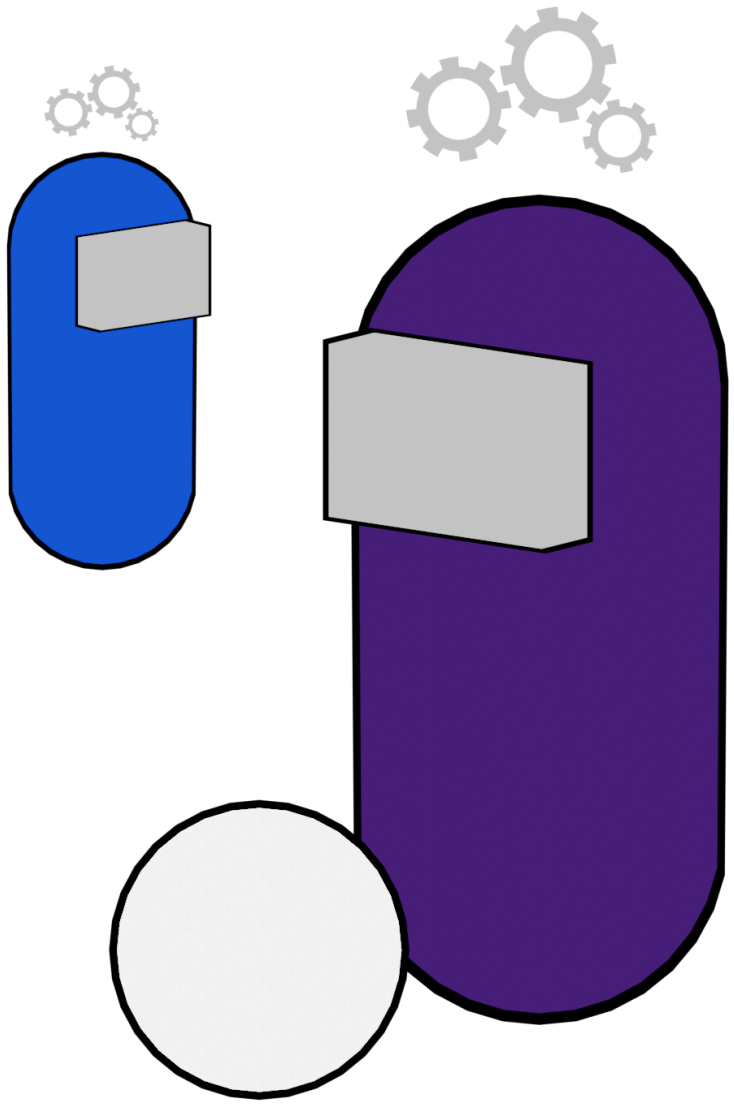


AI EXAM

By

Kristofer Pedersen
cph-kp278

Mads Teddy Francois Knudsen
cph-mk682

Michael Frederiksen
cph-mf315

Søren Merved
cph-sm428

Github link:
https://github.com/Mfknudsen/AI-Exam

Introduction:

For our final project for artificial intelligence, we have made a product where we implement three different models. Our product is a video game made in the Unity game engine and focuses around a game of soccer being played by a model with extra functionality added on by other models. We have trained our models on data we have generated ourselves from within the Unity game engine.

Business case:

We wanted to make a piece of interactive media using different kinds of AI models. Within game development and the interactive media industry the use of AI models are increasing. We have therefore chosen to make our own product to illustrate their uses where we could interact with the models as we thought the idea, while still relatively new, has lots of possibilities for the future.

Models:

In this section we will go over what model we have chosen and the process for each model, their setup, data and training.

Reinforced Learning with Unity MLAgents:

For our first model we decided to train a model using reinforced learning with the use of Unity and their MLAgent library. We have decided to use Unity as we all have some experience using the game engine and because we find the thought of having a trained model play a game interesting and educational.

Our goal for this model is for it to be able to play a game of soccer within Unity. The game is a two vs two with each team containing one agent filling the role of striker and another filling the role of goalie. There should be no consistent bugs, errors or cheats that it uses to “game the system”.

Keywords:

- **Unity:**

Unity is a game engine where in the game world are constructed using game objects. Scripts made to work within this world are written in C# and must inherit from MonoBehaviour.
It contains different global variables such as Time.deltaTime, which can be used to ensure a consistent move speed across multiple frames.

- **GameObject:**

Game objects are the building blocks in Unity and by default they only contain a name, a tag, a physics layer and transform(position, rotation and scale) components, all of which cannot be removed. Scripts inheriting from MonoBehaviour can be added onto game objects as components to create logic.

- **Rigidbody:**

Rigidbody is a built-in component and is used to introduce physics to a game object. You can disallow effects for individual axes for both rotation and position, or be able to make it completely stationary.

- **Tag:**

Tags are strings and are often used to check against a collider when one game object hits another game object using their collideres, either through directly hitting with most colliders or with intersection with trigger colliders which doesn't stop colliders moving through.

- **MLAgents:**

MLAgents are a library made by Unity themselves and are their solution for using AI modules, such as reinforced learning, within Unity as components. The training components are python based and are executed outside of Unity.

Setup:

The setup described is the one used in the final product and doesn't show how it evolved over time. That will instead be described in some parts in "Training and Rewards". When going over the setup we will first name the part and then the components it contains for its logic and finally we will give a small description around how it is supposed to work and how it works with other parts of the setup.

Agents:

- MLAgent Components.
- Tag: blueAgent/purpleAgent depending on team
- Forward and backwards sensors
- Capsule collider
- Rigidbody

Agents are the game objects which contain the necessary components for training and using the resulting generated models during gameplay. It is the aim for the model to make them play soccer while maintaining two different roles, striker and goalie.

Field:

- Two goal with the tag blueGoal/purpleGoal depending on team
- Walls made with box colliders and visual elements.
- Four agents, two purple and two blue.
- One ball.
- Field environment.

The field is where the game takes place. It uses walls to determine the play area for the agents and the ball. It also contains the logic for group training. The goals themselves don't contain any logic but have the necessary tags for the scoring logic.

Ball:

- Tag: ball
- Sphere collider
- Rigidbody

The ball is a simple sphere with its own physics material placed on the rigidbody. The physic material allows us to specify that we want the ball to be more bouncy than objects normally are using their default settings. The ball also contains the logic needed for scoring of goals.

Settings:

To ensure a consistent training environment within Unity, we have a game object with a settings script which applies some settings we specify within the Unity editor. An example of a setting would be changing the Time.deltaTime, which is the time between the current frame and the previous, to be consistent, as running the training results in many long lag spikes. Without these changes to the settings we would for example see the ball and players run past the walls of the field each time a spike occurs.

Soccer.yaml:

- Max steps: 10.000.000
- Self Play

When training it will use the default settings from a generated .yaml file. With Soccer.yaml we are able to control how our training works outside of Unity. In our .yaml file we have increased the max step count which controls how long it will train and we added self play settings which is needed when training more than one team against each other. In our case it would be the blue team against the purple team. It is used by specifying the .yaml file when running the training command in the virtual environment. The max steps that we have specified results in the training taking about 5,5 hours to complete.

