

Bipedal Walking - Reinforcement Learning

Roberto Figueiredo

April 2022

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Acknowledgement

 Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1	Introduction	4
1.1	RoboCup	4
1.2	Soccer League	4
1.3	Bold Hearts	5
1.4	Introduction to the Project	6
1.4.1	Problem	6
1.4.2	Proposed Solution	6
1.4.3	Aims and Objectives	7
2	Background Research	8
2.1	Reinforcement Learning	9
2.1.1	Markov Decision Process	9
2.1.2	Bellman equation	11
2.2	Learning Algorithms	11
2.3	Training Framework	12
2.4	Previous Implementations	13
2.5	Logging and Reproducibility	13
3	Design	14
3.1	Development Structure	15
3.1.1	Cartpole	15
3.1.2	2D Walker	15
3.1.3	3D Walker	15
3.2	Environment Definition	17
3.2.1	Cartpole	17
3.2.2	2D Walker	18
3.2.3	3D Walker	19
4	Implementation	21
4.1	Cartpole	22
4.2	2D Walker	29

4.3	3D Walker	33
5	Experimental Results	34
5.1	Cartpole Outcomes	35
5.2	2D Environment Outcomes	40
5.3	3D Environment Outcomes	42
5.4	Reward Function	42
6	Future Research	43
6.1	Empowerment	44
6.2	Reward Function Development	44
6.3	Policy Gradients	44
6.4	Mujoco Implementation	44
6.5	Real Robot Training	44
7	Project Evaluation	45
8	Conclusion	46

Chapter 1

Introduction

In this chapter the targeted problem, along with the proposed solution will be introduced. To better understand the explored problem, key relevant background concepts will be introduced.

1.1 RoboCup

RoboCup is an attempt to advance the field of robotics by providing a common problem and an environment for sharing knowledge and collaborate. The Objective of RoboCup is to achieve fully autonomous soccer playing robots that are able to defeat the FIFA World Cup champions by 2050.

RoboCup has since evolved from just soccer and now includes multiple fields, Rescue, Soccer, @Home, Industrial and Junior. [5]

1.2 Soccer League

RoboCup Soccer is split into multiple leagues, each with different challenges and focuses. The Small Size league uses small wheeled robots, each team is composed of six robots and play using a orange golf ball while tracked by a top-view camera; This enables the robots to abstract from challenges such as complex vision detection, walking and others, enabling the teams to focus on strategy and multi-robot/agent cooperation and control in a highly dynamic environment with a hybrid centralized/distributed system.

Middle size League assimilates to small size league, but in this case, the robots are of a larger size and must have all sensors on board; The main focus of the league is on mechatronics design, control and multi-agent cooperation at plan and perception levels.

RoboCup also has simulation for most of its leagues, allowing the teams to focus on software and avoid the difficulties originated by using real robots hardware.

The Standard Platform is a step-up from the simulation league as while it uses real robots, NAOs, it allows the teams to focus mainly on software while using real robots and the challenges of using real robots without having to develop custom hardware. Each robot is fully autonomous and takes its own decisions.

The Humanoid league, assimilates the most to humans, using robots assimilating its shape and unlike robots outside the Humanoid League, the task of perception and world modeling is not simplified by using non-human like range sensors, making this, the most transversal league, requiring hardware and software development. In addition to soccer competitions technical challenges take place. Dynamic walking, running, and kicking the ball while maintaining balance, visual perception of the ball, other players, and the field, self-localization, and team play are among the many research issues investigated in the Humanoid League.

1.3 Bold Hearts

The Bold Hearts are the RoboCup team from the University of Hertfordshire. The team researches and develops software and hardware in the Humanoid league, specifically in the teen size league. The robots used by the team are based on the Darwin-OP but incrementally developed using custom 3D printed parts and a new computing unit, Odroid-XU4 and camera, the Logitech C920 Pro. The Robots use ROS2 as an operating system. [11]



Figure 1.1: Boldbot

1.4 Introduction to the Project

1.4.1 Problem

Robotic locomotion has, until a few years ago, been focused on wheel-based movement. Although it is very stable and easy to implement, it lacks flexibility, the ability to move on uneven, unpredictable terrain and overcome obstacles such as stairs.

As a member of the Bold Hearts team our robots must be able to walk and, in addition to the high complexity of this challenges, RoboCup rules periodically change, these changes are put in place as the RoboCup objective is to achieve the most realistic environment. Rule changes affect both robots, changing the required height, sensors and others, as well as affecting the environment such as moving from flat ground to synthetic grass.

One of the main challenges faced by the team is walking, which is one of the most complex movements performed by humans, requiring simultaneous control of multiple joints to move while maintaining balance. This is one of the most energy and time-consuming problems faced by the team aggravated by the rules changes and continuous improvements this requires updates to the robot's structural architecture, leading to new updates to the walking algorithm.

1.4.2 Proposed Solution

Walking algorithms can be developed using various techniques, including explicit programming, supervised learning and unsupervised learning. Walking is very complex as there are a lot of variables involved in it, such as the ground contact, maintaining balance and the complex gait movement. The two main aspects important to highlight are that the walking algorithm requires changes when both the robot or external factors change and that it requires manual work from the team to achieve this.

The best way to solve the first mentioned problem requires having a robot and environment agnostic approach, this is possible by using Reinforcement Learning as the principles of the movement maintains therefore it should be possible to develop an agnostic reward function. From the moment a reinforcement learning solution is successfully implemented, the team should be able to use the same implementation to retrain the policy using the updated robot/environment that has been modeled in the simulator. This also allows to reduce the complexity of the problem as the input into the system is sensory data and the output is a set of actions to execute on the joints.

1.4.3 Aims and Objectives

This project proposes to develop a reinforcement learning implementation for walking, to achieve the objectives of the project, three main topics will be covered, robotics biped locomotion, reinforcement learning algorithms and the training framework.

While achieving a working walking pattern would be desireable, the project aims to develop an implementation of reinforcement learning to achieve walking, along with it aims to develop resourcefull documentation on the process and its results, allowing for reproducibility and further development and adaptations of the current implementation to learn not only walking but any suitable problem.

Chapter 2

Background Research

2.1 Reinforcement Learning

Reinforcement learning is one of the main machine learning paradigms, alongside supervised learning and unsupervised learning. The objective of reinforcement learning is essentially to map states to actions while maximizing a reward signal, in some cases, actions may affect not only the present but future situations. To learn how to map the states to actions the agent must try them, this characteristic, trial-and-error and delayed reward are the most distinguishable features of reinforcement learning.

