A MULTIAGENT COOPERATION GAME USING REINFORCEMENT LEARNING

Abdullah. Alabdullatif. Author

3rd Year Project Final Report

Department of Electronic &
Electrical Engineering

UCL

Supervisor: Prof. Wong

30 March 2022

I have read and understood UCL’s and the Department’s statements and guidelines concerning plagiarism.

I declare that all material described in this report is my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures and computer programs.

This report contains 17 pages (excluding this page and the appendices) and 11353 words.

Signed: Abdullah Alabdullatif Date: 30/3/2022

(Student)

Abstract

In this project, a game environment called Google Research Football was chosen to research the possibility of cooperation between agents using multi-agent Reinforcement Learning. A series of experiments were conducted that led to step-by-step improvements in the Reinforcement Learning models. Starting with single-agent experiments on empty goal scenarios in the game, up to multi-agent experiments in 3 vs 1 scenarios similar to those in the literature. The main findings from the project include that when using Tensorflow and Keras, predictions made using Model() are more efficient (35x faster) than using Model.predict(). Another main result is that the checkpoint reward function is a disadvantage in some certain cases, when the scenario is too easy, since it leads to exploitation and overfitting. As for model training accomplishments, in the empty goal scenario, the issue of overfitting caused the scenario to be solved within 64k steps (compared to 1M steps in the literature), as the player only performs one action which is running straight into the goal. As for the main multi-agent result, a 2-player model learnt to pass through a defender by passing to each other, after training for 5M steps.

Introduction

Artificial intelligence (AI) has been an integral part of video games since the 1950s. However, it used to consist mainly of rule-based AI, which are hard-coded algorithms. As opposed to rule-based AI, Machine Learning is a relatively new and exciting field of study that is a subset of AI, with a massive impact across different industries. It could be defined as the development of computer algorithms that improve automatically through processing large amounts of data, as opposed to hard-coded computer algorithms that follow a set of rules and cannot self-improve. Machine Learning is split into three main categories: Supervised learning, Unsupervised learning, and Reinforcement learning. This project focuses on Reinforcement learning, which is, put simply, a trial and error approach to solve problems. The agent is placed in an environment and is required to solve or achieve a specific goal by choosing an action to take given a current state in the environment. With each action taken, a reward is given to the agent, if this reward is positive, then the agent learns that this was a suitable action to take in that state and vice versa.

There are different types of environments to be researched. Single-agent environments are ones where an environment is explored by a single agent. This project focuses on multi-agent environments, where multiple agents are training and interacting within the same environment. The environment chosen in this project is a game that requires cooperation between agents, which is Google Research Football.

The motivation for research in this field the importance it holds for the development of technologies such as self-driving cars, as it is a prime example of a multi-agent environment that could be developed with Reinforcement Learning.

Literature review

An interesting research paper within the field of multi-agent Reinforcement Learning is the OpenAI Multi-Agent Hide and Seek game [1]. In this environment, agents play a hide-and-seek game where there are two teams, one with two hiders and the other with two seekers. While agents were training using Reinforcement Learning algorithms, six distinct strategies emerged as a result. At the start, all agents were making random movements and decisions, up until a certain point where the chasers managed to catch the hiders and learnt from that reward. From that, strategies began to develop as shown in Figure 1 below.

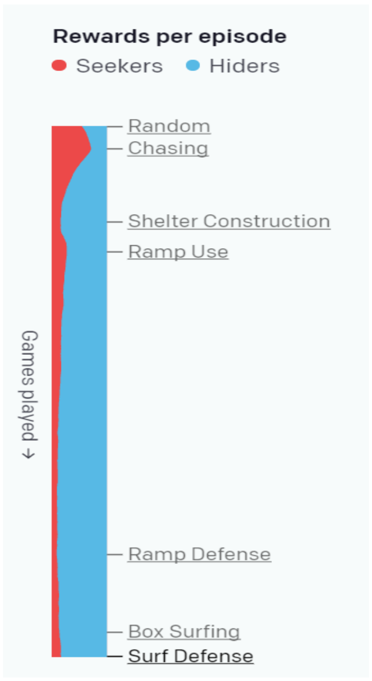


Figure 1. Development of different strategies as agents learn. [2]

Future research outlined in the paper includes reducing sample complexity, as well as better policy learning algorithms that could be used to improve sample efficiency and performance on transfer evaluation metrics. [1]

Another research paper that is more relevant to this project is a paper by Google Research's Brain Team titled: "Google Research Football: A Novel Reinforcement Learning Environment" [3]. The environment created solves several issues that existing environments have. Some environments are too easy to solve, which does not make for a good model of the real world, and models cannot be benchmarked since most models will solve the game. Another issue this environment solves is the need to have access to large amounts of computational resources. The game provides a set of different difficulties, with the easy option being solvable on single machines in a reasonable amount of training time. The environment also provides different representations of the game such as pixels, mini maps, and floats, which also helps in reducing the required computational power.

The environment has two pre-defined reward functions that can be used to train agents, with the possibility to define custom reward functions. The first reward function is scoring, in which scoring a goal results in a reward of +1, and scoring an own goal results in a reward of -1. The second reward, called Checkpoint, is more sophisticated than scoring, as it considers advancing across the pitch and augments the scoring reward with an additional auxiliary reward contribution for moving the ball close to the opponent's goal [3].

The paper shows benchmark results for three RL algorithms: Proximal Policy Optimization (PPO), IMPAPA, and Deep Q-Network (DQN) being used to train agents using both the Scoring and Checkpoint reward functions against different opponent difficulties. Results are shown in Figure 2 below.

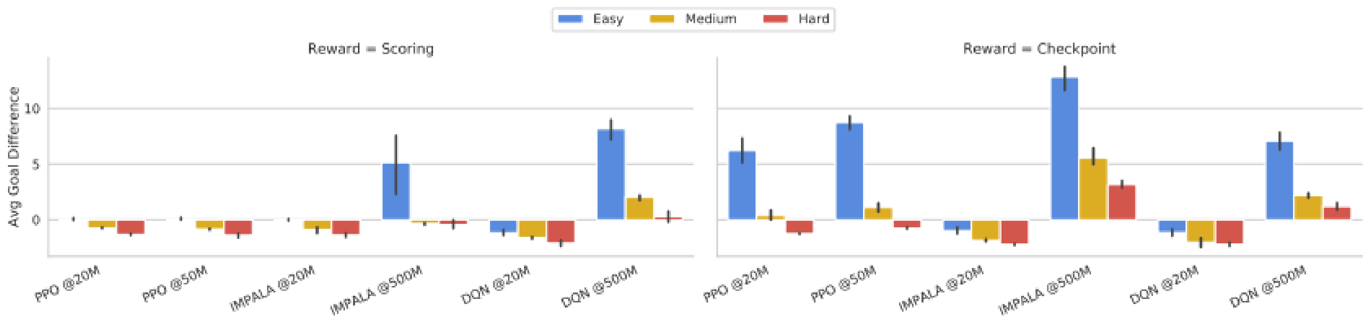
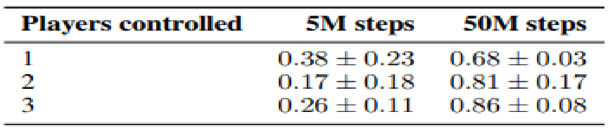


Figure 2. Average Goal Difference for IMPALA, PPO and Ape-X DQN with both the Scoring and Checkpoint reward functions. [3]

It can be shown that the environment difficulty significantly affects the training complexity and that the Checkpoint reward function appears to be helpful for speeding up the training for policy gradient methods (PPO and IMPALA) but does not seem to benefit as much the Ape-X DQN.

The previous results were done using Football Benchmarks (a complete 11 vs 11 match). It is also possible to use Football academy, which provides simpler training scenarios such as a single player to score against an empty goal. These are helpful for researchers to quickly iterate on new research ideas [3] since the training time will be significantly less than a complete match. The said scenario was solved by both PPO and IMPALA with both reward functions using only 1M steps, and only within hours of training.

As for experiments involving multi-agent RL, the environment also allows for controlling several players from one team simultaneously. Experiments were conducted using the 3 versus 1 with Keeper scenario. The number of players that the policy controls was varied from 1 to 3 and trained with IMPALA. Training was initially slower when more players were controlled, but the policies eventually learn more complex behaviours and achieve higher scores [3]. Numerical results are presented in Table 1 below.



