CM3 FRAMEWORK FOR DEEP MULTI-AGENT REINFORCEMENT LEARNING IN FOOTBALL

SHIVANI PATEL

Thesis supervisor: MIGUEL MORALES (Georgia Institute of Technology)

Thesis co-supervisor: CHARLES ISBELL

Tutor: ANNA RIO DOVAL (Department of Mathematics)

Degree: Master Degree in Artificial Intelligence

Master's thesis

School of Engineering
Universitat Rovira i Virgili (URV)

Faculty of Mathematics
Universitat de Barcelona (UB)

Barcelona School of Informatics (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

19/10/2023

Acknowledgements

I would like to express my sincere gratitude to all those who have contributed to the successful completion of this Master's thesis. The research exchange at Georgia Institute of Technology has been both challenging and rewarding, and I am indebted to the individuals and entities for their invaluable support, guidance, and inspiration, without whom it would not have been possible. The project was a very enriching experience for me - both academically/professionally as well as personally.

First and foremost, I extend my heartfelt thanks to my thesis advisor, Mr. Miguel Morales, and Dean Charles Isbell, for their expert guidance, unwavering support, and mentorship throughout this research endeavor. Their profound knowledge in the field of Deep Multi-Agent Reinforcement Learning has been instrumental in shaping the direction of my research. I sincerely appreciate my fellow DMARL lab mates at Georgia Tech and classmates from the Masters in AI at UPC for their valuable insights and feedback during the course of the project.

My family and friends deserve special recognition for their unwavering support, understanding, and encouragement, especially during the challenging times pertaining to administrative tasks and the logistics of the exchange.

Abstract

Collaboration amongst agents in various multi-agent cooperative and mixed environments has been extensively studied in the field of Deep Multi-Agent Reinforcement Learning. This cooperative behavior and roles emerging out of such cooperation could be beneficial for the agents collectively when they align their individual objectives towards a common goal, share resources effectively, and communicate efficiently to optimize their combined efforts. Research spans across various sub-areas, namely communication in MARL (Comm-MARL), intrinsic rewards, exploration in MARL, curriculum learning, reward shaping, and emergent behavior.

Cooperative Multi-Goal Multi-Stage Multi-Agent RL, abbreviated as CM3 [1] is one such framework that uses curriculum learning and a specialized policy function to tackle the issues of efficient exploration and credit assignment respectively. It has been tested on 3 multi-agent environments to demonstrate its power by learning significantly faster than direct adaptations of existing algorithms.

As part of this thesis, we have hypothesized if the domain of football from a multi-agent perspective benefits from CM3. Taking notes from the intersection of reinforcement learning and football [2][3], and some of the current state-of-the-art football algorithms, such as TiKick [4] and WeKick [5] which are based primarily on PPO, we see how actor-critic algorithms like A2C and PPO compare when used in our multi-agent environment. For this demonstration, we have leveraged a modified version of the Unity ML-Agents' SoccerTwos [6] environment. We also propose an additional enhancement to the original CM3 framework by extending the training further to a 3rd stage when the reward is independent of the goal. We hypothesized that it could enhance coordination because there'd be a single common, collective goal for the team - to win the match - as opposed to the individual goals of scoring or saving.

Table of contents

Introduction

Over the last decade there has been an increase in interest towards analytics in many team-sports but with the largest fan-following across the globe, football takes the first stance. As one of the most challenging sports to analyze due to the number of players, continuous events, low frequency of points, and highly stochastic nature - having both the notion of coordination among the members of the same team and competition across teams - makes it an especially interesting choice for research on Deep Multi-Agent Reinforcement Learning.

Cooperative Multi-Goal Multi-Stage Multi-Agent RL framework, or CM3 has been proven to have better (or comparable) results [1] in three challenging multi-goal multi-agent problems: cooperative navigation tasks in difficult formations with the multi-agent particle environment (PettingZoo MPE), negotiating multi-vehicle lane changes in the SUMO traffic simulator, and strategic cooperation in a Checkers environment.

The objective of this research is to demonstrate the power of curriculum learning - particularly the CM3 framework in the domain of football. We also look forward to seeing if coordination amongst players and competition across teams could be improved and introduced sooner in the multi-agent setting. We also tried to see if the use of the custom credit function in the later stages of the curriculum benefits (or not) the overall quality of the policy trained. This goes to imply if a finer credit assignment is indeed beneficial for a mixed football environment or not.

Lastly, we try to improve on the original CM3 framework and see if a third stage of training would benefit the overall coordination amongst the agents and would lead to a better policy or not. This stage is characterized by the reward being independent of the individual goals. We hypothesize that it could actually enhance coordination as there'd be a single common, collective goal for the team - to win the match - as opposed to the individual goals of scoring or saving.

Related work

There have been numerous attempts to develop RL agents to play football (the whole game or a part thereof, i.e. "academy scenarios"). Research and corresponding results in this area contribute to a deeper understanding of how agents can learn to collaborate effectively, and can then be applied to real-world scenarios where they could assist coaches and players in making data-driven decisions and optimizing team strategies.
We explored multiple such end-to-end football AI and found the below to be of much importance to our project.

WeKick

WeKick was developed in 2020 as a response to the Kaggle competition held by Google Research and Manchester City [5]. The framework used is fairly similar to that of the AI agent Tencent Solo which was designed for MOBA games by Deheng Ye et al [7] and was also bolstered with feature engineering, some clever reward shaping, and most importantly self-play with league training.

To enhance training efficiency, asynchronous architecture was used which also provided flexibility to modify computing resources during training. PPO is the algorithm of choice, with the neural network architecture consisting of a few Dense-256 layers, an LSTM block (32 steps, 256 hidden layer size), and a fixed LR of 0.0001 with Adam optimizer.

TiKick

Published as the first end-to-end learning-based RL system to complete a full game in Google Research Football environment, TiKick is proven to accelerate the training process of modern multi-agent algorithms and achieve state-of-the-art results in GRF academic scenarios. It is the first one to have multi-agent control - when its predecessor WeKick only had single-agent control.

The authors first generated a large replay dataset from the league training of single-agent experts. Then this single-agent dataset is used to develop a distributed learning system where the multi-agent environment is trained with offline algorithms.

Offline RL is when standard supervised learning methods are used to train RL agents from past good experiences. Algorithms like GAIL by Ho and Ermon [8] similarly used generative adversarial training to mimic a policy. TiKick is arguably the first one that aims to learn multiple cooperative RL agents from incomplete (past experience) data.

Initial setup

During the literature review/research phase, we found out that the most comprehensive environment for RL football is the Google Research Football [9][10] environment. It is a flexible and powerful open-source simulation platform designed for research on RL and MARL algorithms. It provides a physics-based simulation environment where agents can interact with a soccer game, with the aim of maximizing their scores by scoring goals or preventing the opponent team from scoring. The platform provides a customizable interface to modify game settings such as team size, field size, and other environmental factors. Besides, it also includes a variety of evaluation metrics to measure the performance of agents, such as the win rate, goals scored, and the average reward per episode.

Fig. 1: Full 11 v 11 game on the Google Research Football environment

However, the sheer bulkiness of the environment and the long list of precisely versioned dependencies makes installation difficult, and more so across different OS. Google's team has provided a Docker image to partially alleviate these issues, which did not prove satisfactory in our case. We faced a number of issues in the installation, from Docker hoarding all C drive space for its images even when the actual installation is on another drive, to *gfootball* not rendering on Windows.

Hence, after evaluating multiple MA mixed environments (Annex A) and before formulating the thesis topic, the final verdict was in favor of leveraging Unity's ML-Agents framework. The additional tools and software and learning curve required to modify the environment according to our needs far outweigh the benefits - especially the lightweight assets and rendering. More details on the environment can be found in the following sections.

Preliminaries

Unity ML-Agents Toolkit

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. The environments themselves are written in C#, and it also provides PyTorch implementations of state-of-the-art algorithms to enable easy training of intelligent agents for 2D, 3D, and VR/AR games. However, the simple-to-use Python API could also be leveraged with external libraries of RL algorithms, imitation learning, or neuroevolution - which is what we did.

Unity Editor is the recommended platform for RL tasks, however, with the Gym and PettingZoo compatible Python APIs, training is straightforward in any editor of choice using pure Python scripts. There is ample support for training single-agent, multi-agent

cooperative, and multi-agent competitive scenarios via several Deep Reinforcement Learning algorithms (PPO, SAC, MA-POCA, self-play), however, we used the implementations from Ray's *rl-lib* [11][12] as discussed in detail in the later sections.

For our use case, we've chosen the SoccerTwos multi-agent football environment.

SoccerTwos environment

This is a football environment where agents compete in a 2v2 fashion, all agents being homogenous in the context of the observation and action spaces. The aim is to get the ball in the opponent's goal while preventing it from entering its own. There are 2 multi-agent groups with 2 agents in each group.