The agent must be able to perform actions affecting the environment followed by a perception, this perception should indicate to what state the robot has transitioned and the reward signal associated with the result of the action taken given the previous state. This interaction is summarized in the following figure.

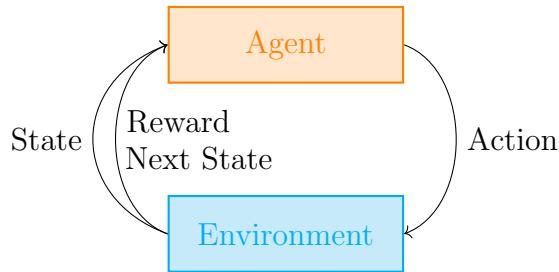


Figure 2.1: Reinforcement learning model

One unique challenge in reinforcement learning is balancing exploration and exploitation. To succeed a task the agent must exploit actions that, through experience have yielded the most rewards, although, to have experience and perform better in the future the agent must explore new actions. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task [12]

2.1.1 Markov Decision Process

Reinforcement learning can be described using the Markov Decision Process(MDP). MDP is the final summary concept of the individual elements:

- The Markov Decision Chain
- The Markov Property
- The Markov Reward Process

Markov Decision Chain

A Markov chain is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. The defining characteristic of a Markov chain is that no matter how the process arrived at its present state, the possible future states are fixed. In other words, the probability of transitioning to any particular state is dependent solely on the current state.

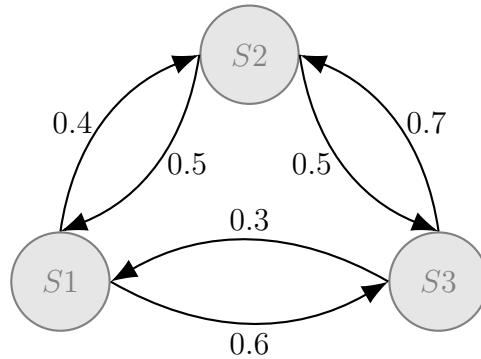


Figure 2.2: Example of a Markov Decision Chain

As can be observed by the example Markov Decision Chain, the transition probabilities are fixed and are only dependent on the current state, this is the Markov property.

$$P(S_{t+1}|S_t)$$

Definition of the probability of transitioning to any particular state given the current state.

Markov Property

This means that the transition to state $t+1$ from state t is independent of the past, meaning that our current state already captures the information of the past states. Defined by the following equation:

$$P[S_{t+1}|S_t] = P(S_{t+1}|S_1, \dots, S_t)$$

Markov Reward Process

As it suggests, the Markov Reward Process is a Markov process with the difference that it includes a reward system that indicates how much reward is accumulated through a particular sequence. An additional factor is applied, the **discount factor** γ that indicates how much the future reward is discounted. if $\gamma = 0$ then the agent will only consider the immediate reward, if $\gamma = 1$ then the agent will consider all subsequent rewards. In practice this extreme rewards are ineffective and γ is usually set to values between 0.9 and 0.99

2.1.2 Bellman equation

The bellman equation is the basis to solve RL problems, it helps us solving the MDP, it is defined as:

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

The Bellman equation sums to the present reward the discount factor multiplied by the best next value outputed by the value function.

The optimal value function $V^*(s)$ maximizes the expected reward. To achieve this the Bellman equation is solved by iteratively updating the value function until $V^*(s)$ is reached.

2.2 Learning Algorithms

One of the main decisions in implementing a reinforcement learning algorithm is the learning algorithm, to understand the algorithm decision its important to understand how this differ.

The first major split in RL algorithms is whether it is model-free or model-based, model-based algorithms use a model of the environment, which is capable of mimicking its behaviour, this allows inferences to be made about how the environment will behave given an action and what the next reward will be. This method of reinforcement learning isn't adequate for the current problem due to the high complexity of the environment in which the agent is interacting, making it impossible to model it.

The second decision is between **policy optimization** and **Q-Learning**. While both aproaches have similar concepts and are driven by MDP they are internally different.

In Q-learning, the goal is to determin a single action from a set of discrete actions by finding the action with the highest Q-value. While in policy optimization the goal is to learn a map from state to action, this can be stochastic and, opposed to Q-Learning, works in continuous action spaces.

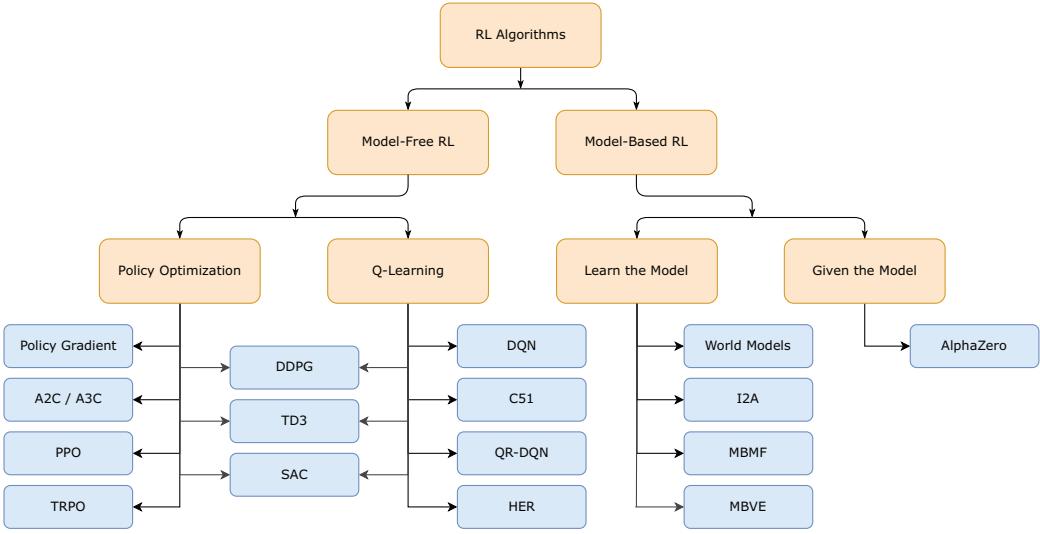


Figure 2.3: A non-exhaustive, but useful taxonomy of algorithms in modern RL. [8]

To the problem in study, Q-Learning was a better fit as the action space was discretized(covered further in chapter 3), apart from the action space, the Implementation of Q-learning is simpler and more common.

The algorithm decision was set on Deep Q-Network (DQN). The main difference between DQN and Q-Learning is that DQN implements a neural network replacing the Q-table. This is a benefit due to the complexity of the environment and the total number of possible states, using standard Q-learning would require a very large q-table, which is a huge memory and computational burden.