Learning:

- Virtual environment.
- ELO rating

When learning, there can be multiple field instances active in the scene and they will all add to the training of the model. Since we have two teams competing against each other during the training, we need a way to determine which model iteration is more likely to win. Default ELO rating starts at 1200 when running the training. We are looking for model iterations with a higher rating with the largest being the desired result.

Training and Rewards:

During training we had multiple iterations and reward setups as we tried to get the model to produce a result we wanted. We had more iterations than described in this section but most have little differences compared to others, so we have selected a few we believe contain something important to the process of training or if we felt they had moved closer to our final goal for the model. We will first write how they receive their rewards, increasing or decreasing, and then a description of the progress.

Unexpected but fun:

Reward setup:

- Goalie: Increase the longer the game goes.
- Striker: Decrease the longer the game goes
- Kick: Increase if they hit the ball.
- Goal score: Increase for their group if their team scored.

During the training we had the agents receive an increasing reward for the goalie for the longer the game went on and a decreasing reward for the striker the longer the game went on. This was meant to push them towards specific roles resulting in the goalie leaving the goal in search of the ball and then it would attempt to keep the ball from other players by walking in circles with the ball. This resulted in the striker not learning as the reward would be decreased too much from the game going on for so long.
It was rather a fun and learning full experience to watch how good the goalies were at walking in a near perfect circle with the ball.

When looking online for other model training projects, it is often a topic of how the model surprises the programmer and in what ways the models managed to cheat the system they were placed in.

Closer to the goal:

Reward setup:

- Goalie: Increase the longer the game goes, but only if it remains close to its own goal.
- Striker: Decrease the longer the game goes.
- Kick: Increase if they hit the ball, with the amount determined by how close the kick direction is towards the opponents goal.
- Wall: Decrease if they touch the wall.
- Goal score: Increase for their group if their team scored.

With a more refined reward setup the agents began to more and more behave like we wanted but there were still more errors to be removed through refining. Though some errors occurred not because of the agents but more how their environment was set up. During training it became clear quickly that if the ball reached one of the four corners then the team belonging to the closest goal would both try to kick it out of the corner but due to their size and the size of the corner, they would end up keeping the ball stuck in the corner, thereby freezing the game.
There are two possible solutions that could be implemented to solve this issue. The first one is to increase the size of the corner and the second is to change the shape of the agents to squares. These two changes could affect how the model decides to move when training the next time.

Goal:

Reward setup:

- Goalie: Increase the longer the game goes, but only if it remains close to its own goal and not in its own corner.
- Striker: Decrease the longer the game goes.
- Kick: Increase if they hit the ball, but if goalie then only if it remains close to its own goal..
- Wall: Decrease if they touch the wall.
- Goal score: Increase for their group if their team scored.

With this iteration we finally see them playing in a way that is acceptable. They appear to be actively seeking out the ball in an attempt to hit it and end games quicker by hitting a goal. They are more active in comparison to previous iteration and are no longer trapped in corners with the ball. This new behavior also produces a more exciting game where you can't be sure who will win based on theirs and the ball's current position.
The goalie has been seen trying to block the ball when it gets close to their own goal with the striker not contesting its own corner, therefore preventing deadlocks, and allowing it to be ready to receive the ball. This behavior was already beginning to appear about ⅛ through the training with more refined decisions.

Final thoughts:

Though not near perfect, the model has learned to play the game to an acceptable degree with no noticeable consistent quirks, bugs or cheats like we saw in some of the first iterations. It does still struggle a little bit when the ball is in a corner or along the edge.

Looking back at how we rewarded and punished the agents during training, we found that rewards based on more complex logic saw the agent having more difficulty learning. When trying to reward the agents based on if they kicked the ball closer to the opponents goal, we saw the agents having a reduced interest in kicking the ball, as it most likely could figure out how to work out how to get the biggest reward instead of the greatest punishment giving when they would kick it away from the opponents goal, which used the dot product. With this knowledge we reworked our rewards to be focused around if statements and saw an instant improvement in the learning process.

There also doesn’t appear to be any kind of cooperation between the members in a team. They both just do what they want and can therefore go against each other during some interaction. The lack of cooperation isn't surprising as we don’t reward them or punish them based on any interaction between team members or even opponents.

Lastly the agents don’t really look around when the ball is not in front of them. They are most likely relying too much on their back sensors, which may work but are also reducing their playing effectiveness as they are more likely to miss the ball when it's behind them.

Image classification:

The MLagents have been set up with several sensors. Each sensor is able to calculate the distance to the target and determine what the target is with simple scripts. However to make a true sophisticated AI we want to replace these sensors with camera(s). AI models will listen to each camera and classify what the camera is looking at, and the distance to the target. For a solution we have decided to make a CNN classification model and regression model.

Data:

To obtain the data we needed to train our model, we created a script to take images of the different objects within our game. The script would circle around each object a set number of times and take images of the target object from different angles and distances.
Since we used the in game camera from the Unity editor, we had extra options when deciding the output image result. This gave us the ability to control how the shots were taken, isolating the target, removing background or adding the background at different places.

Training and data gathering:

The process of creating this solution can be separated into 4 different methods, they are as followed ordered chronologically. More details on the training process can be read within the jupyter notebooks provided in the links.

Model 1 - without context

link:
https://github.com/Mfknudsen/AI-Exam/blob/main/AI%20Exam/CNN/model_no_noise.ipynb

We initially tried to create a classification model that can recognize the 5 game objects we had. The data gathering process was the simplest of all of the prosseces. We took pictures of the subjects 360 degrees at 4 different distances. No background was included in the pictures.

The model proved to be very simple to train and on our first attempt the model performed very well. We suspect that the model may be biased towards the data since there is little variation between the images. This is why we decided to challenge ourselves by expanding on this later on.

Following image is an example image from the dataset:

Model 2 - hybrid model

link:
https://github.com/Mfknudsen/AI-Exam/blob/main/AI%20Exam/CNN/model_hybrid_regression_cnn.ipynb

Because we want the model to also determine the distance we want to make a hybrid model, a CNN image classification and a regression model that can predict distance to the target.
We collected data again, a similar process to the previous step. We labeled the images with the distances to the target as well as took more images at different distances.
We were able to create a hybrid model that is able to classify classes however it was terrible at predicting distances. There was no sign of improving this, therefore this model and process was scrapped.

