Solving Control Problems using Reinforcement Learning

Kian Parnis
University of Malta
Pembroke, Malta

1 INTRODUCTION

1.1 What is Reinforcement learning?

Reinforcement Learning (RL) is the process of learning what to do, by interacting with the environment. This involves the recognition of similarities and certain characteristics of certain situations and mapping these situations to actions, with the overall goal being to maximize some utility that is of interest. RL is typically broken down into an interaction loop (see Figure 1.1.1), this involves our agent taking some action in an environment based on some policy that transitions its current state to a new state while collecting some reward as well as an observation.

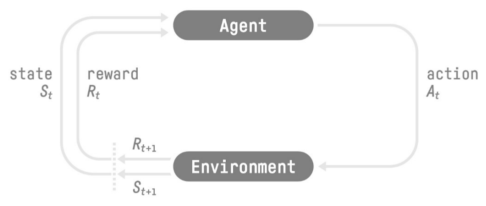


Figure 1.1.1: RL Interaction Loop [1]

The policy typically denoted as π, is a function that maps a state s_t to an action A_t (i.e., how the agent acts in each different environmental state by taking a decision to move into another state s_{t+1}, where t is the current time step). Rewards are represented as numerical values giving back to the agent as a result of the action taken in a state as R_{t+1}. Rewards R_t are used to improve the agent's policy in deciding what action to take in a particular state to reach its target goal, the optimal policy π^* and is achieved by maximizing an expected cumulative reward (The sum of all rewards achieved after a particular time step) [2]. The environment is a scenario that is typically modeled around aiding the agent to train efficiently to reach its desired objectives/goal. For example, the AlphaZero algorithm had managed to achieve a superhuman level of play in Chess and Shogi within 24 hours and this was achieved with the use of Deep Convolutional Neural Networks and playing self-play games utilizing RL [3]. In this scenario, the environment can either be the real world or a simulation of it, with the latter being a cheaper, faster, and safer way to conduct training.

Deep Reinforcement Learning (Deep RL) is a sub-field of machine learning that combines Reinforcement Learning (RL) and Deep Learning. Deep RL is often used when the state space is not feasible to engineer due to space constraints. For example, the game of chess has $\sim 10^{47}$ states while continuous actions are considered to have an infinite state space. Deep Learning aids by utilizing an approximation function which is parameterized by a vector of weights and is typically characterized as Artificial Neural Networks (which includes CNNs and other variants of Deep Neural Networks).

Artificial Neural Networks (ANN) are a collection of perceptions and activation functions. The Network is comprised of an input layer, an output layer, and several hidden layers (see Figure 1.1.2) that map the input to its output. [4].

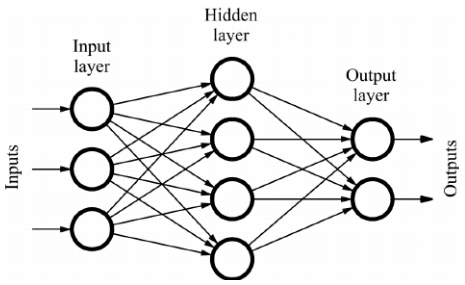


Figure 1.1.2: Artificial Neural Network

A Perceptron is a fundamental unit in ANN's, they receive a number of real values with each input given a weight, with the weight determining the contribution of the input to the output. Its computations are the linear combination of these inputs plus a certain bias to produce a single output Y.

$$Y = \sum(weight * input) + bias$$

Since perceptrons are linear [4], activation functions are also utilized to make neural networks nonlinear techniques which determine when a perceptron should fire. A number of these step functions exist with the simplest being the Step Function. The Step function [5] is a binary classifier that fires a one if the summation Y is greater than a certain threshold and a zero otherwise, which makes this activation function piecewise linear. The Sigmoid Function [5] is a smoothened version of the step function that's used to represent non-linear functions between 0 and 1. The Sigmoid function is denoted as:

$$\sigma(Y) = \frac{1}{1 + e^{-Y}}$$

The Tanh Function [5] is a scaled version of the Sigmoid Function, this function maps the values between -1 and 1 which makes its gradients more stable. The Tanh function is denoted as:

$$\tanh(Y) = \frac{2}{1 + e^{-2Y}} - 1$$

The problem with the Sigmoid and Tanh functions is that they fire all the time making an ANN heavy. The Rectified Linear Unit (ReLu) [6] is a less computationally expensive activation function that maps the input Y to the max of either zero or Y. This lets big numbers pass making a few perceptrons stale. The ReLu is denoted as:

$$\sigma(Y) = max(0, Y)$$

Within RL, an ANN's input layer corresponds to states (for example an input image), while the outputs represent our actions (Move up, Move down etc.).

1.2 How does Reinforcement Learning differ from other Machine Learning approaches?

Machine learning techniques such as Reinforcement Learning (RL) differ from other methods like supervised and unsupervised learning in a variety of ways. In Reinforcement Learning, the agent learns through trial and error by interacting with the environment and getting feedback in the form of rewards or penalties for its actions, as opposed to supervised learning and unsupervised learning, where the agent learns from labeled examples and unlabeled data, respectively.