2.3 Training Framework

Followed by the new advances in RL there was a growing need for a common benchmarking framework, that would allow for comparing algorithms and implementations.

Gym was released by the OpenAI team to enable this, by providing an accessible API and common environments. Gym is now the most widely used tool in RL research, it has a large community and documentation and a growing number of environments.

Gym was chosen as the framework to standardize the implementation which allows for an easier understanding and comparison, beyond this, gym

also facilitates the implementation by using all the pre-built functions.

2.4 Previous Implementations

To develop this project it was important to first study previous implementations, what problems were faced and the reasoning behind there decisions, this brought light to many problems and details of implementing reinforcement learning in special for walking.

One important implementation was the **Deep Q-Learning for Humanoid Walking**[13], an implementation of reinforcement walking on the Atlas platform. this implementation higlights some of the general pre-conceptions regarding the reward system and how to handle the complexity of controlling all the joints simultaneously.

2.5 Logging and Reproducibility

One of the most important aspects of machine learning and Reinforcement Learning in particular is data logging and reproducibility. This project required extensive testing of hyperparameters and reward systems. To understand the results and its correlation with the variables in testing its important to log all the results and to be able to reproduce the results.

Requirements for logging:

- Hyperparameters
- Performance metrics
- Code
- Models
- Renderings

From previous experiences with machine learning and logging platforms MLflow was the first option analysed, although, Weights & Biases (WandB) was the choosen platform, WandB offered an hosted version option compared to MLflow, WandB also integrates with Keras allowing for a seamless implementation. WandB fills all the requirements and allows for a comparison of the renderedings and performance results between runs.

Chapter 3

Design

3.1 Development Structure

Due to the complexity of the project, a development structure has been put in place, this includes multiple steps of increasing complexity and realism, the increasing complexity allows for detecting problems at earlier, simpler stages, making the transition and understanding the problems easier.

3.1.1 Cartpole

Cartpole is a classic exercise of reinforcement learning, it consists in balancing a pole in a cart moving on an horizontal plane by applying a force on the right or left side of the cart making it move in the oposite direction.

The cartpole environment allowes for the implementation and testing of the reinforcement learning algorithm, different implementations and its comparison, at this stage it was also used to implement the logging and reproducibility interface.

3.1.2 2D Walker

At this stage the complexity of the environment increases as the environment starts to assimilate the target problem. Although, this stage eliminates some of the complexity such as using a more complex 3D environment and implementing the training with the robot controll interface. This allows for the implementation of a neural network, learning algorithm that is capable of handling multiple simultaneous outputs as it is required to control all the joints of the robot and understanding how to efficiently calculate the best action.

To achieve this it is necessary to develop a custom 2D environment of a simplified humanoid in order to train a walking behaviour. New challenges from this stage such as implementing a custom reward system, rendering and step functions are an important step in order to transition to 3D simulation.

3.1.3 3D Walker

The final stage of the project is the implementation on a 3D simulated robot, this is the combination of the previous stage with extre complexity, not only due to the inherited complexity of a higher diensional world but because this should be able to integrate with the real robot from the Bold Hearts team and therefore use its control interface. Robot simulation is the main platform for developing software for robotics, it has many benefits,

developing software and testing it directly on a real robot can be a very slow process and can even lead to breaking the robot.

3d simulation brings new challenges, such as a larger range of motion and more joints to controll, along with a more complex environment, requiring more processing power and more time to solve the problem. Along with this it requires a more complex reward system as a new dimention poses new problems.

3.2 Environment Definition

One of the main reasons for the use of OpenAI Gym in this project is the ability to standardize the environments. In this section each Gym environment will be described.

3.2.1 Cartpole

The cartpole environment is a very simple exercise, consisting of a pole in a cart moving on an horizontal plane, the objective is to balance this pole by applying a force on the right or left side of the cart making it move in the opposite direction. [7]

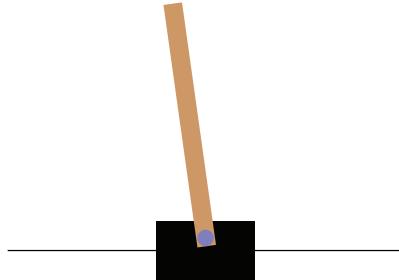


Figure 3.1: Rendered Cartpole Environment - OpenAI Gym

Cartpole Environment definition:

- **Observation Space**

Table 3.1: Observation Space for the Cartpole environment

Num	Observation	Min	Max
0	Cart Position	-4.8	+4.8
1	Cart Velocity	-Inf	+Inf
2	Pole Angle	-24°	+24°
3	Pole Angular Velocity	-Inf	+Inf

- **Action Space**

Table 3.2: Action Space for the Cartpole environment

Num	Action
0	Push cart to the left
1	Push cart to the right

- **Reward:** In the cartpole environment the reward is attributed per timestep survived, being awarded 1 point per timestep.

3.2.2 2D Walker

The 2D environment requires a 2D Humanoid, this has been defined with 8 different joints, 2 in the shoulder, 2 in the hips, 2 in the knees and 2 for the ankles.

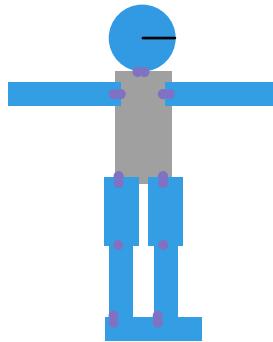


Figure 3.2: Representation of 2D humanoid

Environment definition:

- **Observation Space:** The Observation Space is a vector of size 8, containing the angles for each of the 8 joints of the humanoid.

Table 3.3: Observation Space for the 2D Walker environment

Observation	Min	Max
Joint Position	-20°	+20°

- **Action Space:**

Table 3.4: Action Space for the 2D Walker environment

Num	Action
0	Move the joint counterclockwise
1	Maintain joint position
2	Move the joint clockwise

- **Reward:**

Table 3.5: Reward system for the 2D Walker environment

Action	Points
Moves Back	- 2
Stays in Place	-1
Moves Forward	1
Loses contact with the ground	cumulative -2
Reaches target	10
Falls	-10

3.2.3 3D Walker

In the 3D environment, the robot needs to simulate the one used by the Bold Hearts team, given that the team already uses a simulator, gazebo, this will be used to interact with the robot given that it provides ROS2 integration.