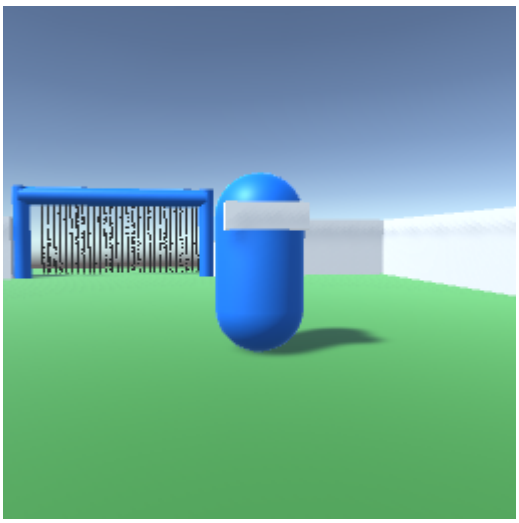
Model 3 - with context

link:
https://github.com/Mfknudsen/AI-Exam/blob/main/AI%20Exam/CNN/model_with_noise.ipynb

In this model we wanted to diversify the images, in an attempt to combat biases from our first model. To do this we took more images of the subjects, this time we took images of them in the football field as well as them with the other classes in the foreground.

It was difficult to create a model that was able to properly classify the object. When using this model in the virtual environment of the game, it predicted almost all objects as the Goal Posts wrongly. Our conclusion was that it was wrong to train the models on images that had several classes at once, with a label for only one of the classes.

Following image is an example image from the dataset:



Model 4 - with semi context

https://github.com/Mfknudsen/AI-Exam/blob/main/AI%20Exam/CNN/model_semi_noise.ipynb

For this process we collected new data, similar to the previous process. We took images of the subject classes in the field with different backgrounds, however we isolated the subject so that no other classes would disturb the image.
This model proved to be the best, both statistically (ignoring the first model) and in the virtual environment in-game.

Following image is an example image from the dataset:

Model 5 - with context updated

https://github.com/Mfknudsen/AI-Exam/blob/main/AI%20Exam/CNN/model_with_noise_updated.ipynb

This process involved going back to model 3 - with context and looking at the data. This was because there was a suspicion that the dataset had outliers. When looking at the data we discovered that there were many outliers with the subject not inframe. After cleaning up the

images and retrained the model, the trained model performed much better than the earlier examples, proving that data cleaning is not a bad thing at all.

LLM:

For the user to be able to give the agents commands we use vector embeddings. Vector embeddings give us the opportunity to find semantic relationships between objects. By using this method we can associate the user's prompt with a command that is predefined. For the model we are using openAIs ADA-02.

Setup:

A list of commands are defined before the game is started with a list of commands with actions the agents can perform. Commands will have a subject agent (the command is given to a specific agent) and some commands have an object agent (the command will have a target). Each potential combination of command, target object and target subject is stringified for example “player 1 stun player 2”.

At the start of the game each combination of commands is vectorized via fetch request and stored in the command controller script. The user's command is vectorized during the game, when the user prompts the game. The prompt’s vector embedding is then compared to the commands’ vector embeddings. To calculate the similarity, cosine similarity is calculated between the prompt’s vector embedding and the commands’ vector embeddings. The command with the smallest distance to the prompt is of course chosen as the most similar prompt and that command with the agent performing the corresponding action.

Data:

For this model we couldn’t prepare any data beforeh
+and for the training of the model. This is because the model itself doesn’t need to be trained by us, since it is available through openAi.
For the vector data regarding the commands, we decided to generate it on each startup of the game, since we continuously added new actions and changed values such as their names.

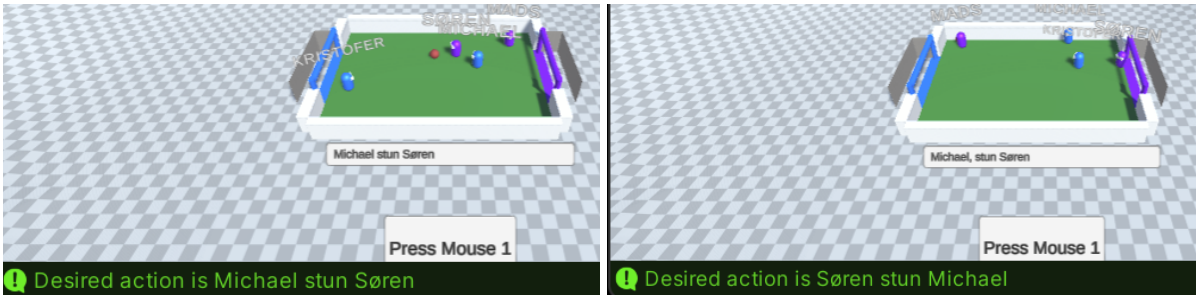
Quality Assessment

Reinforce Learning:

With the way we trained our reinforced learning model we can't produce any models or run any analysis using python like we normally would.
The reason is because when using Unity's python training scripts they don't produce any information we are capable of analyzing.
When running the training scripts there were three different values displayed each recorded step interval: Mean reward, Mean group reward and ELO.
Mean reward refers to the average reward among the active agents during the training. Here we expect an increase over time as the agents begin to understand what they need to do to get their rewards. It started out in the negative as the agents would move about randomly, then increase rather rapidly and finally begin to slow their increase towards an average increase per step interval to zero, as there is a limit to how much they can gain during one episode.
We can't really use mean group reward as a group either ends with -1, 0 or 1.
ELO starts at a value of 1200 and indicates the prediction value of a model's likelihood of winning. The change isn't expected to be very large between each interval but an increase over time indicates the model's improvement. This value is used by the training program to decide which models to compete against each other.

LLM:

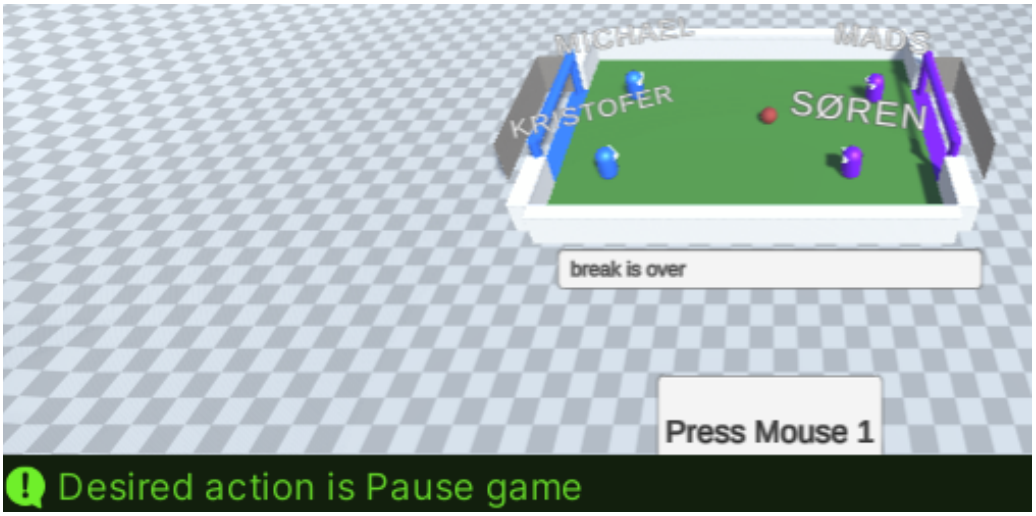
The large language model turned out to work really well. It is capable of providing the most similar action to the command written by the user as intended. It is able to predict general actions like pausing and resuming the game, making specific players emote, attack or defend as well as making a specific player target another.
The large language model does however seem to have a few instances where it struggles to provide the desired result. A somewhat uncommon example is an issue regarding comma included sentences. An example of this is shown in the picture below.