While supervised and unsupervised learning tasks are commonly thought of as separate and equally distributed, RL typically involves sequential decision-making, where the agent's actions in the present can influence the rewards or penalties it receives in the future. While supervised learning and unsupervised learning are mostly employed for prediction and feature extraction or dimensionality reduction, respectively, RL is goal-oriented, with the agent's objective being to learn a policy that maximizes cumulative reward over time. In contrast to supervised learning, which is often a batch process, Reinforcement learning (RL) is typically an online learning process, allowing the agent to acquire experience over time.

1.3 Brief introduction to Value-Based, Policy-Based and Actor-Critic

Value-based models, such as Q-learning, estimate the value of being in a certain state or taking a certain action. These models determine the best action to take by choosing the action that leads to the highest estimated value. Policy-based models, such as REINFORCE, directly output the probability of taking a certain action given a certain state. These models determine the best action to take by choosing the action with the highest probability. Actor-Critic models are a combination of value-based and policy-based models. They have two components: an "actor" which determines the action to take based on the current state, and a "critic" which evaluates the quality of the action taken by the actor. The critic provides feedback to the actor, which is used to adjust the actor's behavior over time. Thus, the main difference between each model is that Value-based models determine the best action based on the estimated value, and Policy-based models determine the best action based on the probability while Actor-Critic models determine the best action based on both the estimated value and probability.

2 BACKGROUND

2.1 Value Based Methods

As previously discussed, Value-based methods are a class of Reinforcement Learning (RL) algorithms that aim to approximate the optimal value function which is denoted by $V^*(s)$, which maps states to the expected cumulative reward when following a certain policy. The main idea behind these methods is to estimate the value of being in a given state and then choosing the action that maximizes the value function. Value-Based methods are based on the Bellman equation, which expresses the relationship between the value of a state and the values of the next states and the rewards obtained by transiting them. The Ballman equation for the value function is:

$$V^*(s) = max_a(R(s, a) + \gamma V^*(s'))$$

Where s is the current state, a is the current action, $R(s, a)$ is the immediate reward, s' is the next state and γ (gamma) is the discount factor, a value ranging between 0 and 1 that determines the importance of future rewards. In order to solve the above equation, we need to know the transition function denoted as $P(s^{'}|s, a)$ which gives the probability of going to state s' from a state s taking action a. This is where Markov Decision Process(MDP) [7] comes into the picture.

An MDP is a mathematical framework for modeling decision-making problems, where the current state of the system, the available actions, and the probabilities of transitioning to the next states are known. An MDP is defined by a tuple (S, A, P, R, γ), where:

S is a finite set of all possible states.
A is a finite set of all possible actions.
$P(s^{'}|s, a)$ is the probability of transitioning to s' when applying action a to state s.
$r = \mathbb{E}[R|s, a]$ is the expected reward of applying action a to state s.
$\gamma \in [0, 1]$

The goal of value-based methods is to find the optimal value function V^* and the optimal policy π^*(s) that maximizes the expected cumulative reward. The optimal policy is defined as:

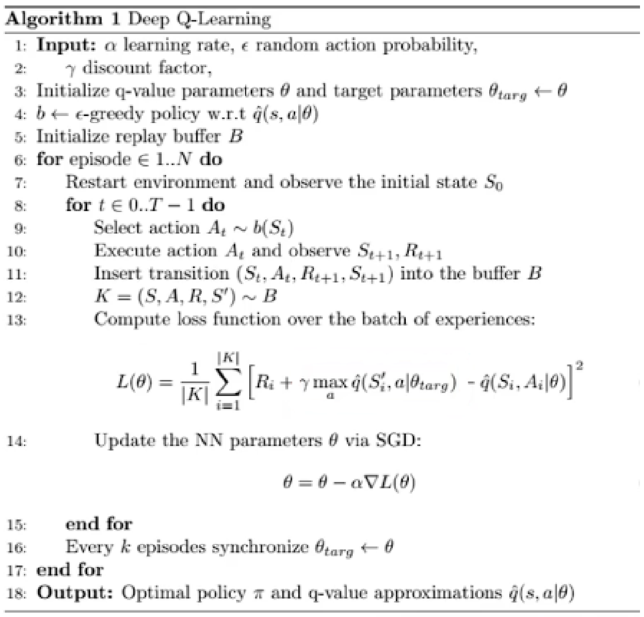
$$\pi^*(s) = argmax_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1})|S_t = s, A_t = a]$$

Where $V^*(s)$ and $\pi^*(s)$ are the optimal value function and optimal policy respectively, with the RHS returning the best a that maximizes the expectation.

