment.

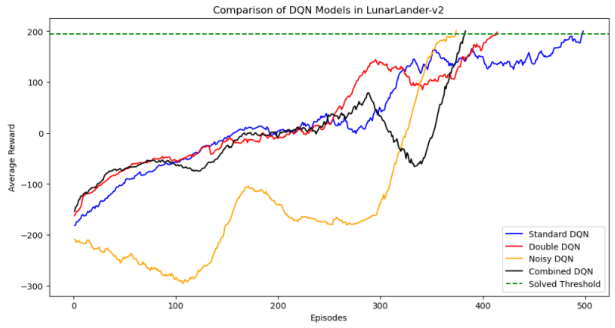

Figure 18. Performance comparison of all DQN models in the LunarLander-v2 environment.

4.2 Experiment 2

In the LunarLanderContinuous-v2 environment, the Stochastic Policy approach using PPO-GAE solved the task in 190 episodes, significantly outperforming the Deterministic Policy approach using DDPG, which required 449 episodes to reach the solution. The PPO-GAE approach was more than 57% faster compared to the DDPG method. This efficiency gain emphasizes the effectiveness of stochastic policy methods in environments with continuous action spaces. Figure 19 provides a visual representation of the learning curves for each method, illustrating the quicker convergence of the PPO-GAE approach to the solution threshold.

Model	Episodes to Solve
Stochastic Policy (PPO-GAE)	190
Deterministic Policy (DDPG)	449

Table 8. Number of episodes required to solve the LunarLanderContinuous-v2 environment.

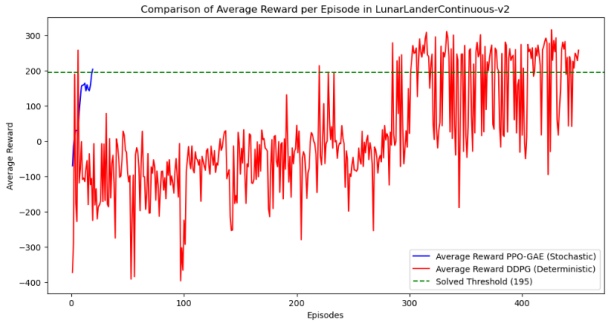

Figure 19. Combined learning curves of the PPO-GAE and DDPG models in the LunarLanderContinuous-v2 environment.

References

[1] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* (1996), pp. 237–285.

[2] Bashar Rajoub. "Supervised and unsupervised learning". In: *Biomedical Signal Processing and Artificial Intelligence in Healthcare*. Ed. by Walid Zgallai. Developments in Biomedical Engineering and Bioelectronics. Academic Press, 2020, pp. 51–89.

[3] Daniel B Or. *Value-based methods in deep reinforcement learning*. Medium. 2023. URL: https://towardsdatascience.com/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1.

[4] Hugging Face. *What are the policy-based methods? - Hugging Face Deep RL Course*. Hugging Face. 2023. URL: https://huggingface.co/learn/deep-rl-course/unit4/what-are-policy-based-methods.

[5] Dinesh Karunakaran. *The Actor-Critic Reinforcement Learning Algorithm*. Medium. 2023. URL: https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14.

[6] Yanhua Huang. "Deep Q-Networks". In: *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Singapore: Springer Singapore, 2020, pp. 135–160. ISBN: 978-981-15-4095-0.

[7] Ofir Nachum et al. "Bridging the Gap Between Value and Policy Based Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[8] Chintan Trivedi. *Proximal Policy Optimization Tutorial (Part 2/2: GAE and PPO Loss)*. Medium. 2023. URL: https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-2-2-gae-and-ppo-loss-fe1b3c5549e8.

[9] Spinning Up in Deep RL, OpenAI. *Deep Deterministic Policy Gradient*. Spinning Up in Deep RL, OpenAI. 2023. URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html.