National and Kapodistrian University of Athens
Interfaculty Program of Postgraduate Studies in Control and Computing,
Faculties of Physics and Informatics

Application of Reinforcement Learning methods in solving a Lunar Landing game

Vasiliki Mavrikou
2020514

Supervisor:
Nikolaos Vlassopoulos, Ph.D

Reviewers:
Dionysios Reisis, Associate Professor
Anna Tzanakaki, Associate Professor

September 2022

Contents

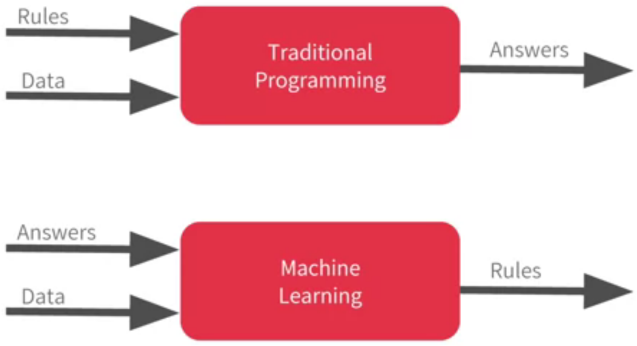
1 Abstract

In this thesis we observe the performance of several Reinforcement Learning algorithms, as well as a PID controller trying to solve a Lunar Landing problem.
First we develop our Lunar Landing game in Python3 using the pygame module.
Then we implement and compare the performance of Deep Q-Learning and Double Deep Q-Learning algorithms trying to solve it. For the development of our learning agent, we create a custom Gym environment based on the game we created, using OpenAI's gym module. Moreover, we review how different reward functions affect the agent's learning process.
Finally using the same custom Gym environment, we develop and tune a PID controller, that tries to control the altitude and the angle of the lander, in order to guide it to a soft landing.

A Lunar Landing problem is the task of controlling the fire orientation engines of a spacecraft in order to carefully land it on the landing pad. In our implementation, the player can maneuver the lander in a two-dimensional (2D) space using four possible actions: do nothing, fire the right or the left orientation engine in order to change the landers' angle, and fire the main engine. The landing pad is always in the same position, in the middle of the image, while the shape of the rest of the landing surface is randomly generated at the beginning of every new game. Our goal is to train the agent so that it achieves a soft pinpoint landing in all situations. We define a soft landing as the magnitude of the landing velocity below 1m/s, the velocity vector directed primarily downward with negligible deviation from an upright attitude and zero rotational velocity.

2 Machine Learning (ML)

Machine learning is an application of Artificial Intelligence that enables systems to learn and improve from experience without being explicitly programmed. Specifically, ML focuses on the development of models that use data to learn for themselves through a training process. During this training, we supply our model with training data, the model processes it, looking for patterns in it, trying to understand entities, domains and the connections between them. As a result of that process, we get a model that is trained to make inferences or predictions on any similar, unseen input we feed to them.



This makes it possible for computers to learn autonomously and adjust actions, without human intervention or assistance. In our case, those actions should lead to winning our Lunar Lander game. There are many ways to train ML models, and one of them is Reinforcement Learning:

3 Reinforcement Learning (RL)

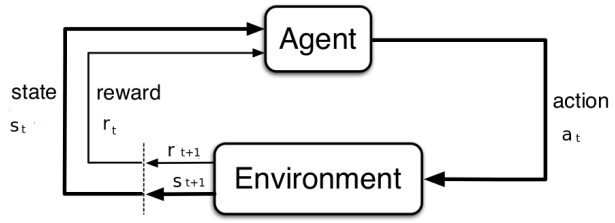
Reinforcement Learning is the training of ML models to make a sequence of decisions. Specifically, RL refers to goal-oriented algorithms, which learn how to achieve a complex objective (goal) by maximizing a given reward over many steps; for example, they can maximize the points won in a game over many moves. That is exactly what we want in our case as well: we want the computer to learn the steps/actions that will eventually lead to maximization of the gained score, which practically means winning the game by safely landing the spacecraft on the landing pad, for all shapes of landing surface.
RL involves an agent that learns a strategy, that is, a sequence of actions, by interacting with an environment. During training, the agent takes actions and interacts with its environment. Moreover, RL is based on the existence of a rewarding system that assesses the quality (usefulness) of every action, and updates the strategy (action selection policy) accordingly. In this way, the agent learns on its own which actions help, and which don't help it reach its goal, without getting any hints or suggestions from us on how to win the game. The only way we are involved is by defining the reward function at the beginning of the training process.

RL is based on the hypothesis that our goal can be achieved with the maximization of the agent's long-term accumulative reward.

In this manner, all the agent needs to do is learn to map states of the environment to actions, so that it eventually discovers the sequence of actions that maximizes its reward. The training process starts with the agent choosing totally random actions and can end up with it developing some quite sophisticated tactics which may even lead to superhuman performances.

4 Markov Decision Processes (MDP)

Markov Decision Processes provide a mathematical framework for modeling RL problems. So, in our model of an agent learning to achieve a goal by interacting with its environment, we consider that each such interaction takes place in discrete time steps t = 0, 1, 2, 3, ... At each time step t, we consider the process being in some state s_t, and the agent choosing to take an action a_t, based on a mapping policy $\pi(a_t|s_t)$. The process responds at the next time step by moving into a new state s_{t+1} using state transition probability $P(s_{t+1}|s_t, a_t)$, and also by giving the agent a corresponding reward r_{t+1}, according to a reward function R(s, a). This reward affects its total reward, thus determining whether the decision of choosing action during time step t, was good or not. During a game/episode, rewards are given at each step, until the agent reaches the terminal state, either by winning or losing the game, and then this process is repeated as another episode begins.
The goal of the agent is to maximize the long-term return expectations of each state, and thus its total reward.



In the case that sets S,A,R all contain a finite number of instances, then we say that we have a finite MDP, and s_t and r_t are random variables of discrete probability distributions that only depend on s_{t-1} and a_{t-1} . In other words, we consider that at each time step t-1, the environments next state s', depends only on the current state s and the agents chosen action a, while it is conditionally independent of all previous states and actions. In this case, we say that the state transitions of our MDP satisfy the Markov property: the probability that we get a pair (s', r) during time step t depends only on pair (s, a) of time step t-1, and not on any other previous states and actions. We define the probability that action a in state s during time step t-1, will lead to state s' at time t, as:

$p(s', r|s, a) = \Pr[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a]$, for all: s, s'∈S, r∈R, a∈A(s). So function p : SxRxSxA → [0,1] is a probability distribution for all pairs (s,a), and thus we have:

$\sum_{s' \in S} \sum_{r \in R}$ p$(s', r|s, a) = 1$, for all: s, s'∈S, a∈A(s).

5 Reinforcement Learning definitions

RL introduces many new concepts. Some of them, are thoroughly explained and connected to our Lunar Lander task. In what follows, we use capital letters to denote sets of things, and lower-case letters to denote a specific instance of that thing; e.g. A is all possible actions, while a is a specific action contained in the set.

- Agent: it is practically the algorithm, the one that takes the actions. In our case the agent is represented by the spacecraft.
- Action a ∈ A(s) : A is the set of all possible moves the agent can choose from and perform. Set A consists of 4 discrete actions a: do nothing, fire the right orientation engine, fire the left orientation engine, fire the main engine.
- Environment: The world through which the agent moves, and which responds to the agent taking an action. The environment takes the agent's current state and action as input, and returns as output the agent's reward and its next state. In our case, the environment can be thought to be the laws of physics that process the agent's actions and determine the consequences of them, the landing pad, as well as the lander's fuel which diminishes each time we fire the main engine.
- State s ∈ S: s is a concrete and immediate situation in which the agent can find itself; i.e. a specific place and moment, an instantaneous configuration that puts the agent in relation to its environment. The state space for our Lunar Lander implementation is an eight-dimensional vector containing information about:
  - the agent's position in space, x and y
  - its horizontal and vertical velocities, vx and vy
  - the agent's angle θ
  - the agent's angular velocity w
  - two flags: left-contact-ok and right-contact-ok indicating whether the left and the right foot of the lander respectively, are in contact with the landing pad
- Reward r ∈ R: A reward r is the feedback by which we measure the success or failure of the agent's chosen action a in a given state s. From any given state s, the agent sends an output a in the form of an action to the environment, and the environment returns the agent's new state s' which is the result of this

action a, as well as a reward r' that effectively evaluates the agent's decision to choose action a.

- Goal: The Markov Decision Problem is considered solved when the agent has adopted a policy that results in the average score on a number of consecutive landing attempts being equal or above the winning score (which is defined by the reward function).

- Expected discounted return G_t: the goal of the learning agent at every time step t, is to choose the a_t that will lead in the maximization of its total reward, that being the sum of the rewards r_t it gets at every time step t, until the end of a game/episode. In this manner, we define the expected discounted return as: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{T} \gamma^k R_{t+k+1}$, where:
  - $0 \leq \gamma \leq 1$ gamma is a parameter called discount rate. All future rewards are multiplied by this γ, in order to dampen their effect on the agent's choice of action, the further into the future they occur. So, G_t is designed to make future rewards worth less than immediate rewards. A discount factor of 1 would make future rewards worth just as much as immediate rewards
  - T is the last time step of an episode

- Policy π: A policy is a strategy that the agent employs to determine its next action based on its current state. It is a mapping: (state) $\rightarrow$ (probability to choose an action), particularly, the action that promises the highest reward r at time step t. Policy defines the agent's behavior during all phases of training, and it changes as the agent gains more experience through the interaction with its environment.
  Solving an RL problem practically means finding an optimal policy, that being a policy that results in gaining the highest expected return for all states.