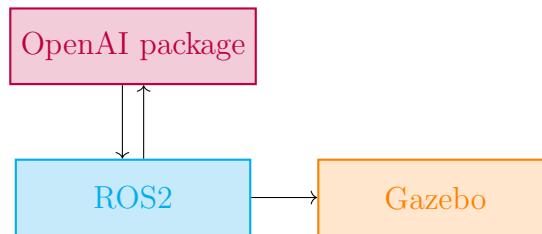


Figure 3.3: Integration of ROS2, Gazebo and OpenAI Gym

To achieve this integration, OpenAI Gym should be able to subscribe ROS2 topics containing the world observations. The OpenAI Gym should also be able to publish to ROS2 topics, allowing it to control the robot's joints, as well as calling services in order to control the simulation, including, pause and reset the simulation. One of the main aspects of this integration is to allow to train not only walking but also any relevant task by the team, to achieve this the OpenAI Gym package should be split into three different parts:

- Robot
- Environment
- Task

The robot file should contain all the setup required for the Boldbot, the teams robot. On the Environment file all the environment setup should be

defined, such as the observation space, action space, reward system, etc. Finally the task file should be specific for the task trying to be achieved, allowing the team to setup just the task without requiring setting up the robot and environment everytime. [3]

Chapter 4

Implementation

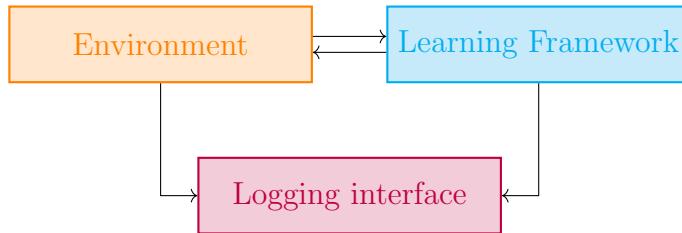
4.1 Cartpole

Cartpole is a pre-built environment from the gym library, therefore, it just has to be imported using:

```
gym.make('CartPole-v1')
```

In this experiment the version 1 was preferred over version 0 as it provides more timesteps, setting the limit to 500 timesteps per episode.

There are three main parts in stage 1:



Each of this should interact for a successfull implementation of the first stage. After the environment is imported, the learning framework is imported, this is a framework that is used to train the agent. In this stage, two main implementations were explored, Keras-rl and a manual implementation of the keras API.

Keras-rl

To implement keras-rl[6] in the specific hardware used in this project, it was required to be installed from source inside a miniforge environment. To be able to setup the model used by keras-rl tensorflow was required, once again, the hardware required a custom installation, the instructions can be found on the manufactures website [4].

As the requirements were met, the next step is to setup a model, this was setup using keras, imported from tensorflow.

Listing 4.1: Setting up the model in the cartpole environment

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input

states = env.observation_space.shape
actions = env.action_space.n

def build_model(states, actions):
    model = Sequential()
    model.add(Input(states))
    model.add(Flatten())
    model.add(Dense(12, activation='relu', input_shape=(1,4)))
    model.add(Dense(12, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model

model = build_model(states, actions)
model.build(states)
```

The code displayed above is used to import the required packages for the model, proceeded by a function where the model is defined, here, we can see that the model receives the inputs, flattens this followed by a fully connected layer with 12 neurons and relu activation function, this layer is followed by another fully connected layer with 12 neurons and relu activation function.

The next step on setting up the learning framework is to define the agent, to do this keras-rl needs to be imported

Listing 4.2: Setting up the learning agent

```
from rl.agents import DQNAgent
from rl.policy import EpsGreedyQPolicy
from rl.memory import SequentialMemory
from tensorflow.keras.optimizers import Adam

def build_agent(model, actions):
    policy = EpsGreedyQPolicy(eps=.3)
    memory = SequentialMemory(limit=50000, window_length=1)
    dqn = DQNAgent(model=model, memory=memory,
                   policy=policy, nb_actions=actions,
                   nb_steps_warmup=100, target_model_update=1e-2)
    return dqn

dqn = build_agent(model, actions)
dqn.compile(Adam(learning_rate=0.01), metrics=['mae'])

dqn.fit(env, nb_steps=50000)
```

In the code above the agent, exploration policy and memory are imported from the keras-rl library and the Adam optimizer imported from Keras.

The *build_agent* function is used to set the DQNAgent, the exploration policy is initialized and ϵ value is set to 0.3. the memory, replay buffer, is initiated using the SequentialMemory class from keras-rl, the memory length is set to 50000 samples and the window length is set to the number of samples at each time, in this case 1. The agent is then compiled using the Adam optimizer where the learning rate is set to the optimal value of 0.01, here the metric use is also set to the mean absolute error. the DQNAgent is now initialized using the model, policy and memory previously defined, along with this the number of actions is passed as a parameter, the number of warmup steps used to avoid oscillating parameters and the target model update is set to 1e-2, this indicates how the target model is updated based on the model being trained.

The last line is used to train the agent, the environment is passed as a parameter and the number of steps is set to 50000.

Manual Agent implementation

During the cartpole stage, another implementation was tested against Keras-rl, this was a manual implementation of the agent. An agent can be split into multiple parts:

- Replay buffer
- Action selection
- Target-q model update
- Training

The **replay buffer** is used to store the following sets of data (state, action, reward, state', done), than this is sampled in batches, this batches are used to train the neural network, the replay buffer is implemented to allow learning multiple times from the same actions and it also helps to break correlations.

Listing 4.3: The following function is used to store the sets of data in the replay buffer

```
def remember(self , state , action , reward , next_state , done):
    item = (state , action , reward , next_state , done)
    self.memory.append(item)
```

Listing 4.4: The following function is used to replay actions from the buffer and train the neural network using this

```

def replay(self , batch_size):
    batch = random.sample(self .memory , batch_size)
    state_batch , target_batch = [ ] , [ ]

    for state , action , reward , next_state , done in batch:
        target = self .get_target_q_value(state)
        if done:
            target [0][action] = reward
        else:
            Q_future = max(  

                self .get_target_q_value.predict(next_state) [0])

            target [0][action] = reward + Q_future * 0.9
        state_batch.append(state)
        target_batch.append(target)
    self .q_model .fit(state_batch ,
                        target_batch ,
                        batch_size=batch_size ,
                        verbose=0,
                        epochs=1,
                        callbacks=WandbCallback())

```

In the replay function, the buffer is sampled, the sample obtained from the buffer contains a batch of states, actions, rewards, next states and done values. This samples are iterated and the target values are predicted based on the state using the target model. if the state sample is done, the calculated target q-value for the action used at the current state is set to the reward value obtained, if the simulation is not done, the target q-value for the action used is set to the reward value plus the q-value for the future state is multiplied by γ which is set to 0.9.

The **action selection** in the manual implementation is applied as follows:

Listing 4.5: The following function is used to balance exploration and exploitation to select an action