2.2 The Deep Q-Network algorithm

Deep Q-Network (DQN) [8] is a variant of the Q-learning algorithm that uses a neural network to approximate the Q-function. The Q-function represents the expected cumulative future reward for a given action in a given state. The goal of Q-learning is to find the optimal Q-function that maps states to actions such that the expected cumulative reward is maximized. In Q-learning, the Q-function is typically represented as a table of values, where each entry corresponds to the expected cumulative reward for a given state-action pair. However, in real-world problems, the state space can be very large and complex, making it infeasible to represent the

Q-function as a table. DQN addresses this issue by using a neural network to approximate the Q-function.



```
Algorithm 1 Deep Q-Learning
1:  Input: $\alpha$ learning rate, $\epsilon$ random action probability,
2:      $\gamma$ discount factor,
3:  Initialize q-value parameters $\theta$ and target parameters $\theta_{targ} \leftarrow \theta$
4:  $b \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
5:  Initialize replay buffer $B$
6:  for episode $\in 1..N$ do
7:      Restart environment and observe the initial state $S_0$
8:      for $t \in 0..T-1$ do
9:          Select action $A_t \sim b(S_t)$
10:         Execute action $A_t$ and observe $S_{t+1}, R_{t+1}$
11:         Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer $B$
12:         $K = (S, A, R, S') \sim B$
13:         Compute loss function over the batch of experiences:
                $L(\theta) = \frac{1}{|K|}\sum_{i=1}^{|K|}\left[R_i + \gamma \max_a \hat{q}(S_i', a|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)\right]^2$
14:         Update the NN parameters $\theta$ via SGD:
                $\theta = \theta - \alpha\nabla L(\theta)$
15:     end for
16:     Every $k$ episodes synchronize $\theta_{targ} \leftarrow \theta$
17: end for
18: Output: Optimal policy $\pi$ and q-value approximations $\hat{q}(s, a|\theta)$
```

Figure 2.2.2: The Deep Q-Networks algorithm

As DQN extends q-learning, it also follows an off-policy learning strategy, meaning that it will explore the environment using an exploratory policy that will be epsilon greedy with respect to the estimated q values $\hat{Q}(s, a|\theta)$ where θ is denoted as the network's parameters. This is shown in Figure 2.2.2 lines 4 and 9, And is translated as code:

```python
def e_greedy(state, env, net, epsilon=0.0): #The E-Greedy Policy
  if np.random.random() < epsilon:
    action = env.action_space.sample() #take a random action
  else: # 1 - e
    state = torch.tensor([state]).to(device) #Pass a batch of observations | device -> CPU or GPU
    q_values = net(state) #Produce q_values from NN
    _, action = torch.max(q_values, dim=1) #Take the max q_value action | torch.max passes two values, the actual value and the index (we only care about the index)
    action = int(action.item()) #make sure its a integer
  return action
```

Called within class *DeepQLearning*(*LightningModule*) as:

```python
if policy: #if a policy is present get action selected by the policy
    action = policy(state, self.env, self.q_net, epsilon=epsilon)
else: #if not pick a random action
    action = self.env.action_space.sample()
```

To update the network's parameters θ, DQN uses the following loss function: (Shown in line 13 of Figure 2.2.2)

$$L(\theta) = \frac{1}{|K|}\Sigma_{i=1}^{|K|}[R_i + \gamma max_a \hat{q}(S_i^{'}, a_i|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)]^2$$

where:

$L(\theta)$ is our loss.
K is the size of the batch.
R_i is the current batch reward.
$\gamma \in [0, 1]$
$max_a \hat{q}(S_i^{'}, a_i|\theta_{targ})$ is the current batch target Q-value.
$\hat{q}(S_i, A_i|\theta))$ is the current batch expected Q-value.

This loss function is translated into code within class DeepQLearning (*LightningModule*) as:

```python
def training_step(self, batch, batch_idx):
    states, actions, rewards, dones, next_states = batch #retrieve data from batch
    actions = actions.unsqueeze(1) #create an extra dim to solve size diff
    rewards = rewards.unsqueeze(1)
    dones = dones.unsqueeze(1)

    state_action_values = self.q_net(states).gather(1, actions) #get Q(s,a) value from neural network

    next_action_values, _ = self.target_q_net(next_states).max(dim=1, keepdim=True) #keepdim keeps orginal dim, get target q(s', a') values
    next_action_values[dones] = 0.0 #set done values to zero

    expected_state_action_values = rewards + self.hparams.gamma * next_action_values # gamma is the discount factor

    loss = self.hparams.loss_fn(state_action_values, expected_state_action_values) #compute loss values with the mean sq diff (varient)
    self.log('episode/Q-Error', loss)

    return loss

return loss
```

Also, note that θ_{targ} represents the fixed target network's parameters. The idea is to keep the weights of the target Q-network fixed for a certain number of steps and then update them to the weights of the Q-network being trained. The algorithm uses this fixed target network to prevent the Q-values from oscillating or diverging. Thus, in Figure 2.2.2 lines 3 and 16 show the instantiation and synchronization of θ_{targ} respectively.

Instantiation code step:

```python
self.target_q_net = copy.deepcopy(self.q_net)
```

Synchronization code step:

```python
if self.current_epoch % self.hparams.sync_rate == 0:
    self.target_q_net.load_state_dict(self.q_net.state_dict())
```

To improve the stability of the training process, DQN uses a technique called experience replay. In experience replay, the agent stores a dataset of past experiences (i.e., state, action, reward, next state) in a replay buffer. The agent then samples random mini-batches of experiences from the replay buffer to use in the training process, rather than using the most recent experience. As shown in

Figure 2.2.2 in lines 5, 11, and 12. The class for the replay buffer is written as follows:

```
class ReplayBuffer: #Store enviroment observations (kinda
    like a database) | Stored as [State, Action, Reward
    , Next State]
  def __init__(self, capacity): #defining the capacity of
    the replay buffer
    self.buffer = deque(maxlen=capacity) #Using a queue
    data structure

  def __len__(self):  #return the length of the buffer
    return len(self.buffer)

  def append(self, experience): #save experience in the
    database
    self.buffer.append(experience)

  def sample(self, batch_size):
    return random.sample(self.buffer, batch_size) #get a
    sample of explisit buffer size
```

To summarize, the Deep Q-Network algorithm is a variant of Q-learning that uses a neural network to approximate the Q-value function, and employs experience replay and fixed Q-targets to improve the stability of the training process.