Players controlled	5M steps	50M steps
1	0.38 ± 0.23	0.68 ± 0.03
2	0.17 ± 0.18	0.81 ± 0.17
3	0.26 ± 0.11	0.86 ± 0.08

Table 1. Scores achieved by the policy controlling 1, 2 or 3 players respectively, after 5M and 50M steps of training.

Future research outlined in the paper includes self-play, multi-agent learning**,** sample-efficient RL, sparse rewards, and model-based RL. It is noted that both research papers outline sample efficiency as a promising line of future work. As for my project, it will build upon the paper by focusing on multi-agent learning, since it is also an important line of research. Experiments to be designed include scenarios such as 3 vs 3 matches, where each team trains using a different RL algorithm.

Goals and objectives

Below is a list of goals, each with a set of objectives within it, set at end of Term 1 in the interim report.

- Single-agent AI training
    1. Improving the PPO model to solve the empty goal scenario
    2. checking how many steps is necessary to achieve that on my local machine.
    3. compare that with training on a GPU server, to check how faster it is to achieve the same results.
- Use Existing Multi-Agent RL algorithms
    1. conduct experiments using multi-agent RL algorithms on suitable scenarios such as 5 vs 5 or 3 vs 3 matches.
- Introducing novelty in the research
    1. Creating a custom reward function to improve the performance of RL models, such as shooting on target.

The PPO model was successfully improved, and the empty goal scenario was solved. As for the server, it turned out to have a similar speed to a single machine (when only 1 core is used) but the code had to be more efficient by using different model prediction methods. As for multi-agent experiments, a 3 vs 3 scenario was created but was not executable on the UCL sever due to using a docker container. Therefore, an existing 3 vs 1 was used since that was the scenario used in the research paper for multi-agent research. As for the custom reward functions, their fairness was discussed in the Experimentation section, and it was difficult to implement it since it requires editing the source code of the game/environment.

Background Theory

The process of understanding the underlying theory behind the project starts with compiling a list of resources to study Reinforcement Learning, compare them, and decide which ones to use. The most comprehensive resources to use were the Introduction to Reinforcement Learning with David Silver course created by UCL and DeepMind [4]. The course is based on the Reinforcement Learning: An introduction textbook by Sutton and Barto [5].

The main difference between Reinforcement Learning and other machine learning paradigms is there is no supervisor, only a reward signal determines the agent's actions. Also, the agent's actions affect the subsequent data it receives [6], since actions affect the environment, which in turn affects the new state being input to the agent.

A reward is a scalar feedback signal Indicating how well agent is doing at a given time-step. The agent's objective is to maximise the cumulative reward. An example in a reward is a reward of +1 when scoring a goal in a football game. A policy is a map from state to action, which defines the agent's behaviour. As for the value function, it is a prediction of future rewards and is used to evaluate how good a certain state is [6]. An actor-critic model is one that combines a policy with the value function. Such methods used are referred to as Model-Free Reinforcement Learning. On the other hand, there are Model-Based Reinforcement Learning algorithms that make use of the rules of the environment to predict upcoming rewards or state. For example, in the game of chess, the agent could use model-based RL to predict the move resulting in the best reward, by using tree search to find the optimal policy. Figure 3 below shows a useful taxonomy for RL algorithms.

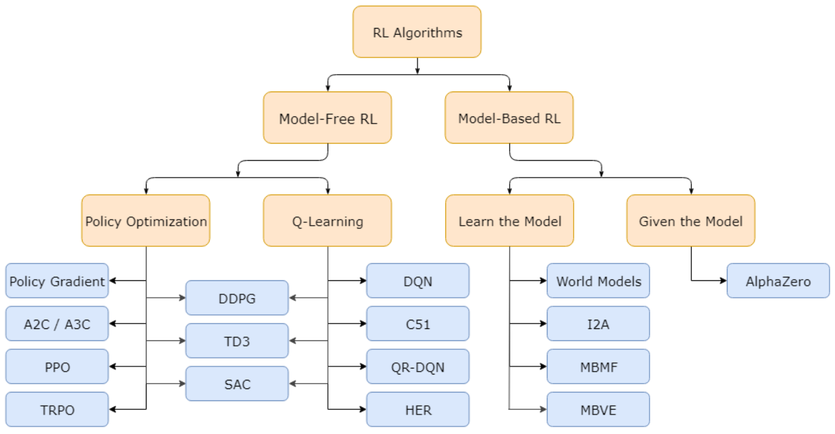


Figure 3. A non-exhaustive, but useful taxonomy of algorithms in modern RL [7].

An important aspect to be considered in Reinforcement learning algorithms is the trade-off between exploration and exploitation. Exploration is when the agent learns more about the environment. On the other hand, exploitation is when the agent exploits its current knowledge to maximise rewards. The main objective of the agent is to maximise rewards, but this is not possible without more exploration, as the agent could be stuck in a local maximum if it exploits its available knowledge. Examples of this trade-off that humans experience on daily basis include restaurant selection. Exploitation would be going to your favourite restaurant, and exploration is trying a new restaurant [6]. In the chosen environment for this project, exploitation would be choosing the same series of actions that lead to scoring a goal, and exploration is trying a new move or plan.

Research in Reinforcement learning could also be split into two areas, single-agent, and multi-agent. In Multi-agent Reinforcement learning, from the perspective of one agent, the rest of the agents are simply part of the environment. It could also be split into two types: cooperative and competitive. Cooperative is when agents work together towards a certain goal. On the other hand, competitive is when agents compete with one another to accomplish a goal. In this project, both types will be explored as there will be experiments of 3 vs 3 football matches.

Experimental Setup

Choosing a suitable game environment

The first objective of the project was to choose a suitable game/environment. The chosen game for the project should be at least a 2 vs 2 game, so that I can experiment with multi-agent algorithms. Therefore, games like Chess and Connect 4 would not work. Multi-agent Reinforcement Learning research in the past have been performed on various games such as: 2v2 football, Pacman (the monsters being the agents), as well as simple independent games created by the researcher.

There is this trade-off between either choosing a game that is too complex where training will take weeks (which is not suitable since the project ends in March) or choosing a simpler game but that might be "solved" to a certain degree (meaning that most algorithms perform well on the game to the point that they are all similar).

At the start, I was between two choices, either creating a simple game, or using a game that is already made. The advantage of creating one would be having full control of the game parameters and being able to change them when something goes wrong (e.g., when the agent breaks the game). The disadvantage is that it will be very time consuming to create a full game and might have a lot of issues compared to already tested games.

To start, I investigated creating my own game, which involved research of tutorials on creating games for machine learning, such as the Run Forrest game. One of the ideas I had for a game to code was a Chess variant that involves four players (2 vs 2) instead of two. However, by the nature of this variant, the agent would just treat the two opponent's pieces as the pieces of one player, which could be argued that it doesn't satisfy the requirement of the project to have at least two players in each team.

I became more inclined to code my own game. This is because I don't know how to use already created games for my own and train ai agents within them (e.g., StarCraft) as well as those games might take up too much memory or processing power to the point that they affect execution time.

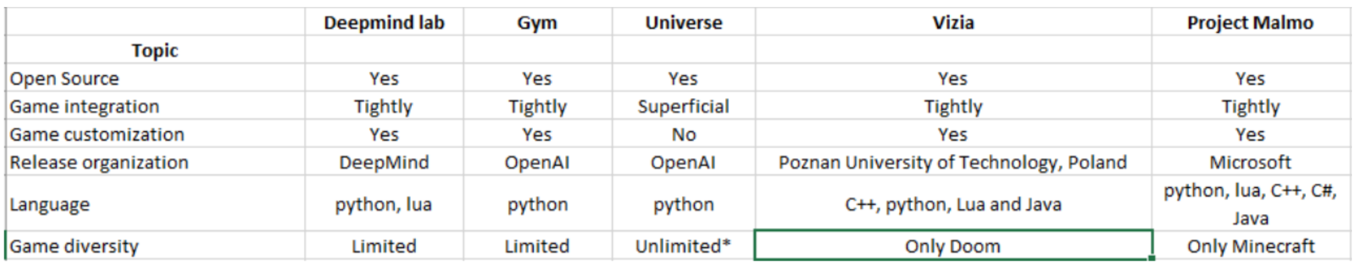
At this point, I had three main game ideas to code:

1. 2 vs 2 Chess: The issue with this game is that it might be very difficult to interpret the agents' moves and prove if they cooperate or not, since it is even more complicated than regular chess.
2. Baloot: A popular Arabic 2 vs 2 card game. The issue could also be that it is difficult to show agent cooperation to someone who is not familiar with the game. Another issue possibility of the game being easily solved by AI agents, which means it cannot be used for the project since there is no benchmark to compare machine learning models if all of them perform equally well by solving the game.
3. Pac-man: By using the ghosts as AI agents. The issue is that it has been used before by other people so it might not be original.