```

def act(self , state):
    if np.random.rand() < self .epsilon :
        return self .action_space.sample()
    q_values = self .q_model .predict(state)
    action = np.argmax(q_values [0])
    return action

```

The action selection process implemented above is based on the epsilon greedy exploration strategy, to achieve a balance between exploration and exploitation an ϵ . To select an action a random number between [0,1] is generated and compared to ϵ , if the random number is smaller than ϵ a random action is sampled from the environment, this is called exploration.

If the random number is higher than ϵ the model is used to predict the q-value for each action, the highest q-value is selected using the numpy function argmax which returns the index of the higher q-value (and therefore the best predicted action) instead of its value.

It is important to **update the target-q model** improving the quality of the future predictions, to do this the target model should be updated at a set number of iterations.

Listing 4.6: The following function is used to update the target-q model

```
def update_weights(self):
    self.target_q_model.set_weights(self.q_model.get_weights())

    if self.replay_counter % 10 == 0:
        self.update_weights()
```

In the cartpole implementation the target-q model is updated every 10 times the regular model is updated, the target-q model is updated by setting its weights to the current regular model.

The last implementation step is the **training**, as can be seen in the code bellow:

Listing 4.7: The following function is used to update the target-q model

```
nr_episodes = 200
for episode in range(nr_episodes):
    state = env.reset()
    done = False
    total_reward = 0
    step_count = 0
    while not done:
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        step_count += 1
    if len(agent.memory) >= batch_size:
        agent.replay(batch_size)
```

The code above starts by setting the number of episodes to train for, followed by the for loop, itterating for the number of set episodes over the training code.

the episode starts by reseting the environment, which returns the state of the world, when each episode starts the total reward, done state and step counter are reset.

The while loop repeats until done is set to True, ending the episode, this

can be either due to the pole falling, the cart moving out of the window limits or reaching 500 steps.

At each step, an action is selected using the act function shown before, which is inside a class Agent. After selecting an action a step is performed in the environment by passing the selected action, the environment step returns the state it transitioned to, the reward obtained, whether the episode has finished and a dictionary with extra information which, in the cartpole environment, is not set to return anything.

After each step the information retrieved is added to the buffer and the environment state is set to the state it has transitioned to, followed by incrementing the step counter.

At the end of each episode if the number of samples in the environment is higher than the batch size required the agent samples the buffer and updates the model.

WandB - Weights & Biases

To implement the logging and enable reproducibility the platform WandB - Weights & Biases was used, this platform has integrations with all the major machine learning tools, including Keras, Tensorflow, PyTorch, Lightning, Hugging Face, Fast.ai, Scikit and XGBoost.[1]

WandB allows to both host the platform locally or use the free wandb cloud based version. In order to be free from relying on external services a self-hosted version was setup on the same machine used to run the training. To setup the self-hosted version, docker is required as the platform is dockerized, WandB can be installed using the pip command:

```
$ pip install wandb  
$ wandb login
```

To log data to wandb a project needs to be created:

Listing 4.8: Code used to create a wandb project and setup logging

```
import wandb  
from rl.callbacks import WandbLogger  
run = wandb.init(  
    config={  
        "gamma": 0.9,  
        "epsilon": 0.3,  
        "target_reward": 500.0,  
        "batch_size": 64,  
        "win_trials": 100,  
        "units": 12,  
    })
```

```

    "learning_rate": 0.01
    "metrics": "mae"
},
project="cartpole", id="run_1")

dqn.fit(env, nb_steps=50000, callbacks=[WandbLogger()])

```

After importing the wandb package when using keras-rl we can import a pre-built callback, WandbLogger.

By calling the wandb.init function, there are three parameters passed, config, project and id.

The project parameter is the name of the WandB project, all the runs using this project name will be grouped in the interface, allowing them to be compared.

The second parameter is the id, corresponding to the run id, the name we want to attribute to the current session.

The config parameter is a dictionary containing any hyperparameters and values transversal across a session that we want to log, logging the parameters this way allows us to correlate the results to the hyperparameters and reproduce the results.

After initiating the session we can log data using pre-built callbacks both built into keras and keras-rl.

Apart from the data being logged, it was necessary to log other data and artifacts.

Listing 4.9: Logging data and artifacts to WandB

```

run.log(
    {"video": wandb.Video(filename, fps=30, format="mp4")})

run.log({
    "reward": total_reward,
    "epsilon": agent.epsilon,
    "average_reward": np.average(reward_history),
    "mean_score": mean_score,
    "minimum_reward": np.min(reward_history),
    "max_reward": np.max(reward_history),
})

```

As can be seen above it is possible to log both files, and data manually, in the first example we can see how a rendered video is logged to wandb, this is helpful as it allows to observe the progress of the training.

The second example shows how extra metrics can be logged.

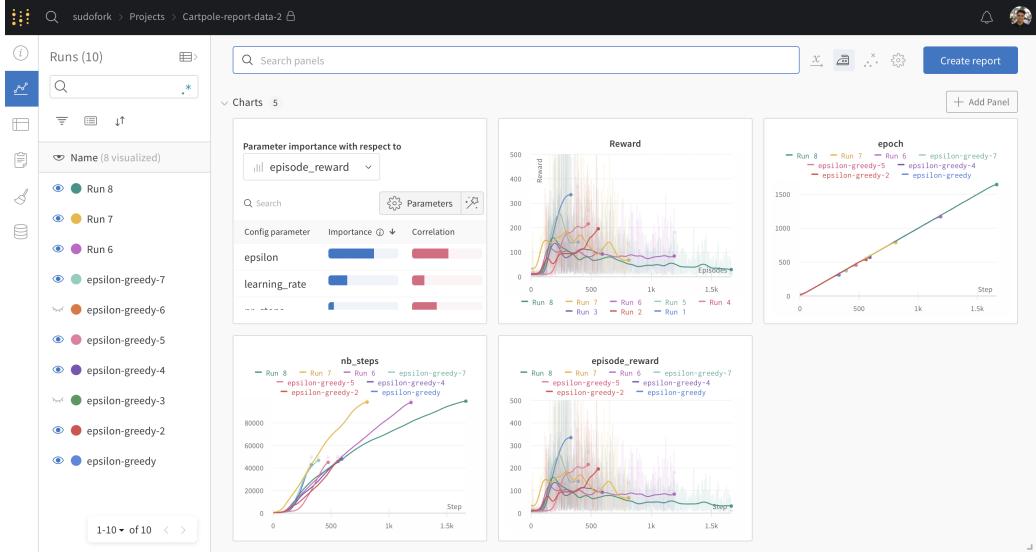


Figure 4.1: Logging data and artifacts to WandB

4.2 2D Walker

The implementation of the 2D walker environment has one major difference from the cartpole, as this requires the development of a custom 2D simulation and gym environment. After research the tools chosen for the simulation development were Pymunk[10], a python implementation of Chipmunk[2] , and Pygame[9] as the later is integrated in pymunk, allowing for a smoother implementation of the rendering avoiding duplicate code. Once an element is implemented in pymunk it can be directly simulated in pygame.

After the simulation was developed the next step was to implement it in a gym environment. The first step is to define the observation space and action space. As defined in the Design chapter, the robot has 8 joints each with 20 degrees of freedom and each joint can perform one of three actions(decrease, maintain and increase the angle):

Listing 4.10: Import necessary gym modules and initiate action space and observation space