2.3 Policy Based Methods

Policy Based Methods are also a class of Reinforcement Learning (RL) algorithms whose goal is to learn a policy function, denoted as a stochastic policy $\pi(a|s)$, which is the probability of choosing a in a state s, $P(a|s)$. This can also be extended to approximate π using parameter θ which represents the parameters of our policy as $\pi(a|s, \theta) = P(A_t = a|S_t = s, \theta_t = \theta)$. Given this policy approximation function, we would want to find the best parameters for θ. This is achieved with the use of Stochastic Gradient Accent which utilizes the score function $\nabla \ln \pi_\theta(s, a)$ and combines it with the return to adjust the parameter vector θ as follows:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi_{\theta t}(S_t, A_t)$$

Where G_t is the return for state S_t and α is the step size hyper-parameter. As discussed, one popular algorithm that uses a policy-based method is REINFORCE. REINFORCE uses the gradient of the expected return with respect to the policy parameters, θ, to update the policy. This is written as:

$$G = R_t + \gamma G$$
$$\hat{J}(\theta) = \gamma^t G \ln \pi(A_t|S_t, \theta)$$
$$\theta = \theta + \alpha \nabla \hat{J}(\theta)$$

This can be mapped to REINFORCE's implementation in the following:

```
for row in range(self.samples_per_epoch -1, -1, -1): #
    from samples_per_epoch -1 -> -1 (moving backwards) (
    last to first)
      running_return = reward_b[row] + (1 - done_b[row])
    * self.gamma * running_return # return in each
    timestep in a signle backwards pass
      return_b[row] = running_return
```

```
prob_b = self.policy(obs_b)
    log_prob_b = torch.log(prob_b + 1e-6) #calculate our
    log probabilities for our loss function
    action_log_prob_b = log_prob_b.gather(1, action_b)

    entropy = - torch.sum(prob_b * log_prob_b, dim =-1,
     keepdim=True)

    pg_loss = - action_log_prob_b * return_b #minimize
     negative -> maximize
    loss = (pg_loss - self.hparams.entropy_coef * entropy
     ).mean()
```

The main advantage this gives over Value-Based methods is that Value-Based methods cannot represent stochastic policies. This also means whereas, for example, Q-Learning would need to utilize its own exploration strategy such as an e-greedy policy, which can be quite inefficient, Policy Based can use its stochastic policies for exploration. Another advantage is that the policy changes more smoothly during learning, for value-based methods when the Q-value changes, they choose a new action 100% of the time. On the other hand, in policy-based methods, the probability of choosing an action changes in small increments. This is not to imply that value-based methods are worthless, policy gradients' high variance estimates of the gradient updates are one of their main drawbacks. This results in extremely noisy gradient estimations and can make learning unstable.

2.4 Actor-Critic Methods

In state-of-the-art implementations, Actor-Critic is mainly utilized. These methods have two main components: an actor and a critic. The actor represents the policy, and the critic represents the value function. The actor learns to select actions, while the critic learns to evaluate the actions taken by the actor. The two components are trained together and influence each other.

Value-based methods are generally more sample efficient and converge faster than policy-based methods because they only need to estimate the value of a state or state-action pair, rather than the full policy. However, they can be sensitive to the choice of function approximator and may have difficulty dealing with large or continuous action spaces. Policy-based methods, on the other hand, can handle large or continuous action spaces more easily, but they tend to have high variance estimates and converge more slowly than value-based methods.

The combination of these two allows the agent to balance the trade-off between exploration and exploitation. Actor-Critic methods generally have a lower variance than pure policy-based methods and are more able to handle large or continuous action spaces than pure value-based methods.

2.5 Deep Deterministic Policy Gradient (DDPG)

DDPG (Deep Deterministic Policy Gradient) [9] is an algorithm for training agents to perform a task in a continuous action space. It is an extension of the DQN (Deep Q-Network) algorithm, which is used for training agents in discrete action spaces. The difference between discrete and continuous actions is that for continuous actions, there is an infinite amount of possible actions to be taken.

In order to solve these kinds of tasks, the DDPG algorithm assumes that the Q function $Q(s, a(s))$ is differentiable w.r.t $a(s)$. This means that for example, if we had an action [-1, 1] and [1.0001, -1], DDPG would assume that they are very similar values ($max_a Q(s,a) \approx Q(s, \mu(s))$), which ends up being a good assumption in the real world. With this, since the Q function is differentiable, optimization can be done with the use of a stochastic gradient accent. Similarly to the REINFORCE algorithm, the policy is also represented with a neural network $\pi(s|\theta)$ and optimized via gradient ascent so that it produces the most valuable actions.