In the first image, Michael is stunning Søren, which is the action requested by the user. In the second image however, the same sentence with only a comma difference, results in the person performing the stun to be Søren. In this example, the same sentence with and without comma thereby swaps the roles of the stunner and target. This isn't always the case, but it seems the queries with the comma included occasionally change the roles of the

action. This does happen rarely though and since it is a rare occurrence, fixing it wasn't prioritized.

Another issue the model has is sentences including specific words that have a different meaning due to the context they are in. A great example is shown below.



The sentence "Break is over" would to most people convey a message saying that the game is to continue. The LLM, on the other hand, takes the word "Break" too literally, without including the context and the crucial words that follow. This results in the opposite of what the user requested, however this is but a small issue, since in order to resume the game, it has to already have been paused, making pausing it again unintentionally, completely indifferent.

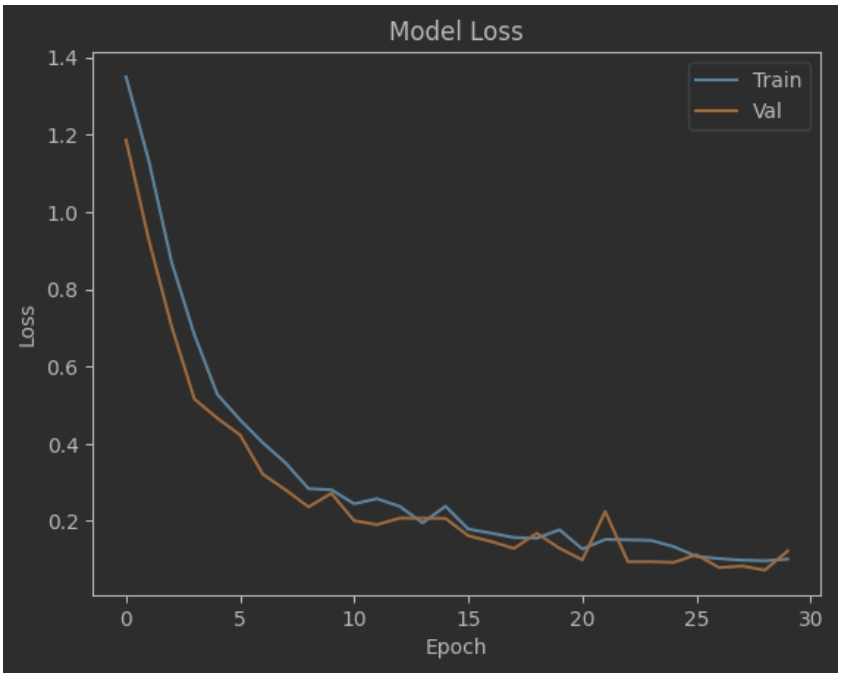
Generally speaking the large language model does provide the desired action to be performed with only rare instances of performing an unintentional action. For the best chance of achieving the desired outcome, it is recommended using short, precise messages without commas, since these make misunderstandings highly unlikely and the model overall very reliable.

Classification:

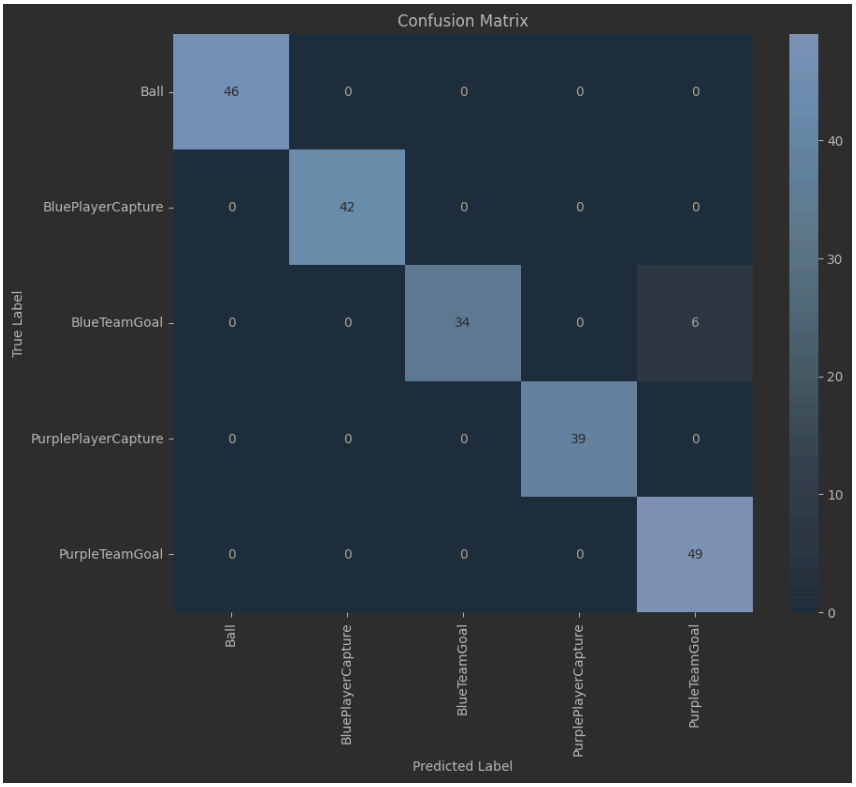
Details of quality and statistics of the Classification models can be found in their respective jupyter notebook files (linked earlier in the report) and this section will therefore not go into great detail.We will focus on the best models from each process.

Model 1 - no context

The following graph shows the validation training loss plot. Overall the plot is good, the lines converge and stabilize after several epochs.

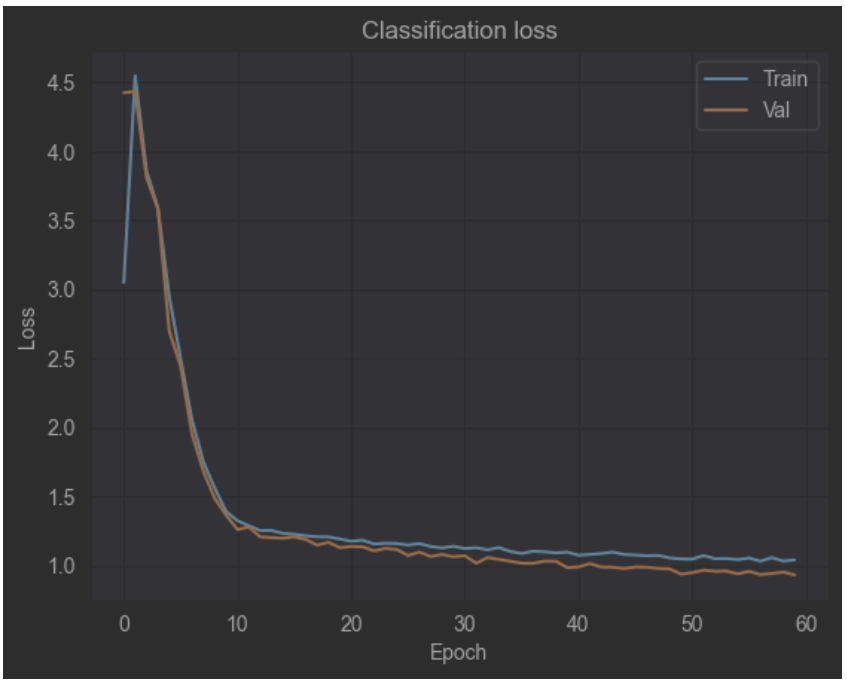


The following plots show the confusion matrix. There is a strong concentration of true positives in the diagonals.

