Bipedal Walking - Reinforcement Learning

Roberto Figueiredo

April 2022

Abstract

This report covers the attempt to develop a bipedal walking pattern using reinforcement learning.

This project developed a series of experiments progressing from a testing ground using the CartPole environment to a 2D walker and implementation of OpenAI Gym on the RoboCup Team, Bold Hearts, ROS2 environment enabling training for any reinforcement learning task. The experiments cover the testing of two different learning implementations and hyperparameter tuning.

Although the objective of walking could not be achieved in the target time frame with the limited resources available, the documentation covers the problems found along with the development and how the difficulties were overcome. The implementation chapter will cover how each of the stages was developed, with coding examples for each of the main components.

The results and findings of the experiment will be explored and an attempt to understand what could be changed and what is responsible for not achieving the desired outcome.

Acknowledgement

It is important to start by thanking Professor Daniel Polani as project supervisor and Doctor Alessandra Rossi as team captain and mentor, not only for all the support and guidance but for making me passionate about science making and for piquing my curiosity.

It is also important to mention all my fellow teammates from Bold Hearts for the patience and for all I've learnt from them during the last four years.

Lastly, my family and friends for always supporting my decisions and supporting me over the hardest times.

Contents

Chapter 1

Introduction

RoboCup is a worldwide competition introduced to bridge the gap between robotics and new advancements in AI technology. Each year thousands of RoboCupers get together to compete and share knowledge and advances in robotics. The Bold Hearts team competes in the RoboCup Soccer Humanoid League. In this Chapter, these topics will be introduced to understand the problem at study better.

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 collaboration. 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 an 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.

The middle size League assimilates to the small-size league, but in this case, the robots are of a larger size and must have all sensors on board; The league's primary focus 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. Unlike robots outside the Humanoid League, the task of perception and world modelling 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, UH RoboCup Team

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. [13]

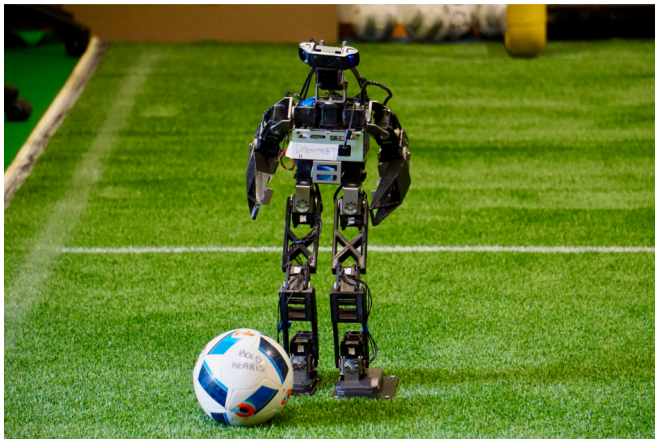


Figure 1.1: Boldbot, the robot built and used by the Bold Hearts team

1.4 Introduction to the Project

1.4.1 Problem, Walking for Humanoid robots

Until a few years ago, Robotic locomotion was 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.

The Bold Hearts teams robots must be able to walk and, in addition to the high complexity of these challenge, RoboCup rules periodically change. These changes are implemented as the RoboCup objective is to achieve the most realistic environment and 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. Walking is one of the most complex movements performed by humans, requiring simultaneous control of multiple joints to move while maintaining balance. Not only due to its natural complexity but also aggravated by the rules changes and continuous improvements leading to new updates to the walking algorithm, this is one of the team's most energy and time-consuming problems.

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 maintain, 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 modelled in the simulator. This also reduces 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.

This project aims to develop an implementation of reinforcement learning to achieve a walking pattern for humanoid robots. It aims to develop detailed documentation on the processes and their results, allowing for reproducibility, problem mitigation, and further development. The last objective for this project is to develop an integration of OpenAI Gym and ROS2. This allows not only to develop a walking algorithm but may also be used by the team to develop any reinforcement learning task.

Chapter 2

Background Research

This chapter introduces key background research necessary to understand reinforcement learning and the decisions made. It will cover reinforcement learning, the Markov Decision Process, the Bellman equation, learning algorithms, training frameworks, previous implementations and logging platforms.

2.1 Reinforcement Learning

Reinforcement learning is one of the main machine learning paradigms, alongside supervised learning and unsupervised learning. Reinforcement learning aims to map states to actions while maximising 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. These characteristics, 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 summarised in the following figure.

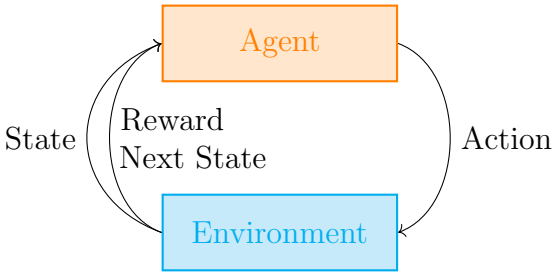


Figure 2.1: Reinforcement learning model.

One unique challenge in reinforcement learning is balancing exploration and exploitation. To succeed in 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 [15]

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 Property
- The Markov Decision Chain
- The Markov Reward Process

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 all the relevant information from the past. Defined by the following equation:

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, ..., S_t)$$

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 arrives 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.

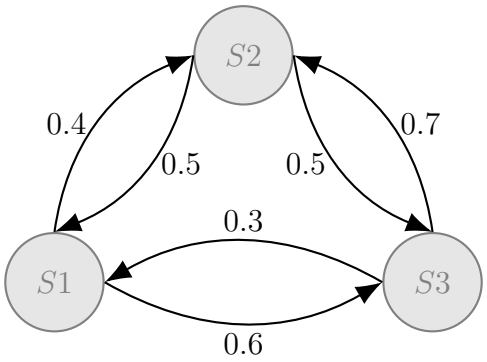


Figure 2.2: Example of a Markov Decision Chain.