The problem with creating my own game that I did not realise earlier was that it has to be a suitable environment to carry out experiments and Reinforcement Learning research. This meant I had to use an already existing environment that supports RL research, as it is way beyond my scope of skills and time frame to develop it. I started by searching for suitable opensource AI Reinforcement Learning platforms. Here is a list of the main five platforms (the rest are notable mentions):

1. DeepMind Lab
2. OpenAI Gym
3. OpenAI Universe
4. Project Malmo
5. VizDoom
6. RL-Glue
7. CommAI
8. Burlap
9. Rlenvs

Figure 4 below summarises the five main platforms.



Topic	Deepmind lab	Gym	Universe	Vizia	Project Malmo
Open Source	Yes	Yes	Yes	Yes	Yes
Game integration	Tightly	Tightly	Superficial	Tightly	Tightly
Game customization	Yes	Yes	No	Yes	Yes
Release organization	DeepMind	OpenAI	OpenAI	Poznan University of Technology, Poland	Microsoft
Language	python, lua	python	python	C++, python, Lua and Java	python, lua, C++, C#, Java
Game diversity	Limited	Limited	Unlimited*	Only Doom	Only Minecraft

Figure 4. Summary of information about the five major AI Reinforcement Learning platforms. [8]

Since VizDoom and Project Malmo only play single games, they were not considered. This is because their games are not suitable for the project. We need a multiplayer game that is simple enough for our computers to handle. Such as the football game from DeepMind Lab.

The most popular and supported platforms are Deepmind Lab and OpenAI Gym. However, Deepmind Lab and OpenAI Gym do not support Windows. This is an issue since the machine I am using has a Windows OS. Therefore, the best solution was to use dual-booting, which means having both Linux and windows on my machine. Finding the most suitable platform is not enough, as I need to find a suitable game within the environment that supports multi-agent Reinforcement Learning and acts as a benchmark for research. I found a list of RL benchmarks [9], and compiled a list of the ones that support multi-agent Reinforcement Learning, along with their properties:

1. Google Research Football – Multi-task; Single-/Multi-agent; Creating environments.
2. Meta-World – Meta-RL; Multi-task.
3. Multiagent emergence environments – Multi-agent; Creating environments; Emergence behaviour, e.g. Hide & Seek game.
4. OpenSpiel – Classic board games; Search and planning; Single-/Multi-agent.
5. RLCard – Classic card games; Search and planning; Single-/Multi-agent.
6. StarCraft II Learning Environment – Rich action and observation spaces; multi-agent.
7. The Unity Machine Learning Agents Toolkit (ML-Agents) – Create environments; Curriculum learning; Single-/Multi-agent; Imitation learning.

Google Research football (GRF) seems like the most supported game out of the ones compiled, and it is compatible with the widely used OpenAI Gym API [10]. At first, I had an issue with running python from the command prompt, so I fixed it in order to pull repositories from Github and try Google Research football. I faced issues loading Google Research football game into Windows, but it worked on Linux. However, the question was whether it was suitable for my research. It is an 11 vs 11 football game, which meant the multi-agent RL algorithms might be too complicated for my own research. This issue could be solved if the number of players could be reduced. I started reading more about the game and I found out that it is possible to pick different scenarios for the game, meaning that it does not have to be an 11 vs 11 match. I also found out that there are different difficulty settings within the game, which allows single machines to train AI agents against the easy option, and arrays of machines against the hard option.

Another option was the MuJoCo soccer game by Deepmind. I tried to run the game, but I had issues with importing libraries and library paths. I also faced the same issue with Google Research Football on Linux, which hindered my progress as I spent time troubleshooting. At this point, I took a step back and did a 3-hour Reinforcement Learning course using Python, with the intention to find a solution to my problem as well as finding other environments to use. I found a suitable 2 vs 2 soccer game within the Unity Machine Learning Agents Toolkit, which became my back-up plan in case Google Research Football does not work.

I decided to work with the Google Research football (GRF) game, even with the issues I faced with it, because other decent options were also from the OpenAI gym platform, meaning that the problems I face will be the same on other games. Also, the game is suitable for this project since it is not easy to solve, not computationally expensive, not single-player, and does not lack open-source licensing [3].

The main problem I had with training agents in GRF was that the GPU version of TensorFlow required an NVIDIA GPU, which was not available in my machine, and training with the CPU version of TensorFlow is not practical since it will be too slow. This, along with less training time, led me to consider other methods of running my code, such as using UCL's resources or the cloud.

Logistics of training AI agents

As mentioned in the previous section, my machine was not suitable to train AI agents for two main reasons. Firstly, the type of GPU is not compatible with TensorFlow. Secondly, due to the time constraint of the project, I will need more powerful machine to train agents faster, especially when multi-agent algorithms are used. For reference and comparison, AI agents took around 4 days to solve a simple Atari game using GPU [6]. This is enough evidence to show that training AI agents for this project requires more than a single machine.

The first solution I had was to use UCL's computer clusters. However, after contacting the staff responsible for them, I was redirected to UCL EEE's GPU resources to use. The GPU servers available were from NVIDIA, which is what I needed for TensorFlow to work.

I started the process of learning how to access the remote server by editing my ~/.bashrc file on Linux. I tried to access UCL's GPU by ssh but I got this message: "kex_exchange_identification: Connection closed by remote host". Then, I figured out how to connect to UCL's servers by typing this into the command prompt: ssh -X zceesa@london.ee.ucl.ac.uk

Then, I needed to learn how to run my python script after connecting to the remote server. I did it by connecting to the server within the code itself using the paramiko python package (Shown in the Appendix as Code 1). However, I do not see the game being rendered since it is not running on my machine anymore.

As an alternative solution, I tried to use Google Colab's GPU servers. It provides free GPU but only up to 12 hours continuously. I decided to test the GRF game on it and see if it renders on my screen. The first thing I learnt was that Bash commands can be run on Google Colab by prefixing the command with '!'. However, I had issues running the code on Google Colab as it doesn't allow installing gfootball, which is the library for GRF, nor cloning the GitHub repository. As a last resort solution, I tried to manually upload all the necessary libraries from my machine to Google Drive, but this also did not work. Then, I tried to use Kaggle's GPU to run my code, but I faced the same issue I had with Google Colab, which is that I cannot import gfootball and hence cannot create the environment.

The best solution for now is to use UCL's GPU servers. However, I need to learn how to export a video of the rendered game, as well as the outputs from the code such as the trained model.

It seems that ToServer.py code is not working as expected. It connects to the remote server but does not execute train.py. However, I managed to connect to the remote server and run train.py using the following cmd command on PyCharm's terminal:

```
cat train.py | ssh zceesaa@athens.ee.ucl.ac.uk python
```

The issue is similar to that with Kaggle and Google Colab servers, where I cannot use the modules imported, and they need to be installed in the remote server.

I got in touch with the IT services in our department and got offer some help. It turns out that they need to install the SDL (Simple DirectMedia Layer) dependency on the server as it won't work without it. They managed to make it work only on one server called medusa. The solution on the medusa server requires the use of Docker. I started learning how to use it in order to use the server. Finally, the server stopped working due to issues in memory space, which hindered my last multi-agent experiment using RLLIB and OpenAI baselines.

Experimentation

Due to the nature of this project, this section will showcase a series of experiments along with the setups and reasonings made before and after them, in chronological order, to show the development and improvements made both in the models and the knowledge of the experimentalist.