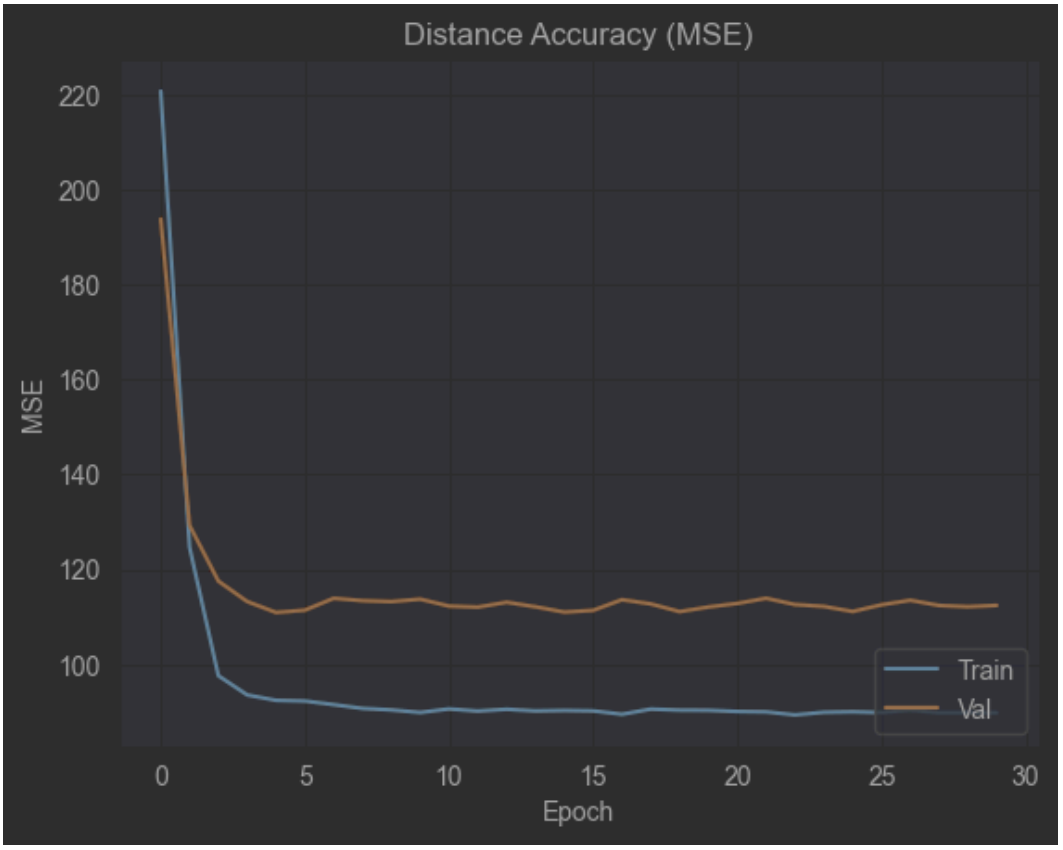


Model 2 - hybrid model

The following graphs represent the model with 16 filters



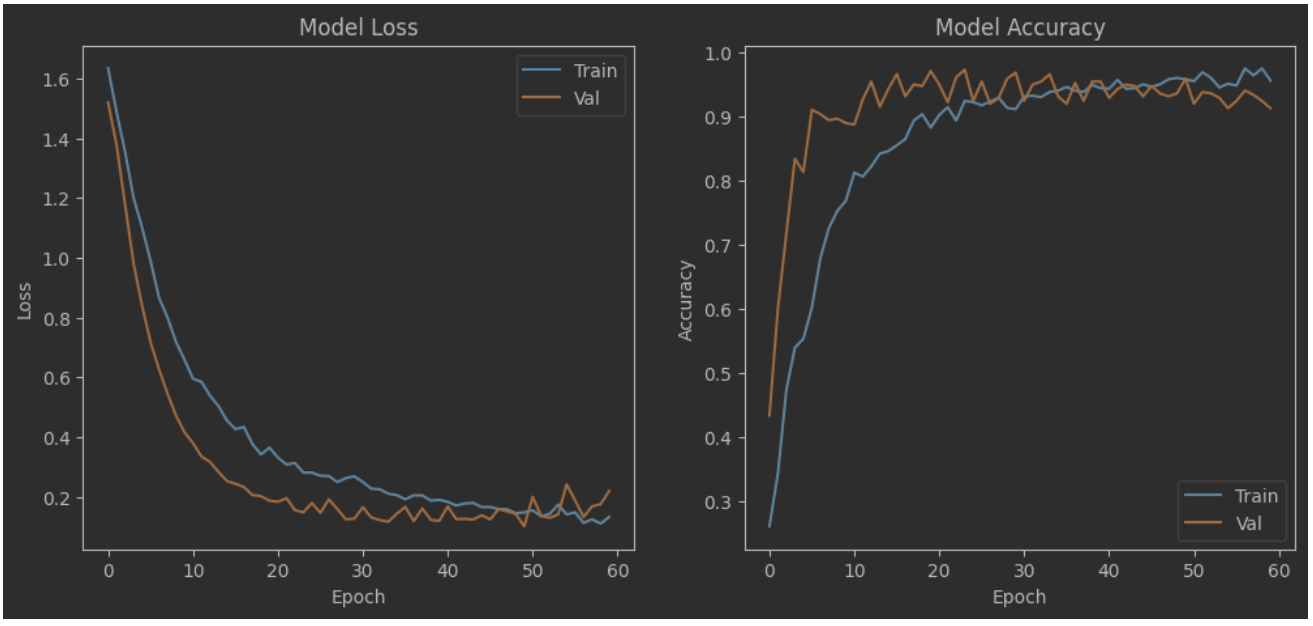
The plot above shows the validation training plot. The plots don't quite converge; it may be because of regularization or dropout that the training plot is worse than Validation.