- Action value function for policy $\pi : Q_\pi(s, a)$: is the expected discounted return we get if we start from the current state s choosing action a, under policy π. It maps state-action pairs to rewards, and is defined as:

  $$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{T} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

  We define this expected long-term discounted return, as opposed to the short-term reward R. Reward is an immediate signal that is received in a given state, while $Q_\pi(s, a)$ is the sum of all rewards we might anticipate from that state. It is a long-term expectation, and it differs from an immediate reward in their time horizons.
  There can be states s, where $Q_\pi(s, a)$ and reward may diverge. For example, the agent might receive a low, immediate reward even when it moves to a position with great potential for long-term value; or instead, it might receive a high immediate reward that leads to diminishing prospects over time. And this is exactly the reason why $Q_\pi(s, a)$, rather than immediate rewards, is what RL

seeks to predict and control.

- Optimal action value function Q*(s,a): is the expected discounted return we get if we start from the current state s choosing action a, under an optimal policy . This is what we need in order to reach our goal: to win the game:
  $Q^*(s,a) = max_\pi Q_\pi(s,a) = E[R_{t+1} + \gamma max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a] =$
  $= \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} Q^*(s', a')]$

Generalized Reinforcement Learning algorithms do well in finding optimal policies for stochastic Markov Decision Problems (MDPs).
Q-Learning is a popular RL algorithm:

6 Q-Learning

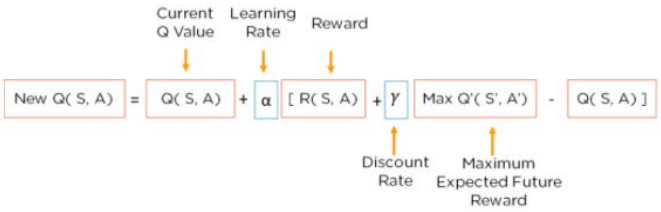
It is a straightforward approach to learn state-action pairs. Q-learning maintains a Q-table that is iteratively updated in order to maximize the reward for every state it encounters.
According to this algorithm, the optimal action-state value Q*(s,a) is the current reward r, and the discounted maximum future reward possible. This strategy is captured by the Bellman Equation:
$Q^*(s,a) = E_{s'}[r + \gamma max_{a'} Q^*(s', a') | s, a]$

In the case of finite discrete input spaces, the number of Q values to compute would be finite and the problem can be solved iteratively in a dynamic programming styled approach:
$Q_{i+1}(s,a) = E[r + \gamma max_{a'} Q_i(s', a') | s, a], \lim_{i \to \infty} Q_i = Q^*$

So, according to the Q-Learning algorithm, we can approach optimal action value function A*(s,a) using:

$A(s_t, a_t) \leftarrow A(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{a_{t+1}} A(s_{t+1}, a_{t+1}) - A(s_t, a_t)]$, where:

- $Q(s_t, a_t)$ is the action value function during step t
- r_{t+1} is the reward for the action taken during step t
- $0 < \alpha \leq 1$: is the learning rate specifying the rate measure at which the old value will be replaced by the new value
- $max_{a_{t+1}} Q(s_{t+1}, a+t+1)$ is the maximum value of the action-value function at step t+1



In the Q-Learning case, the algorithm attempts to learn the optimal action value function Q*(s,a). When the action space is discrete, for each state-action pair, there exists a separate Q*(s,a) value, and so we can create a look-up table with the rewards of each pair of state-action, which we will be constantly updating during training.

The studied action-value function Q directly approximates the optimal action-value function Q*, regardless of the policy followed. This simplifies algorithmic analysis and enables early proof of convergence. The policy still affects the determination of which state-action pairs to visit and update. However, all it takes for that convergence, is that all pairs are constantly being updated. This is a minimum requirement in the sense of what method is guaranteed to find the optimal behavior in the general case should require it. Under these assumptions and variants of the usual stochastic forecast conditions on the order of the step size parameters, Q has been shown to converge to Q*.

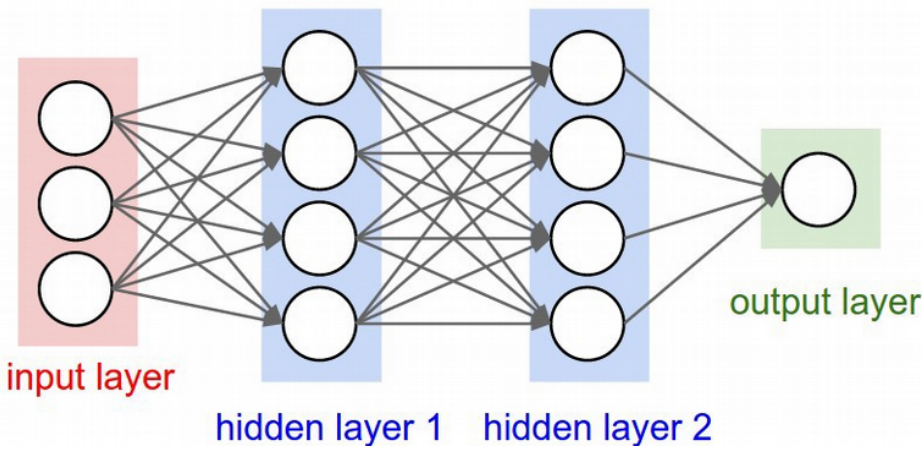
For MDP problems with small state and action space, Q-Learning is an effective solution. However in most cases where the state or the action space is very large or stochastic, it usually makes traditional RL algorithms slow down or fail to converge to optimal policies. In our Lunar Landing case, where the state representation is continuous, employing a Tabular Method (Q-table) to solve this MDP is practically hard.

Instead, we will be exploring a much simpler solution for our Lunar Landing Environment, which involves using function approximation techniques.

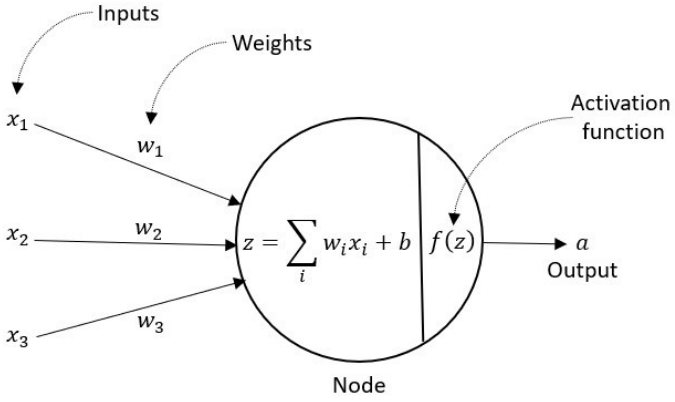
In what follows, we will be discussing Deep Neural Networks as function approximators. Then, then we are going to implement our own model which we will use as a value function approximator that given the agent's states we feed to it, it will be providing us with value predictions for Q*, in order to extend the capability of the Q-Learning method.

7 Artificial Neural Networks (ANNs)

Artificial Neural networks are a subset of machine learning and the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.
ANNs are structures comprised of layers of nodes: they contain an input layer followed by one or more hidden layers, and then an output layer.
They accept data as input, which flows through the nodes of the network and gets processed in order to produce new information as output, which will most probably be a prediction/decision based on the provided input and the problem we are trying to solve.



Each node of the network performs a relatively simple calculation. However, deep ANN architectures (those with many hidden layers) are capable of solving quite complicated problems.

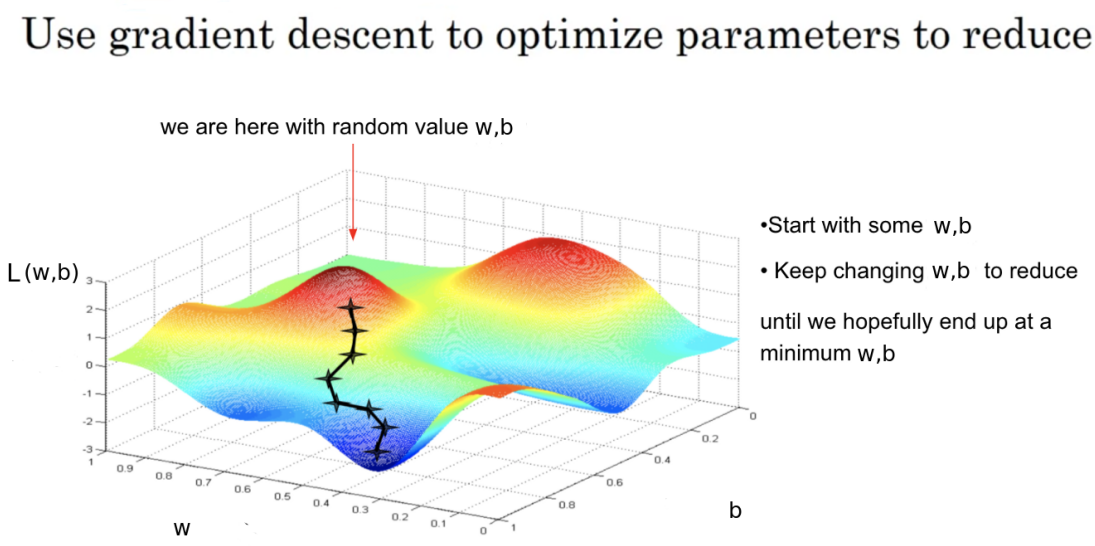


The output we get from our model depends on the values of vectors w_i, b of the network's nodes. Those values are not known to us in the beginning, when we start developing our model, and so we initialize them with random values.

In order to find the right values for vectors w_i, b that make our model capable of producing useful results that help us solve a problem, an ANN needs to be trained.
Training of ANNs is an iterative procedure during which we present some training data to our model, which it processes, and learns from them. This data is usually sets of input, as well as its corresponding output pairs (the data we would ideally want our network to be able to produce by itself), and through this learning process, the values of w_i, b are constantly being adjusted so that our model becomes more and more capable of performing the mapping:
(input data) $\rightarrow$ (output data), by itself. We call vectors w_i, b training parameters.