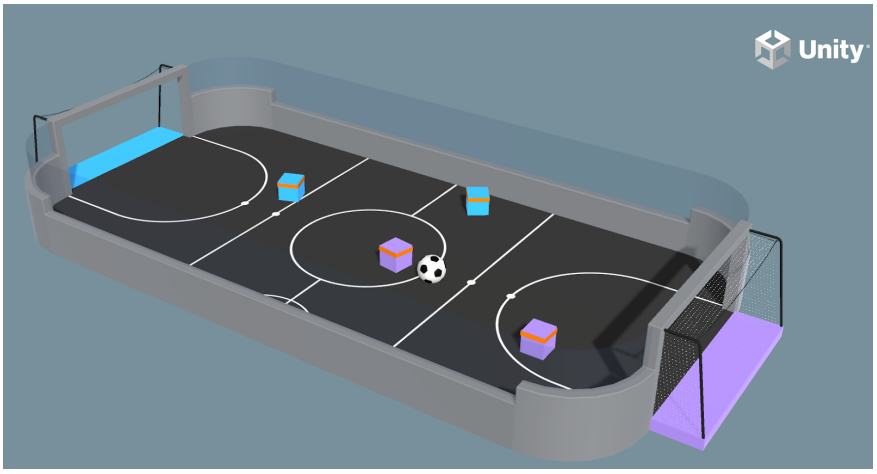


Fig. 2: Unity ML-Agents SoccerTwos 2v2 game

As is evident from the rendering of the environment above, since the agents/players are essentially blocks, there are lesser degrees of motion and physics complexity as compared to Google Research Football. This is a huge benefit for our use case where we're more limited with respect to the observation and action space complexity, and more concerned with the comparative performance of algorithms as opposed to having more realistic simulations.

Original observation space

The original observation space is a vector of size 336 corresponding to 11 ray-casts distributed forward over 120 degrees and 3 ray-casts distributed backward over 90 degrees, each detecting 6 possible object types, along with the object's distance. The forward ray casts contribute 264 state dimensions and backward 72 state dimensions over three observation stacks.

Original action space

3 discrete branched actions (each with 3 possible values 0, 1, or 2) corresponding to forward/backward, sideways movement, as well as rotation.

Original reward function

1. (1 - accumulated time penalty) When the ball enters the opponent's goal, the accumulated time penalty is incremented by (1 / MaxStep) every fixed update and is reset to 0 at the beginning of an episode.
2. -1 When the ball enters the team's own goal.

Modifications to the environment

In order to make the environment better suit the requirements for our experiments, the following changes were made to the scenes in Unity3D editor (see Annex B) and the assets exported:

1. *Common changes*
   There is an additional property added to the agents termed as “role” - with possible values as “Goalie” or “Striker”. It is assigned randomly to an agent at the beginning of the episode and is fixed for that episode. Added as a one-hot encoded value to the policy network as a “goal signal”, it conditions the policy according to the goal of the agent (refer to eq. (3)) (see Annex D for more details on goal signals). This results in the observation space of the agents to be (264, 72, 2), instead of the original (264, 72).

2. *Creating Stage 1*
   The environment for Stage 1 to contain two players (one in each team). The blue player is the agent to be trained, and the purple is a heuristic player, with straightforward +/-1 rewards. The agent is supposed to behave according to the role assigned at the beginning of the episode - i.e. either attempt to score a goal or defend as a goalkeeper.
   The *AgentSoccer.cs* script contains the said hard-coded behavior for the purple player under the overridden method *Heuristic()*.

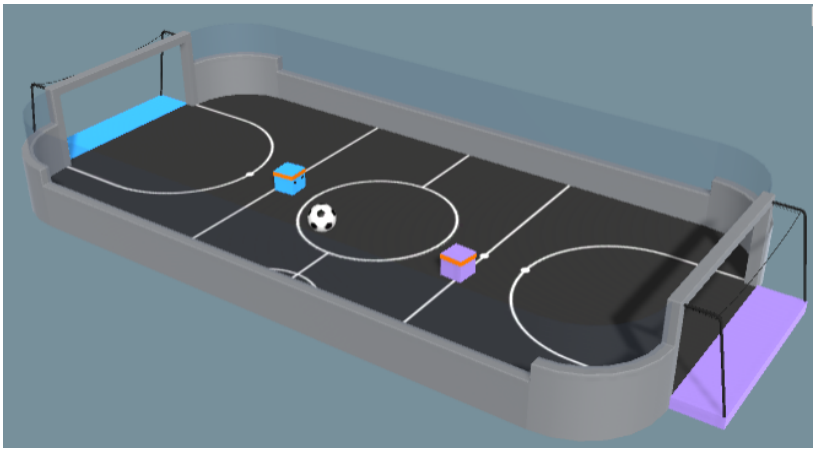


Fig. 3: Stage 1 induced single-agent environment (1v1)

3. *Creating Stage 2*
   Very similar to the original SoccerTwos environment with 2 players in each team. The roles of striker & goalie are assigned so that every team contains a role of each kind

(managed centrally via the flags in *SoccerSettings.cs*). The reward given to the agent depends on the role:

   a. In case a goal is scored, the striker agent gets a +1, the goalie gets a 0 reward.
   b. In case a goal is conceded, the striker gets a 0, the goalie gets a -1 reward.

   This enforces the goal-directed behavior in this "multi-goal" environment so that strikers are more proactive in scoring, whereas the goalkeeper is more interested in protecting their own goalpost to prevent the opponent from scoring.

4. *Creating Stage 3*
   Similar to as Stage 2, except the reward function is not dependent on the roles. Both agents of a goal-scoring team get a +1, and the opposing team gets a -1 each - which is the same as Stage 1 but in a multi-agent setting.

It is worth noting that from the environmental perspective, it is a zero-sum game, but not from a policy/training point of view. In other words, because we are always training only the blue team against the heuristic purple, so purple rewards have essentially no impact on training.

Besides these, the change common in all cases is the setting *brain* parameter for every agent/player as None instead of *Striker NN/Goalie NN/SoccerTwos NN* for that agent to be trained from scratch and not use pre-trained NN "brains" provided by Unity.

Additionally, we also used SideChannels [13] for communication between Unity and Python outside of the traditional machine learning loop. We required it outside the ML loop as we needed to track the win counts at an iteration level - like the other metrics - and not at episode level. One logical iteration can have one or more episodes. At the episodic level, the win count would always be either a draw (0-0) or a win/lose (1-0/0-1) since episodes terminate on one team scoring.

More details on the exhaustive list of changes can be found in Annex B, C, and on SideChannels in Annex E.

Ray rl-lib

Ray's *rl-lib* is an open-source library for reinforcement learning, offering support for production-level, highly distributed RL workloads while maintaining unified and simple APIs for a large variety of industry applications. Users have the option to use any deep learning framework of their choice as it supports both TensorFlow (both 1.x with static-graph and 2.x with eager mode) as well as PyTorch.

One of the core strengths of this library over others is that it supports parallelization of provided algorithms out of the box, by setting the *num_workers* config parameter, enabling the workload to run on multiple CPUs/nodes - speeding up learning.

Furthermore, *rl-lib* can auto-vectorize Gym or PettingZoo environments via the *num_envs_per_worker* config. Environment workers can then batch and thus significantly

speed up the action computing forward pass. On top of that, the *remote_worker_envs* config to create single environments (within a vectorized one) as ray Actors, further parallelizes even the env stepping process.

But to us, perhaps the most useful advantage is the library's ability to convert even custom Gym/PettingZoo environments into multi-agent ones via a few simple steps using the "multiagent" property in the config. Then, the training of agents can be either cooperative with shared or separate policies and/or value functions, adversarial scenarios using self-play and league-based training, or independent learning of neutral/co-existing agents. As will be discussed in the following sections, we experimented with the first two extensively in our work.

WandB

Experiments and model performance were tracked by Weights and Biases (WandB). In its simplest terms, it's a metric logging tool that helps track runs in a collaborative way and organize the same in a central dashboard. By keeping track of all the details of your experiments, including the code used, the hyperparameters set, and the data collected, ensures the reproducibility of experiments at a later time - which is extremely essential for scientific research.
Another very useful tool by WandB was leveraged for hyperparameter tuning, known as *Sweeps*. It works by creating a configuration space of hyperparameters to explore and then running multiple experiments with different combinations of those hyperparameters. With a few lines of code, it is possible to perform:

1. *Automated experiment management*
   Multiple experiments with different hyperparameters in parallel could be run, all managed and tracked in a single dashboard. Furthermore, if re-using a single *sweep_id*, such WandB Sweep agents could be spread across multiple nodes (even not connected otherwise), cores, or GPUs, because these combinations of parameters, or "runs" are managed centrally on the WandB server. This makes it easy to keep track of which experiments are running, which have finished, and which are performing best.

2. *Experiment insights*
   WandB Sweeps provide visualizations that make it easy to compare and analyze the results of multiple experiments. This can help gain insights into which hyperparameters are most important and which ones have the greatest impact on the results.

Experimental setup

The entirety of the work undertaken as part of this thesis has been divided into the following sections:

1. Choosing baselines

2. Implementation of CM3
3. Tweaking CM3 with respect to
    a. Underlying algorithm - A2C, A3c or PPO
    b. Extention of curriculum
4. Experimentation

Baselines

There were 3 proposed algorithms to be used as potential baselines - QMIX, MADDPG, and COMA.

OpenAI's MADDPG [14][15] is proven to perform poorly with any environment besides the particle environment on which it was proposed, hence it was dropped from the list.

COMA (Counterfactual Multi-Agent Policy Gradients) [16] could have been leveraged as a baseline to further substantiate our findings. But the lack of implementation in a library as extensive and robust as *rl-lib*, and the effort required to implement a baseline from scratch made it out of scope for this thesis.