After reading the Google Research Football paper, I visualised that the best way for me to start training agents is to reproduce the easy experiments done using the three algorithms (PPO, IMPALA, DQN) on the empty goal scenario. This is because it could be trained within hours since it is an easy scenario. Also, by reproducing these results, I will learn how to use the environment properly and it will prepare me to start carrying out my own experiments with different scenarios such as the 3 vs 1 scenario, where I can use multi-agent Reinforcement Learning algorithms.

The game environment was not easy to understand and there were not any official tutorials to get started. However, there was a useful tutorial on creating a Proximal Policy Optimization (PPO) model using the actor-critic method to train a player to score against an empty goal [11]. I made a python file for each step in the process so that I can understand it and apply it step by step. I already created a virtual environment and installed the system dependencies and python packages required for this project. I also already tested the game running on my machine, and it worked on Linux.

Now, in the python code, we start by importing the gfootball library:

```
import gfootball.env as football_env
```

Then, we create an environment object (env):

```
env = football_env.create_environment(env_name="academy_empty_goal",
representation="pixels", render=True)
```
env_name chooses the scenario that will be used. academy_empty_goal scenario is where our player spawns at half-line and has to score in an empty goal on the right side. representation='pixels' means that the state that our agent will observe is in the form of an RGB image of the frame rendered on the screen.

After that, we can check the state dimensions and the number of possible actions the agent can take at that state:
```
state_dims = env.observation_space.shape
print(state_dims)
n_actions = env.action_space.n
print(n_actions)
```
The output printed is:
```
(72, 96, 3)
19
```
where the frame dimensions are 72x96, and 3 since it is in RGB. 19 is the number of actions the agent can take.

After that, we incorporate the Actor-Critic models. Figure 5 below shows the structure of the PPO algorithm. The actor model takes as input the current observed state and decides what action is best to take. This action is fed into the environment, which produces a new state, and a reward is obtained. This reward could be positive (if a goal is scored), or negative (if an own goal is scored). This reward is the input to the critic model, which evaluates if the action taken by the Actor led our environment to be in a better state or not and gives its feedback to the Actor, hence its name. It outputs a real number indicating a rating (Q-value) of the action taken in the previous state. By comparing this rating obtained from the Critic, the Actor can compare its current policy with a new policy and decide how it wants to improve itself to take better actions.

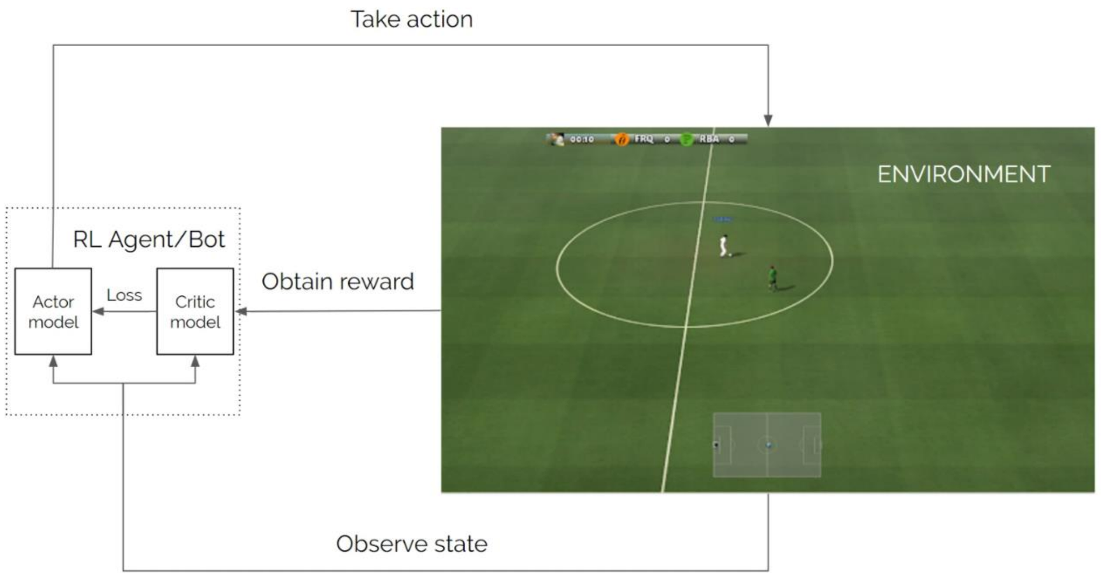


Figure 5. Actor-Critic model of the PPO algorithm [11]

Before coding the two models, we start by letting the agent pick random actions in the environment for 128 steps (Shown in the Appendix as Code 2). Then, a function is defined for the actor model (Shown in the Appendix as Code 3). The function uses pretrained MobileNet CNN in order to process the input image, which is the state of the environment, and uses mean squared error as the loss function. Then, the function is called to construct the actor model:
```
model_actor = get_model_actor_image(input_dims=state_dims)
```
After that, the PPO loop is edited to take into account the actor model, rather than picking random actions in each state (Shown in the Appendix as Code 4).

Then, the critic model is incorporated, which is similar in code to the actor model, the difference is that there is only one output node from the neural network, which is the q_value, and it is a real number (Shown in the Appendix as Code 5). Then, the function is called to construct the critic model:

```
        model_critic = get_model_critic_image(input_dims=state_dims)
```

After that, the PPO loop is edited to take into account the critic model, and new data is appended to existing lists (Shown in the Appendix as Code 6).

After that, the tutorial implements the Generalized Advantage Estimation (GAE) algorithm that drastically improves the evaluation of rewards. For example, if the player crosses the ball towards the goal, this gives a reward of zero based on the scoring reward function defined in the game. However, if the cross is successful and a goal is scored, then a reward of +1 is given as an indirect result of the cross. This is taken into account when the GAE algorithm is used. Then, the tutorial implements a custom PPO loss function instead of MSE.

With the addition of the GAE algorithm and the custom PPO loss function, I started getting data cardinality errors when fitting the models. The issue does not happen in the tutorial even though it uses the same code. Therefore, it is due to the default reshaping of data for the model to be fitted. I could not fix the issue given how complex the code became with the added features. Therefore, I made a simpler version of the tutorial's code to fix it. I used the MSE loss function instead of the custom PPO loss function and managed to fix the error by reshaping the states array being inputted to fit the model:

```
        actor_loss = model_actor.fit([np.reshape(states, newshape=(128, 72, 96, 3))],
        [np.reshape(actions_onehot, newshape=(-1, n_actions))], verbose=True,
        shuffle=True, epochs=8,callbacks=[tensor_board])
        critic_loss = model_critic.fit([np.reshape(states, newshape=(128, 72, 96, 3))],
        [np.reshape(returns, newshape=(-1, 1))], shuffle=True, epochs=8, verbose=True,
        callbacks=[tensor_board])
```

The code works and the agent is training (referred to as Experiment 1). The first thing I noticed was that the player got stuck at one point in training where he decides not to move (he only performs action 13). This is probably due to the simple model used. However, the objective now is to get to the end of the code where the model is supposed to save, then test loading the model. Training took around 30 minutes to do 50 iterations of 128 steps each. I think it took a long time because the representation was in pixels. Then, I managed to save the model, load it, and test it using Code 7 shown in the Appendix.

The model still needs improvement in order to solve the scenario and score goals. The complete code used to train the agent is shown in the Appendix as Code 8. I made a GitHub repository for my project [12], to be able to easily use the code on multiple machines and to act as a backup location as well. The same code could be found there with the python file name SimplerModel.py.

I found a way to solve the data cardinality error and it was by reshaping all inputs and outputs. However, I then got an error indicating that a Keras tensor data type is being passed to the model.fit() function, where the model fits the inputs to the outputs. I found a way around that issue, which is to disable eager execution from tensorflow:

```
        from tensorflow.python.framework.ops import disable_eager_execution
        disable_eager_execution()
```

I then found an error in my code causing the avg_reward variable to always be zero. It is because of the test_reward() function, which is supposed to test a model by letting the player take a specific number of steps in the environment, and calculate the average reward. However, it does not do enough steps to test the model so the player has no time to score and get any rewards. I decided to abandon the test_reward() function and simply sum the rewards of each iteration such that this becomes the new indicator of performance for the iteration.

I trained the PPO model for 100k steps using the modified code (Experiment 2). However, only around 24k steps were done, and my machine became slow such that the experiment did not complete. It is worth noting that the processing speed decreased with time. for example, 1000 steps at the start were done in less than 3 minutes, compared to an hour for 1000 steps later.