During training we supply the input training data to our model, and we get an output $\hat{y}$ which is our models prediction for the corresponding input. Then we compare this prediction $\hat{y}$ with the true output value y (target value) which is also part of our training dataset. We use a loss function L in order to perform the comparison between y and $\hat{y}$.

What we actually want is to minimize the difference between the values of y and $\hat{y}$. So the whole training process practically is a minimization procedure of our loss function L($\vec{w}$,b), with respect to our training parameters $\vec{w}$ and b. We usually do that using the gradient descent algorithm. Gradient descent is an iterative optimisation algorithm used to finding a local minimum of a differentiable function.



Our goal is that after the end of the learning process we get a model that is capable of generalizing, meaning to be able to produce a correct output for all input data we feed to it (both training-labeled and test-unlabeled data).

8 Deep Q-Learning

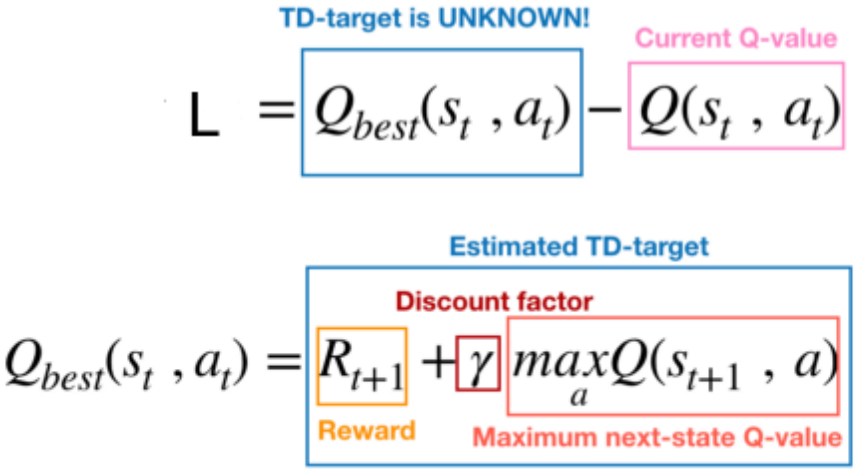
Deep Q-learning algorithm involves a deep Neural Network (DQN) that learns an optimal policy for a finite MDP. The network takes a state vector s as input, and it returns an estimation for Q*(s,a) for all 4 possible actions. Consequently, the agent performs the action corresponding to the largest estimated Q*.

During training, at each step of the process, we feed our model with the agent's current state, and according to the policy followed, we get a prediction Q for the next best action. The agent performs this action, and gets a reward that assesses whether the our models prediction was good or not. Accordingly, this may lead to changes in the training parameters and thus the followed policy. By continuously updating the neural network's parameters, we can eventually approximate the optimal path of actions (optimal policy) that leads to solving the MDP. The neural network is trained by minimizing the Mean-Squared Error (MSE) for the loss function:

$L = [(r + \gamma Q^*(s', a')) - Q(s, a)]^2$, where:

- Q(s,a) is our model's prediction
- $(r + \gamma Q^*(s', a'))$ is the target value we use for fitting

This way, throughout multiple iterations, the Q-network is smoothened and an optimal Q-function is learned.



Unlike supervised learning where the true target values of our training set are part of the training dataset, and thus they are known to us, in our case, we have to produce them ourselves, using the same model that also makes the predictions. This makes our target values dependent to the weights of our model, which of course are being changed during the training process.

When implementing the Deep Q-Learning algorithm, there are some techniques we employ in order to facilitate the learning process:

Epsilon-Greedy Exploration-Exploitation Strategy

Exploration (selecting a random action) and exploitation (selecting a greedy action, that is, the action that yields the highest short-term reward) is a crucial component of the Deep Q-Learning algorithm, as it helps make sure that the agent explores a wide range of all possible state-action combinations.
It is a strategy we employ in order to chose the agent's next action during training. According to it, during every step of the training process, the agent takes a randomly chosen action (random policy) with probability ϵ, and a network-based action (optimal policy) with probability (1-ϵ). At the beginning of the learning process the value of ϵ is equal to 1, meaning that the agent takes only random actions (exploration phase), but it decays over time (with an epsilon decay rate ϵ_d).In this way, as the training process goes on, the agent relies more and more on the decisions made by our model (observation phase). Exploring random actions for a long time will slow down the training process.

Experience Replay - Batch Learning

In a reinforcement learning environment, sequential states are strongly correlated. This is very true in our problem as well: the action the agent takes at one step, is strongly correlated to the actions it takes in the near future and past. However, it is generally more efficient to train our model with states that are more independent one from the other. In order to do that, we use an experience replay buffer: during training, for every state s of the agent that we feed to our model, we keep the values we get for (s,a,r,s') to a replay memory.
Furthermore, at every step, we also randomly pick a number (batch size) of those (s,a,r,s') values from replay memory and we feed them to our model in order to both produce the target values, and also fit our model to them. This random sampling we perform using the replay memory, helps us tackle the problem of autocorrelation which can lead to unstable training.
In addition to that, experience replay utilizes history experience more efficiently, as it reminds older states to our model, making it learn them multiple times.

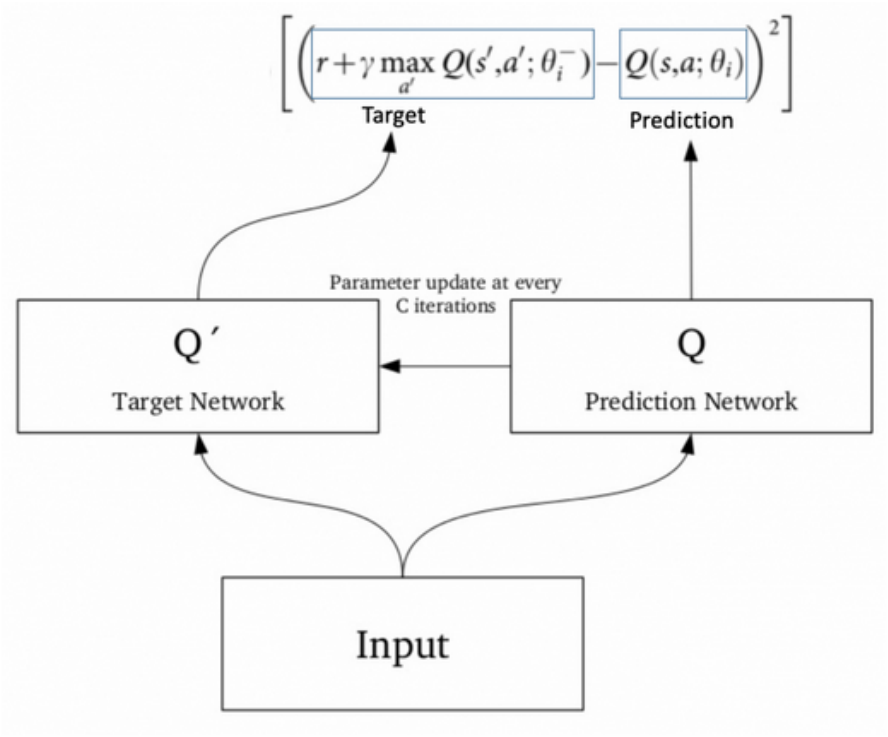
Target Network

In order to avoid training our model using target values that are constantly changing, we can perform training using two ANN architectures, instead of just one: our main network (Model) and a target network (Target Model).

- Model: as described before, we use a neural network that takes our agents states as inputs, and it gives an estimation for its next best actions. During training, as the agent explores more and more of its environment, the Model is learning better state-to-action mappings, and so its estimations for Q*(s,a) are improving with time.
- Target Model: this network is a clone of our Main Model, and it is used in

order to produce the target values used for the fitting of the main model.

More specifically, as before, we train our Main Model in order to use it as an approximator for the optimal action value function Q*(s,a). During training, we only train this Main Model, with its weights being updated from the difference between its predicted value for current state s, and the maximum possible rewards for next state s' (target values), produced by the Target Model.
So, we only use the Target Model during training, for producing the target values used for the fitting of the Main Model. The Target Model doesn't get trained at all, however, every now and then we update its weights by replacing them each time with the corresponding values of our Main Model's training parameters, in order to improve the quality of the target values it produces.
We use the same network architecture for both Model and Target Model.



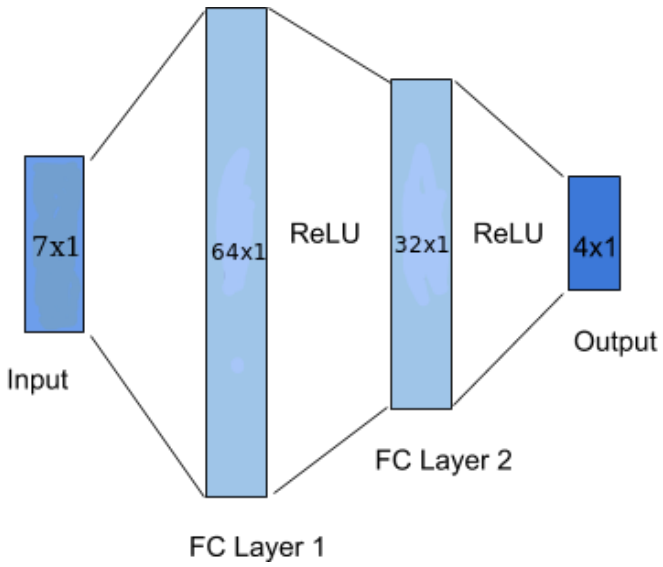
In this way, we get a more stable learning environment, where the weights of the Target network are updated every so many episodes, with a copy of the Main network. The update frequency (upd-freq) of the weights of Target Model, is another parameter of this whole model, comprised of 2 ANNs.