Thus, DDPG learns a Q-function as well as a policy concurrently which are both neural networks with the use of stochastic gradient accent (Making it an actor-critic algorithm). The Q-network is used to evaluate the actions taken by the policy network and provide feedback to update it, while the policy network generates actions to be taken by the agent, based on the current state. In addition, these, DDPG similarly to DQN also utilizes a replay buffer to store experiences, so that the network can learn from them in order to improve its performance. It also uses a target network, which is used to stabilize the training process. This gives us two prominent equations the first is calculating the MSBE (mean-squared Bellman error) loss for our Q Learning which is given by:

$$L(\phi, B) = \underset{(s,a,r,s',d)\sim B}{\mathbb{E}}[Q_{\phi(s,a)} - (r + \gamma(1-d)Q_{\phi_{targ}}(s', \mu_{\phi_{targ}}(s'))^2]$$

where:

$L(\phi, B)$ is our loss.
B is our previous experiences.
$\mu_{\phi_{targ}}$ is the target policy.
$\gamma \in [0, 1]$
d is our terminal state, (True == 1, False == 0) .

This maps to the following lines:

```
if optimizer_idx == 0:

    action_values = self.q_net(states, actions)
    next_actions = self.target_policy.mu(next_states)
    next_action_values = self.target_q_net(next_states,
    next_actions)

    next_action_values[dones] = 0.0 #no extra rewards

    target = rewards + self.hparams.gamma *
   next_action_values

    q_loss = self.hparams.loss_fn(action_values, target
   )
    self.log('episode/Q-Error', q_loss) #compute loss

    return q_loss
```

The second is calculating our deterministic Policy by performing gradient ascent (with respect to our policy parameters only) to solve:

$$\underset{\theta}{max} \underset{s\sim B}{\mathbb{E}} Q_\phi(s, \mu_\theta(s))$$

which maps to the following:

```
elif optimizer_idx == 1:

    mu = self.policy.mu(states)
    policy_loss = - self.q_net(states, mu).mean() #
   negative to achieve minimize to maximize in pytorch
    self.log('episode/P-Error', policy_loss) #compute
   loss
    return policy_loss
```

2.6 Proximal Policy Optimisation (PPO)

The PPO algorithm [10] is an improvement over the vanilla policy gradient method. Algorithms such as REINFORCE carry out a 'Line Search Method' with the step:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi_{\theta_t}(S_t, A_t)$$

PPO, similar to Trust Region Policy Optimisation (TRPO) considers better data efficiency and reliability in taking data steps which classifies them under Trust Region Methods. The goal of PPO is to reduce the amount of change made to the policy at each training epoch in order to increase the training stability of the policy (smaller policy updates during training are more likely to converge to an optimal solution).

This is achieved with the term "$r_t(\theta)$" referring to the ratio of the current policy to the old policy. It is used to measure the change in the policy network during the update step. The current policy is represented by the probability distribution of actions given the current state, which is estimated by the policy network with the current set of parameters θ. The old policy is represented by the probability distribution of actions given the current state, which is estimated by the policy network with the old set of parameters θ_{old}. $r_t(\theta)$ is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

The ratio $r_t(\theta)$ represents how much the current policy differs from the old policy. A ratio close to 1 indicates that the current policy is similar to the old policy and a small change is made, while a ratio different than 1 indicates that the current policy is different from the old policy and a large change is made. The PPO algorithm uses this ratio to ensure that the update of the policy network is not too large, by limiting the values of $r_t(\theta)$ between $1 - \epsilon$ and $1 + \epsilon$ where ϵ is a hyper-parameter called the clipping factor. This helps to prevent the algorithm from overreacting to the samples collected in a single episode, making the algorithm more stable and less likely to oscillate. This is then used to define the objective function as follows:

$$J(\theta) = \hat{\mathbb{E}}[min(r_t(\theta)\hat{A}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A})]$$

This can be mapped to the following lines:

```
rho = torch.exp(log_prob - prev_log_prob)

    surrogate_1 = rho * advantages
    surrogate_2 = rho.clip(1 - self.hparams.epsilon, 1
   + self.hparams.epsilon) * advantages

    policy_loss = - torch.minimum(surrogate_1,
   surrogate_2) #PPO loss using clipping
```

However, the original PPO algorithm uses the Monte Carlo method to estimate the advantage function, which can lead to high variance and bias in the estimation. The Generalized Advantage Estimation (GAE) [11] (an enhancement of the PPO algorithm) addresses this issue by introducing a parameter λ that controls the trade-off between bias and variance in the advantage function. The GAE advantage function is defined as:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \Sigma_{(l=0)}^{\infty} (\gamma\lambda)^l \delta^V{}_{t+1}$$

where:
$\hat{A}_t^{GAE(\gamma,\lambda)}$ is the advantage function at time step t.
δ is the temporal difference error.
$\gamma \in [0, 1]$.
λ is the GAE paramater.
this can be mapped to the following:

```
 for row in range(self.samples_per_epoch -1, -1, -1):
  #from samples_per_epoch -1 -> -1 (moving backwards)
  (last to first)
   running_gae = td_error_b[row] + (1 - done_b[row].
  int()) * self.gamma * self.lamb * running_gae
   gae_b[row] = running_gae
```

3 METHODOLOGY

3.1 Problem Definition

Both Environments depict a classic rocket trajectory optimization problem. There are two variations[1] to this environment 'LunarLander-v2' which deals with discrete actions and 'LunarLanderContinuous-v2' which deals with continuous actions. This means the only difference between the two environments is the different actions they produce. LunarLander-V2 has a Discrete(4) action space, these are: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. On the other hand, LunarLanderContinuous-v2 has a Box(2) action space which corresponds to $Box(-1, +1, (2,), dtype = np.float32)$. The first coordinate of an action determines the throttle of the main engine, while the second coordinate specifies the throttle of the lateral boosters.