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 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,

these extreme values are ineffective, and γ is usually set to values between 0.9 and 0.99

2.1.2 Bellman equation

As the agent moves through the Markov decision chain, one problem develops, how to choose the path that maximises future rewards from the current point onwards.

To solve this problem, a recursive equation is used, the Bellman equation.

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

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

The optimal value function $V^(s)$ maximises 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.

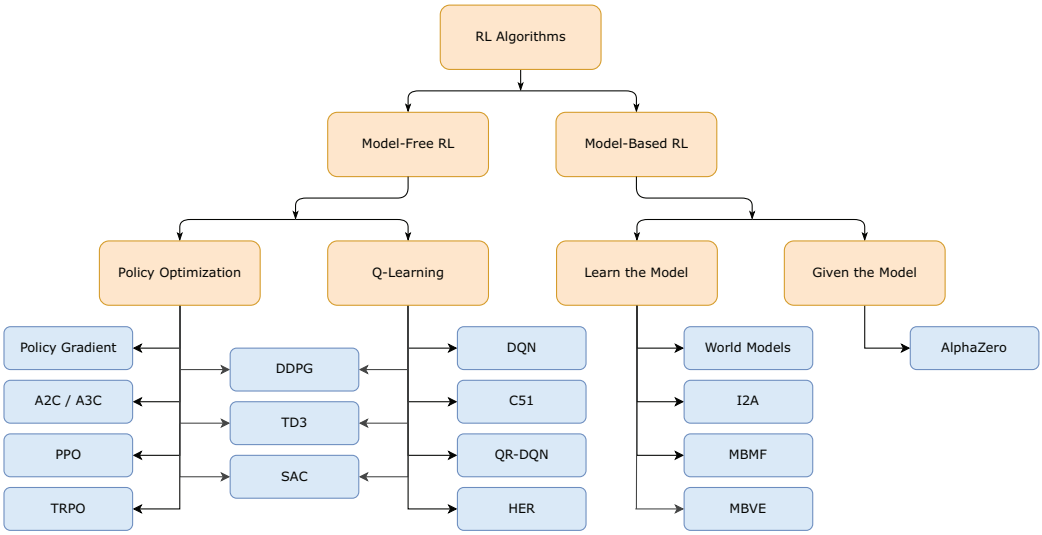


Figure 2.3: A non-exhaustive, but useful taxonomy of algorithms in modern RL from OpenAI[10].

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 reinforcement learning method is inadequate 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 approaches have similar concepts and are driven by MDP, they are internally different.

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

Q-Learning was a better fit for the problem in study as the action space was discretised. 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.

The OpenAI team released Gym to enable this, by providing an accessible API and standardised 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[8].

Gym was chosen as the framework to standardise the implementation which allows for an easier understanding and comparison, beyond this, the Gym also facilitates the implementation by using all the pre-built functions[1].

2.4 Previous Implementations

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

One important implementation was the **Deep Q-Learning for Humanoid Walking**[16], an implementation of reinforcement walking on the Atlas platform. This implementation highlights 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 is data logging and reproducibility. This project required extensive testing of hyperparameters and reward systems. To understand the results and their correlation with the variables in testing as well as being able to

reproduce them, it is important to log all the hyperparameters and 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 chosen platform, WandB offered a 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 renderings and performance results between runs.

Chapter 3

Design

This chapter will be split into two main sections, development structure and environment definition, the first will define how the project will be developed and explain each stage. The second section, will define the design for each of the environments used in the development.

3.1 Development Structure

Due to the project's complexity, 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 more manageable.

3.1.1 CartPole

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

The CartPole environment allows for implementing and testing 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 control interface. This allows for the implementation of a neural network, a learning algorithm capable of handling multiple simultaneous outputs as it is required to control all the joints of the robot and understand how to calculate the best action efficiently.

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 essential steps to transition to 3D simulation.

3.1.3 3D Walker

The final stage of the project is the implementation of a 3D simulated robot. This is the combination of the previous stage with extra complexity, not only due to the inherited complexity of a higher dimensional 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 primary 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 break the robot.

3d simulation brings new challenges, such as a larger range of motion and more joints to control, 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 dimension poses new problems.

3.2 Environment Definition

One of the main reasons for using 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 a 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. [9]

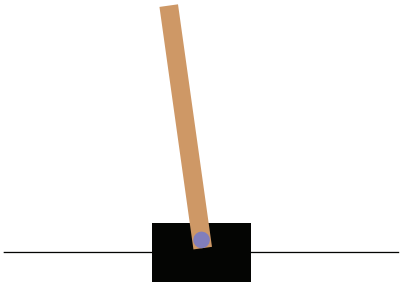


Figure 3.1: Rendered CartPole Environment - OpenAI Gym

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.

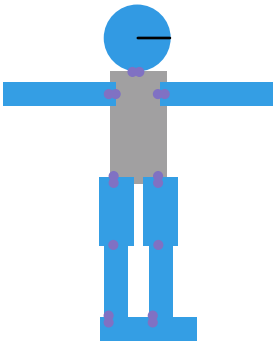


Figure 3.2: Representation of 2D humanoid

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	8
Stays in Place	9
Moves Forward	10
Loses contact with the ground	cumulative -2
Reaches target	16
Falls	0

The reward is calculated at each timestep, given a threshold θ, the position of the robot is compared to the last position and calculated the offset. If the robot loses contact with the ground with both feet, the reward is subtracted 2 points.

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.