In what follows, we are implementing the Deep Q-Learning algorithm using a deep ANN architecture (Deep Q-Learning Network- DQN). We will implement DQN both with (Standard DQN) and without (Vanilla DQN) the use of a Target network. After that we are going to compare the results we get from our models.

8.1 Deep Q-Network (DQN)

For our problem, we construct a relatively simple Neural Network comprised of:

- Input layer: 8-dimensional state vectors
- Hidden layer 1: 64 nodes with 'ReLu' as activation function
- Hidden layer 2: 32 nodes with 'ReLu' as activation function
- Output layer: 4 nodes, with 'linear' activation function



We optimize our model using Adam optimizer with learning rate α and batch size bs (the number of training samples we use each time for the fitting of our model). Every episode will be finished either when the agent wins or loses the game (done=True from the gym environment), or when the episode's steps reach a predefined maximum step number, equal to 2500.

Our model has many hyperparameters that need to be tuned. The most important of them are: batch size bs, learning rate α, epsilon decay ϵ_d , discount rate γ and Target network update frequency.
There isn't a rule of thumb when choosing the values of such parameters, so before we start training our model, we first need to perform some experiments (hyperparameter tuning), in order to find out the effect that different values of those paramereters have on our model's performance, and then tune it accordingly. After that, we can start training our final model.

8.2 Reward Function

As we have already mentioned before, the only way we intervene to the learning process, is by defining the reward function that assesses the actions of our agent. The reward function is a rule defined by the nature of the problem we are trying to solve, with the purpose of directing the search of the agent towards the goal, so that it becomes more efficient.

In our case, we are going to observe the performance of our agent for two different reward functions:

Reward Function 1:

This reward function is the one provided by the Gym module for its LunarLander-v2 environment:

```
reward = 0

shaping = (
    - 100 * np.sqrt(x * x + y * y)
    - 100 * np.sqrt(vx * vx + vy * vy)
    - 100 * abs(0)
    # 10 points for legs contact
    + 10 * left_contact_ok
    + 10 * right_contact_ok)

if self.prev_shaping is not None:
    reward = shaping - self.prev_shaping

self.prev_shaping = shaping

# Encourage lander to use as little fuel as possible
if(fire_main_engine):
    reward -= 0.30
elif(fire_side_engine):
    reward -= 0.03

done = False

# You Lost, out of screen or fuel
if((lander_exploded) and (not lander_hit_terrain)):
    done = True
    reward = -100.0

if (lander_hit_terrain):
    done = True

    # You Won
    if(landeR_landed):
        landing_reward = 100.0
        # Multiply percentage of remaining fuel
        reward = landing_reward * fuel_conservation

    # You Lost, out of landing pad
    elif(lander_velocity_ok and lander_angle_ok):
        reward = -10.0

    # You Lost, lander crashed
    else:
        reward = -100.0
```

During each step of an episode, the agent gets a reward defined by the values of its current state vector. Particularly, the purpose of the shaping vector we define above, is to capture how the of the agent's state's values change from

step to step and reward it accordingly (considering the fact that ideally, at the end of an episode we want the values of x, y, vx, vy, θ and w to be equal to 0, and the values of left-contact-ok and right-contact-ok to be equal to 1).
Moreover, at each step the agent will get a reward of -0.3 points for firing the main engine, or a reward of -0.03 points for firing one of the side engines, in order to discourage it from spending too much fuel.
At the end of an episode, the agent gets a reward of +100 for safely landing on the landing pad (and thus winning the game), or a reward of +10 for safely landing outside the landing pad, or a reward of -100 points if it crashes on the surface of the moon, or if it runs out of fuel.
Each leg with ground contact is +10 points.
The agent receives a total reward of over 400 points for safely landing on the landing pad, thus winning the episode.

Reward Function 2:

The second reward function we are going to try out is much simpler than the previous one:

```
# Encourage lander to use as little fuel as possible
# total fuel = 500
fuel_conservation = self.game.lander.fuel / 500

reward = 0.0

# Encourage agent to approach the surface instead of hanging in the air
if(vy >= 0):
    distance_reward = 1 - abs(y)**0.5
    reward = distance_reward * fuel_conservation

done = False

# You Lost, out of screen or fuel
if((lander_exploded) and (not lander_hit_terrain)):
    done = True
    reward = -100.0

if (lander_hit_terrain):
    done = True

    # You Won
    if(landeR_landed):
        landing_reward = 100.0
        # Multiply percentage of remaining fuel
        reward = landing_reward * fuel_conservation

    # You Lost, out of landing pad
    elif(lander_velocity_ok and lander_angle_ok):
        reward = -10.0

    # You Lost, lander crashed
    else:
        reward = -100.0

```

During each step of an episode, the agent gets a (positive) reward only if the lander moves downwards (which can be translated to vy being > 0).
At the end of an episode, the agent gets a reward of +100 for safely landing on the landing pad (and thus winning the game), or a reward of +10 for safely landing outside the landing pad, or a reward of -100 points if the lander crashes on the surface of the moon, or if it runs out fuel.
All of the positive rewards the agent gets are multiplied by the fuel-conservation factor, in order to encourage it to spend as little fuel as possible.
The agent receives a total reward of more than 400 points for safely landing on the landing pad, thus winning the episode, with the variation range depending on its trajectory until the end of the episode.

8.3 Difficulties we faced / Other experiments

When trying to design the custom environment as well as the ANN model in order to solve our Lunar Landing problem, we faced many difficulties. Most of them arise from the fact that we train our models on a CPU, which results in long training hours (on average, training our model for 1000 episodes, using the architecture defined in section 8.1, takes up to 10-12 hours).
This prevents us from training our models for too many episodes, which in many cases is required in order to reach convergence. Moreover, it doesn't allow us to perform all of the experiments needed in order to explore as many combinations of ANN architectures, hyperparameter values and reward functions as possible, until we find the optimal one.

However, given the time we had, and the computational resources available, we tried to perform as many experiments as possible, before we make our final decisions on the models we are going to use. More specifically:

- We tried using a different network architecture, comprised of 2 hidden layers of 512 and 256 nodes respectively, only to find out that this more complex model needed double the time to be trained, and the performance of the agent didn't show any signs of improvement, comparing to our chosen, much simpler architecture.
- Before we start training any model used, we perform hypeparamater tuning. During tuning, we try out three different values for each different hypeparameter. For each such value, we train our model for 350 episodes and record the cumulative reward of the agent on each episode, in order to compare the results we get for each of them. In the following table, we present the values we are going to try out for each hyperparameter:

epsilon decay	0.990	0.995	0.999
gamma	0.8	0.9	0.99
learning rate	0.0001	0.001	0.01
batch size	32	64	128
update frequency of Target net	5	20	50

9 Double Q-Learning

According to the Q-Learning algorithm, the optimal policy of the agent is always to choose the best action in any given state, that being the one with the maximum expected/estimated Q-value.
However, at the beginning of the training process, the agent knows nothing about its environment, yet it still has to somehow estimate Q(s,a), in order for it to get updated with each iteration. Such estimated Q-values are very noisy and we can never be sure whether the action with maximum expected/estimated Q-value is really the best one. As a matter of fact, in most cases, the best action often has smaller Q-values against the non-optimal ones, which results in the agent taking a non-optimal action in a given state, only because it has the maximum Q-value. Such problem is called overestimation of the action value (Q-value).
Moreover, the target values we use, are also estimated using the reward added to the next state maximum Q-value we have which is noisy. All this results in a messy Loss Function that contains positive biases caused by different noises.
This has a large impact on the weight update procedure and thus the learning of our model, which can be very complicated and messy.

Seperating the process of selecting the optimal action from the estimation of optimal actions is a simple way to reduce the impact of overestimation as much as possible. The Double Q-Learning algorithm tries to address this concept by using two different action-value functions, Q and Q', as estimators:

* Q function is for selecting the best action a with maximum Q-value of next state
* Q' function is for calculating expected Q-value, using the action a selected above

The update procedure is slightly different from the basic version, as we update Q function using the expected Q-value of Q' function as demonstrated below:

Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \boxed{Q'(s_{t+1}, \boxed{a})} - Q(s_t, a_t))$$

estimated/expected Q-value

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

Using these independent estimators, may cause an underestimation of expected values, but it could help the agent to yield convergence to optimal policies faster in comparison to Q-learning.

Double Q-Learning implementation with Deep Neural Network is called Double Deep Q-Network (DDQN). DDQN uses two identical networks, a Behavioral and a Target model:

* The Behavioral model (qnet) learns from experience replay during the training process, and it is used for the selection of the best action α, by taking the maximum Q-value of next state:

$$a = \max_{a} Q_{qnet}(s_{t+1}, a)$$

* The Target network (tnet) is used for calculating the estimated/expected Q-value with action α selected by the Behavioral model.
  This Q-value is then used in order to update the Q-value of the Behavioral model:

calculate estimated Q-value with action from QNET by using Target Network

$$Q_{qnet}(s_t, a_t) \leftarrow R_{t+1} + \gamma\, Q_{tnet}(s_{t+1}, a)$$

  Target model is not trainable, but we update its learning parameters every now and then (according to the Target network update frequency parameter: upd-freq), by replacing them each time with the corresponding weights of the Behavioral model.

For our Behavioral and Target models, we use the same network architecture we defined in section 8.1

10 Implementations and Results

In what follows we present the results we got from the implementation of: (1)Vanilla DQN, (2)Standard DQN, and (3)DDQN algorithms as they are described above.