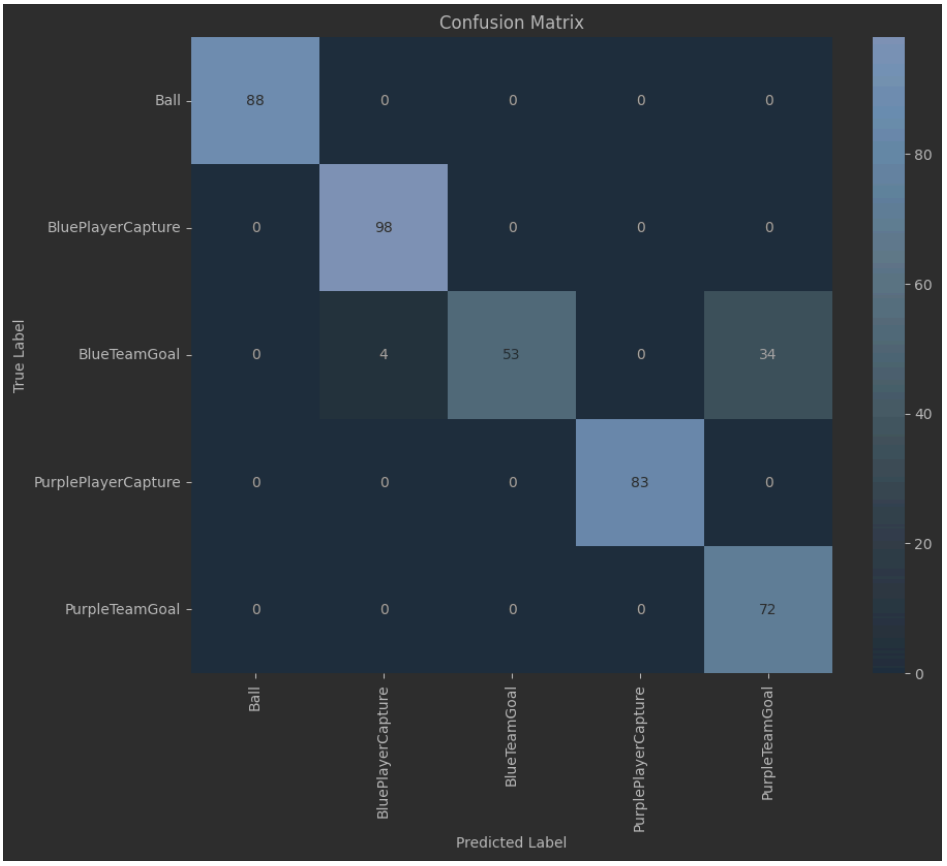


As this model is a combined model with both regression and cnn, this model also has a regression output. The plot above shows the MSE for the regression, the graph unfortunately shows that the model performs terribly when it comes to measuring distances.

Model 3 - with noise

The plots represent the final model in Model 3 with noise.

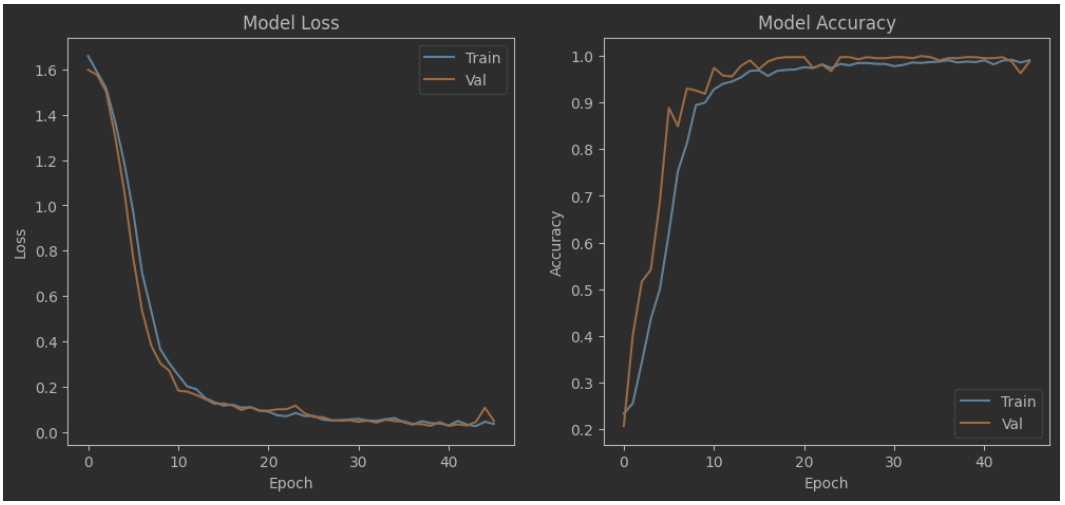




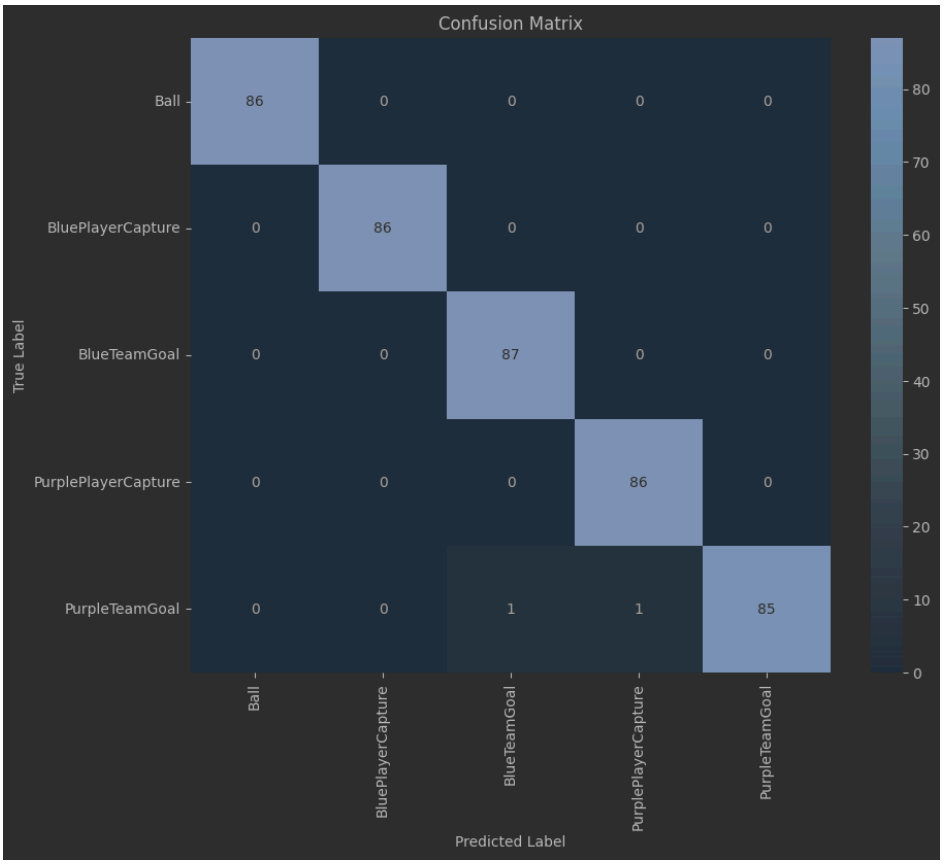
The confusion matrix shows promise with strong diagonals and good accuracy . It incorrectly classifies blue team goals several times as purple team goals. However when using this model in the virtual environment of the game, it performed terribly. Almost most of the objects were classified as Purple Team goals making the model unusable.
To figure out why the blue team goals keep on being mislabeled as purple team goals we displayed the mislabeled images. This showed some anomalies in the images which led to the investigation of looking through our data and finally data cleaning.