Due to this disadvantage, I trained the PPO model for two iterations, each consisting of 10k steps (Experiment 3). The outcome of the experiment was that the player does not move, the action taken is always 18 (Action list is shown in the Appendices) [13]. This raised the question, what is the reason for only picking one action?

The answer was that this is caused by two reasons. Firstly, each iteration in training was 10k steps, which is very high, and only two iterations were done. The way the model works is that after the first iteration (N steps) the model is trained using model.fit() with the data from N steps. After fitting the model, the probability distribution for the list of actions changes, and the probability of good actions start to dominate after every iteration. Therefore, a higher number of iterations is required, leading to less steps per iteration (since there is a limit to the number of steps taken, due to time and processing power). A good compromise is 128 steps per iteration. Secondly, in the testing code, the action is chosen as the action with the maximum probability, from the action probability distribution:

```
action = np.argmax(action_probs)
```

the maximum probability was always for action 18. replacing that line with the following lets the agent pick other actions, with a probability based on the action probability distribution predicted by the model at a specific state.

```
action = np.random.choice(n_actions, p=action_probs[0, :])
```

All above were reasons to train the same model for 150 iterations of 128 steps each (Experiment 4).

As we have seen so far, there are only two reward functions defined in the environment, scoring and checkpoint. One of the goals of the project was to create a custom reward function to improve the performance of RL models. However, would that be a fair way to improve the performance of the models? This is because it could be argued that it is a method of “hard-coding”, and that it could be used to let the model behave in a specific way. For example, adding a reward for shooting within a specific range. Also, all models produced then cannot be compared to the benchmark since they are at a disadvantage of not having the custom reward function.

By looking at the source code of the game (wrapper.py file to be specific), I found how the checkpoint reward function is computed based on distance, in a very complicated way that prevented me from learning how to define custom rewards. However, I learnt how to use the checkpoint reward function, by passing this parameter when creating the environment using the function: football_env.create_environment():

```
rewards='scoring,checkpoints'
```

As for multi-agent learning, I found that in order to train a policy controlling multiple players, one has to pass this parameter to the 'create_environment' function defining how many players to control (such as 2 players here):

`number_of_left_players_agent_controls=2` to `football_env.create_environment()`

Then, instead of calling 'env.step' function with a single action, it is called with an array of actions, one action per player. Therefore, I started adding this to my code, but this produced numerous errors, all of them related to the fact of having more players to control, which changed the structure of most variables in the code. An example of an error is: (in single-agent, n_actions is equal to 19, which is the number of actions available to the player)

```
n_actions = env.action_space.n

AttributeError: 'MultiDiscrete' object has no attribute 'n'
```

To fix the error, I found the MultiDiscrete class in the source code and checked the case with more than one player. In order to fix the error, I needed the class responsible for single-agent learning, which was difficult to find, so I did a nice trick where I changed the code back to one player and purposefully got this error, which shows you the class ‘Discrete’ for single players:

```
n_actions = env.action_space.m

AttributeError: 'Discrete' object has no attribute 'm'
```

I ended up fixing all the errors and re-formatting the structure of the data being used to train the model, especially the ones related to the Keras neural network. I also managed to create a new scenario in the environment called 3_vs_3.py to carry out my multi-agent experiments in. Each team has two players and a goalkeeper, so it’s much less computationally expensive that the options provided in the game which are either a full 11 vs 11 game or a 5 vs 5 game. However, I could not use custom made scenarios in the UCL server as I was using docker containers to run the environment, which gave me errors when trying to edit the source code of the game (adding more files).

Since using a single machine proved to be difficult and slow, I finally managed to use the UCL server (around mid-February). I tested the speed on the medusa UCL server, and the speed is almost exactly equal to that of my local machine. It also experiences the same problem of becoming extremely slow after just a couple of iterations. Therefore, I knew the issue had to be within my code. I decided to pinpoint where exactly in my code the program start to become slow, by using the time library in python and calculation the time taken to execute specific lines of code. After testing different positions in the code, it turns out that the model.predict() function takes more time as the code runs, which means as more data gets collected, predicting takes a longer time. For example, after 3 iterations of 256 steps each, the delay is around 0.14 seconds for predicting a single action for the actor model. I started to figure out why this is the case by looking at the documentation for Tensorflow/Keras, and I found that there is an alternative way to predict an action for a model by calling the Model(), instead of using the .predict() function. This gives the same output, but much faster. I tested it and it completed 256 steps in one second, meaning it is around 35 times faster than Model.predict(), and it also does not become slower as the model collects more data.

This method however provides the prediction in the form of a tensor, which is converted to a numpy array using .numpy(). This conversion is only possible by enabling eager execution. Therefore, I had to deal with the error in model.fit() which I solved before by disabling eager execution. There are two possible ways to convert from a tensor to a numpy array. First one is to use .eval() and make a graph, which is complicated, the other is to access each element in the tensor then use .item() on it to access only the value without the tensor, then assign that to my numpy array. However, none of those methods worked since my code required eager execution in one part and disabling it in another.

After careful consideration, I found the source of the problem. In my code, model_critic.fit() works perfectly, but model_actor.fit() gives the error. This is because when compiling the models (model.compile()), the critic model uses a simple MSE (Mean Squared Error) loss, whereas the actor uses the PPO loss function. This confirms that the Keras tensor error is caused when using the PPO loss function.

While I fix the error, I trained a model on the empty goal scenario using the MSE loss function, making use of the fast training now by using Model() instead of Model.predict(). I trained it for 100k steps, split into 781 iterations of 128 steps each (Experiment 5). Training time was around 12 minutes only. However, the result was very bad as the player did not score a single goal in training. The reason behind that is elaborated in the analysis section.

I also tested whether loading the model and continuing to train it would increase predicting time when using model(x) and model.predict(). It turned out that using model(x), it takes around 0.002 seconds and one whole iteration was 1.8 seconds. As for model.predict(), it takes around 0.03 seconds to predict and one whole iteration was 10.2 seconds. This was important to see whether Model(x) scales with more data or not. As written online, they say it does not scale well, but given that 100k steps is enough, it does.

As for the PPO loss function issue, a solution proposed is to pass this parameter to the model.compile() function: `experimental_run_tf_function=False.`

The solution worked. The model is currently training for the empty goal scenario for 100k steps using the PPO loss function and the checkpoint reward function (Experiment 6).

I also trained a similar model for 1M steps, but only using the scoring reward function (Experiment 7).

Looking back at the research paper, the only multi-agent experiment was using the 3 vs 1 scenario using the IMPALA algorithm. My plan is to come up with a similar experiment to compare models using one, two, or three players, but using the PPO algorithm instead, and comparing between the two algorithms. I started by training a model controlling two players in the 3 vs 1 scenario using scoring only as a reward (Experiment 8). The time taken to finish training was 3 hours and 53 mins. Training took an extra hour compared to 1M steps on empty goal scenario because there are two players controlled by the agent which doubles the amount of data being trained each step. I tested the model saved at step 1M, and the player moves back and stays still. As for the model saved at step 100K, the player walks to the top left until he leaves the field.

I then trained models in the 3 vs 1 scenario, controlling 1, 2, and 3 players, with and without the checkpoint reward. This gives a total of 6 models being trained for 5M steps, trained simultaneously on the server (Experiment 9).

After testing the models, I found out that they all overfit very early on in training. For example, in the model controlling three players with the checkpoint reward, the action probability distribution has one action as a probability of 1, from iteration 600 (step 150K).The source of the overfitting issue is not known, it might be the epochs parameter in model.fit() being equal to 1 instead of 8, or the new method of predicting model(), or the experimental_run_tf_funtion=false, or something else. I tested training three players using the checkpoint reward for 800 iterations using epochs=8 and this also overfits from the 600th iteration, so the source of the overfitting issue is not epochs.

I had a step back and started researching how multi-agent experiments are being done and what libraries are being used. I found an example code for multi-agent learning within the Github repository for Google Research Football, and they do not use tensorflow or keras directly, but they use openAI baselines, which is a much simpler library to use that provides the algorithm for the user and it uses tensorflow within it, which hides some of the complexity. I tried to run the example code but I faced issues installing and configured OpenAI stable baselines.