10.1 Vanilla DQN

We implement the Vanilla DQN algorithm for reward functions 1 and 2. In both cases we train our agent for 1000 episodes using the same model. Before training, we conducted experiments in order to tune the values of some important training parameters of our model, which we present in the following table:

epsilon decay	0.995
gamma	0.99
learning rate	0.001
batch size	128

To analyze the training process, we record the cumulative reward of each episode, as it reflects the change of the agent's control ability during training:

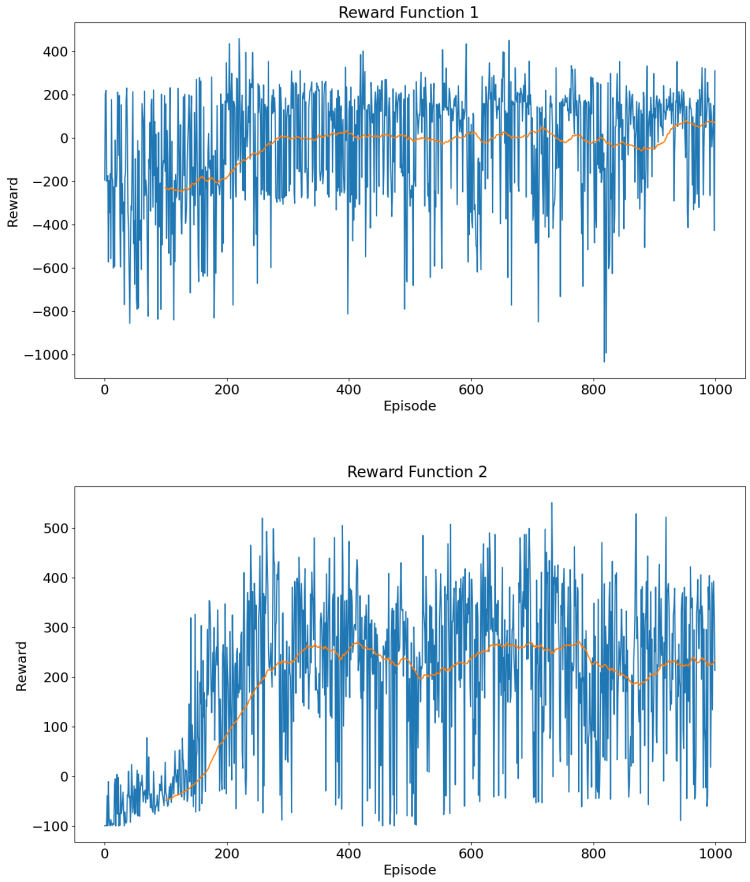


Fig 1: Rewards for each training episode

Fig 1 shows the reward values per episode during training. The blue line-denotes the rewards for each training episode, and the orange line shows the average reward of the last 100 episodes.

In both cases we can observe that for the first 300 episodes the value of the average reward increases, which means that the agent keeps learning with time.
Specifically, the average reward in the earlier episodes is mostly negative, as the agent has just started learning. Eventually, it starts performing relatively better and the average reward goes up and becomes positive. After about 300 episodes, the orange line becomes more stable as the average reward practically stops increasing significantly, meaning that the agent doesn't learn much for the rest of the training process. At this point of training, ϵ has faded almost completely. However, if its value was a little higher, and the agent had the opportunity to explore some new state-action combinations by picking a random action every now and then, maybe this would help unstuck the learning process.
Despite the fact that during training we observe the mean reward increasing with time, and even our agent winning on some episodes, it doesn't manage to converge to a winning policy. This can also be seen from the testing of our trained models. Indicatively, we present the performance of our trained models during testing for 100 episodes:

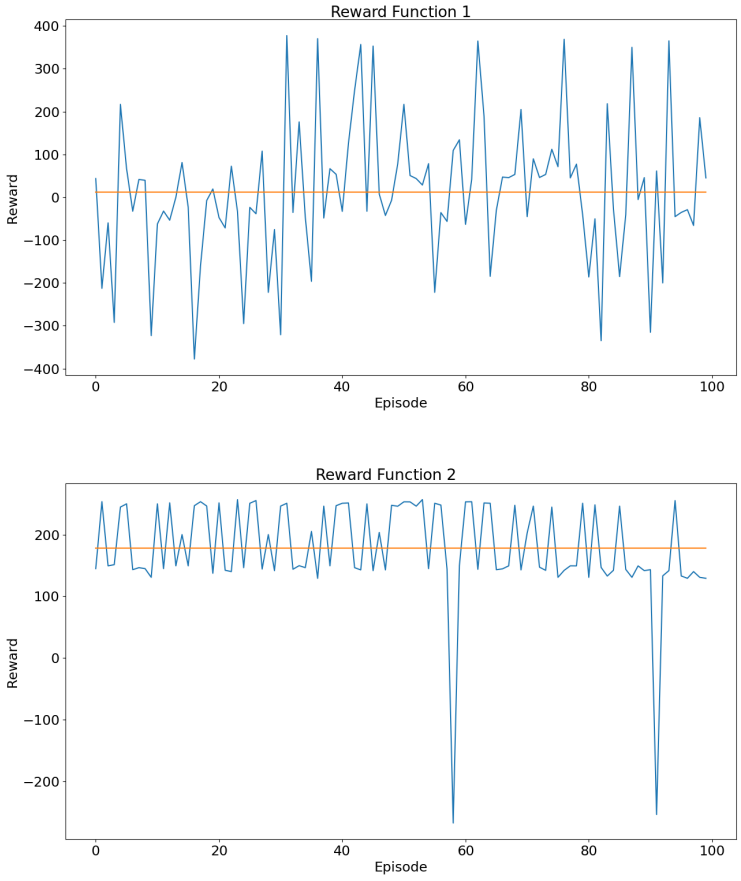


Fig 2: Rewards for each testing episode

By comparing the performance of our agent for the two different reward functions, we can easily observe that even thought Reward Function 2 is much simpler than Reward Function 1, it manages to outperform it. This manifests in the graphs we get for the training and the testing processes:

Specifically, for Reward Function 2, we observe that the average reward of our agent increases by more than 300 points during the first 300 episodes of the training process, and then it becomes more stable, oscillating from 200 to 300 points for the rest of training. Despite the fact that during training the average reward never reaches 400 points (the minimum winning score for an episode), and so agent doesn't find a winning policy, it still manages to win on many seperate episodes and even get rewards of over 500 points. Fig 2 supports all of the above, as it shows that during testing the agent performs well enough to get positive rewards of $\sim$ 180 points on almost all testing episodes, which means that our model has improved during training. However it hasn't converged to a winning policy, as the agent doesn't manage to win on any episode.

With Reward Function 1, during training we get a smoother average reward which manages to increase by 200 points in the beginning of the process, before it stabilizes after the first 300 episodes to an average reward that is slightly above 0, for the rest of the training process.
As a matter of fact, the two reward functions we employ, have different ranges of values of the total rewards per episode they offer, and so we can't make direct comparisons between them. However, from Figs 1 and 2, we can clearly see that the performance of our agent is worse under this reward function.
More precisely, despite the fact that in the beginning of the training process, the average reward starts from a lower point and it clearly increases, until it finally stabilizes to a positive score, this is far from the average score of more than 400 points, needed in order to assume that our agent has found a winning policy. Moreover, unlike the case of Reward Function 2, here, we observe that during the whole training process, the agent rarely manages to win an episode, even on seperate occasions.
From Fig 2, we observe that during testing, our trained model gets an average reward that is slightly above 0. This is far from converging to an optimal policy, however we can see that our model has indeed learnt something, as the minimum rewards it now receives are much better than the worst ones it got during training.

10.2 Standard DQN

We then implement the Standard DQN algorithm for reward functions 1 and 2. In both cases we train our agent for 1000 episodes using the same model. Before training, we conducted experiments in order to tune the values of some important training parameters of our model, which are presented below:

epsilon decay	0.99
gamma	0.9
learning rate	0.001
batch size	128
update frequency of Target net	20

To analyze the training process, we record the cumulative reward of each episode:

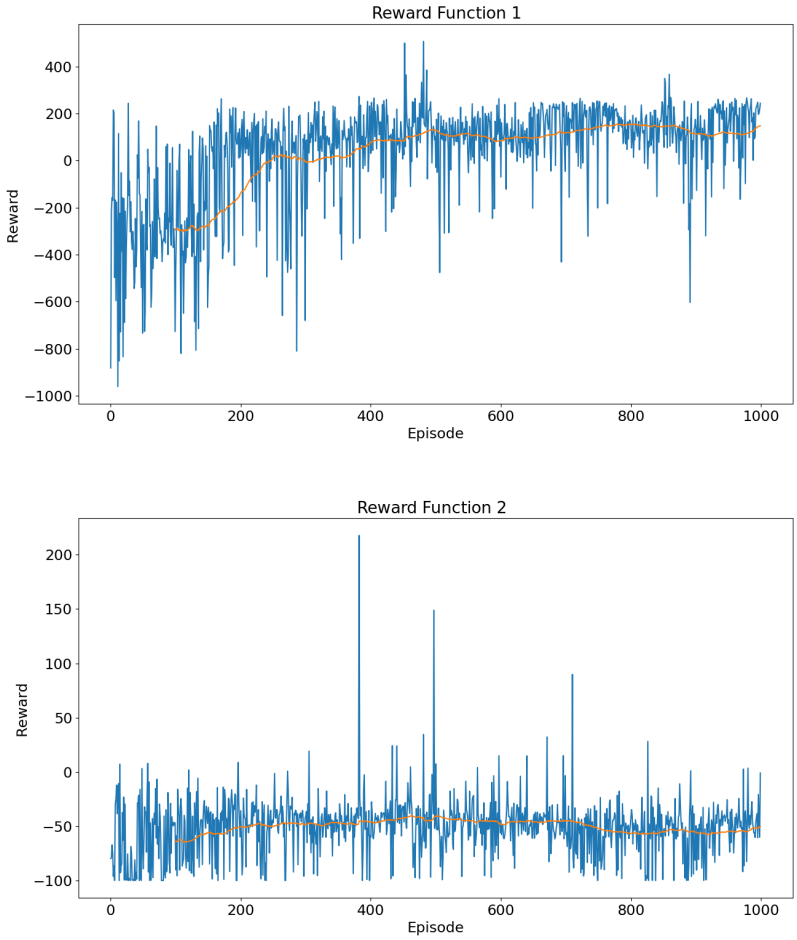


Fig 3: Rewards for each training episode

As we can observe from the graphs, the results we get for Reward Function 1 are very similar to the ones we got from the implementation of the Vanilla DQN algorithm: during the first 250 episodes, the value of the rolling mean significantly increases by about 300 points. After that point, it becomes more stable, oscillating from 0 to 100 points for the rest of the training process. This means that the agent kept learning almost until the end of training.
On the other hand, the results we get for Reward Function 2 are much different, as the average reward doesn't seem to increase during the whole training. This means that the agent hasn't learnt much, as the rewards it gets from episode 100 until the end of training, when the value of epsilon has decayed enough for us to consider that the agent has entered the exploitation phase, become only slightly better than the ones it got in the very first episodes, during the exploration phase, where the actions it took were mostly random.
In the following graphs, we present the performance of our trained models during testing for 100 episodes:

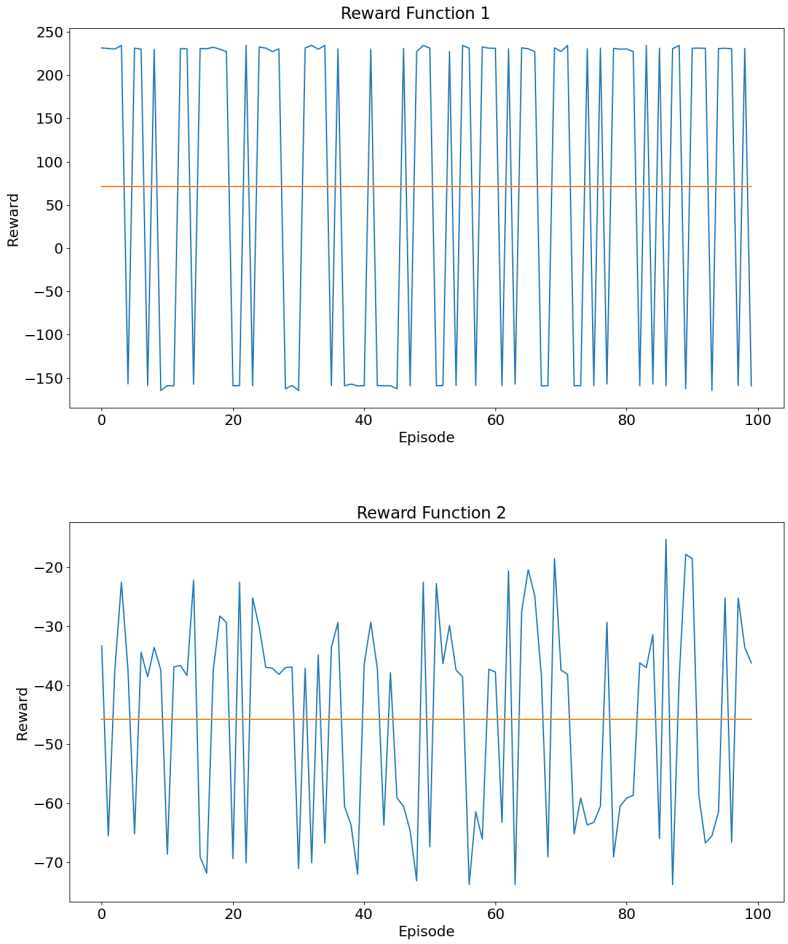


Fig 4: Rewards for each testing episode

By comparing the performance of our agent for the two different reward functions, it is obvious that Reward Function 1 performs much better than the second one.

Specifically, for Reward Function 1, we observe that despite the fact that in the beginning of the training process, the average reward is clearly increasing, until it finally stabilizes to a positive score, which is also higher than the one achieved with the Vanilla DQN implementation, this is far from the average score of more than 400 points, needed in order to assume that our agent has found a winning policy. As a result, during the whole training process, the agent rarely manages to win even on seperate occasions.
From Fig 4, we observe that during testing, our trained model gets an average reward of ≈ 75 points. This is far from converging to an optimal policy, however we can see that our model has indeed learnt something, as the minimum rewards it now receives are much better than the worst ones it got during training.

The results we get for Reward Function 2 are very bad: during training the agent doesn't seem to learn much, as the average reward remains about the same during the whole process. This also manifests in Fig 4, where the average reward of the agent during testing is ≈ -45 points, which is only slightly better than the one it got during training. During both training and testing processes, the agent doesn't manage to win even at one single episode

10.3 DDQN

Finally, we implement the DDQN algorithm for reward functions 1 and 2. In both cases we train our agent for 1000 episodes using the same model. Before training, we conducted experiments in order to tune the values of some important training parameters of our model, which are presented in the following table:

epsilon decay	0.995
gamma	0.99
learning rate	0.001
batch size	32
update frequency of Target net	5

To analyze the training process, we record the cumulative reward received at each episode:

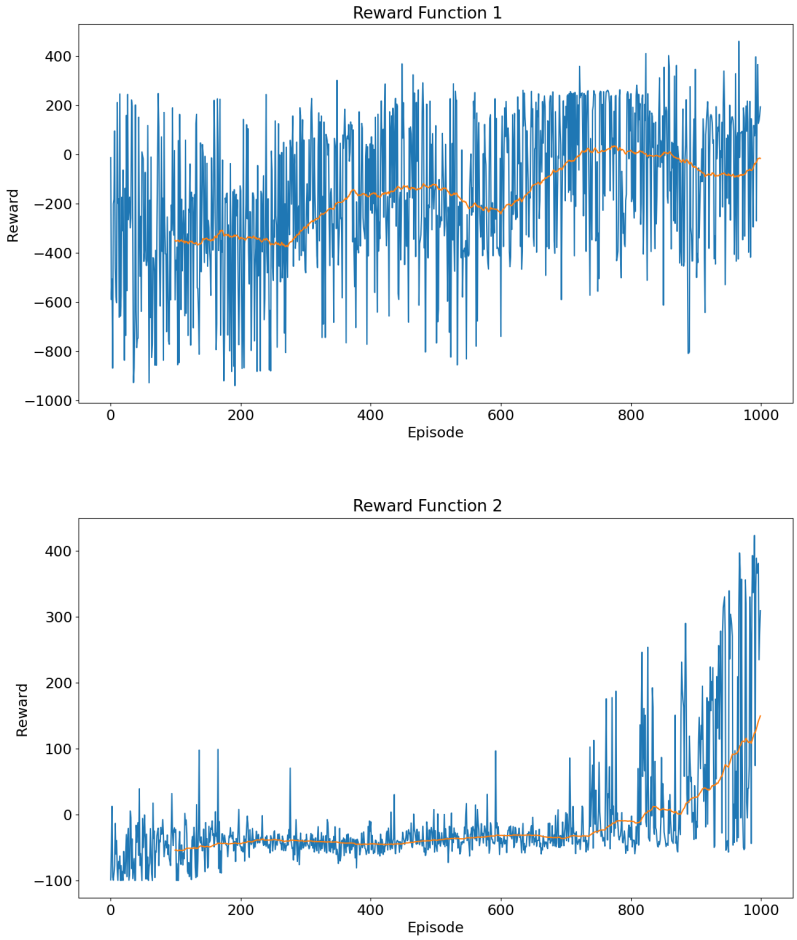


Fig 5: Rewards for each training episode

As we can observe from the graphs above, in both cases of reward functions, the average rewards don't seem to stabilize during the last training episodes, which means that we could continue training our models for more than 1000 episodes, as the performance of our agents might become better. In the case of Reward Function 1, we can see that despite the fact that the average reward doesn't yield a very stable line, it keeps increasing until the end of the training process.
On the other hand, the average reward for Reward Function 2 remains stable, without showing any signs of improvement from the beginning, until a very late stage of the training process, during which -pretty much like in the case of the Standard DQN algorithm- the agent doesn't seem to learn much. Then, from episode ≈ 750 and on, the average reward starts steadily increasing, and by the end of the training process, it has become much higher than the average rewards of Reward Function 1.
In the following graphs, we present the performance of our trained models during testing for 100 episodes:

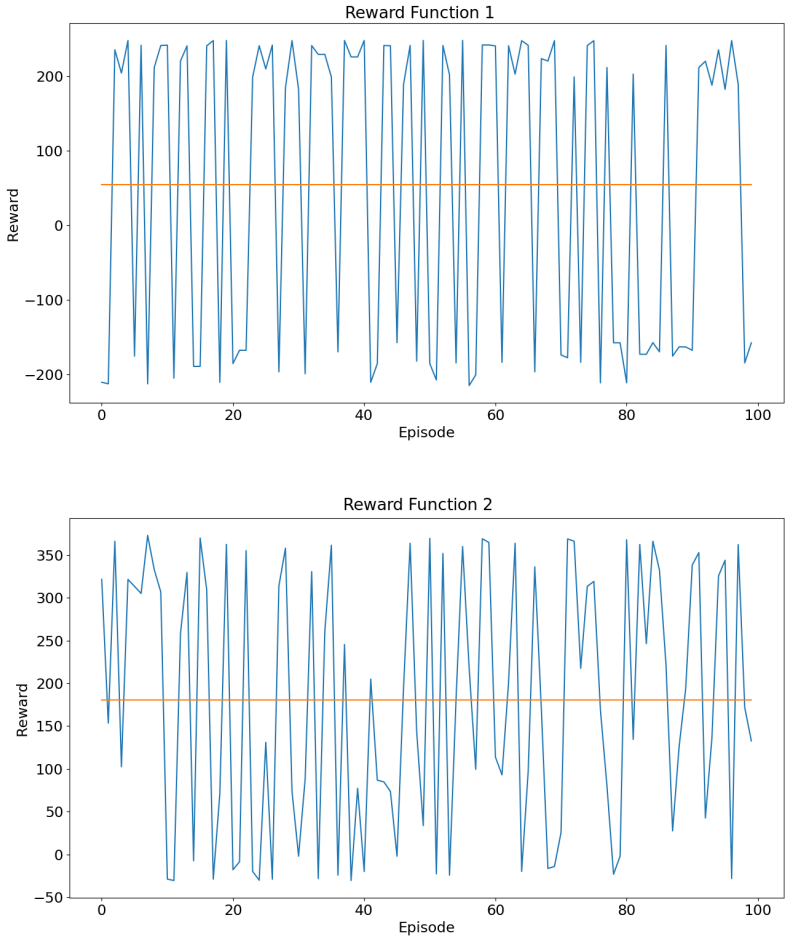


Fig 6: Rewards for each testing episode

By comparing the performances of our models during testing, for the two different reward functions, we can easily observe that even thought Reward Function 2 yields very bad results for most part of the training process, it gives a model that outperforms the one that was trained using Reward Function 1, which had a more "typical" learning process with an average reward that generally kept increasing from the beginning, until the end of the process.