Hence, QMIX (the implementation from *rl-lib*) was ultimately the one being chosen. It is worth noting that although the baseline algorithm code was available, making it work for our modified environment was equally challenging. A series of modifications were applied in order for our SoccerTwos environment to be suitable for QMIX:

1. QMIX requires a discrete action space - SoccerTwos has a multi-discrete one. So the environment was edited in UnityEditor to accept "flattened" actions and a subsequent mapping from MultiDiscrete(3, 3, 3) to Discrete(27) was applied.
2. QMIX requires agent groups, and hence the blue team players were grouped logically into one, and the heuristic purple payers into another. The group name then behaves as a single agent. The action and observation spaces are now a Tuple consisting of individual agent spaces repeated as many times as the number of agents in that group. More information is in Annex C.

One limitation of this implementation of QMIX is that groups can only have homogeneous agents - which in our case did not prove to be a hindrance.

CM3 implementation

For the second section, we began by implementing our own version of the CM3 framework for the chosen Unity environment. Do note that although the original paper has code associated with it [17], it could only be used as a reference because of its poor quality and strong coupling to the environments the authors used for demonstration.

Our implementation comprises three parts:

1. **Custom credit function**

Mathematically, the credit function in CM3 for an agent n taking action a^n, towards fulfilling a goal g^n, where other agent m takes action $a^m \in A^m$ is:

$$Q_n^{\pi}(s,\ a^m) :=\ E_{\pi}[\ \sum_{t=0}^{\infty} \gamma^t R_t^n \mid s_0 = s,\ a_0^m = a^m\ ] \tag{1}$$

This explicit credit assignment is backed by the notion that actions taken by agents other than n also contribute towards attaining the goal g^n, and so just learning $Q_n^{\pi}(s, \boldsymbol{a})$ is not enough.

So, for all $m, n \in [N]$, (1) satisfies the following relation of the form of the Bellman expectation equation:

$$Q_n^{\pi}(s,\ a^m)\ =\ E_{\pi}[\ R_t^n\ +\ \gamma Q_n^{\pi}(s_{t+1}, a_{t+1}^m) \mid s_t = s,\ a_t^m = a^m] \tag{2}$$

Parameterised by θ_{Q_c}, the credit function is learned by optimising the standard loss as:

$$L(\theta_{Q_c})\ =\ E_{\pi}[(R_t^n\ +\ \gamma Q_n^{\pi}(s_{t+1}, a_{t+1}^m;\ \theta_{Q_c})\ -\ Q_n^{\pi}(s_t,\ a_t^m;\ \theta_{Q_c}))^2\ ] \tag{3}$$

By letting $\boldsymbol{\pi}$ be parameterised by θ, the overall objective $J(\theta)$ is maximised by ascending the following gradient, and thus the policy is trained using:

$$\nabla_{\theta} J(\pi)\ =\ E_{\pi}[\ \sum_{m,n=1}^{N} (\nabla_{\theta} log \pi^m(a^m \mid o^m,\ g^m))A_{n,m}^{\pi}(s, \boldsymbol{a})] \tag{4}$$

Where $A_{n,m}^{\pi}$ is the advantage for agent n based on all counterfactual actions $\hat{a}^m$ for agent m:

$$A_{n,m}^{\pi}(s, \boldsymbol{a}) := Q_n^{\pi}(s, \boldsymbol{a})\ -\ \sum_{\hat{a}^m} \pi^m(\hat{a}^m \mid o^m,\ g^m) Q_n^{\pi}(s,\ \hat{a}^m) \tag{5}$$

This means that for a fixed agent m, the inner summation over n considers all agents' goals g^n and updates m's policy based on the advantage of a^m over all counterfactual actions $\hat{a}^m$, as measured by the credit function for g^n. The strength of interaction between action-goal pairs is captured by the extent to which $Q_n^{\pi}(s, \hat{a}^m)$ varies with $\hat{a}^m$, which directly impacts the magnitude of the gradient on agent m's policy. The derivations of all these equations can be found in the original CM3 paper [1].

The credit function is used as the critic within the policy gradient. In settings where an agent must learn to solve different tasks or have multiple goals that are similar in some aspects - like in this case, these goals can be either to score a (football) goal or

to save the ball by preventing the opponent to score. The agent will generalize better for the overall objective of winning the game because it will learn to reuse learnings from such different tasks.

In our case, this is implemented by passing an additional observation to the agents, called as the "goal signal".
Furthermore, the conditioning of the joint reward $R(s_t, \mathbf{a_t}, g^n)$ is done by modifying the reward function to penalize goalies more and strikers less so on opponent scoring and rewarding strikers more strongly than goalies if the team scores itself - in the second stage of the curriculum.

2. **The curriculum**
   The multi-stage curriculum is designed and trained as follows: we first trained one single A-C pair for a generic player agent against a heuristic (purple) opponent. Spawning the ball and the agent at random locations and rotations throughout the field, with random roles as the "goal", the agent just learns to score the goal in the opponent's goal post when it is assigned the role of a striker and save the ball from entering its own post when assigned the goalie role.

   Stage 2 then comprised replicating the trained policy obtained in the previous stage 2 times, and continuing training against a heuristic team (as compared to a single heuristic player in Stage 1).

   In Stage 3, we removed the dependence of the reward function on the goal, keeping the policy network intact (i.e. still goal-dependent) and continued training.

3. **Function augmentation**
   Bridging the 2 stages (as described in the original paper) is not needed in our case because the observation of the agent in our environment is purely egocentric and the only way it "detects" other agents is if its own ray casts hit other agents. In that case, only the "*tag*" property of the object it hits distinguishes if it is another agent, a wall, a ball, or any other object in the environment.
   The additional OHE goal signal as observation is the same in both stages. In short, the number of parameters/size of the observation space vector is not changing.

Impact of algorithm selection across curriculum stages

The original CM3 paper stated that it used actor-critic but did not mention which algorithm precisely. So for our CM3 implementation, we had 3 choices of algorithms: A2c, A3C, and PPO.
Given the current state-of-the-art baseline on Google Research Football (TiKick and WeKick) used PPO and it has also been proven to be surprisingly effective in cooperative environments [18], we experimented with A2C, A3C, and PPO (with best hyperparams obtained from the corresponding sweeps) in the full 2v2 environment (essentially Stage 2 directly) for 500k environment steps - this was repeated 10 times and the metrics were averaged over to minimize the potential impact of random seed variations. The reasoning

behind this comparison was to try to leverage the strengths of each of these algorithms in the hopes of obtaining the best results in the final version of CM3.

Impact of extending the curriculum

The original CM3 paper had 2 stages of the curriculum. We propose a third one where we remove the dependency of the rewards on the goal signal but keep the policy network intact (i.e. still impacted by the goal of the agent). We hypothesize that the collective objective of all the agents in a team would encourage better coordination and hence there would be a wider score difference characterized by a better win rate.

Experimentation methodology

To determine good hyperparameter configuration for the curriculum stages, a broad sweep was first run for all 3 algorithms - QMIX, A2C, and PPO - with around 25-30 configs/runs in each case with varying values for learning rate, neural network architecture (layers and number of units in each layer), gamma (discount factor), lambda (GAE parameter used to reduce variance in training), activation function, using LSTMs or not, and sharing layers for value function or not. The sweep runs were trained for 1.5mn steps (~200 iterations), and checkpointing after every 10 iterations.

"Good" runs were determined first by considering higher *episode_reward_mean*, then by *low vf_loss,* and then by lesser *policy_entropy* (so as to have a more stable configuration).

Once good hyperparameters were finalized, we started with the baseline training.

1. QMIX
   The full multi-agent environment (2v2) was trained with QMIX against heuristic, for 3000 iterations (~18mn environment steps)

2. MAPPO
   We implemented MAPPO using *rl-lib*'s PPO with the multi-agent setting for the 2 players (centralized training, decentralized execution). In essence, a single "agent" takes in the observations for both the players during training and learns a policy that implicitly encodes how all agents should cooperate and coordinate their actions. This learned policy is then used during execution by each agent in a decentralized manner, making decisions based on its local observations and the learned coordination.
   Rl-lib simplified this greatly as we only needed to provide the observation and action spaces for the individual players in the "multiagent" section of the config and it took care of the rest depending on the number of players the environment has.

   In a similar setting, MAPPO training ran for 3000 iterations too.

After the baselines, we ran the algorithm comparison scripts for A2c, A3C, and PPO for 500k environment steps on the multi-agent scenario directly to determine which algorithm performed the best comparatively. The criteria were the same as previously - using higher *mean_episodic_reward*, then lower *vf_loss,* and then lower *policy_entropy*.

Finally, once we obtained the best underlying algorithm for CM3, the training began by following the multi-stage curriculum as below:

1. Stage 1 (induced single-agent environment) was trained for 1000 iterations (~4mn environment steps) and checkpointed at every 10 iterations.
2. Stage 2 began by restoring the weights obtained after Stage 1 - replicating the said policy weights to both the blue team players, & continued training in the multi-agent setting for another 2000 iterations.
   This marked the end of training for the original CM3 framework.
3. Stage 3 restored the weights from the 1000th iteration in Stage 2 (i.e. when the agent has already been trained for a total of 2000 iterations) and then continued the multi-agent training for another 1k iterations. The key difference here is the reward function not being goal-dependent.