As for the overfitting issue, I came up with two approaches to limit it by forcing the agent to explore more. The first approach is setting a maximum possible probability (say 50%) for each action so that the agent does not always choose the same action and get stuck. I manually changed the action distribution within the for loop that takes a step in the environment and set the maximum threshold. The second approach to solve this problem is an epsilon greedy approach where the agent takes random actions based on the value of epsilon, where the higher the value the more probably the agent will pick actions randomly. The first approach was tested on models controlling three players, and the second was tested on models controlling two players (Experiment 10).

Below is shown the code for the maximum probability approach. The code was repeated for each player, since each player has an action probability distribution.

```
if max(action_dist) > 0.5:
   diff = max(action_dist) - 0.5
   share = diff / (len(action_dist) - 1) # distribute the difference to other actions
   index = np.argmax(action_dist)  # index of max
   for i in range(len(action_dist)):
       if i == index:
           action_dist[i] = 0.5
       else:
           action_dist[i] = action_dist[i] + share
```

Below is shown the code for the epsilon greedy approach to pick a random action:

```
if np.random.uniform() < epsilon: # epsilon greedy approach
   action_player1 = np.random.choice(action_dims[0]) # take a completely
random action
   action_player2 = np.random.choice(action_dims[0])
else:
   action_player1 = np.random.choice(action_dims[0], p=action_dist[0, 0, :])
action_player2 = np.random.choice(action_dims[0], p=action_dist[0, 1, :])
```

Based on the results of the previous experiment, I used the maximum threshold approach along with the epsilon approach (with epsilon set to 0.2) to all 6 scenarios and train for 5M steps (Experiment 11).

Since using my own implementation of the PPO model on Keras did not get good results. I had another go at the example code provided that uses OpenAI baselines. I figured that the issue is with the version of python and tensorflow used. Using python 3.7 and Tensorflow 1.15, I was able to install OpenAI baseline's library and run the example code (run_ppo2.py) [13]. Then, I copied the example code into my repository customised it for my experiment (Code shown in appendix). The issue is that this only enables single-agent learning, as multi-agent with baselines requires the use of the ray library, as shown in the example (run_multiagent_rllib.py) [13]. Therefore, I started training models using one player in the empty goal scenario (for 1M steps) and the 3 vs 1 scenario (for 5M steps), with and without the checkpoint reward function (Experiment 12). I also created a new python file (Code shown in appendix) to test the models. The code was made by using parts of run_ppo2.py and baseline/run.py, refining the code, and removing parameters when running the code as they are not necessary and add extra complexity. The test code worked after adding the final fix, which is passing the parameter `allow_early_resets=True` to the Monitor() function. this is so that the environment can be reset at the start. I was planning to calculate the average goals score and the variance for the 5M steps model in the 3vs1 scenario and compare it with the IMPALA results quoted in the paper. However, UCL's server stopped running my code due to memory issues in the docker container.

As for the multi-agent example using ray, I had errors when running the example code, so I figured that the error can only happen based on using different software versions. Therefore, I installed the version of ray which was the latest when the example code (run_multiagent_rllib.py) was uploaded, which was ray 0.7.5. After that, I copied the code and edited it such that it works in the UCL server (by specifying memory and CPU/GPU). The python file could be found in my GitHub repository with the name gfootball_multiagent.py. However, I faced issues running the code in UCL's server due to low memory in the docker container.

Results

This section shows the results from some of the experiments and model training on Google Research Football. All shown models and iterations can be found in my GitHub repository in the /venv/models/ folder [12].

Starting with Experiment 4, the final model saved managed to solve the scenario and score the goal. Find full code on GitHub repo. The 61st iteration of the model scores a goal every time since the max probability action is to shoot the ball. Another useful output from the test was the action probability distribution for each saved model (iteration). It is an array of 19 elements, each one is the probability of picking a certain action at a specific state. Below is the action probability distribution for iteration 48 at a random state. It is shown that action 5 (running to the right), and action 12 (Shooting) are the ones with the highest probability (22% and 18% respectively).

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
48	0.0046	0.035	0.0019	0.00096	0.22	0.0018	0.012	0.083	0.025
Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0.021	0.13	0.0072	0.18	0.0015	0.0012	0.0024	0.0033	0.034	0.23

Below is the action probability distribution for iteration 92 at a random state. It is shown that action 5 (running to the right) is no longer favoured by the player, but action 4 (running top right) and action 12 (Shooting) the ones with the highest probability (24% each).

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
92	0.0012	0.0036	0.00036	0.00011	0.24	0.00047	0.0021	0.16	0.007
Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0.012	0.14	0.0014	0.24	0.0003	0.00025	0.00064	0.00067	0.012	0.17

Below is the action probability distribution for iteration 147 at a random state. It is shown action 12 (Shooting) has become the action with the highest probability (33%).

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
147	0.00059	0.00098	0.00011	0.000044	0.14	0.00018	0.0009	0.20	0.004

Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0.00094	0.17	0.00067	0.33	0.0008	0.00006	0. 00032	0.00023	0.0084	0.13

In Experiment 6, the best version of the model runs towards the corner. As for Experiment 7, time taken to finish whole training: 10516.5s = 175 mins = 2 hours and 55 mins. From the saved models, the rewards were consistently 0 up until the 490th iteration, after that it is consistently a reward of 1 This means it took around 500*128 = 64K steps for the PPO model to solve the empty goal scenario.

This is the action probability distribution of the final saved model (1M steps) at a random state: (small numbers to the power of -25 were approximated as zero here)

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
7812	0	0	0	0	0	1	0	0	0
Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0	0	0	0	0	0	0	0	0	0

The model solved the scenario by favouring action 5, which is running to the right, as shown by the array above.

As for Experiment 10, to test the maximum probability approach, below is the action probability distribution (for player 1) for one of the iterations of the three-player model using the checkpoint reward:

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
x	0. 0089	0. 0079	0. 0082	0.0081	0. 0078	0.45	0.0082	0.0079	0.0072
Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0.0083	0.0072	0.0079	0.42	0.0089	0.0073	0.008	0.0075	0.0072	0.008

As shown, there are two actions with a probability around 40%, which is even better than expected! Instead of overfitting to only choosing one action, the model re-distributes the excess probability to other actions.

As for the epsilon greedy approach, testing the model for two players with the checkpoint reward gives the following action probability distribution:

Iteration	Action 0	Action 1	Action 2	Action 3	Action 4	Action 5	Action 6	Action 7	Action 8
X	0	0	0	0	0	0	0	0	0
Action 9	Action 10	Action 11	Action 12	Action 13	Action 14	Action 15	Action 16	Action 17	Action 18
0	0	0	0	0	0	0	1	0	0

As shown, there is still overfitting, meaning that this approach does not work by itself.

As for Experiment 11, there are many action probability distribution tables collected, and so the results will be states without printing all tables. For the 1-player model without the checkpoint reward, the last iteration scored a goal! Also, in the probability distribution, running to the left has the least probability, since it causes an own goal to be scored and a reward of -1. As with the 1-player model with the checkpoint reward, the player always runs to the top and goes outside. The player thinks it is a good action due to the checkpoint reward, so exploitation happens and the agent does not explore other actions such as shooting to score a goal.

For the 3-player model with the checkpoint reward, actions all have equal probabilities for the most recent iteration. As for the 3-player model without the checkpoint reward, iteration 5400 has a good distribution, where shooting has the highest probability of around 62%. The issue is, after a couple of iterations, this probability drops to 50% due to the limit set. As for the 2-player model with checkpoint reward, it learns that shooting is a good action from iteration 400 onwards, with a probability of around 43%. As for the 2-player model without checkpoint reward, iteration

18200 was interesting, the two players learnt to pass the ball to each other until they go through the defender! They do not shoot towards the goal after that, however. Favoured actions by this model are running to the right and passing.

As for Experiment 12, the 1-player model in the 3vs1 scenario learnt that passing the ball is the most optimal action to pass through the defender, and they manage to score goals most of the time. I also tested the empty goal scenarios, and both (with checkpoint and without) solved the scenario by shooting twice towards the goal.

Analysis

This section shows some analysis for some experiments whose results were only states without explanation. The rest of the analysis is implied within the experimentation and results sections.

In Experiment 5, The 100k step MSE model was bad, as expected since actions are only evaluated by their immediate rewards. On the other hand, within the PPO algorithm, instead of evaluating an action based on the immediate reward obtained, it considers rewards obtained in the future. Actions taken closer to the reward are more significant. This is executed within the get_advantages() function in the code.