For Reward Function 1, we observe that despite the fact that it yields an ever increasing average reward (overally, during training, the average reward increases by more than 250 points), which means that the agent keeps learning during the whole training process, again, this is still very far from the average score of more than 400 points, needed in order to assume that we have converged to a winning policy. However, since the performance of the agent doesn't seem to stabilize during the last episodes of training, maybe it can still be improved and eventually even converge to an optimal policy if we keep training our model.
From Fig 6, we can see that our trained model has indeed improved its performance, yielding an average reward of ≈ 60 points during testing, which is very close to the results it got from the implementation of the Standard DQN algorithm.

With Reward Function 2, the average reward of our agent increases by more than 200 points during the last 250 episodes of the training process. From Fig 6, we can see that during testing, our trained model gets an average reward of 180 points which is the about the same with the reward it got with the Vanilla DQN implementation, only that now, our model has the potential to continue improving its performance, and maybe even reach convergence, if we continue training it.

10.4 Conclusions

From the three experiments conducted, we can draw some important conclusions:

* Neither the sophisticated Reward Function 1, nor the more naive Reward Function 2 used, result in a model that manages to converge to a winning policy when training for 1000 episodes. This holds for all 3 RL algorithms implemented, and it means that more research needs to be done, in order to design a proper reward function that solves our Lunar Landing problem. The only promising results, were the ones obtained from the implementation of the DDQN algorithm, where for both reward functions used, we observed that the performance of our agents continues improving during the last episodes of the training process, which hint that it could maybe increase even more had we kept trained our models.

* A different epsilon decay strategy, maybe one where the value of ϵ decays as training goes on, but it eventually stabilizes to a small value, rather than completely fading, would allow some more space for exploring new state-action combinations during the later stages of exploitation phase, which might help boost the performance of the agent, and "unstuck" the training process, during phases where, we observe the average reward stabilizing and the agent not learning.

* With Reward Function 1, the implementations of Standard DQN and DDQN algorithms yielded very similar results which slightly outperform the ones we got from the implementation of Vanilla DQN algorithm.

* With Reward Function 2, the implementations of the Vanilla DQN and the DDQN algorithms yielded very similar results during testing. However, the corresponding learning processes were very different from each other. The implementation of the Standard DQN algorithm yielded very bad results, as the performance of the agent does not improve at all during training.

* Overally, the combination of Reward Function 2 and the DDQN algorithm yield the most promising results. After that, follows the combination of the Reward Function 2 and the Vanilla DQN algorithm.

11 Proportional–Integral–Derivative (PID) controller

A PID controller is a control loop feedback mechanism that automatically applies an accurate and responsive correction to a control function. It continuously calculates an error value e(t) by subtracting the current measured value of a process variable from a goal value (setpoint). This error is used to apply a correction on the system under control. The correction is a weighted sum of three terms:

* Proportional: this term is proportional to the error, meaning that small errors will give small corrections, while bigger errors will result in bigger corrections.

* Integral: this term accounts for past errors. It is used to correct for any biases in the system due to a constant force (like a misaligned component, or uncalibrated sensor, etc).

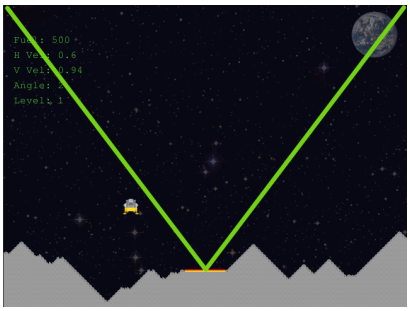
* Derivative: it attempts to predict the controller response. It is used to dampen the response of the Proportional and Integral terms in order to avoid overshoots.

In our case, we implement a PID controller in order to guide our lander to a soft landing and solve our Lunar Landing problem. In order to do this, we decide that the process variables to be controlled are going to be

the amounts of thrust to the main and side engines, which are used for controlling the altitude and angle of the lander respectively.

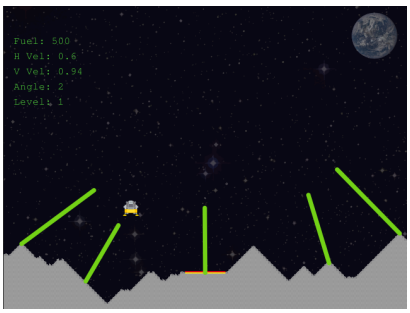
What we want is at every step of an episode, the PID to be controlling the lander, so that it follows a moving setpoint the best it can. At each step, the lander is given an action from the control loop. The choice for this action is determined by a 2-dimensional vector corresponding to the adjustments for the thrust outputs of the main, and the side engines, produced by the PID controller. The control of the lander is implemented as described below:

* The first step is to calculate the setpoints (targets) of our process variables:
    - We choose the altitude target to be equal to: $|x|$, where x is the horizontal distance of the lander from the landing pad:



      This is because when the lander is below the green cone, we want it to increase its height back into it, and when it is above the cone, to generally decrease its height.

    - Similarly, we want the agent to always be pointing towards the landing pad. We know that the angle is the one determining what direction the main engine will send the lander to. Specifically, if the lander is above the landing pad, we want its angle to be equal to 0, and if it is at the extreme edges of the screen, we want it to be +/-45 degrees. For those reasons, we choose the angle setpoint (target) to be: pi/4 * (x + vx).



* The measured values of the altitude and the angle of the lander are known to us at each step from the state vectors. Therefore, the error can be calculated as the difference between our setpoints and the current measured altitude/angle: e = process variable - setpoint.

* In order to add proportional control (kp_y, kp_{angle}) to our control system, we adjust the boosters by some scalar multiple of the errors (i.e. a correction proportional to the error).

* Since we are assuming that we have 100% sensor values in our problem, the Integral component is not needed for the solution.

* In order to add derivative control (kd_y, kd_{angle}), we add the derivative of each control value to our adjustment. The derivative of position (altitude and angle) is velocity, and in our case both vy and the angular velocities are availiable to us as at every step, as they are compontnets of the lander's state vector. We add those derivative terms to our adjustments with an additional parameter.
  Derivative control helps the craft slow its rate of fall as it approaches the landing.

The adjustments produced by the PID are calculated using:

$y_{adj} = kp_y * y_{error} + kd_y * vy$
$angle_{adj} = kp_{angle} * angle_{error} + kd_{angle} * w$

In order to calculate the adjastments from the equations above, we need the values of parameters $kp_y, kd_y, kp_{angle}, kd_{angle}$ which however are not known to us from the beginning.
In order to approximate their values, we use an optimization technique called Hill Climbing. It is a simple technique where we start off by assuming that the value of all four parameters is equal to 0 (no control). We then proceed on trying to land the agent while observing our score. At each episode, we alter the parameters by some small, random amount. If the agent gets a better score we keep those new values and repeat the process. Otherwise we throw them out and try adding random noise again. This technique allows the optimizer to slowly move the values of the parameters 'up the hill', towards to a solution.

The 2-dimensional vector containing the output control values produced by the PID, isn't directly the engine gimbal altitude and angle, but merely a lander-related control. As we have already mentioned above, those values correspond to adjustments for the thrust outputs of the main, and the side engines, which the controller uses in order to choose the agent's next action. Specifically, given a PID output: out = (main, side), where $-1 < main, side < 1$:

* The main engine doesn't work with less than 50% power, so it will be turned off completely if $main < 0$. If $0 < main < 1$, then action 'fire main engine' will be taken.

* The side engines don't work if $-0.5 < side < 0.5$. If $main < 0$ and $side < -0.5$, then action 'fire left engine' will be taken.
  Similarly, if $main < 0$ and $side > 0.5$, then action 'fire right engine' will be taken.

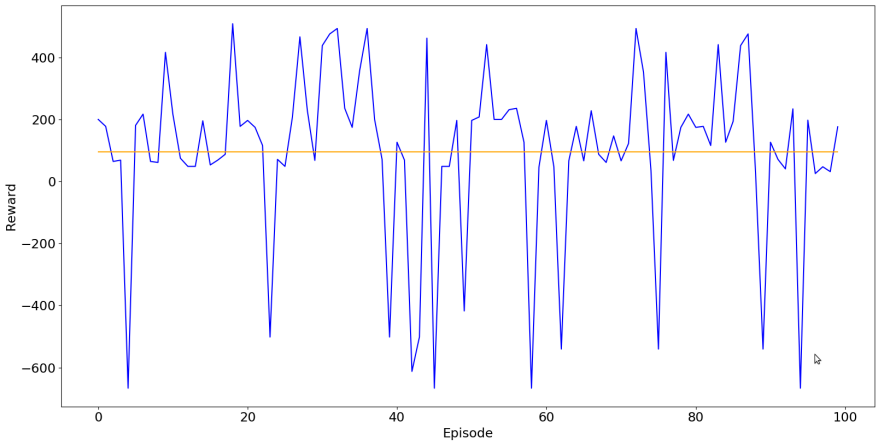
* If main < 0 and $-0.5 < side < 0.5$, then action 'do nothing' will be taken.