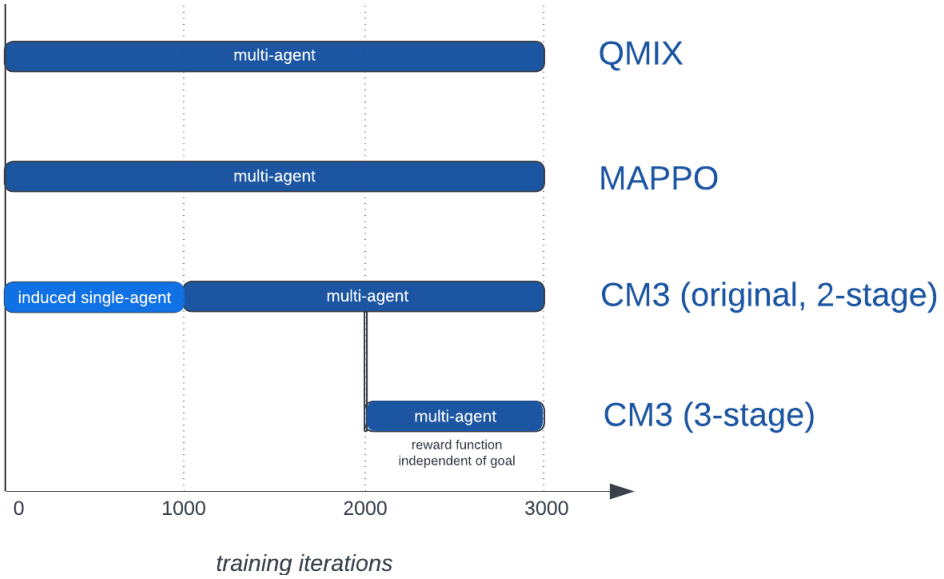


Fig. 4: Training recipe for baselines and CM3

After the training completions, all 4 versions were made to compete head-on with each other for 1000 episodes. The win counts were collected for each game and converted to win rates to determine how they compared.

Results

Sweeps for hyperparameter tuning

A broad sweep was run for PPO and A2C with the following sweep settings that allowed us to (randomly) search for good hyperparameters for our environment for subsequent experiments:

```
method: random
parameters:
```

```
_lambda:
  distribution: uniform
  max: 0.99
  min: 0.8
fcnet_activation:
  values:
    - tanh
    - relu
fcnets:
  values:
    - - 256
      - 256
    - - 512
      - 512
    - - 256
      - 512
    - - 512
      - 256
    - - 256
      - 256
      - 256
gamma:
  distribution: uniform
  max: 0.99
  min: 0.8
lr:
  distribution: log_uniform_values
  max: 0.1
  min: 1e-05
use_lstm:
  values:
    - true
    - false
vf_share_layers:
  values:
    - true
    - false
```

Of the resulting 22-25 runs for each algorithm, the top 5 performing runs for PPO (considering high *episode_reward_mean*, low *vf_loss* and low *policy_entropy*) as an example, are shown below:

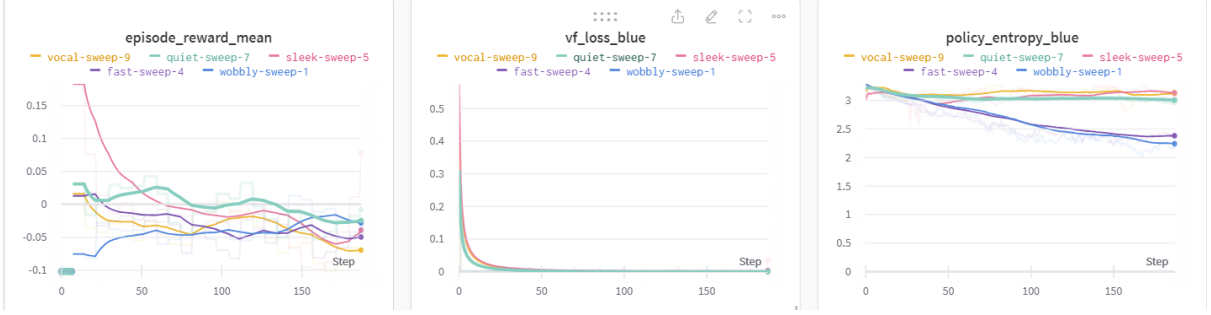


Fig. 5: Fig. Top 5 runs for PPO sweep

The final neural network configurations chosen for A2C and PPO can be found in Annex F and G.

Choice of algorithm: A2C vs A3C vs PPO

The algorithm comparison on the 2v2 SoccerTwos version resulted in the graph below:

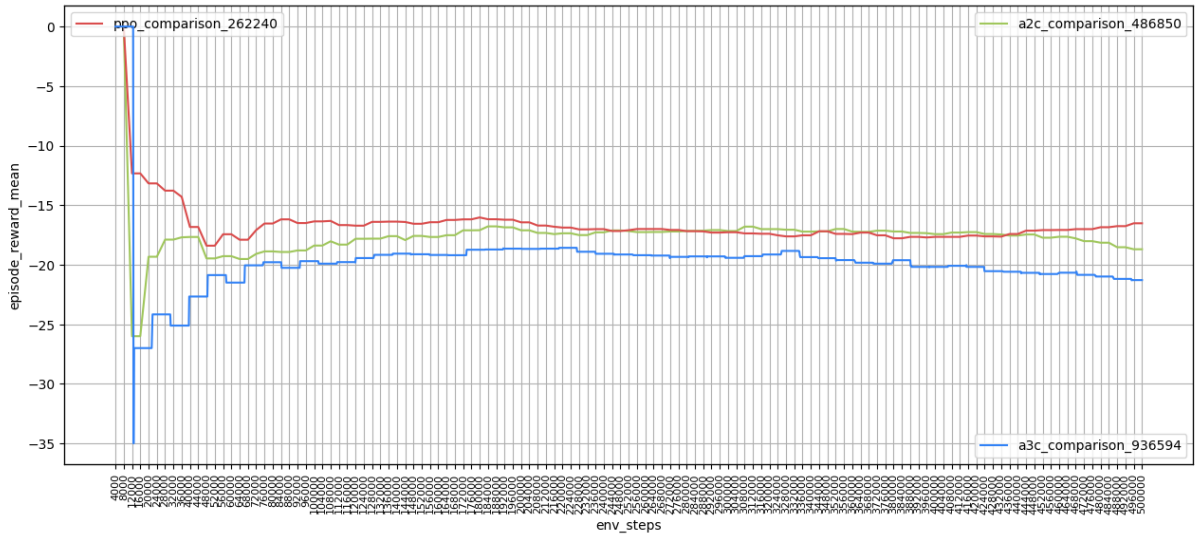


Fig. 6: mean_episodic_reward for A2c, A3c, and PPO on SoccerTwos 2v2 version

A3C was clearly a bad choice. Between A2C and PPO with an almost similar performance in the middle, the graph for PPO started having an upward curve later. Since the actual training would be far greater than 500k steps, it tipped our choice more towards PPO.

Further comparison of A2C and PPO with respect to value function loss and entropy yielded the below results - which are in favor of PPO as well:

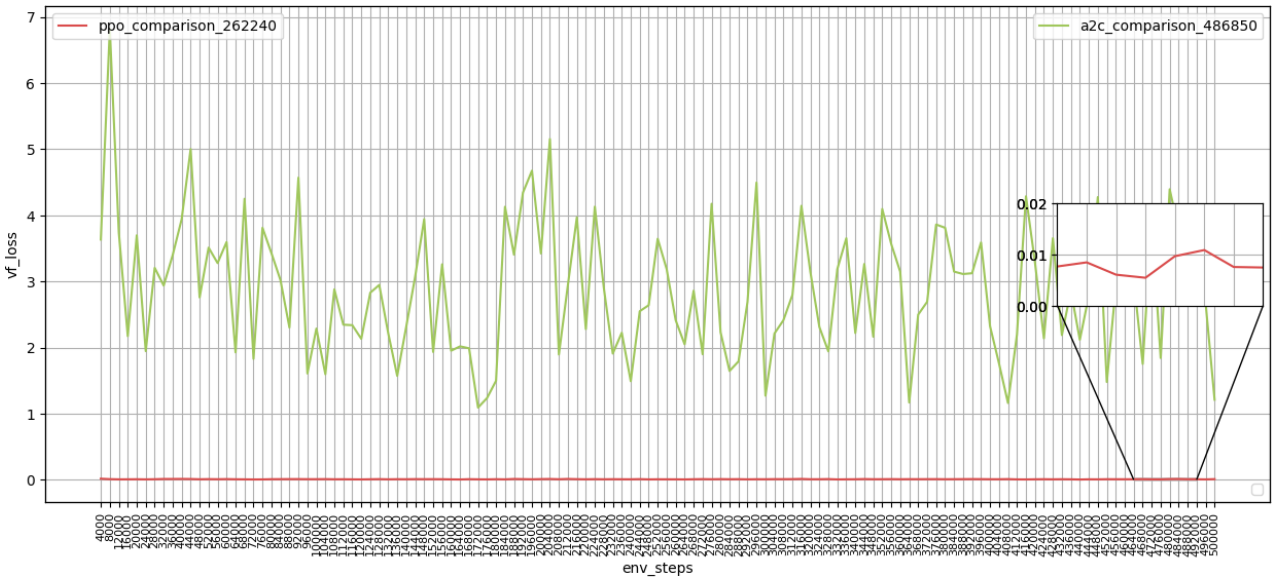


Fig. 7: vf_loss for A2c and PPO on SoccerTwos 2v2 version

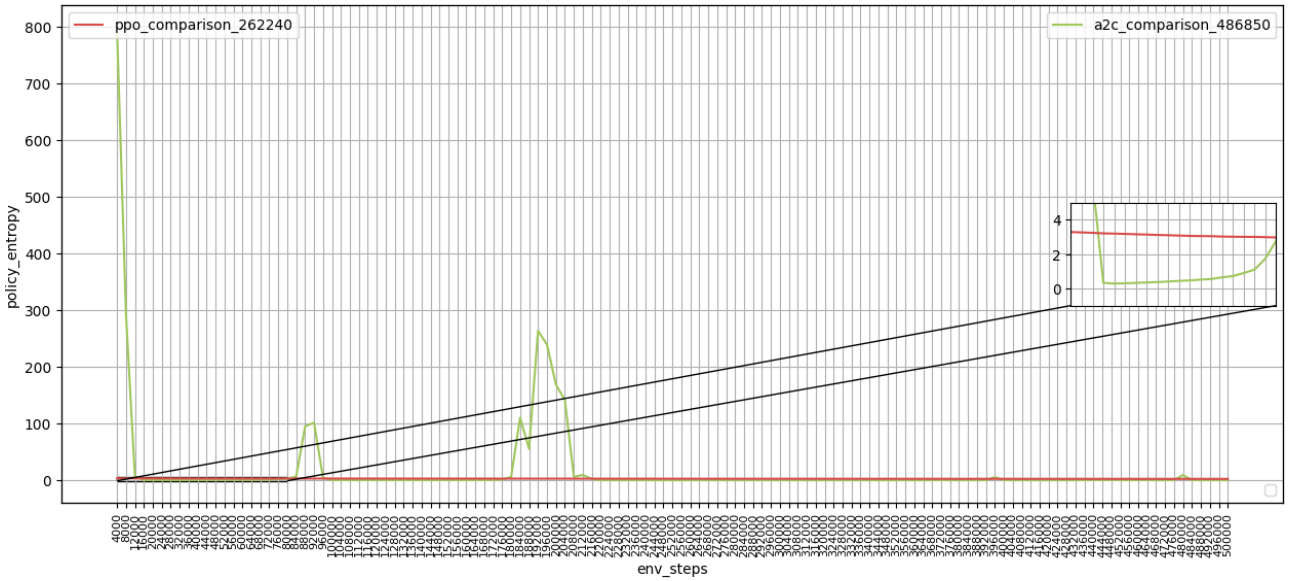


Fig. 8: policy_entropy for A2c and PPO on SoccerTwos 2v2 version

Additional reasons also include:

1. It has been proved that A2C can be considered as a special case of PPO [19]. If this is the case, we would want the algorithm of our choice to be applicable more broadly.
2. PPO is more sample efficient than A2C, so our relatively short training (due to limited hardware resources) would be better suitable for the former.

Thus, of the available actor-critic algorithms, PPO was determined to be “good enough” for the Unity SoccerTwos environment.

CM3 vs baselines

Based on the *episode_reward_mean* graph as shown below, it is evident that both versions of CM3 outperform the two baselines in terms of a higher average reward. The sudden changes at iteration 1k and 2k is due to the change in stage.

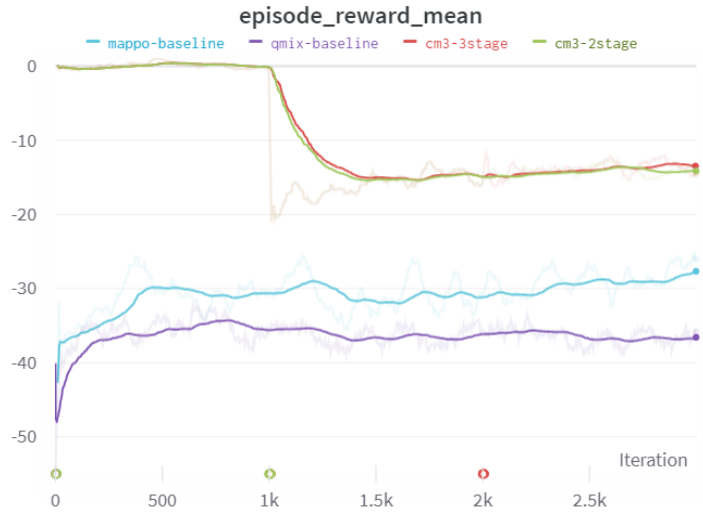


Fig. 9: Mean episodic reward during training of CM3 and baselines across 3k iterations (smoothing 0.95 applied)

As for the variants themselves, the 3-stage version of CM3 is comparable to the original 2-stage version, with only slightly higher mean rewards.

Regarding the other metrics of comparison, the value function loss is lower too in the case of the CM3 variants as compared to the baseline, however, the policy exhibits slightly higher entropy. Please note that *rl-lib*'s implementation of QMIX does not return the below metrics and hence has not been captured for comparison.

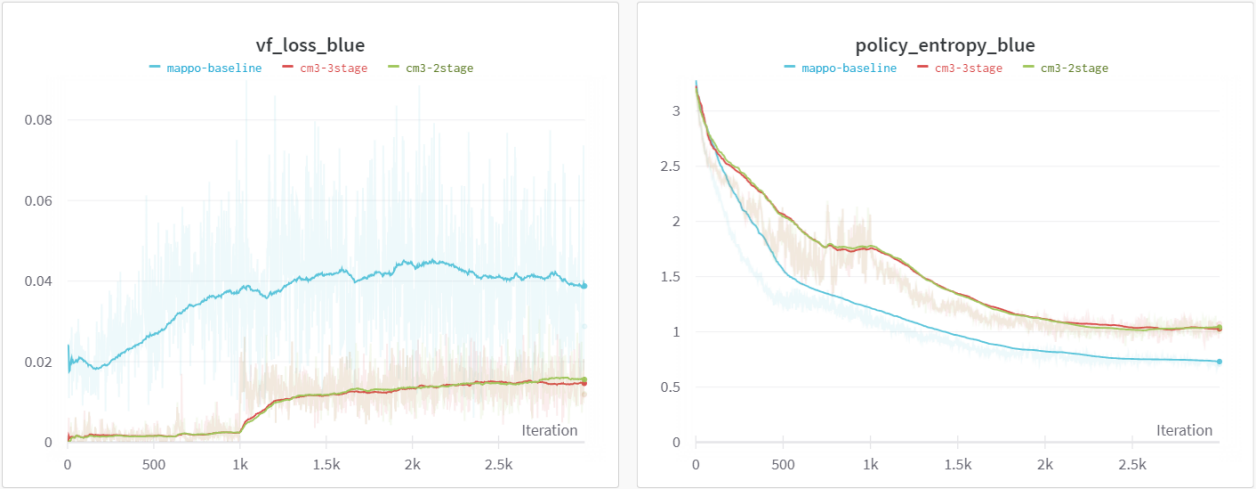


Fig. 10: Value function loss and policy entropy during training of CM3 and MAPPO baseline across 3k iterations

For a stage-wise comparison, the below graphs signify that although Stage 1 had an overall higher mean episodic reward, it could be attributed to episodes when the training was against a relatively passive heuristic goalie (as compared to a very strong heuristic striker). Furthermore, the fluctuation of episodic reward around 0 signifies that the Stage 1 training could've been a little longer or tweaked for the agent to have *actually* learned something meaningful.

Fig. 11: Training metrics of CM3 across the 3 stages

Our extension in Stage 3 showed a relatively more stable training based on the lower entropy of policy (continuing from Stage 2) and slightly higher episodic rewards - however not enough to justify a better performance just yet.

Win counts

Win counts are tracked at an iteration level using Side Channels, and they're written first to a file against every worker's UUID. This is done for 2 reasons:

1. To not call the synchronous WandB logging API multiple times, from every *rl-lib* worker as that'd significantly slow down the training, and
2. Counts need to be consolidated across workers.

At the end of the training, a win count logger script consolidates the counts and logs one metric per iteration to our WandB project.

To determine how the training progresses, we collected the win counts of the team *while being trained* against the heuristic. It can be deduced that a steeper graph signifies more wins - better and faster training. Thus, in the beginning, as compared to the baseline MAPPO and QMIX, the induced single-agent setting (Stage 1) performs worse. However, the later stages have a steeper curve denoting better training.

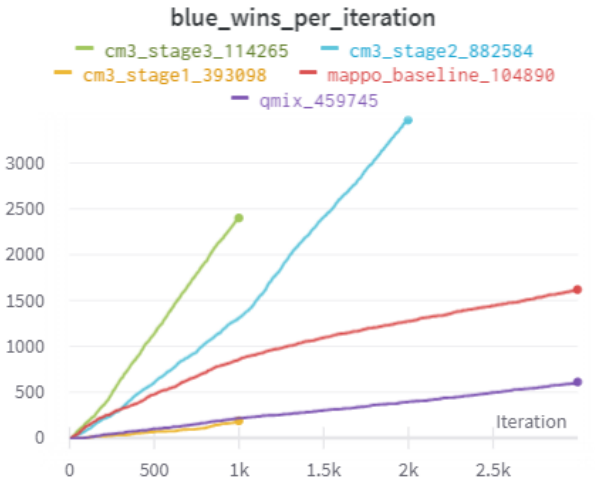


Fig. 12: Cumulative blue team win counts (absolute) during training of CM3 and baselines across 3k iterations

However, absolute win counts per iteration are still not the best way to make the comparison. A better way is to have the win rate which would also account for the total number of games being played per iteration. Thus, converting the win counts to a win rate results in the below graph - making it clear and signifying better training when the reward function is made agnostic of the role the agent is assigned (a goal-independent 3rd stage). This proves that our extension to the original CM3 framework indeed results in a better policy.