In Experiment 6 and 7, in can be seen from the results that the model using only scoring as a reward performed much better than the one using the checkpoint reward. Confusing the agent with rewards that are more detailed than needed was a disadvantage in such a simple scenario (empty goal). To illustrate this, after 100k steps of training using the checkpoint reward, the player was moving towards the corner, which is close to the goal, so he receives a high reward (close to 1). This makes the scoring reward of 1 become not so important. If the scoring reward could be scaled to like +10 then using both would result in a better combination. The checkpoint was also a disadvantage in Experiment 16, when the player was running to the top and leaving the pitch, as the agent was getting a reward for this action so there was no exploration.

In addition, using Model() instead of Model.predict() for predicting actions increased exploitation in training and decreases exploration. This can be interpreted by looking at the rewards for each iteration. Using model.predict(), more randomness was evident in the rewards, but when Model() was used, all models before 490th iteration had 0 reward, and after than all iterations had a reward of 1.

Conclusion and future work

In Summary, many obstacles were faced during the project, especially regarding the logistics of training the agents (issues with the server) and troubleshooting issues regarding the environment setup. One of the most important findings was the efficient way to make a prediction when using Tensorflow and Keras. Predictions made using Model() are more efficient (35x faster) than using Model.predict(). Another main finding is that the checkpoint reward function was a disadvantage in some certain cases, when the scenario is too easy, since it leads to exploitation and overfitting. As for model training accomplishments, in the empty goal scenario, the issue of overfitting caused the scenario to be solved within 64k steps (compared to 1M steps in the literature), as the player only performs one action which is running straight into the goal. As for the main multi-agent result, a 2-player model learnt to pass through a defender by passing to each other, after training for 5M steps.

As for future work, I will conduct experiments using multi-agent RL algorithms by making use of stable baselines and the ray RLLIB (Reinforcement Learning Library) on scenarios such as the 3 vs 1, up to 50M steps and compare it with the IMPALA results from the research paper and examine how the models interact and learn, or on other scenarios such as a full match, a 5 vs 5 match, or 3 vs 3 if more computationally practical given the available resources. Finally, since the environment only has two pre-defined reward functions, it will be useful to create a custom reward function (such as passing or shooting on target) as this is at the core of a Reinforcement Learning algorithm and will significantly improve models.

References

[1] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powel, B. McGrew and I. Mordatch, “Emergent Tool Use From Multi-Agent Autocurricula,” 2019.

[2] A. Pilipiszyn, “Emergent Tool Use from Multi-Agent Interaction,” 17 September 2019. [Online]. Available: https://openai.com/blog/emergent-tool-use/. [Accessed 13 October 2021].

[3] K. Kurach, A. Raichuk, P. Stanczyk, M. Zajac, O. Bachem, L. Espeholt, C. Riquelme, D. Vincent, M. Michalski, O. Bousquet and S. Gelly, “Google Research Football: A Novel Reinforcement Learning Environment,” 2019.

[4] D. Silver, “Introduction to Reinforcement Learning with David Silver,” [Online]. Available: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver.

[5] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, The MIT Press, 2018.

[6] D. Silver, “RL Course by David Silver - Lecture 1: Introduction to Reinforcement Learning,” [Online]. Available: https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB&index=3. [Accessed 23 November 2021].

[7] “Part 2: Kinds of RL Algorithms,” [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below. [Accessed 22 November 2021].

[8] J. Shaikh, “Getting ready for AI based gaming agents – Overview of Open Source Reinforcement Learning Platforms,” 15 December 2016. [Online]. Available: https://www.analyticsvidhya.com/blog/2016/12/getting-ready-for-ai-based-gaming-agents-overview-of-open-source-reinforcement-learning-platforms/. [Accessed 18 October 2021].

[9] P. Januszewski, “Best Benchmarks for Reinforcement Learning: The Ultimate List,” 16 July 2021. [Online]. Available: https://neptune.ai/blog/best-benchmarks-for-reinforcement-learning. [Accessed 4 November 2021].

[10] K. Kurach and O. Bachem, “Introducing Google Research Football: A Novel Reinforcement Learning Environment,” Google, 7 June 2019. [Online]. Available: https://ai.googleblog.com/2019/06/introducing-google-research-football.html. [Accessed 29 October 2021].

[11] C. Trivedi, “Proximal Policy Optimization Tutorial (Part 1/2: Actor-Critic Method),” [Online]. Available: https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-1-actor-critic-method-d53f9afffbf6. [Accessed 6 December 2021].

[12] “GitHub repository for the project,” [Online]. Available: https://github.com/asa1420/GRF

[13] “GitHub repository for Google Research Football” [Online]. Available: https://github.com/google-research/football/tree/master/gfootball [Accessed 30 March 2022]

Appendices

Code 1:

```
import paramiko
# Connect to remote host
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect('athens.ee.ucl.ac.uk', username='zceesaa', password='*********')
# Setup sftp connection and transmit this script
sftp = client.open_sftp()
sftp.put(__file__, '/tmp/test.py')
sftp.close()
# Run the transmitted script remotely without args and show its output.
# SSHClient.exec_command() returns the tuple (stdin,stdout,stderr)
stdout = client.exec_command('python /tmp/test.py')[1]
print("success")
for line in stdout:
    # Process each line in the remote output
    print (line)
client.close()
sys.exit(0)
```

Code 2:

```
ppo_steps = 128
states = []
actions = []
values = []
masks = [] # checks if the match is in completed/over state, in which case we
want to restart the game
rewards = []
actions_probs = []
actions_onehot = []
for itr in range(ppo_steps): # collect 128 interactions with the game
   observation, reward, done, info = env.step(env.action_space.sample()) # this
takes a random action in the game and gives as output the observation, reward,
and other info, as well as if the game is done.
   if done:
       env.reset() # reset if the game is done
env.close()
```

Code 3:

```
def get_model_actor_image(input_dims):
   state_input = Input(shape=input_dims)

   # Use MobileNet feature extractor to process input image
   feature_extractor = MobileNetV2(weights='imagenet', include_top=False)
   for layer in feature_extractor.layers:
       layer.trainable = False

   # Classification block
   x = Flatten(name='flatten')(feature_extractor(state_input))
   x = Dense(1024, activation='relu', name='fc1')(x)
   out_actions = Dense(n_actions, activation='softmax', name='predictions')(x)

   # Define model
   model = Model(inputs=[state_input], outputs=[out_actions]) # the model takes
as input the current state and outputs a list of actions
   model.compile(optimizer=Adam(lr=1e-4), loss='mse')

   return model
```

Code 4:

```
for itr in range(ppo_steps): # collect 128 interactions with the game
   state_input = K.expand_dims(state, 0)
   action_dist = model_actor.predict([state_input], steps=1) # uses the actor
model to predict the best actions
   action = np.random.choice(n_actions, p=action_dist[0, :]) # picks an action
based on the action distribution from the actor model
   action_onehot = np.zeros(n_actions)
   action_onehot[action] = 1
   observation, reward, done, info = env.step(action)
   if done:
       env.reset() # reset if the game is done
env.close()
```

Code 5:

```
def get_model_critic_image(input_dims):
    state_input = Input(shape=input_dims)

    # Use MobileNet feature extractor to process input image
    feature_extractor = MobileNetV2(include_top=False, weights='imagenet')
    for layer in feature_extractor.layers:
        layer.trainable = False

    # Classification block
    x = Flatten(name='flatten')(feature_extractor(state_input))
    x = Dense(1024, activation='relu', name='fc1')(x)
    out_actions = Dense(1, activation='tanh')(x) # output node from the neural net
is 1 since it is only the q_value


    # Define model
    model = Model(inputs=[state_input], outputs=[out_actions])
    model.compile(optimizer=Adam(lr=1e-4), loss='mse')

    return model
```