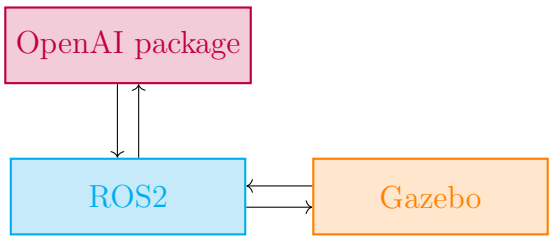


Figure 3.3: Integration of ROS2, Gazebo and OpenAI Gym

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 call services in order to control the simulation, including pause, unpause 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 files:

- Robot

- Environment
- Task

The robot file should contain all the setup required for the Boldbot, the team's robot. On the Environment file, the observation space, action space, reward system and other core Gym functions should be defined. Finally, the task file should be specific for the task trying to be achieved, allowing the team to set up just the task without requiring setting up the robot and environment every time. [3]

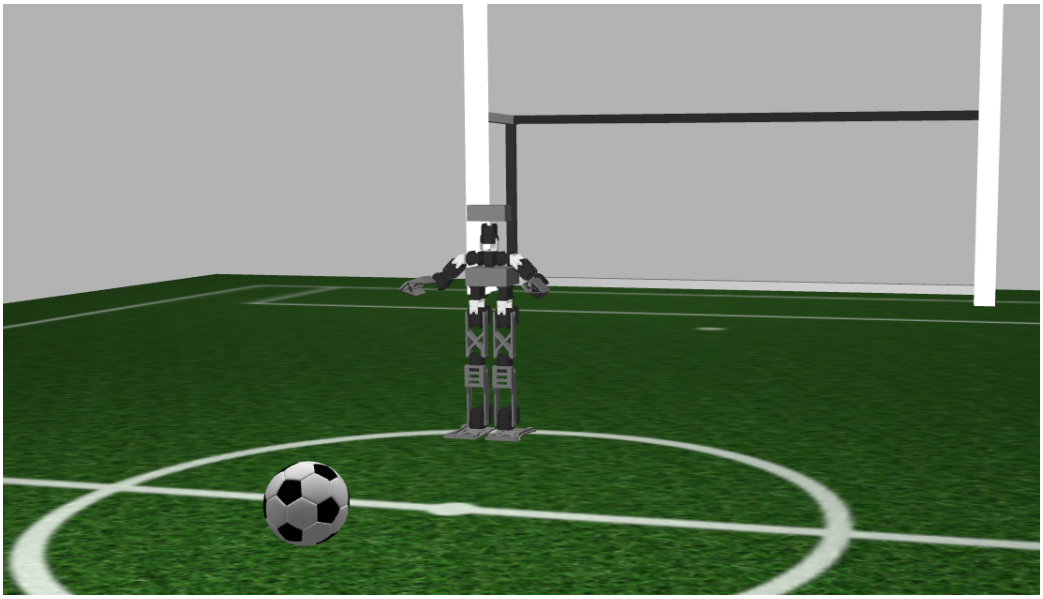


Figure 3.4: Bold Hearts simulation, the robot, field, goals and ball are simulated

The 3D Walker will be using the current simulator, demonstrated above, using the already modelled humanoid and field.

Chapter 4

Implementation

This chapter covers how each stage was implemented, including code snippets, diagrams of how each component is linked and decisions that were made.

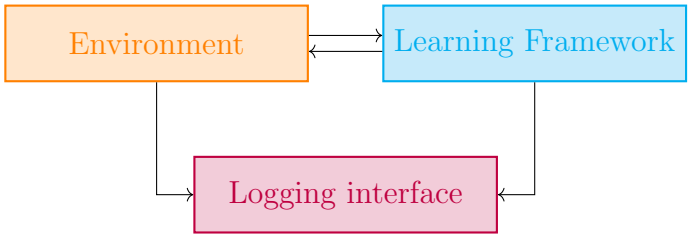
4.1 CartPole

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 version 1 was prefered 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 these should interact for a successful implementation of the first stage. After the environment is imported, the learning framework is imported, this is a framework used to train the agent. Two main implementations were

explored in this stage, Keras-rl and a manual implementation of the Keras API.

Keras-rl

To implement keras-rl[7] in the specific hardware used in this project, it was required to be installed from source inside a miniforge environment. To be able to set up 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.