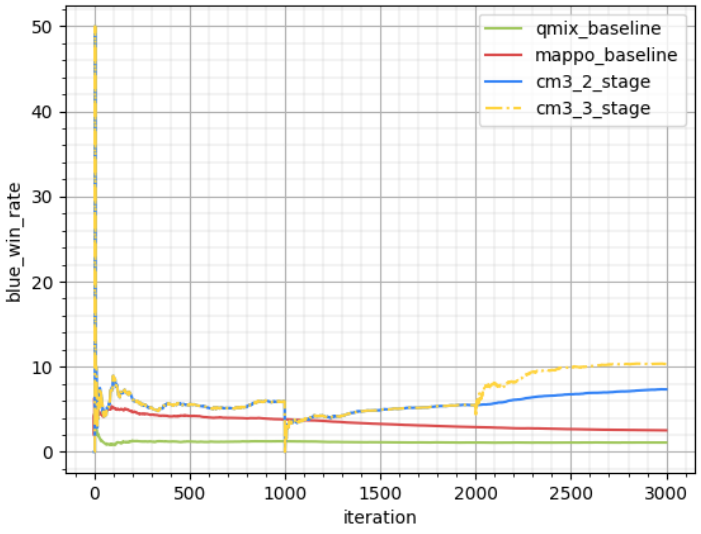


Fig. 13: Win rate during training of CM3 and baselines across 3k iterations

After the completion of training, we also pitched the different versions against each other to see how they compare with a head-on competition. Each pair of teams were made to play for 1000 episodes and the number of games won by the blue team (the first in comparison) divided by the total games completed was used to calculate the winning rate. All variants performed far worse with the heuristic as the training was fairly short to beat the powerful heuristic team.

Blue team vs Purple team	Win rate (blue wins/total games*)
QMIX baseline vs Heuristic	*4.42%*
MAPPO baseline vs Heuristic	*1.69%*
CM3 2-stage vs Heuristic	8.93%
CM3 3-stage vs Heuristic	9.56%
CM3 2-stage vs MAPPO baseline	*77.64%*
CM3 3-stage vs MAPPO baseline	*81.00%*
CM3 3-stage vs CM3 2-stage	53.96%

** draws are not counted*

Table 1: Win rates for head-on competitions post-training (averaged over 1000 episodes)

Consolidating the above data, it is enough to prove our 2 hypotheses:

1. CM3 (both variants) performing better when compared against the heuristic individually and against the baseline directly.
2. A 3-stage CM3 (the enhancement we proposed) results in a policy that is capable of winning more games than the original 2-stage CM3.

It is worth noting that although helpful, the combinations of QMIX baseline vs any of the PPO variants (MAPPO baseline or CM3) could not be possible because of the way the algorithm is implemented in *rl-lib*, which requires a grouped environment. It is not possible to have one team of agents grouped and the other as individual agents with their own policy networks in Unity ML-Agents.

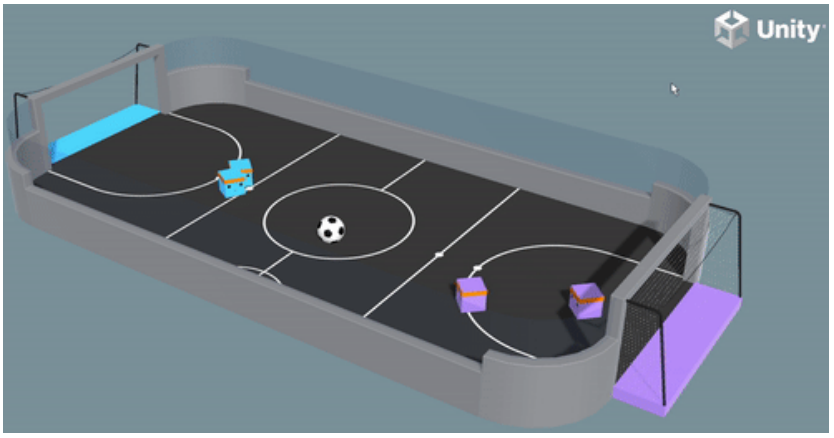
Visual interpretation

Metrics like episodic/policy rewards or win rate are good for quantifying agent learning on paper, although to see the emerging behavior and determine the development of specialized roles, it is important to analyze the results visually.

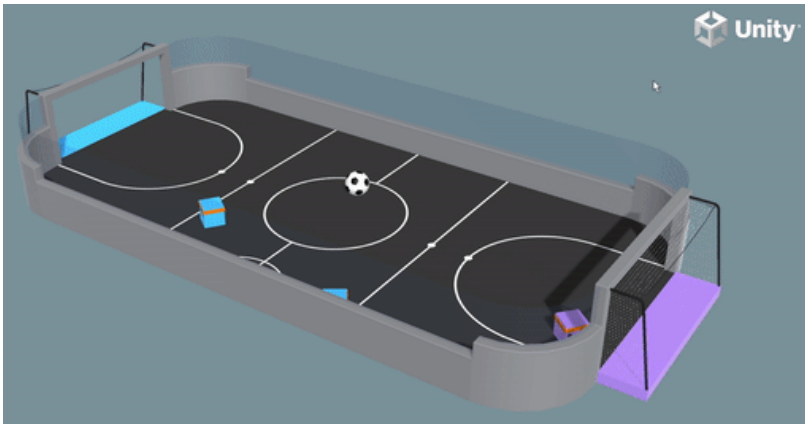
Do note that the below captures as gifs may become static images when the report is viewed in .pdf format. Please refer to the evidences directory on GitHub at https://github.com/shivanip14/unity-football-cm3/tree/master/captures.

We pitched the following combinations of teams against each other with the visuals ON and found the following results:

1. **3-stage CM3 blue vs MAPPO baseline purple**
    a. *Goal scoring*
    The 3-stage CM3 version clearly scored a lot more times than the baseline, whose goals were few and far between:

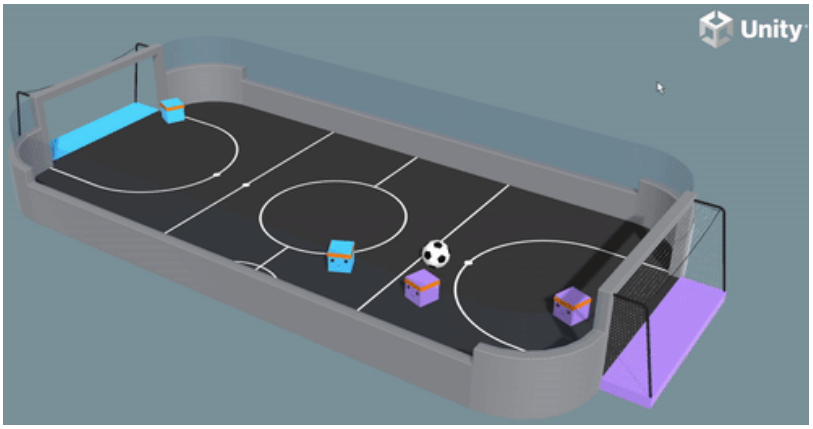


b. *Role differentiation*
We were able to witness a clear designation of roles in the blue team (3-stage CM3), where one player (the so-called "striker") takes the initiative to attempt scoring goals and being on the opponent's front more often, as compared to the other "goalie" which appears to be playing in a more defensive position:

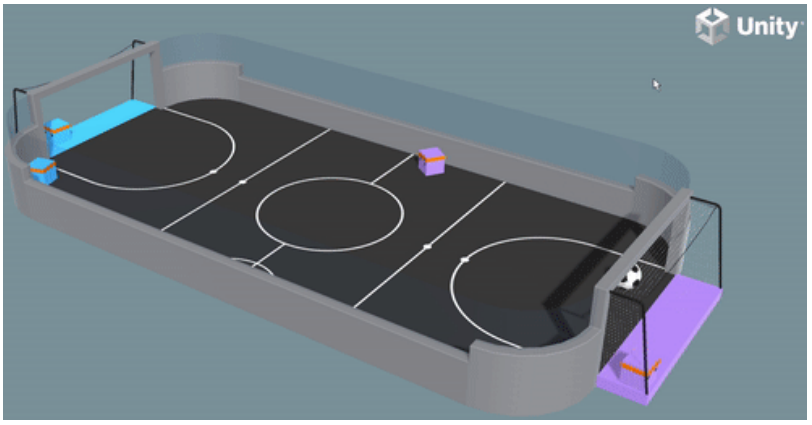


Whereas in case of the purple team backed by MAPPO, both the players are somewhat exhibiting similar behaviors. They also seem to perform poorly visually - as was backed by the win rate data obtained earlier.

c. *Intra-team coordination*
Agents trained by CM3 showed coordination by passing the ball to their teammate in an attempt to score. Even though the training was fairly short, we were able to make the agents exhibit this collaborative behavior in some episodes:

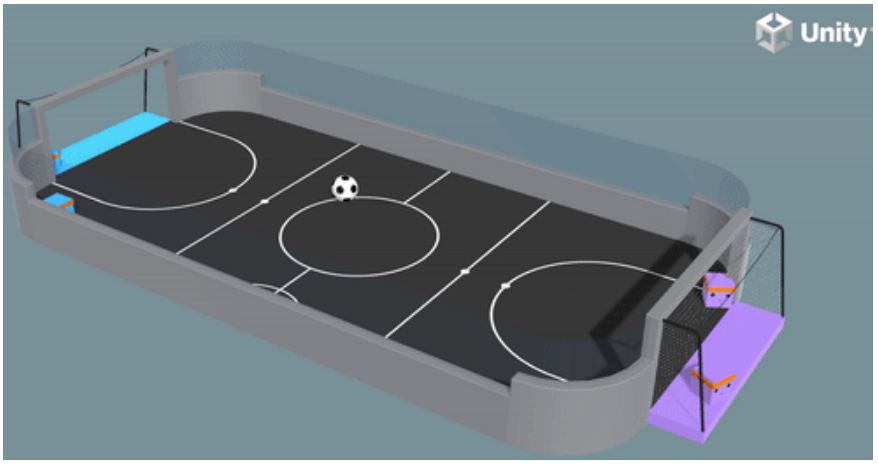


d. *Unusual behavior from both sides*
While the 3-stage CM3 was able to perform much better than its baseline MAPPO rival, there were times when both the teams seemed to be stuck and unable to recognize where the ball was or display any logical behavior:



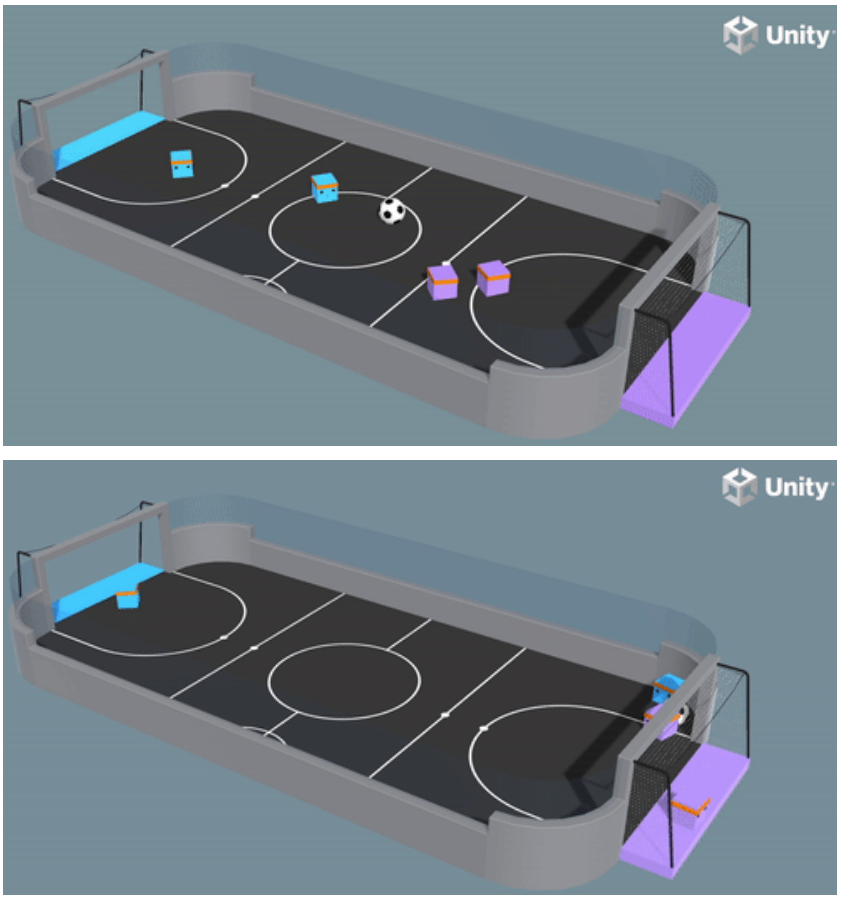
2. 3-stage CM3 blue vs 2-stage CM3 purple

a. *Comparable goal scoring*
Visually, both the teams seemed to be equal in making attempts to score goals, unlike the previous competition against MAPPO baseline that seemed to be a one-sided game:

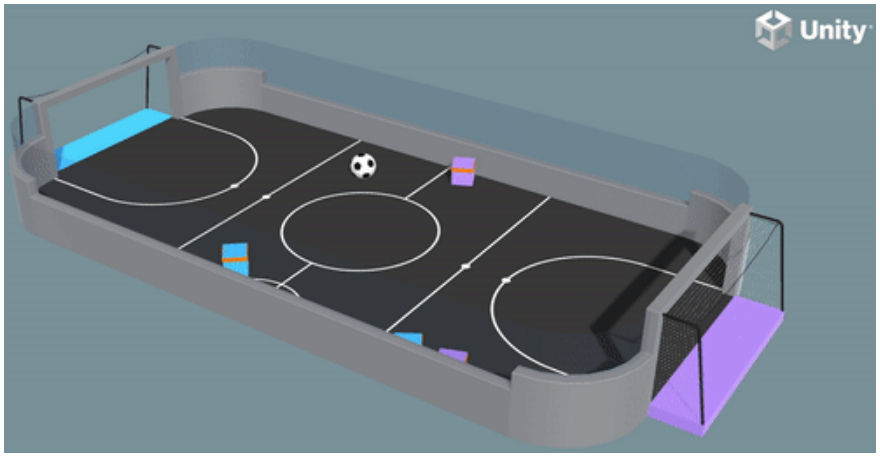


This corroborates our mean episodic reward results which also denoted an almost equal performance.

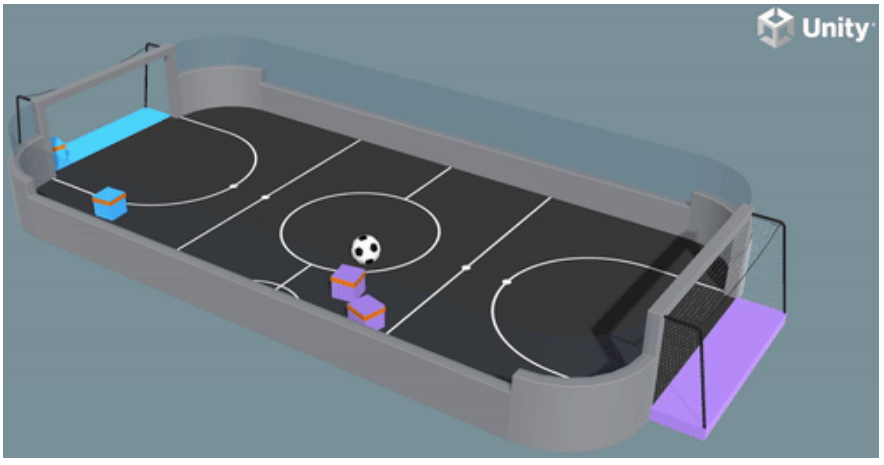
b. *Defensive capabilities*
Since both teams were making attempts to score, it also highlighted the other team's defensive capabilities when one team tried to win the ball from the other:

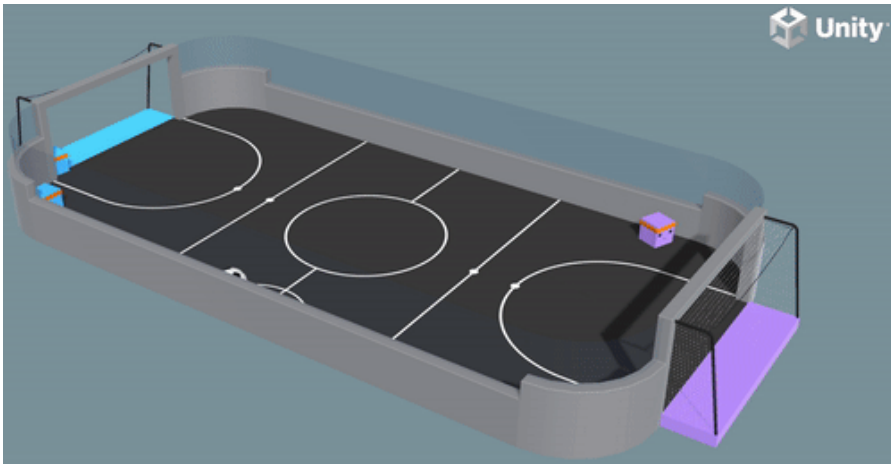


c. *Taking defensive positions*
We saw the emerging behavior in both the teams that made both players in one team (blue) take defensive positions around its goal post when the ball came near it, and clearly distinguishable roles in the other (purple) team which made one of its players take a defensive position and one attacking.