Code 6:

```
for itr in range(ppo_steps): # collect 128 interactions with the game
    state_input = K.expand_dims(state, 0)
    action_dist = model_actor.predict([state_input], steps=1) # uses the actor
model to predict the best actions
    q_value = model_actor.predict([state_input], steps=1)  # uses the critic model
to predict the q value.
    action = np.random.choice(n_actions, p=action_dist[0, :]) # picks an action
based on the action distribution from the actor model
    action_onehot = np.zeros(n_actions)
    action_onehot[action] = 1
    observation, reward, done, info = env.step(action)
    mask = not done

    states.append(state)
    actions.append(action)
    actions_onehot.append(action_onehot)
    values.append(q_value)
    masks.append(mask)
    rewards.append(reward)
    actions_probs.append(action_dist)

    state = observation # changing the state variable into the new observation or
the next state, otherwise we use the same initial state as input to out models.

    if done:
        env.reset() # reset if the game is done
env.close()
```

Code 7:

```
env = football_env.create_environment(env_name='academy_empty_goal',
representation='pixels', render=True)

n_actions = env.action_space.n
dummy_n = np.zeros((1, 1, n_actions))
dummy_1 = np.zeros((1, 1, 1))

model_actor = load_model('my_first_model_actor.hdf5')
model_critic = load_model('my_first_model_critic.hdf5')
```

```
        state = env.reset()
        done = False

        while True:
            state_input = K.expand_dims(state, 0)
            action_probs = model_actor.predict(state_input, steps=1)
            action = np.argmax(action_probs)
            next_state, _, done, _ = env.step(action)
            state = next_state
            if done:
                state = env.reset()
```

Code 8: (SimplerModel.py)

```
import os.path
import gfootball.env as football_env
import numpy as np
import tensorflow as tf
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import backend as K
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2

gamma = 0.99
lambda_ = 0.95

def test_reward():
    state = env.reset()
    done = False
    total_reward = 0
    print('testing...')
    limit = 0
    while not done:
        state_input = K.expand_dims(state, 0)
        action_probs = model_actor.predict([state_input], steps=1)
        action = np.argmax(action_probs)
        next_state, reward, done, _ = env.step(action)
        state = next_state
        total_reward += reward
        limit += 1
        if limit > 20:
            break
    return total_reward

def get_advantages(values, masks, rewards):
    returns = []
    gae = 0
    for i in reversed(range(len(rewards))):
        delta = rewards[i] + gamma * values[i + 1] * masks[i] - values[i]
        gae = delta + gamma * lambda_ * masks[i] * gae
        returns.insert(0, gae + values[i])

    adv = np.array(returns) - values[:-1]
    return returns, (adv - np.mean(adv)) / (np.std(adv) + 1e-10) # the 1e-10 residue is
to make sure not to divide by zero

def get_model_actor_image(input_dims):
    state_input = Input(shape=input_dims)

    # Use MobileNet feature extractor to process input image
    feature_extractor = MobileNetV2(weights='imagenet', include_top=False)
    for layer in feature_extractor.layers:
        layer.trainable = False

    # Classification block
    x = Flatten(name='flatten')(feature_extractor(state_input))
    x = Dense(1024, activation='relu', name='fc1')(x)
    out_actions = Dense(n_actions, activation='softmax', name='predictions')(x)

    # Define model
    model = Model(inputs=[state_input], outputs=[out_actions]) # the model takes as
input the current state and outputs a list of actions
    model.compile(optimizer=Adam(lr=1e-4), loss='mse')

    return model

def get_model_critic_image(input_dims):
    state_input = Input(shape=input_dims)

    # Use MobileNet feature extractor to process input image
    feature_extractor = MobileNetV2(include_top=False, weights='imagenet')
    for layer in feature_extractor.layers:
        layer.trainable = False

    # Classification block
    x = Flatten(name='flatten')(feature_extractor(state_input))
    x = Dense(1024, activation='relu', name='fc1')(x)
    out_actions = Dense(1, activation='tanh')(x)  # output node from the neural net is 1
since it is only the q_value

    # Define model
    model = Model(inputs=[state_input], outputs=[out_actions])
    model.compile(optimizer=Adam(lr=1e-4), loss='mse')

    return model

env = football_env.create_environment(env_name="academy_empty_goal",
representation="pixels", render=True)
state = env.reset()
state_dims = env.observation_space.shape
print(state_dims)
n_actions = env.action_space.n
print(n_actions)

tensor_board = TensorBoard(log_dir='./logs')

ppo_steps = 128
target_reached = False
best_reward = 0
iters = 0
max_iters = 50

model_actor = get_model_actor_image(input_dims=state_dims)
model_critic = get_model_critic_image(input_dims=state_dims)
while not target_reached and iters < max_iters:
    states = []
    actions = []
    values = []
    masks = []  # checks if the match is in completed/over state, in which case we want
to restart the game
    rewards = []
    actions_probs = []
    actions_onehot = []
    state_input = None

    for itr in range(ppo_steps): # collect 128 interactions with the game
```

```
        state_input = K.expand_dims(state, 0)
        action_dist = model_actor.predict([state_input], steps=1) # uses the actor model
to predict the best actions
        q_value = model_critic.predict([state_input], steps=1)  # uses the critic model
to predict the q value.
        action = np.random.choice(n_actions, p=action_dist[0, :]) # picks an action
based on the action distribution from the actor model
        action_onehot = np.zeros(n_actions)
        action_onehot[action] = 1
        observation, reward, done, info = env.step(action)
        print('itr: ' + str(itr) + ', action=' + str(action) + ', reward=' + str(reward)
+ ', q val=' + str(q_value))
        mask = not done

        states.append(state)
        actions.append(action)
        actions_onehot.append(action_onehot)
        values.append(q_value)
        masks.append(mask)
        rewards.append(reward)
        actions_probs.append(action_dist)

        state = observation # changing the state variable into the new observation or
the next state, otherwise we use the same initial state as input to out models.

        if done:
            env.reset() # reset if the game is done
    state_input = K.expand_dims(state, 0)
    q_value = model_critic.predict([state_input], steps=1)
    values.append(q_value)
    returns, advantages = get_advantages(values, masks, rewards)
    actor_loss = model_actor.fit(
            [np.reshape(states, newshape=(128, 72, 96, 3))], [np.reshape(actions_onehot,
newshape=(-1, n_actions))], verbose=True, shuffle=True, epochs=8,
            callbacks=[tensor_board])
    critic_loss = model_critic.fit([np.reshape(states, newshape=(128, 72, 96, 3))],
[np.reshape(returns, newshape=(-1, 1))], shuffle=True, epochs=8, verbose=True,
callbacks=[tensor_board])
    avg_reward = np.mean([test_reward() for _ in range(5)])
    print('total test reward=' + str(avg_reward))
    if avg_reward > best_reward:
        print('best reward=' + str(avg_reward))
        model_actor.save('model_actor.hdf5')
        model_critic.save('model_critic.hdf5')
        best_reward = avg_reward
        if best_reward > 0.9 or iters > max_iters:
            target_reached = True
    iters += 1
    env.reset()
env.close()
```

Actions list:

The default action set consists of 19 actions:

- Idle actions
    - `action_idle` = 0, a no-op action, sticky actions are not affected (player maintains his directional movement etc.).
- Movement actions
    - `action_left` = 1, run to the left, sticky action.
    - `action_top_left` = 2, run to the top-left, sticky action.

  - `action_top` = 3, run to the top, sticky action.
  - `action_top_right` = 4, run to the top-right, sticky action.
  - `action_right` = 5, run to the right, sticky action.
  - `action_bottom_right` = 6, run to the bottom-right, sticky action.
  - `action_bottom` = 7, run to the bottom, sticky action.
  - `action_bottom_left` = 8, run to the bottom-left, sticky action.

- Passing / Shooting

  - `action_long_pass` = 9, perform a long pass to the player on your team. Player to pass the ball to is auto-determined based on the movement direction.
  - `action_high_pass` = 10, perform a high pass, similar to `action_long_pass`.
  - `action_short_pass` = 11, perform a short pass, similar to `action_long_pass`.
  - `action_shot` = 12, perform a shot, always in the direction of the opponent's goal.

- Other actions

  - `action_sprint` = 13, start sprinting, sticky action. Player moves faster, but has worse ball handling.
  - `action_release_direction` = 14, reset current movement direction.
  - `action_release_sprint` = 15, stop sprinting.
  - `action_sliding` = 16, perform a slide (effective when not having a ball).
  - `action_dribble` = 17, start dribbling (effective when having a ball), sticky action. Player moves slower, but it is harder to take over the ball from him.
  - `action_release_dribble` = 18, stop dribbling.