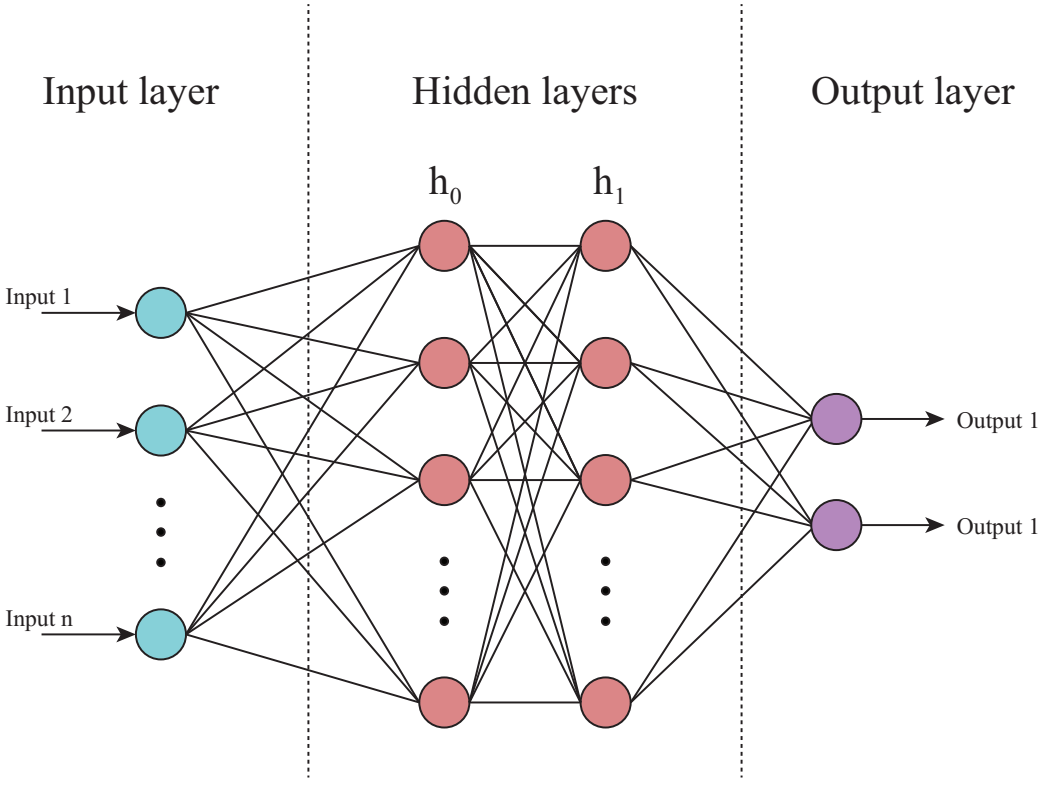


Figure 4.1: Visual representation of the neural network used to solve the CartPole Environment.

An implementation of the neural network used for the CartPole environment, using the structure demonstrated above is shown in the code snippet below.

Listing 4.1: Setting up the model in the CartPole environment

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

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

def build_model(states, actions):
model = Sequential()
model.add(Input(states))
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 and it connect to a fully-connected (Dense) 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 in 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 is imported from Keras.

The *build_agent* function is used to set the DQNAgent, the exploration policy is initiated 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 initiated 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, and the sample obtained from the buffer contains a batch of states, actions, rewards, next states and done values. These 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** to improve 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 ten 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, iterating for the number of set episodes over the training code.

The episode starts by resetting 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 is 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.

Weights & Biases - WandB logging platform

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.[18]

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 set up on the same machine used to run the training. To set up 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 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 additional metrics can be logged.

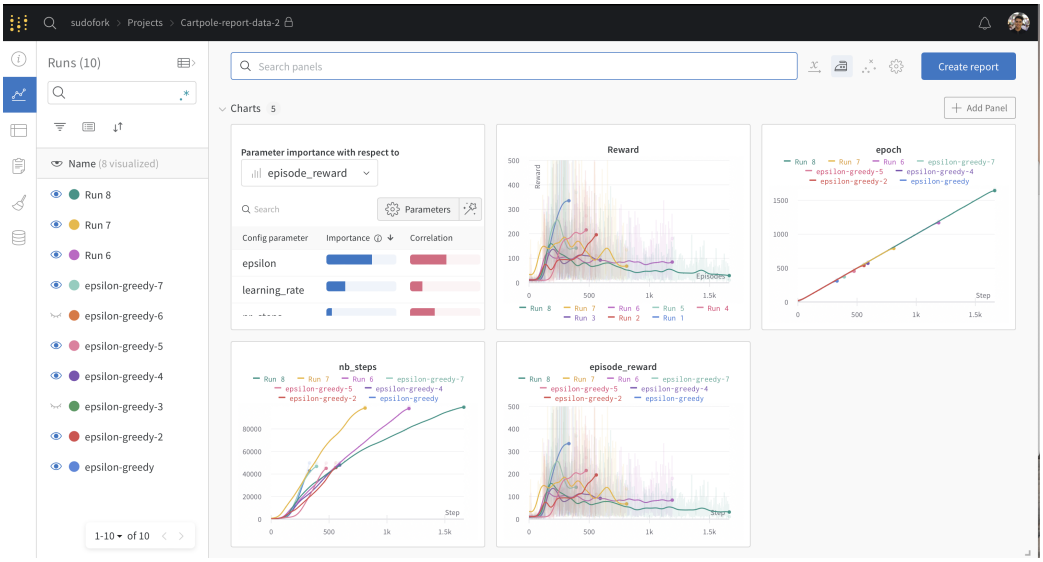


Figure 4.2: Logging data and artifacts to WandB

4.2 2D Walker

Implementing 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[12], a python implementation of Chipmunk[2], and Pygame[11] as the latter is integrated into 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 three distinct possible actions, which can be described using a Discrete space, although, as there are eight distinct joints, the resulting actions space is [3]*8, which 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, the higher bound and shape, hence the [8], as there is an observation for each of the eight 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 is applied to the robot motors, although this will not take an effect before the robot is updated and the space takes 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's 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 deducted 2 points if both feet lose contact with the ground.

Referring 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's vertical position is lower than the initial vertical position minus a threshold (50), the robot is considered to have fallen. If the robot's 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 set 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 horizontal target position defined. If the robot reaches 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 joint's 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 physics 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 by passing the width and weight 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 set up, it was required to have a neural network able to support the eight simultaneous actions.