```
from gym.spaces import MultiDiscrete, Box

self.action_space = MultiDiscrete([3]*8)
self.observation_space = Box(-20, 20, [8])
```

Each joint has 3 distinct possible actions, this can be described using a Discrete space, although, as there are 8 distinct joints, the resulting actions space is $[3]^8$, this can be defined by using a MultiDiscrete space, a space that allows for multiple discrete action spaces.

The observation space can be defined as a continuous observation as it can take any value in the range of $[-20,20]$ therefore it requires a Box space, used to define continuous values in gym environments, the Box space takes as arguments, the lower bound, higher bound and shape, hence the [8], as there is an observation for each of the 8 joints.

There are three main functions in a gym environment: **step**, **reset** and **render**:

Listing 4.11: Defining the step function

```
def step(self , actions):
    actions = [(a-1)*2 for a in actions]
    self.robot.ru_motor.rate = actions[0]
    self.robot.rd_motor.rate = actions[1]
    self.robot.lu_motor.rate = actions[2]
    self.robot.ld_motor.rate = actions[3]
    self.robot.la_motor.rate = actions[4]
    self.robot.ra_motor.rate = actions[5]
    self.robot.lf_motor.rate = actions[6]
    self.robot.rf_motor.rate = actions[7]

    self.robot.update()
    self.space.step(1/50)

    done = False
    reward = self.calculate_reward()

    if self.check_fall():
        done = True
        reward = -10
    if self.check_complete():
        done = True
        reward = 10

    info = {}
    observation = self.robot.get_data()

    self.last_horizontal_pos = self.robot.body.position[0]
    self.last_vertical_pos = self.robot.body.position[1]

return(
    observation ,
    reward ,
    done ,
```

```
info)
```

The step function receives the action array containing the action selected for each of the joints. The first step is to convert the action to the correct form to be applied to the motors, to do this, the action is subtracted by 1 to transform the range from [0,2] to [-1,1] this are then multiplied by the rate of the motor (2).

With the correct motor rates, this are applied to the robot motors, although this wont take an effect before the robot being updated and the space taking a step. The robot step is set to take 50 actions per second, defined by the value passed in the step function.

After the action is executed the reward is calculated using a separate function, defined as follows:

Listing 4.12: Defining the reward function

```
def calculate_reward(self):
    shape = self.space.shapes[-2]
    contact_lf = len(self.robot.lf_shape.shapes_collide(b=shape).points)
    contact_rf = len(self.robot.rf_shape.shapes_collide(b=shape).points)
    if (self.robot.body.position[0] - self.last_horizontal_pos) > 1:
        reward = 1
    elif 1 > (self.robot.body.position[0] - self.last_horizontal_pos) > -1:
        reward = -1
    elif (self.robot.body.position[0] - self.last_horizontal_pos) < -1:
        reward = -2
    if not contact_lf and not contact_rf:
        reward -= 2
    return reward
```

The reward is calculated based on the current robot horizontal position compared to the previous position, if the robot moves forward it is awarded 1 point, if it stays in the same position it is penalized with -1 point, if the robot moves backwards it is penalized with -2 points. The reward is then deducted 2 points if both feet loose contact with the ground.

Refering back to Listing 4.11 after the reward is calculated the step function checks if the robot has fallen using the following function:

Listing 4.13: Defining the function to check if the robot has fallen

```
def check_fall(self):
    robot_xpos = self.robot.body.position[0]
    if self.robot.body.position[1] < self.initial_height - 50:
        return True
    if robot_xpos < 0 or robot_xpos > screen_width:
        return True
    return False
```

If the robot vertical position is lower than the initial vertical position minus a threshold (50) the robot is considered to have fallen. If the robot horizontal position goes outside the screen width, the robot is also considered to have fallen. the result returned by the check_fall function is used to set the done variable and sets the reward to -10 points in case the robot falls.

The next function call is check_complete this checks if the robot has achieved the target horizontal position defined. If the robot reached the target, the episode ends and the reward is set to 10 points.

After calculating the reward and checking if the episode has ended new observation data is obtained from the environment by reading the joints position. Followed by updating the variable holding the last horizontal position.

The reset function in the gym environment is used to reset the environment and simulation when a new episode starts and returns an observation.

Listing 4.14: Defining the reset function

```
def reset(self):
    self.space = pymunk.Space()
    self.space.gravity = (0.0, -990)
    self.robot = Robot(self.space)
    self.robot.add_land(self.space)
    self.initial_height = self.robot.body.position[1]
    observation = self.robot.get_data()
    return observation
```

The reset function starts by resetting the physycs engine space by initiating a new one, followed by setting the gravity applied to the objects. The next step is to re-create the robot in the space followed by the land. The initial_height is set and a new observation is obtained.

To allow us to visualize the environment a render function has been implemented:

Listing 4.15: Defining the render function