Model 4 - Semi Noise

The following section represents the best performing model from Semi Noise models.



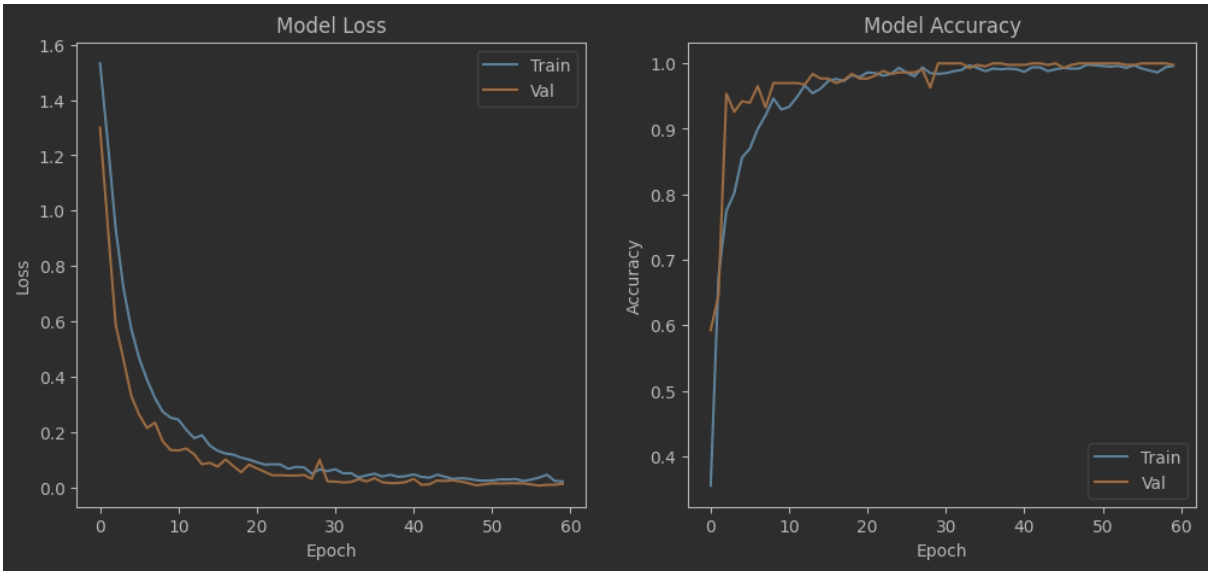
The validation training loss plot above is almost an ideal plot. The lines converge and stabilize and around 0.1 loss.



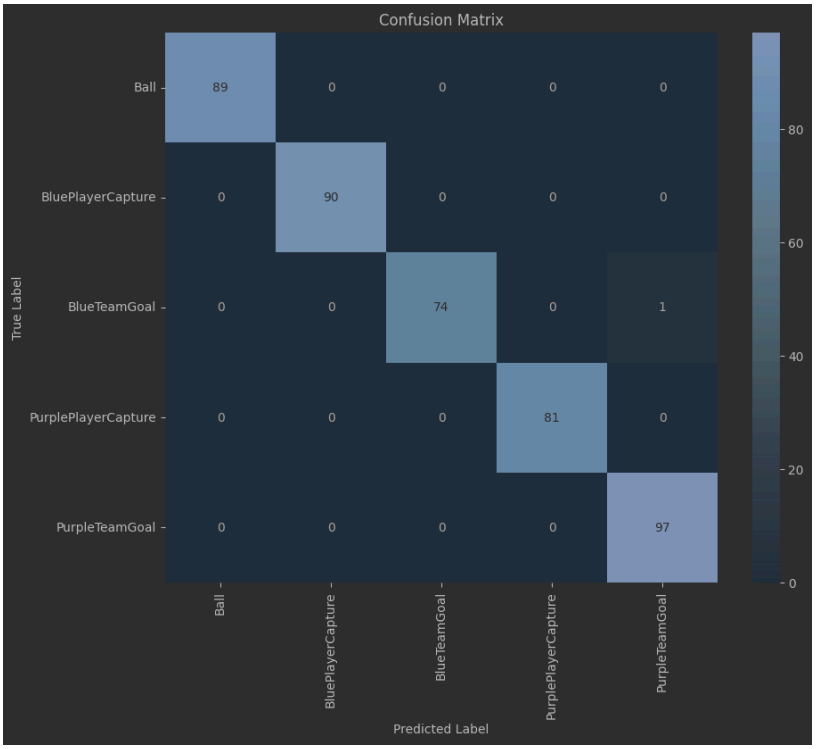
The confusion plot represents a very good model. The error is 0.47% which is good. These plots represent an optimal model and when used in game it had the best results, being able to identify all classes in the game, although not at the same error rate the plots show. For this reason this model was chosen as one of the candidates for the final model.

Model 5 - with noise updated

This section represents the final model we trained within the noise updated process. Where we return to our model or process with backgrounds and other objects in the image.



The plot above shows a very fine validation train loss plot, demonstrating a final loss of close to 0. The model Accuracy is also representative of this and close to 100%.



The confusion matrix shows that when it was tested on the training set, it only made 1 error, which shows that models are quite optimal.
This model demonstrates a very good performance therefore it was chosen as one of the final models.

AI Application:

Implementation of Reinforced Learning:

For the most part our reinforced learning model didn't need any implementation as it was nearly in its entirety made within the Unity game engine with the python scripts used to train the model not being needed to run the model during the games runtime.

We took the entire setup for the training and placed it within our final product scene, which is the scene the player would play within when the game would be built and released. We then went over each of the character controllers, which each individual soccer player had as a component, to make sure everything worked as intended.

Implementation of LLM:

The implementation of the large language model was done by using an openAI API. Through the use of this endpoint, we were able to vectorize both the input of the user and the available action options of the game. At the start of the game, all the available actions were vectorized right away and stored. When the user would then type in command, the input would get vectorized immediately. The vectorized user input would then be compared to each action option and the one most similar to the users input would get deemed the desired one and therefore executed.

Implementation of Classification:

The initial plan was to create a model that could predict both the class of the object its looking at and the distance. While we were able to train a model that is able to predict classes the distance became difficult to measure with one model. For the mlagent to act properly, we need it to also know the distance to the target it's looking at which why this model is difficult to implement in the final model. The mlagents are not able to run entirely off the model without the help of censors which is why this idea was not fully implemented.

To make use of our classification models we used it to predict what the main game camera was looking at. The model would first have to be converted from a keras model to an onnx type. We then use the barracuda inference package to be able to run the onnx model in unity. The user is able to press c to call the model and it will predict whatever they're looking at.