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

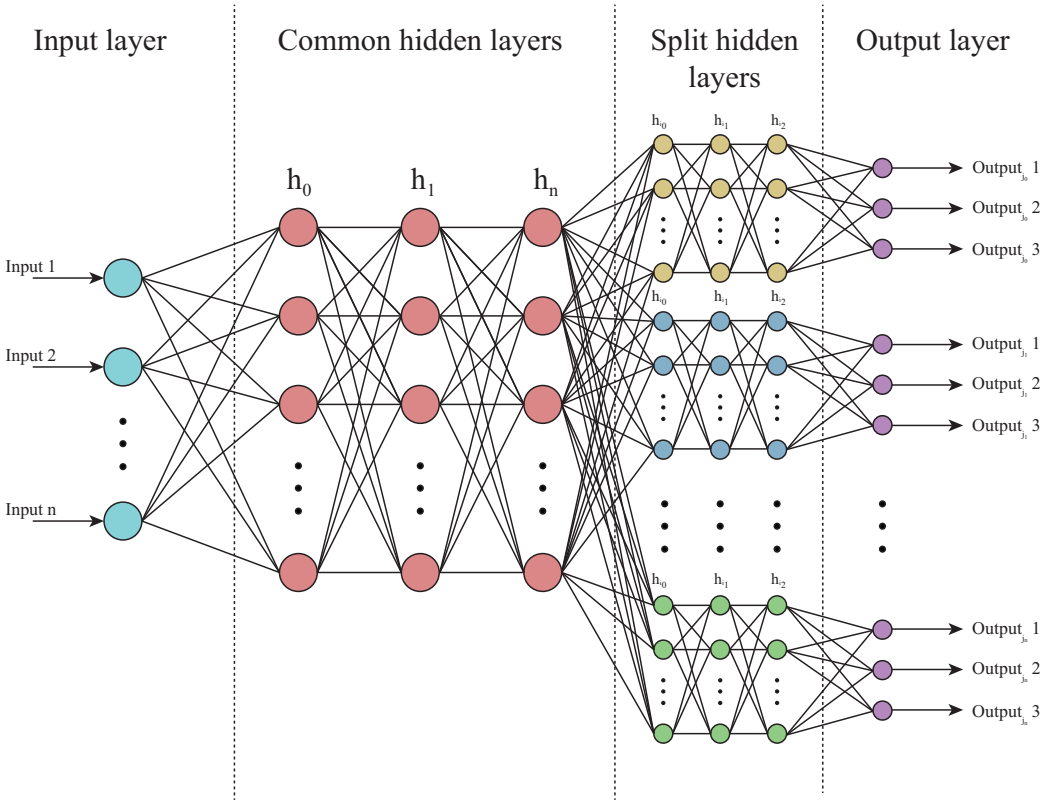


Figure 4.3: Neural network structure used in the 2D environment

Implementation of the neural network using the Keras API:

Listing 4.16: Implementation of the neural network using the Keras API

```
def build_model():
    # Define model layers.
    units = 128
    input_layer = Input(shape=(8,))

    x = Dense(units=units, activation='relu')(input_layer)
    x = Dense(units=units, activation='relu')(x)
    x = Dense(units=units, activation='relu')(x)
    x = Dense(units=units, activation='relu')(x)
    x = Dense(units=units, activation='relu')(x)
    x = Dense(units=units, activation='relu')(x)
    x = Dense(units=units, activation='relu')(x)

    y1 = Dense(units=units, activation='relu')(x)
    y1 = Dense(units=units, activation='relu')(y1)
    y1 = Dense(units=units, activation='relu')(y1)
```

```
    y1 = Dense(units=units, activation='relu')(y1)
    y1 = Dense(units=units, activation='relu')(y1)
    y1_output = Dense(units='3', name='motor_1', activation="linear")(y1)
    y2 = Dense(units=units, activation='relu')(x)
    y2 = Dense(units=units, activation='relu')(y2)
    y2 = Dense(units=units, activation='relu')(y2)
    y2 = Dense(units=units, activation='relu')(y2)
    y2 = Dense(units=units, activation='relu')(y2)
    y2_output = Dense(units='3', name='motor_2', activation="linear")(y2)

    model = Model(inputs=input_layer,outputs=[y1_output, y2_output])

    return model
```

The code above reduces the number of outputs and therefore layers created to avoid code repetition.

One necessity noticed later during testing was how the reward was calculated during training, as the reward is calculated based on the epsilon greedy action selection it doesn't fully represent the state of the model. To fix this problem a testing function was added:

Listing 4.17: Defining the testing function

```
def test_model(self):
    rewards = []
    for _ in range(5):
        state = env.reset()
        reward_accumulater = 0
        done = False
        step_count = 0
        while not done:
            state = np.expand_dims(state,0)
            action = np.argmax(self.q_model.predict(state), axis=2).flatten()
            state, reward, done, _ = env.step(action)
            reward_accumulater += reward
            step_count += 1
            if step_count>900:
                done = True
        rewards.append(reward_accumulater)

    run.log({
        "test_reward_average":np.average(rewards),
        })
```

This function runs everytime the target model is updated and logs the value to WandB, allowing for a better understanding of the state of the model.

4.3 3D Walker

The initial design for the 3D environment was set to use Mujoco, a physics engine that has recently open-sourced when acquired by DeepMind. Although, this does not allow for direct integration with ROS2 and would require developing a new control interface for the robot or a compatibility layer to link the team's control and the simulator.

Given the problems mentioned above and time constraints, using the already working simulator used by the team, Gazebo, was a better option, and therefore the decision to test Mujoco and compare its performance to Gazebo was abandoned.