```
def render(self, mode='human', close=False):
    if self.viewer is None:
        self.viewer = pygame.init()
        pygame_util.positive_y_is_up = True
        self.clock = pygame.time.Clock()
        self.screen = pygame.display.set_mode((screen_width, screen_height))
        self.draw_options = pygame_util.DrawOptions(self.screen)
        self.screen.fill((255, 255, 255))
        self.space.debug_draw(self.draw_options)
        pygame.display.flip()
        self.clock.tick(50)
    return pygame.surfarray.array3d(self.screen)
```

The render function starts by initiating pygame and flipping the vertical coordinates as these are inverted due to the pymunk integration. The clock and screen are initiated passing the width and height parameters for the screen.

After creating the screen, this is filled with white and the objects are drawn on the screen, followed by calling the flip function used to update the screen. After updating the display the clock is ticked, the parameter passed corresponds to the maximum number of frames per second, matching the one set in pymunk.

A 3D array containing the image data is returned so that it can be displayed or added to a sequence creating a video.

After the environment was setup, it was required to have a neural network able to support the 8 simultaneous actions.

This was solved by using a neural network that split into multiple independent layers for each of the joints, as exemplified below:



4.3 3D Walker

Chapter 5

Experimental Results

This chapter will cover the development process, the decisions that have been taken as well as the results and outcomes of the individual development parts.

5.1 Cartpole Outcomes

The first task developed was the classic cartpole environment, this was helpfull in understanding core concepts of reinforcement learning and neural networks, along with it, cartpole was essential in testing and setting up the logging interface as well as testing different implementations of the learning algorithm

Keras-rl

Keras-rl is a community maintained high level implementation of keras agents for reinforcement learning. this was the first implementaiton tested.

The implementation of keras-rl is very easy and doesn't require a deep understanding of reinforcement learning and the leanring structures.

Keras API

The second implementaiton tested was using the plain keras API, while this provides more flexibility it also requires a much deeper understandng of how reinforcement learning works. The implementaiton using the Keras API was essential to develop a necessary knowledge for the project and to progress to the next stage.

While an implementaiton using Keras-rl would be simpler and even possibly ease the itteration process, this implementation provides less flexibility and given the target of the project and desire to develop a deeper understanding of reinforcement learning the implementation using the Keras API was choosen to implemnt the next phases.

Results

The cartpole served as an innitial testing environment for multiple variables and strategies.

The first attempt uses Boltzmann distribution as an exploration policy in this experiment an itteration on the learning rate was performed, along with the number of steps to achieve the target.



Figure 5.1: Reward output for the cartpole experiment - Boltzmann

As can be seen from the above graphs, the learning rate of 0.01 shows much better results when compared to 0.1, although when the model was tested after 5000 steps the results were as follows:

20 episodes test run		
	Learning Rate: 0.01 Steps 5000	Learning Rate: 0.01 Steps: 10000
Episode 1	126.00	500.00
Episode 2	121.00	500.00
Episode 3	129.00	500.00
Episode 4	124.00	500.00
Episode 5	124.00	500.00
Episode 6	123.00	500.00
Episode 7	122.00	500.00
Episode 8	123.00	500.00
Episode 9	124.00	500.00
Episode 10	123.00	500.00
Episode 11	125.00	500.00
Episode 12	116.00	500.00
Episode 13	125.00	500.00
Episode 14	126.00	500.00
Episode 15	125.00	500.00
Episode 16	127.00	500.00
Episode 17	118.00	500.00
Episode 18	125.00	500.00
Episode 19	122.00	500.00
Episode 20	119.00	500.00
	Average: 123.35	Average: 500.00

As we can observe from the table, the training length can have a significant impact on the results, in the case of the cartpole environment it helps to break harmful correlations, stopping the cart from moving off the boundaries.

The second attempt uses a more common exploration policy, epsilon greedy, in this experiment the epsilon, learning rate and metric used were also varied.

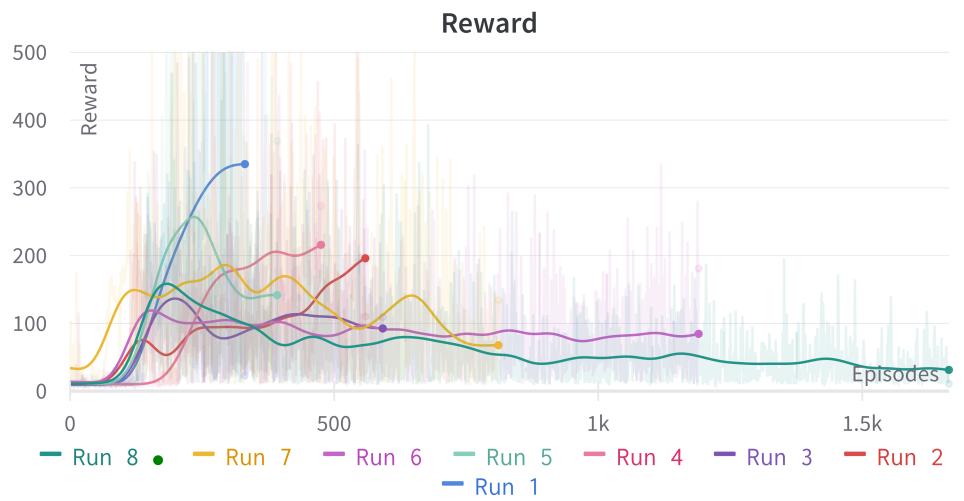


Figure 5.2: Reward output for the cartpole experiment - epsilon greedy

Hyperparameters	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
epsilon	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.3
learning rate	0.01	0.1	0.03	0.001	0.01	0.03	0.01	0.01
metric	mae	mae	mae	mae	accuracy	mae	mae	mae
number of steps	50000	50000	50000	50000	50000	100000	100000	100000

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
Episode 1	500.00	31.00	47.00	390.00	500.00	88.00	500.00	89.00
Episode 2	500.00	28.00	49.00	338.00	500.00	87.00	500.00	95.00
Episode 3	500.00	85.00	33.00	500.00	500.00	84.00	500.00	90.00
Episode 4	500.00	85.00	26.00	166.00	500.00	90.00	500.00	92.00
Episode 5	500.00	28.00	51.00	210.00	500.00	91.00	500.00	88.00
Episode 6	500.00	28.00	57.00	307.00	500.00	88.00	500.00	93.00
Episode 7	500.00	34.00	41.00	335.00	500.00	86.00	500.00	95.00
Episode 8	500.00	27.00	22.00	343.00	500.00	91.00	500.00	92.00