The environment's observations space depicts an 8-dimensional vector, these are: the coordinates of the lander's x and y positioning, its linear velocities in x and y, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not. The lander's starting state is always at the top center of the viewport, a random initial force is applied to its center of mass, and the object for the lander is to land on the landing pad which is always placed at coordinates (0,0), a possibility is also given for the lander to land outside the landing pad. There are three ways an episode might end, these are: 1. If the lander crashes, 2. If the lander goes outside the viewpoint, and 3. If the lander has come to rest (it has landed successfully). Finally, a number of different rewards are given, 100–140 points are awarded for going from the top of the screen to the landing pad and stopping. The lander loses its reward if it moves from the landing pad. The lander loses an additional 100 points if it crashes. If it comes to a stop, it gains an extra 100 points. Points are added for each leg that touches the ground. Each frame when the primary engine is fired costs -0.3 points. Each frame that the side engine is fired costs -0.03 points. The problem is considered solved if it achieves 200 points.

[1]https://www.gymlibrary.dev/environments/box2d/lunar_lander/

3.2 PyTorch-Lightning

The experiments were all implemented with the help of PyTorch-lightning[2]. PyTorch-Lightning is a lightweight wrapper on top of the PyTorch library that allows users to easily scale their models and training scripts across multiple GPUs and machines. It also includes a number of additional features such as automatic logging, model check-pointing, and early stopping, making it a popular choice for developing and deploying deep-learning models.

3.3 Optuna

Optuna[3] is an open-source Python library that was used for hyperparameter optimization. It aims to make it easy to conduct hyperparameter searches by providing a high-level API for defining and running trials, as well as a variety of built-in optimization algorithms and integration with other machine learning libraries. Additionally, Optuna provides visualization tools for understanding the results of the optimization process and can be used with popular machine learning frameworks such as TensorFlow and PyTorch Lightning. This library was used to try out a number of trials (this was set to 20) to try out different hyper-parameters to maximize an 'hp_metric' which is the statistical mean over the previous 100 episodes. This was used throughout with a specific range of values that should be tried out. For example, in DQN the learning rate was suggested to be anywhere between $1e-5$ to $1e-1$. The best results would then be taken and the model would be trained again using these found parameters. The main parameters tested were: batch sizes, learning rates, gamma, and hidden sizes.

3.4 Early Stopping

In reinforcement learning, early stopping can be used to stop the training process of an agent before it reaches the maximum number of epochs, the idea is to stop the training when the performance of the agent starts to degrade, indicating that the agent might be over-fitting to our batch training data. PyTorch Lightning provides an Early-Stopping callback which can be used to implement the early-stopping functionality. This can be used by passing a metric to our trainer class, its patience (which is the number of epochs that should go by without improvement, to stop our trainer), and if we are maximizing or minimizing. The metric which is monitored with our early stopping was the mean return, patience varied from implementation to implementation with different algorithms needing a lower/higher degree of patience, and finally, maximization was used since we want to maximize the amount of mean return received.

3.5 TensorBoard

TensorBoard[4] is a web-based tool developed by TensorFlow that allows users to visualize and analyze their machine-learning models. In the context of reinforcement learning, Tensor-Board can be used

[2]https://www.pytorchlightning.ai/
[3]https://optuna.org/
[4]https://www.tensorflow.org/tensorboard

to visualize various metrics such as the agent's reward over time, the loss, the number of epochs, etc. This will mainly be utilized apart from visualization, to compare the results achieved by the various experiments done.

4 RESULTS

In this section, various results shall be outlined for each of the different algorithms, with tables also used to compare the different hyper-parameters used to train the final models. In the next section, these results shall be used to provide conclusions based on our results.

4.1 The Deep-Q Network (DQN)

As discussed previously, Optuna was used to test various hyper-parameters against each other to maximize our hp_metric. The following is an example of such a test, with different ranges for learning rate and gamma being taken. One can note the drastic difference achieved when changing such parameters which can significantly improve our agent's performance.

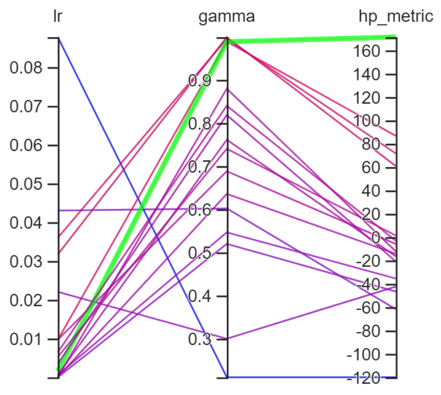


Figure 4.1.1: Optuna testing for DQN

The following are the different results obtained by the DQN algorithm:

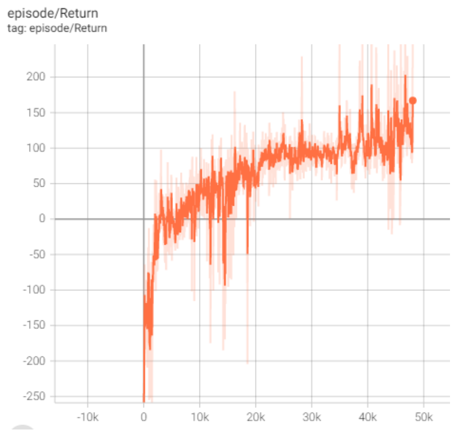


Figure 4.1.2: DQN Return vs Steps

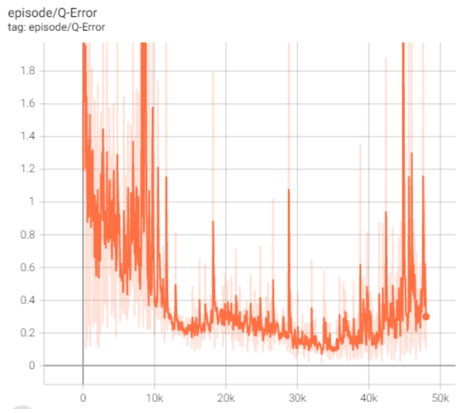


Figure 4.1.3: DQN Value Function Loss

4.2 REINFORCE

The following are the final results obtained by the REINFORCE algorithm.

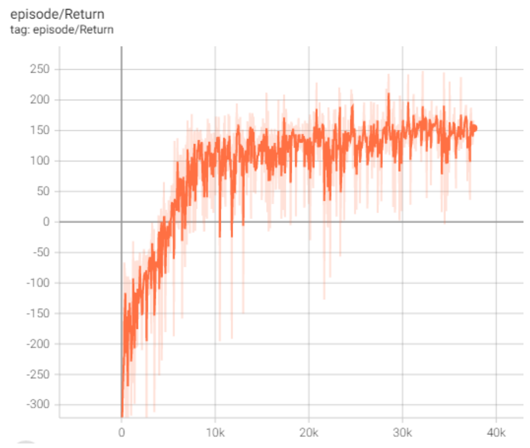


Figure 4.2.1: REINFORCE Return vs Steps

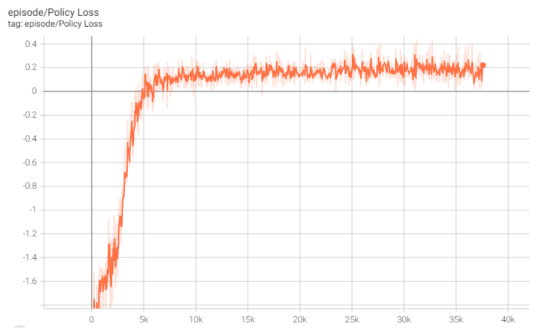


Figure 4.2.2: REINFORCE Policy Loss

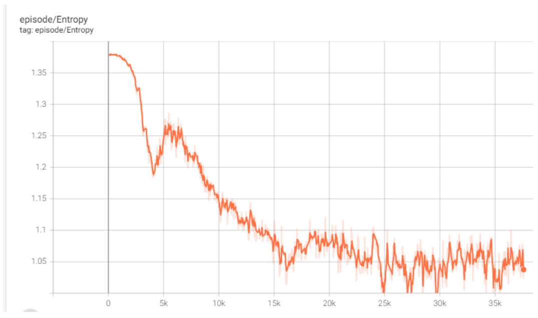


Figure 4.2.3: REINFORCE Entropy

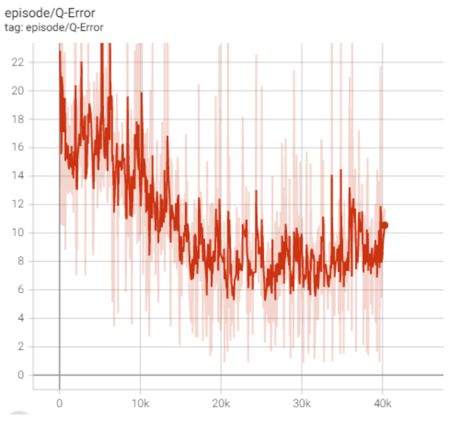


Figure 4.3.3: DDPG Value Loss

4.3 DDPG

The following are the final results obtained by the DDPG algorithm.

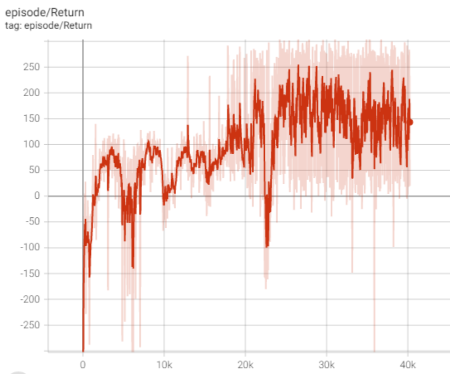


Figure 4.3.1: DDPG Return vs Steps

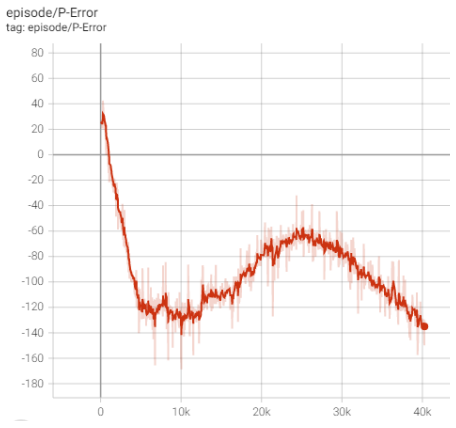


Figure 4.3.2: DDPG Policy Loss

4.4 PPO

The following are the final results obtained by the PPO algorithm.