To complete the 3D implementation it was necessary to develop a new Gym environment and re-purpose the learning code. openai_ros is an existing ROS package that implements the OpenAI Gym environment in ROS, using the following architecture:

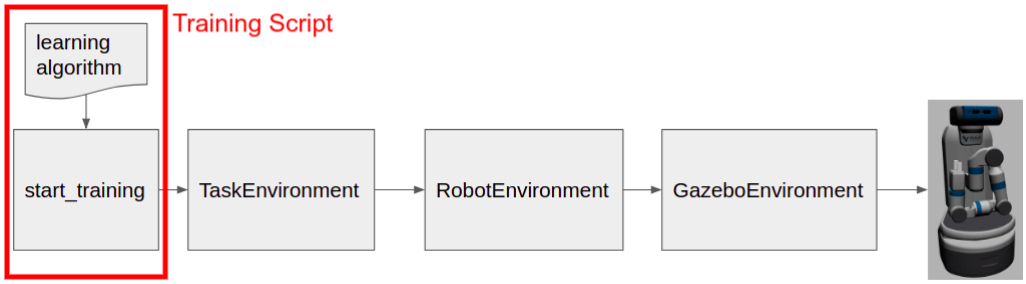


Figure 4.4: openai_ros package architecture[3]

Although the package was developed to be used with ROS and Bold Hearts use ROS2 and due to some incompatible dependencies, it could not be used, although a ROS2 version based on the original was developed [14]. When trying to compile the package on the Bold Hearts ADE it failed as it required Gym and the ADE does not have the package and does not have an internet connection as the docker container is in isolation mode, to fix this problem it was necessary to enable external connections to the docker container.

Listing 4.18: Enabling external connections to the docker container

```
$ pkill docker
$ iptables -t nat -F
$ ifconfig docker0 down
```

```
$ brctl delbr docker0
$ docker -d
$ sudo systemctl restart docker
```

After the docker container was able to connect to the internet the package was able to be compiled.

The first process to be able to control the robot was to understand all the topics and services it would require to control the simulation and the robot.

- **Push commands to the joints**

Listing 4.19: Push commands to the joints

```
self.node.create_publisher(
    JointCommand,
    "/cm730/joint_commands",
    10
)
```

- **Get the joint information**

Listing 4.20: Get the joint information

```
self.node.create_subscription(
    JointState,
    "/joint_states",
    self.joint_status_callback,
    0
)
```

- **Pause the simulation**

Listing 4.21: Pause the simulation

```
self.node.create_client(Empty, '/pause_physics')
```

- **Resume the simulation**

Listing 4.22: Resume the simulation

```
self.node.create_client(Empty, '/unpause_physics')
```

- **Reset the simulation**

Listing 4.23: Reset the simulation

```
self.node.create_client(Empty, '/reset_simulation')
```

With the main controls defined, a proof of concept implementation was put in place by simplifying the architecture and ignoring the robot environment and task environment, concentrating only the basic required code into an environment file and a separate file would initiate the environment, sample random actions and push them to the robot.

The proof of concept was successful and the robot would perform the published random movements. The next step is now to develop the code into the defined architecture, this allows for an abstraction of the robot, environment and task. This architecture is important to avoid code duplication and allows the team just to define a new task when new challenges arise.

Chapter 5

Experimental Results

This chapter will cover the multiple experiments conducted, it will cover a comparison of Boltzmann exploration and epsilon greedy exploration and experiments with different hyperparameters and how this affects the results.

5.1 CartPole Outcomes

The first task developed was the classic CartPole environment, this was helpful 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 implementation tested.

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

Keras API

The second implementation tested was using the plain Keras API. While this provides more flexibility, it also requires a much deeper understanding of how

reinforcement learning works. The implementation using the Keras API was essential to develop the necessary knowledge for the project and to progress to the next stage.

While an implementation using Keras-rl would be simpler and even possibly ease the iteration 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 chosen to implement the next phases.

Results

The CartPole served as an initial testing environment for multiple variables and strategies.

The first attempt uses Boltzmann distribution as an exploration policy in this experiment, an iteration 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:

Table 5.1: 20 test runs for the CartPole experiment, comparing the results of two different training lengths.

	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 of the boundaries.

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

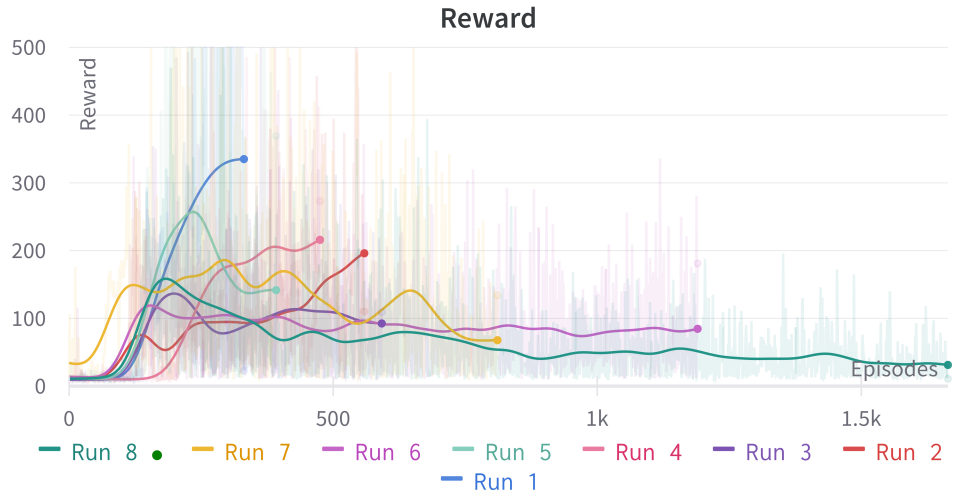


Figure 5.2: Reward output for the CartPole experiment - epsilon greedy.

Table 5.2: Hyperparameters for each of the 8 runs using 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

Table 5.3: Results for 20 test episodes for each of the eight training sessions. In the tests, all actions are chosen using the trained model.

	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

When comparing the two approaches(epsilon greedy and Boltzmann), it is clear that the more common epsilon greedy approach was more successful 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 successful 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 yielded 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 by only supporting scalar actions.

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 eight 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 experientation were helpfull in understading the impact of the changes in the reward system.