Episode 9	500.00	85.00	80.00	307.00	500.00	86.00	500.00	87.00
Episode 10	500.00	33.00	21.00	500.00	500.00	92.00	500.00	92.00
Episode 11	500.00	28.00	26.00	139.00	500.00	29.00	500.00	90.00
Episode 12	500.00	83.00	90.00	437.00	500.00	87.00	500.00	87.00
Episode 13	500.00	30.00	75.00	341.00	500.00	90.00	500.00	91.00
Episode 14	500.00	33.00	56.00	500.00	500.00	88.00	500.00	90.00
Episode 15	500.00	88.00	41.00	493.00	500.00	87.00	500.00	93.00
Episode 16	500.00	31.00	88.00	500.00	500.00	86.00	500.00	91.00
Episode 17	500.00	29.00	39.00	500.00	500.00	81.00	500.00	93.00
Episode 18	500.00	35.00	33.00	500.00	500.00	87.00	500.00	89.00
Episode 19	500.00	83.00	67.00	340.00	500.00	94.00	500.00	93.00
Episode 20	500.00	32.00	77.00	242.00	500.00	87.00	500.00	92.00
Average	500.00	46.80	50.95	369.40	500.00	84.95	500.00	91.10

Table 5.1: test

When comparing the two approaches, it is clear that the more common epsilon greedy approach was more successfull in solving the cartpole environment, achieving the maximum of 500 points in 50000 steps.

As can be observed by the table containing the average reward for 20 test episodes, we can observe that both run 1 and run 5 were successfull with 50000 steps, followed by run 4 with close scores. Both run 1 and 5 have 0.01 as the learning rate and only vary the metric used, any learning rate above 0.01 tested yelded low scores, when testing varying epsilons it was possible to solve the environment with double the steps, the results also demonstrate that setting random exploration to values higher than 20% couldn't be solved under the 10000 steps.

The conclusion drawn show that the most suitable parameters are:

- **epsilon** = 0.1
- **learning rate** = 0.01

- **metric** = mae
- **number of steps** = 50000

5.2 2D Environment Outcomes

When first implementing the 2D environment using the keras-rl library, it was found that the library doesn't support the use of Multidiscrete action spaces, an attempt to implement this environment using tensorflow Agents yielded the same results:

"ValueError: Only scalar actions are supported now"

Given this, it was necessary to implement a custom learning implementation using the Keras API.

The 2D environment was a big challenge to develop, this is due to the big jump in complexity in comparison with a simple, well known and documented environment such as the cartpole. The 2D environment required a custom 2D environment and the implementation of this with the physics engine. When this obstacle was overcome, the second major challenge was to implement the learning algorithm and be able to control the 8 joints simultaneously.

Once the learning algorithm was fully implemented, the reward system and hyperparameters needed to be tested and tuned, this shows the complexity of reinforcement learning. While many iterations over the reward system and hyperparameters were made, given the limited time and resources a full walking motion could not have been achieved. Although, the results obtained with experimentation were helpful in understanding the impact of the changes in the reward system.

Results

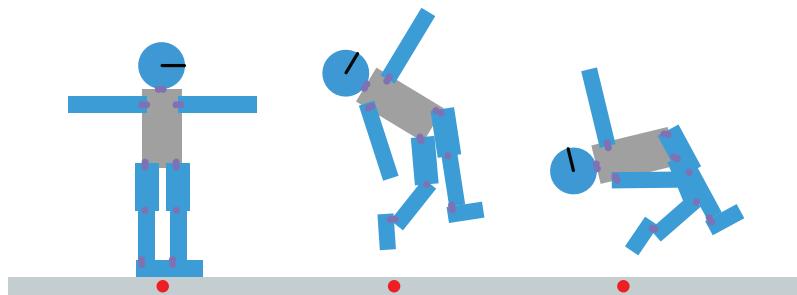


Figure 5.3: Simulation of 2D environment after 199 episodes

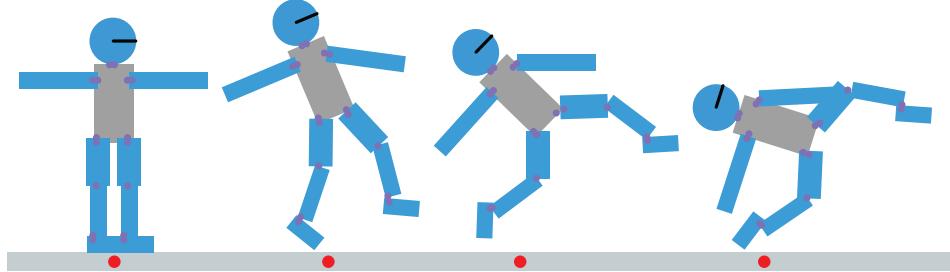


Figure 5.4: Simulation of 2D environment after 10651 episodes

Hyperparameters	
gamma	0.99
learning rate	0.01
epsilon maximum	0.1
epsilon minimum	0.01
batch size	64
units per hidden layer	128
loss function	mean absolute error

After 10000 episodes we start to observe how the agent starts to learn to move forward as it yields the most rewards as opposed to the example of 199 episodes where the agent would jump and fall almost in the starting position. From performing a parameter importance analysis on a larger number of runs using the wandb logger, the perceived notions are confirmed as can be seen bellow:

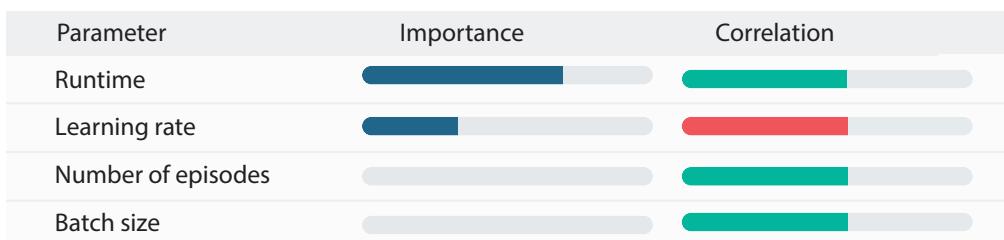


Figure 5.5: Parameter importance analysis of the 2D environment

5.3 3D Environment Outcomes

5.4 Reward Function

Chapter 6

Future Research

6.1 Empowerment

6.2 Reward Function Development

6.3 Policy Gradients

6.4 Mujoco Implementation

6.5 Real Robot Training

Chapter 7

Project Evaluation

Chapter 8

Conclusion

Bibliography

- [1] Weights and biases.
- [2] Chipmunk. Chipmunk. <http://chipmunk-physics.net>.
- [3] Alberto Ezquerro, Miguel Angel Rodriguez, and Ricardo Tellez. openai ros. http://wiki.ros.org/openai_ros, 2016.
- [4] Apple Inc. Tensorflow plugin - metal. <https://developer.apple.com/metal/tensorflow-plugin>.
- [5] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative, 1995.
- [6] Taylor McNally. keras-rl2. <https://github.com/taylormcnally/keras-rl2>, 2021.
- [7] OpenAI. Cartpole. https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py, 2016.
- [8] OpenAI. Introduction to rl. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html, 2018.
- [9] Pygame. Pygame. <https://www.pygame.org>.
- [10] Pymunk. Pymunk. <http://www.pymunk.org>.
- [11] Marcus M. Scheunemann, Sander G. van Dijk, Rebecca Miko, Daniel Barry, George M. Evans, Alessandra Rossi, and Daniel Polani. Bold hearts team description for robocup 2019 (humanoid kid size league). *arXiv:1904.10066 [cs]*, Apr 2019. arXiv: 1904.10066.
- [12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. MIT Press, Nov 2018. Google-Books-ID: uWV0DwAAQBAJ.

- [13] Alec Jeffrey Thompson and Nathan Drew George. Deep q-learning for humanoid walking, Apr 2016.