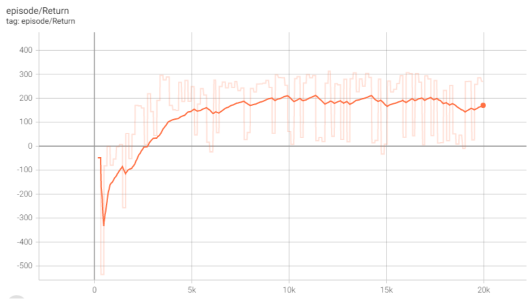


Figure 4.4.1: PPO Return vs Steps

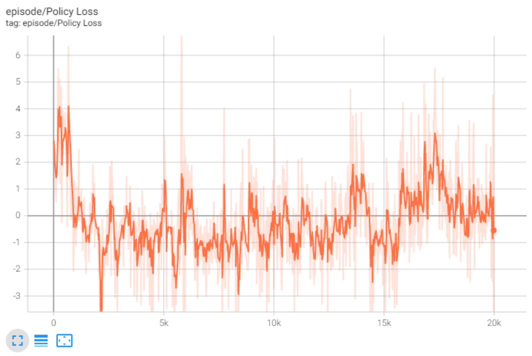


Figure 4.4.2: PPO Policy Loss

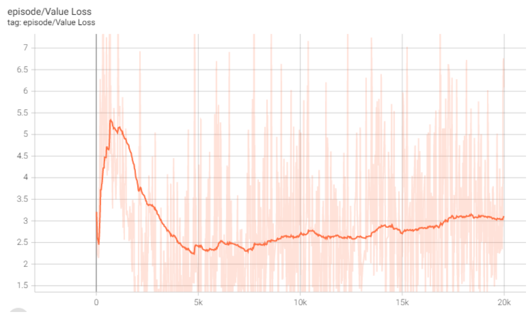


Figure 4.4.3: PPO Value Loss

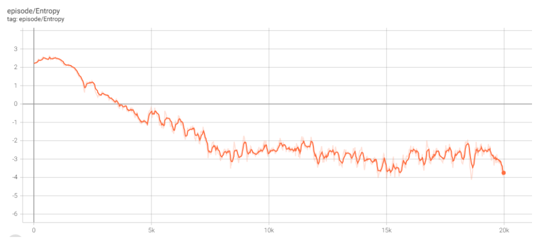


Figure 4.4.4: PPO Entropy

4.5 Comparison

The following is a comparison of the final hyper-parameters found by optuna, as well as a comparison of the results achieved by each algorithm.

Hyperparamaters 1.			
Algorithm	Batch Size	Hidden Size	Num Envs
DQN	256	128	N/A
REINFORCE	256	64	64
DDPG	256	256	N/A
PPO	256	64	64

Table 1: First set of hyper-paramaters

Hyperparamaters 2.			
Algorithm	Gamma	Policy lr	Value lr
DQN	0.99	N/A	1e-3
REINFORCE	0.99	1e-4	N/A
DDPG	0.99	1e-3	1e-3
PPO	0.99	1e-4	1e-3

Table 2: Second set of hyper-paramaters

Training Results			
Algorithm	Best Score	Training Time	Epochs Taken
DQN	296.626	18min	1200
REINFORCE	232	52min	552
DDPG	311.56	26min	1020
PPO	275.89	17min	1550

Table 3: Each algorithms results

5 CONCLUSION

As discussed in section 3.1, for the problem to be considered solved a total of 200 points has to be achieved. Achieving more points past this score means that the Lander is being more efficient and landing at the landing pad more elegantly. Each algorithm for the two experiments managed to converge successfully. For the first experiment, the Deep Q-Network algorithm managed to achieve the best score in the least amount of time without the need for parallelized environments. While REINFORCE did manage to eventually converge, it is good to note that this algorithm proved more sensitive to configurations, sometimes being unable to converge even with a slight change in parameter values. For the second experiment, both PPO and DDPG managed to achieve similar results (Note that one training run may not give the same results as a second run) thus, they both proved effective at solving the problem. What can be noted from Figure 4.3.1 is that DDPG proved to show some instability in its runs, while it did manage to converge, it did so with a very high variance while PPO on the other hand achieved a smoother convergence.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018.
[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.
[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, and others, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.
[4] S. Haykin, "Neural Networks and Learning Machines," Pearson Education India, 2010.
[5] T. M. Mitchell and T. M. Mitchell, *Machine learning*, vol. 1. McGraw-hill New York, 1997.
[6] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.
[7] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, learning, and optimization*, vol. 12, no. 3, p. 729, 2012.
[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, and others, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2016.
[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017.
[11] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.