Results

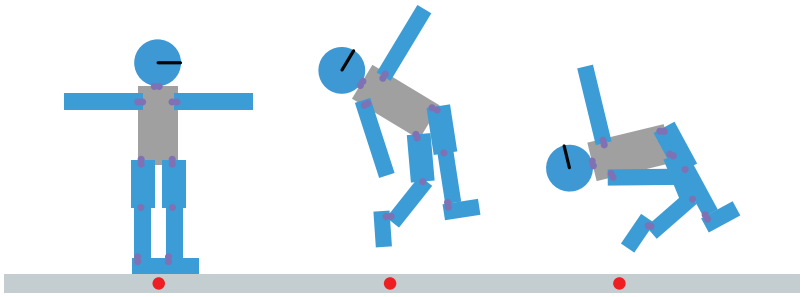


Figure 5.3: Simulation of 2D environment after 199 episodes, the red dot is a reference used to understand the relative movement of the robot to a stationary point. See text for more details.

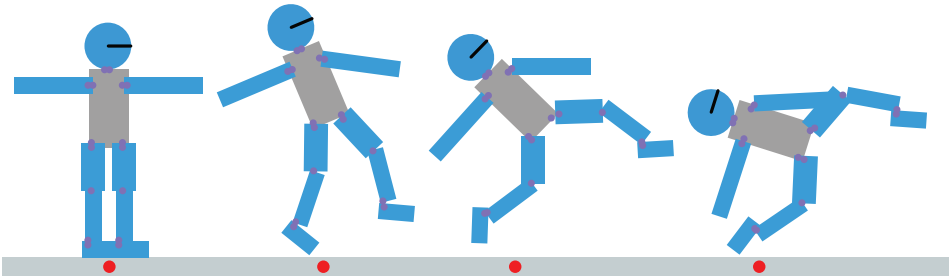


Figure 5.4: Simulation of 2D environment after 10651 episodes, the red dot is a reference used to understand the relative movement of the robot to a stationary point. See text for more details.

Table 5.4: hyperparameters from the run demonstrated above.

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.

After many iterations over the reward system and hyperparameters, the reward system performs better when the range of reward is positive. After removing penalties and setting the rewards in the range [0,16], the agent showed more expectable results and was able to learn to reach targets in close range.

One of the observations made during trained is how the agent rapidly learns to balance itself in the same position. This happens because in the short term, it's more favourable to stay still than to fall. This behaviour converges very fast, although, as learning to walk is a complex pattern, it would require long training sessions with potentially millions of episodes to update the neural network.

When testing why the robot wasn't able to learn to walk, the q-values showed very little variation, independently of the input. The reason for this is that the learning rate, when set to 0.01, would require much lengthier training sessions. When applying the same testing to sessions with learning rates as high as 0.1, the q-values would converge very fast, but the agent wouldn't perform well.

Bellow are the comparison of selected runs to which were applied different hyperparameters, as an attempt to find the most suitable and how varying this affects the results.

Figure 5.5: Comparison of multiple selected runs using different hyperparameters.

From the graph above we can observe that two of the runs perform much better than the remaining, this is explained, confirming previous beliefs, by using an importance and correlation analysis:

Figure 5.6: Importance and correlation analysis of the selected runs.

Table 5.5: Hyperparameters used for the selected experiments explored in this chapter.

Hyperparameters	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Batch size	50	12	12	128	32	50
Epsilon	0.8	0.8	0.8	0.8	0.8	0s.3
Epsilon minimum	0.1	0.1	0.1	0.1	0.1	0.02
Gamma	0.65	0.65	0.65	0.65	0.65	0.99
Learning rate	0.01	0.05	0.01	0.01	0.01	0.01
Loss	mae	mae	mae	mae	mae	mae
Units	128	64	64	64	256	128

The above graph is generated using WandB, it shows the linear **correlation** between the hyperparameter and the reward obtained, the red lines represent a negative value and the green lines represent positive values, therefore, when the hyperparameter has a higher value, the metric also has higher values and vice versa. Although, correlation alone isn't a very good metric as it does not capture second-hand interactions between parameters.[17] Therefore an importance metric is also calculated using a random forest with the hyperparameters as inputs and the metric as the target output and reports the feature importance values for the random forest.

This analysis shows how the number of units in the neural network impacts the results, coming up as the most important hyperparameter in the testing. The second hyperparameter is batch-size, affecting how much data is fed into the network. Gamma and its high correlation with the episode reward

is expected, as a higher gamma values more long-term rewards. The next two hyperparameters, epsilon and epsilon min(minimum), are related, ϵ is the initial value of exploration while epsilon min is the value of exploration at the end of the episode. The data shows that lower epsilons work better for the tests conducted, it was a common observation, not only in the selected runs, but also along the entire development process. In this specific case, higher exploitation works better. The best Run tested, Run 6, ϵ was set to 0.3 and epsilon minimum to 0.02.

5.3 3D Environment

As mentioned, this scenario, due to time constraints, limited resources and unforeseen challenges in implementing the 2D environment, was not tested using a learning algorithm. Although, this stage was implemented in ROS2 as it was important to have the technology ready for development and testing, not only for walking but to be used by the team to train any suitable tasks.

Chapter 6

Result Discussion

6.1 CartPole Results Discussion

In the CartPole stage, two exploration strategies were tested, Boltzmann and Epsilon Greedy. The more popular Epsilon Greedy had the best results being able to converge to an optimal policy in half the number of steps compared to Boltzmann.

The second experiment conducted in CartPole was to find the optimal hyperparameters. As could be seen in the Results section, the optimal parameters found were learning rate of 0.01 and epsilon of 0.1. The learning rate of 0.01 tends to be a starting point for testing hyperparameters in reinforcement learning and is often the optimal value. The value for epsilon(ϵ) is the probability of choosing a random action. There needs to be a balance between exploration and exploitation. The experiment covered exploration rates starting at 10% and tested both 20% and 30%, although 20% exploration was only able to converge after 100000 steps, compared to 10% which converged at 50000 steps. Any value tested over 20% didn't converge in 100000 steps.