11.1 Implementations and Results

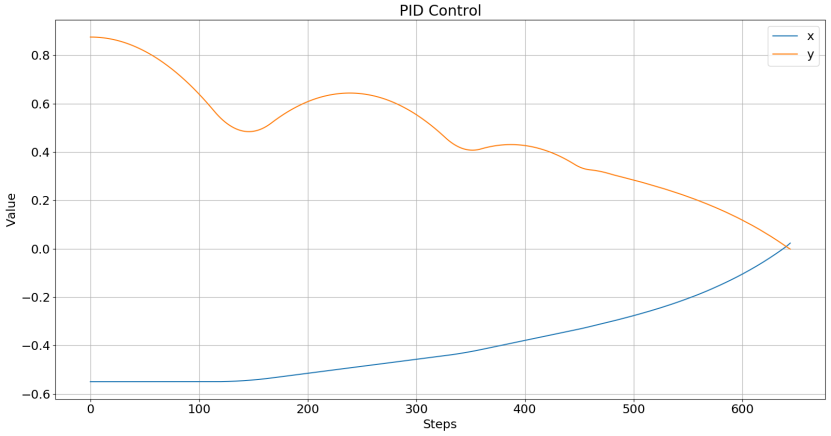
In what follows we first tune our controller and then implement the PID control, for Reward Functions 1 and 2.
During the tuning process, we implement the Hill Climbing optimization technique as described above, for a total of 2000 steps, in order to try to approximate the values of parameters $kp_y, kd_y, kp_{angle}, kd_{angle}$ for which the controller yields optimal results.
We then implement the control (testing) process for a total of 100 episodes.

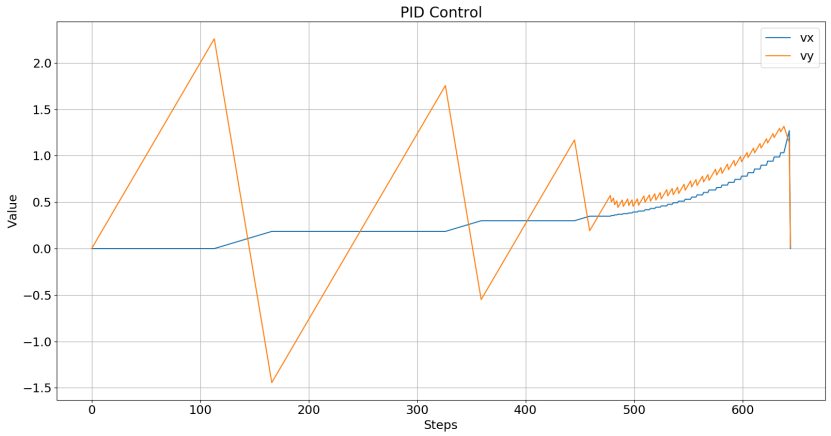
To observe the control process, we record the cumulative reward of the agent on each episode. Moreover, we record the values of the lander's position and velocity during each step of the episode in which the lander earns the highest reward during testing:

Reward Function 1:

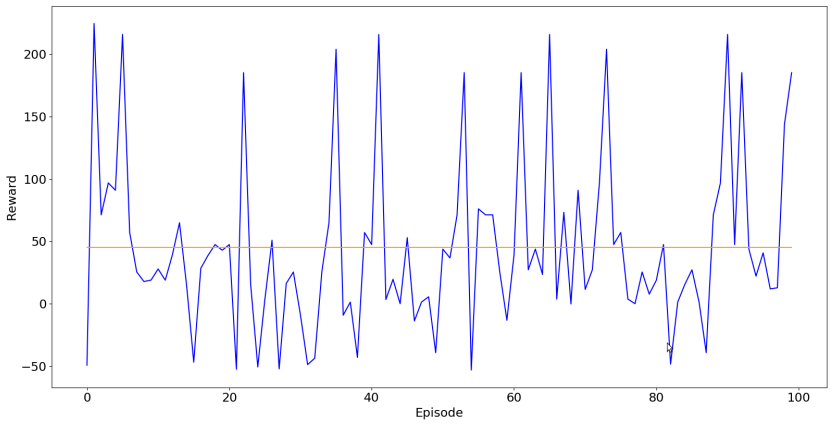


From the figure above, we can see that the PID controller manages to yield an average reward of ≈ 100 points during testing, which outperforms all of our three RL implementations using the same reward function.



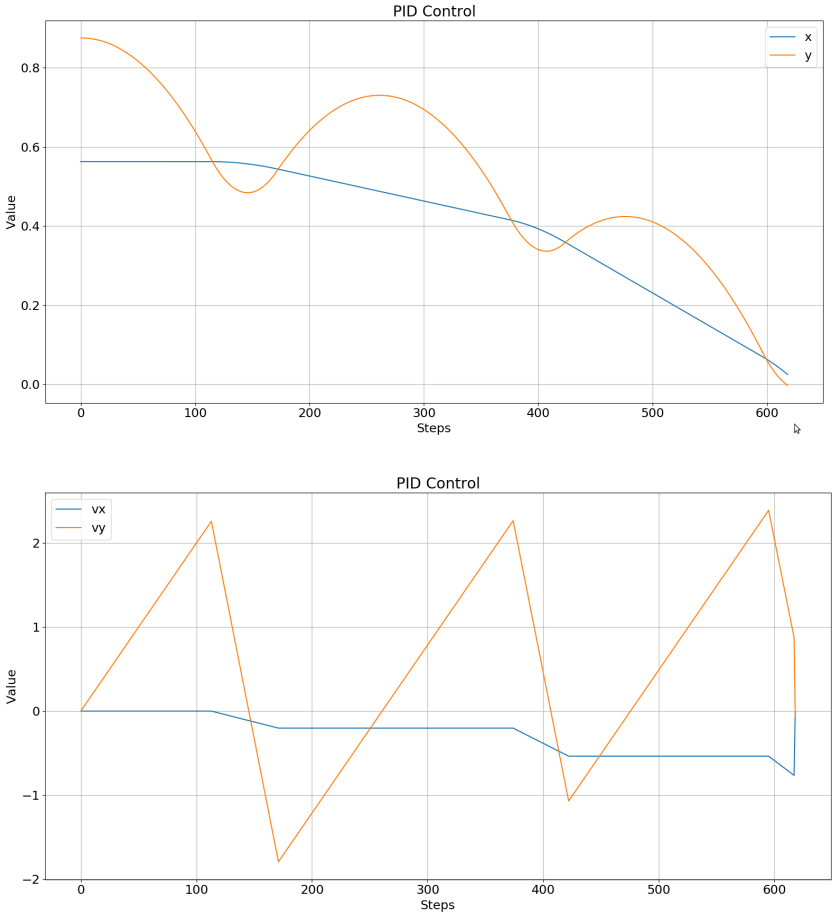


Reward Function 2:



From the figure above, we can see that the PID controller yields an average reward of ≈ 50 points during testing, which is worse than the ones we got from our implementations of Vanilla DQN and DDQN. Despite the fact that the controller performs slightly worse for Reward Function 2, it still

manages to give descent results, especially considering its simple implementation.



It is worth mentioning that while the training of our RL models take many hours to complete, the tuning of the controller, using a naive optimization technique, takes only a few minutes to complete.

Despite the fact that the PID yields quite satisfying results considering its simple implementation, it is still very far from solving our Lunar Landing problem. However, as in the case of our RL implementations, this inability to reach convergence is most probably due to bad design choices in our reward functions.

12 References

* http://repositori.lapan.go.id/964/1/Jurnal%20TD_Larasmoyo_Pustek roket_2021.pdf
* https://mycodeangel.com/free-python-game-projects/jupiter-landing/
* https://github.com/openai/gym/blob/8e812e1de501ae359f16ce5bcd9a 6f40048b342f/gym/envs/box2d/lunar_lander.pyL482
* https://github.com/fakemonk1/Reinforcement-Learning-Lunar_Lander
* https://colab.research.google.com/github/ehennis/ReinforcementLear ning/blob/master/06-DDQN.ipynb
* https://rubikscode.net/2020/01/27/double-dqn-with-tensorflow-2-and- tf-agents-2/
* https://colab.research.google.com/github/NeuromatchAcademy/course -content-dl/blob/main/projects/ReinforcementLearning/lunar_lander.ip ynbscrollTo=ztlMk4u1uEDv
* https://medium.datadriveninvestor.com/training-the-lunar-lander-agent -with-deep-q-learning-and-its-variants-2f7ba63e822c
* https://goodboychan.github.io/python/reinforcement_learning/pytorch/ udacity/2021/05/07/DQN-LunarLander.html
* https://medium.com/@parsa_h_m/deep-reinforcement-learning-dqn-doub le-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9823
* https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep- q-network-double-dqn-b8fc9212bbb2
* https://github.com/wfleshman/PID_Control
* https://blog.nodraak.fr/2021/04/aerospace-sim-3-thrust-vector-control/
* https://blog.nodraak.fr/2020/12/aerospace-sim-2-guidance-law/
* https://www.expert.ai/blog/machine-learning-definition/
* https://www.potentiaco.com/what-is-machine-learning-definition-types- applications-and-examples/
* https://wiki.pathmind.com/deep-reinforcement-learning#reward
* https://www.simplilearn.com/tutorials/machine-learning-tutorial/what- is-q-learning
* https://catalog.us-east-1.prod.workshops.aws/workshops/6d24d10b-9397- 4264-b47a-1099c25aa68a/en-US/bonus/reward

* https://towardsdatascience.com/how-to-design-reinforcement-learning-reward-function-for-a-lunar-lander-562a24c393f6