Images:

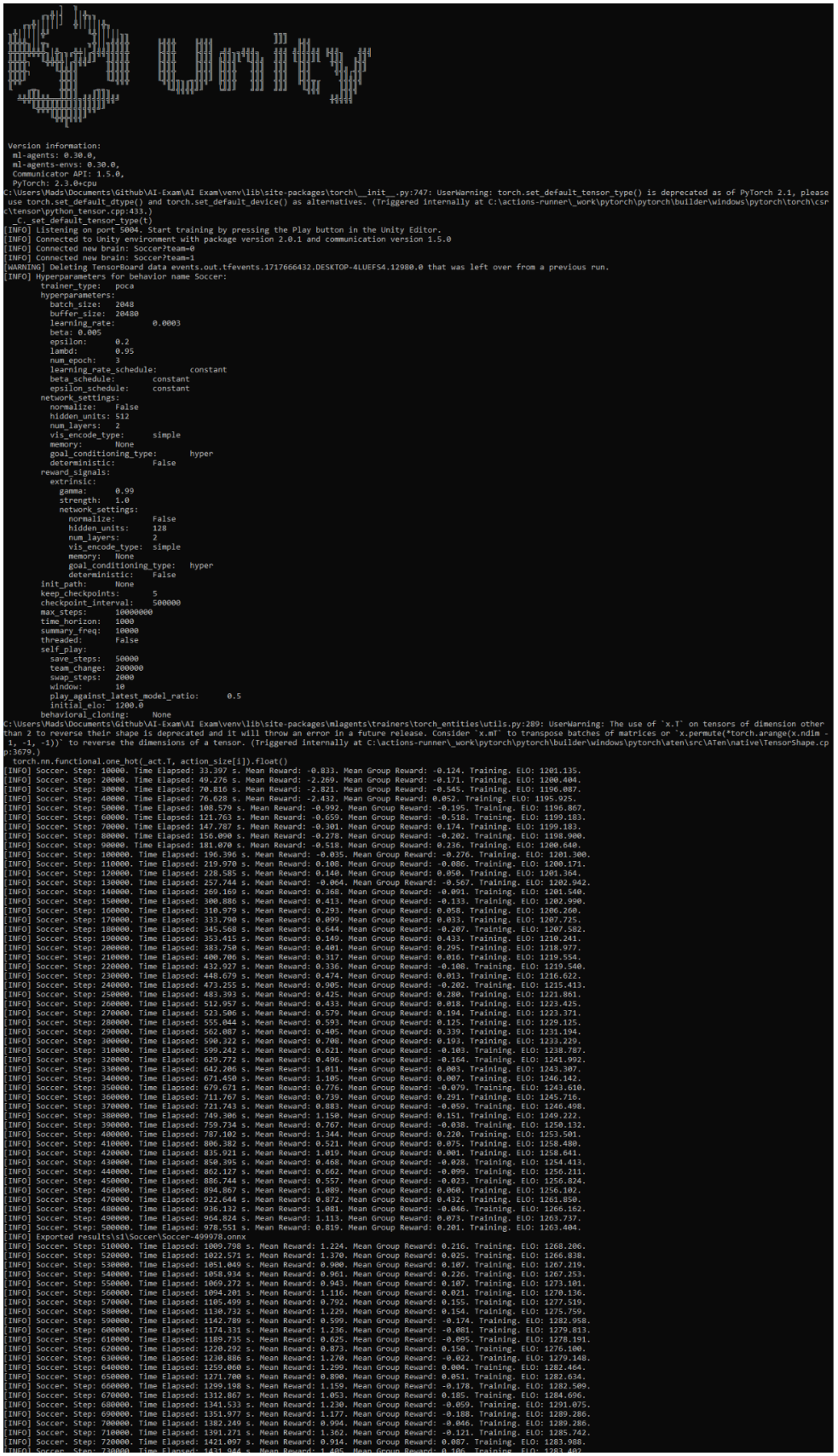


Image A1: Screenshot of the python training scripts running in a virtual environment.

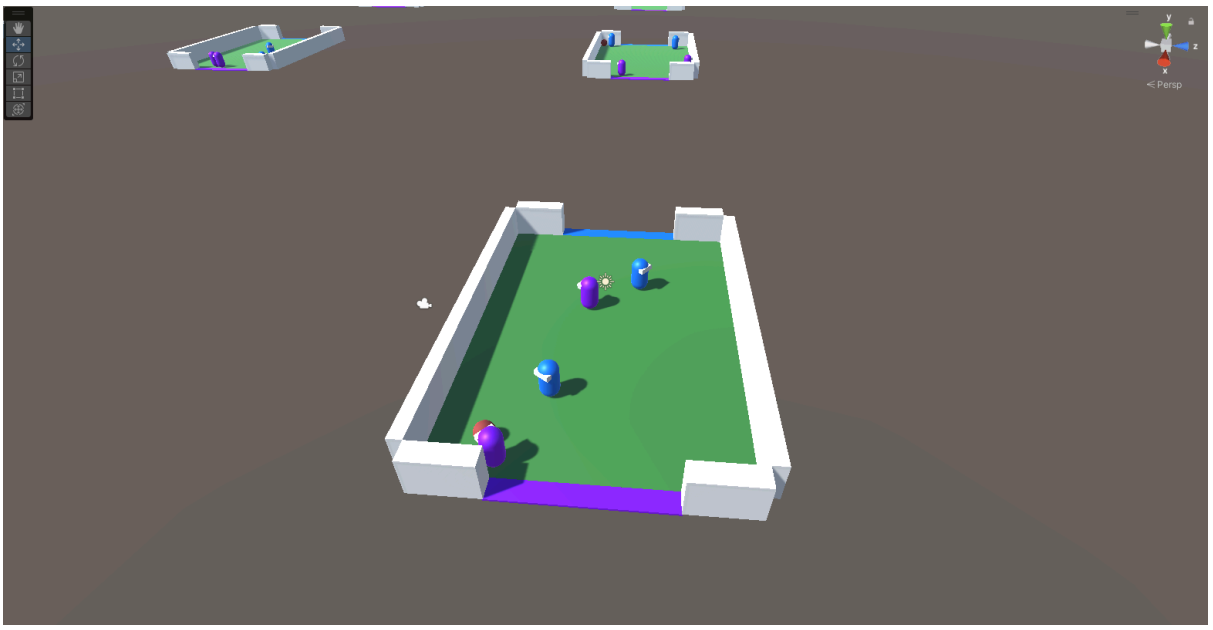


Image A2: Screenshot of Unity with the agents visible during their training.

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

Expected score of player A

Difference between Elo score of B and A

Image A3: Expected score of each player.

$$R'_A = R_A + K(S_A - E_A)$$

New Elo score of player A

Current Elo score of A

Adjustement Rating (K-factor)

Performed score of A

Expected score of A

Image A4: Calculation of new ELO score.