6.2 2D Walker Results Discussion

Most of the experiments in this stage iterated over the reward function. After a lot of iterations using penalties (negative rewards) and shifting the rewards to use only positive rewards, the results became better as the walker stopped bending backwards.

Although, over the experiments realized, when testing the q-values and their variance depending on the inputs, for the learning rate of 0.01, the q-values were very close to each other and changed very little after about 400000 steps tested when testing with a learning rate of 0.1 the q-values converged fast although the results were suboptimal.

Chapter 7

Future Research

Reinforcement learning is a very complex topic, specifically when applied to such a complex movement as robotic bipedal walking. There are many approaches to solving reinforcement learning problems. Although many areas of interest couldn't be covered, this can be explored in future research on Bipedal Robotic walking and reinforcement learning in general.

As was shown in this report, developing an optimal reward function can be complex, which is an unsolved problem in the reinforcement learning field. One promising topic suggested by the project supervisor was empowerment[6] by intrinsic motivations. Empowerment is a technique that aims to overcome the reward problem by equipping the robot with intrinsic rewards, using rewards such as curiosity and empowerment.

The developed project uses discrete actions although, there is a loss of information when using this method, for future research, it would be valuable to compare a policy gradient approach using continuous actions.

The initial design for this project was to use Mujoco as a physics engine. However, due to time constraints, the plan had to be changed to use Gazebo as the environment was already modelled and integrated directly with ROS2. Nonetheless, the use of Mujoco is still of interest to the team, and a comparison against Gazebo can be researched.

One of the problems when developing something as simple as a motion script for the robot is the transition to the real robot, while the script might work perfectly in the simulator, it will require adaptation to work on the real robot. This is a topic of interest for the future and understanding how, using the simulator to train most of the movement, the learning can be transferred to the real robot and how to tune the learned model when testing on the robot.

Chapter 8

Project Evaluation

Walking for bipedal robots using reinforcement learning was an ambitious project. Although walking could not be achieved, this project successfully developed a working 2D walking environment with a simple humanoid, implementing OpenAI Gym with ROS2 and developing documentation and preparation for future research on the topic.

The training for this project was mainly executed using Google colab+, as incompatibilities with the computer architecture (ARM64) made training locally slower in comparison. Although, Google colab constantly crashed due to timeouts and unknown problems, making it impossible to train for extended periods of time. On a retrospective, it would have been a better decision to train locally as even if the training was slower, it would be able to run for very long periods of time.

As already explored, the reward function should have been tested from the beginning exclusively with positive rewards, as this would have allowed for a more efficient experiment.

Although no training was executed, the last stage was a significant achievement as it is a useful tool and development not exclusive to this project but for the team.

Chapter 9

Conclusion

Although the project's main target wasn't achieved, many objectives were successful. A 2D simulator and a 3D implementation of OpenAI Gym and ROS2 were achieved. Along with this, documentation produced shows what problems were faced and how these were mitigated, including training using Google colab, installing openai_ros2 in the team's environment and mistakes to avoid, such as developing a reward function with a big focus on penalties.

Starting with a simple environment such as the CartPole was good for the future of the project as it allowed for developing and testing the code in an already tested environment. It also allowed for experimentation with hyperparameters and strategies without wasting a lot of time in training.

Developing the 2D Walker was a complex task due to the novelty of using new tools such as pymunk and pygame. Pymunk proved to be a good choice for the implementation as it is well documented, easy to use and allows for rendering directly without code duplication in pygame.

The training duration and computational power were the most limiting factors in developing this project. To achieve an optimal policy, it is clear that more and longer runs are required for testing hyperparameters and reward function variations.

A working implementation using openai_ros2 was achieved and is ready for testing using a learning algorithm for future research.

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016. arXiv: 1606.01540.

[2] Chipmunk. Chipmunk. `http://chipmunk-physics.net`. last visited: 2020-03-10.

[3] Alberto Ezquerro, Miguel Angel Rodriguez, and Ricardo Tellez. openai ros. `http://wiki.ros.org/openai_ros`, 2016. last visited: 2022-03-20.

[4] Apple Inc. Tensorflow plugin - metal. `https://developer.apple.com/metal/tensorflow-plugin`. last visited: 2022-04-12.

[5] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative, 1995.

[6] Navneet Madhu Kumar. Empowerment driven exploration. last visited: 2022-04-18.

[7] Taylor McNally. keras-rl2. `https://github.com/taylormcnally/keras-rl2`, 2021. last visited: 2022-04-22.

[8] OpenAI. Gym. `https://gym.openai.com`. last visited: 2022-04-22.

[9] OpenAI. Cartpole. `https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`, 2016. last visited: 2022-04-22.

[10] OpenAI. Introduction to rl. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`, 2018. last visited: 2022-03-18.

[11] Pygame. Pygame. `https://www.pygame.org`. last visited: 2022-03-30.

[12] Pymunk. Pymunk. `http://www.pymunk.org`. last visited: 2022-03-30.

[13] 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.

[14] siw engineering. openai_ros2. `hhttps://github.com/siw-engineering/openai_ros2`. last visited: 2022-04-18.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. MIT Press, Nov 2018. Google-Books-ID: uWV0DwAAQBAJ.

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

[17] WandB. Wandb - parameter importance. `https://docs.wandb.ai/ref/app/features/panels/parameter-importance`. last visited: 2022-04-22.

[18] WandB. Weights and biases. `https://wandb.ai/site`. last visited: 2022-04-22.