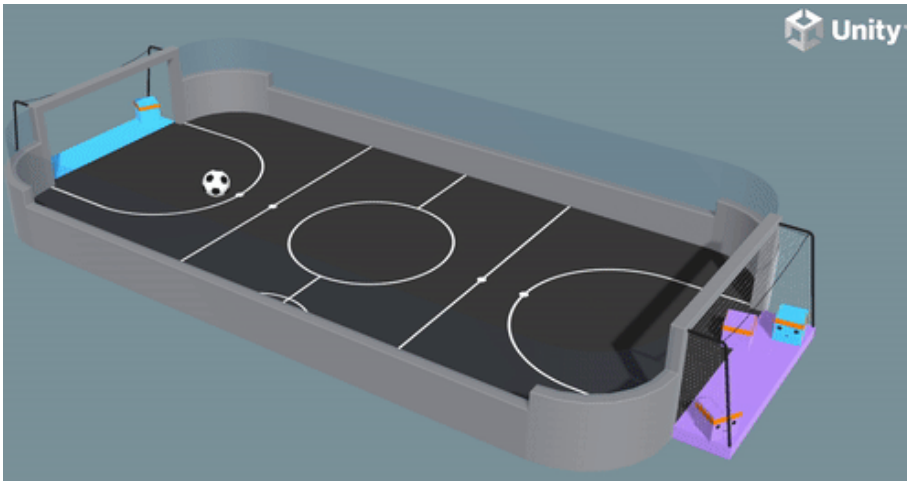


d. *Using walls*
Particularly in the purple team trained by the 2-stage CM3, we witnessed the agents using walls to deflect the ball in the desired direction. This behavior was not very evident in the team backed by 3-stage CM3.





e. *Unusual behavior*
Like in the previous play, there were times when both the teams seemed stuck and not being able to find the ball or take any logical action till the end of the episode. This could possibly be attributed to a lack of sufficient training.



Again, as with obtaining the win rates, a CM3/MAPPO baseline team could not be pitched against QMIX because of the changes required in the environment (pertaining to agent grouping) for the latter due to its implementation.

Conclusions

With the results obtained, the following conclusions can be made:

1. A well-crafted curriculum can make an agent achieve an otherwise difficult and complex task - such as winning a football game - much more easily and converge to a better policy *faster*. This was proved by both the CM3 variants performing better than the QMIX and MAPPO baselines, in terms of a higher mean episodic reward, lower value function loss, and higher win rate against both the heuristic and the baseline.
2. Extension of curriculum results in an even better policy (determined by a higher win rate) as the additional stage of curriculum forces the training in the direction aimed towards a collective, team-oriented goal, rather than individual goals.
3. The enhancement as such did not result in a significant increase in mean episodic rewards, and hence the policy improvement could not be quantified by the usual RL metrics.
4. A visual comparison of the play between CM3 variants and baselines demonstrated that the teams trained using CM3 were able to segregate the roles and hence showed attacking and defensive behavior, leading to it scoring more and conceding fewer goals than the baselines. There was also some emergent intelligent behavior arising out of the CM3 training which was evidently absent from baselines - in which players used walls to deflect and redirect the ball.
5. The visual comparison also highlighted the limitations of the project and insufficient training, when there often were episodes in which none of the agents showed any definitive “football-like” play and the episodes ended after reaching the maximum environment steps limit.

Limitations and future work

While this research project has provided valuable insights into the impact of curriculum learning in the complex mixed environment of football, it is important to acknowledge several limitations that may have affected the scope and generalizability of the findings:

1. The analysis performed to choose the underlying algorithm for CM3 has been very limited. As mentioned earlier, the original paper had no information pertaining to the exact algorithm and the hyperparameters. The initial sweeps and comparison of the three "actor-critic" methods (PPO, A2c & A3C) provided some guidance, however, it can be argued that a more in-depth analysis could've certainly been helpful, particularly so to exaggerate the difference in training metrics between the 2 variants of CM3 (which in our case has not been very evident).
2. The checkpoints we chose after the end of Stage 1 and Stage 2 of CM3 training were after a fixed number of iterations, for the sake of simplicity in dividing 3k iterations across 3 stages. A better approach would have been to choose a better model using the training metrics graph.

Further extension of this project could be aimed to include the below enhancements:

1. The cooperation amongst the agents of the same team could be quantified further by measuring metrics such as the number of passes made, or agents positioning themselves strategically that leads to a goal or a save.
2. Agents/players per team could be increased to a larger number as compared to 2v2, to see evolving intra-team strategies.
3. Heuristic agents' logic could be made more smart and realistic - presently, the striker agent is extremely difficult to train against, whereas the goalie agent is equally passive. This results in a training plateau. An ideal scenario would be to have the heuristic logic somewhere between an aggressive striker who is *also* able to save goals, and whose difficulty (read *aggressiveness to score goals*) gradually increases.
4. Given suitable computing resources, the project could be made more comprehensive when paired with Google Research Football. We believe it can perform comparable to the current state-of-the-art football AI TiKick.

Challenges

Throughout the course of this project, several challenges were encountered that impacted the progress and required careful consideration and adaptation. These challenges encompassed various aspects of the research process and necessitated proactive problem-solving strategies to mitigate their effects. The major roadblocks faced are summarized as follows:

1. *Incompatible and unsupported environments*
   Google Research Football was initially the designated environment of choice since it's a very powerful and flexible open-source simulation platform, especially for Deep MARL research. However, the sheer bulkiness of the environment and the long-list of precisely versioned dependencies makes installation difficult, and more so across different OS. Particularly for a Windows + Nvidia M-series combination, the support is very limited, and rendering issues, memory saturation, and crashes are frequent.

Hence, the alternative was to adapt the entire project to another similar environment which was lighter but still customizable - Unity ML–Agents' Soccer environment. The flip side to this was limited knowledge and forum support on the said relatively newer platform.

Secondly, as the implementation progressed, we encountered instances where the errors faced with the environment and the chosen algorithms presented nuances that were not accounted for in the initial analysis plan. This required reevaluation of both the environment design (state, observation and action spaces, rewards) and the chosen baseline algorithms. Particularly, the QMIX implementation from Ray's *rl-lib* inherently did not support vectorized action spaces and hence the original environment had to be modified to use a similar flattenned action space.

2. *Limited implementations of research papers*
   During the literature review phase of the project, numerous recent research papers were identified to be of importance which spanned across a broad array of research topics in Deep Multi-Agent Reinforcement Learning. Many of them did not have the supporting implementation to reproduce the results. Of the ones that did, it was tightly coupled with the environment in which the original research was carried out. Thus, the lack of generic algorithm implementations prevented them from being used even as benchmarks/baselines.

3. *Lack of computing resources by host institute*
   The project involved running a GPU-intensive simulation environment such as Google Research Football right from the analysis phase. The lack of one provisioned by Georgia Tech meant the work was undertaken in the local system, which significantly limited the pace at which the code and its results could be analyzed. For training purposes, however, access to the mentor's personal computing systems was provided as a workaround which slightly alleviated the challenge.

References

[1] CM3: Cooperative Multi-goal Multi-stage Multi-agent Reinforcement Learning by Jiachen Yang et al (Jan 2020); https://arxiv.org/abs/1809.05188

[2] Train a Reinforcement Learning Agent to play Football by Felix Yu (Dec 2020); https://flyyufelix.github.io/2020/12/02/google-football-rl.html

[3] How does AI play football? An analysis of RL and real-world football strategies by Atom Scott et al (Nov 2021); https://arxiv.org/abs/2111.12340

[4] TiKick: Towards Playing Multi-agent Football Full Games from Single-agent Demonstrations by Shiyu Huang et al (Nov 2021); https://arxiv.org/abs/2110.04507

[5] WeKick: 1st Place Solution for Google Research Football with Manchester City F.C. by Kaggle (Dec 2020); https://www.kaggle.com/c/google-football/discussion/202232

[6] Unity ML-Agents by Unity Technologies (2023); https://github.com/Unity-Technologies/ml-agents

[7] Mastering Complex Control in MOBA Games with Deep Reinforcement Learning by Deheng Ye et al (Dec 2020); https://arxiv.org/abs/1912.09729

[8] Generative Adversarial Imitation Learning by Jonathon Ho and Stefano Ermon (June 2016); https://arxiv.org/abs/1606.03476

[9] Google Research Football: A Novel Reinforcement Learning Environment by krol Kurach et al (Apr 2020); https://arxiv.org/abs/1907.11180

[10] Implementation of experiments and source code for Google Research Football environment by Karol Kurach et al (Apr 2022); https://github.com/google-research/football

[11] rl-lib Documentation by the Ray Team (2023); https://docs.ray.io/en/latest/rllib/index.html

[12] Reinforcement Learning with rl-lib in the Unity Game Engine by Sven Mika (Jan 2021); https://medium.com/distributed-computing-with-ray/reinforcement-learning-with-rllib-in-the-unity-game-engine-1a98080a7c0d

[13] ML Agents Class SideChannel documentation by Unity Technologies (2022); https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/Unity.MLAgents.SideChannels.SideChannel.html

[14] Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments by Ryan Lowe et al (Mar 2020); https://browse.arxiv.org/pdf/1706.02275.pdf

[15] rl-lib 2.2.0 available algorithms - MADDPG by the Ray Team (2023); https://docs.ray.io/en/releases-2.2.0/rllib/rllib-algorithms.html#maddpg

[16] Counterfactual Multi-Agent Policy Gradients by Jakob Foerster et al (Dec 2017); https://arxiv.org/abs/1705.08926

[17] Implementation of experiments for Cooperative Multi-goal Multi-stage Multi-agent Reinforcement Learning by Jiachen Yang (June 2022); https://github.com/011235813/cm3

[18] The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games by Chao Yu et al (Mar 2021); https://arxiv.org/abs/2103.01955

[19] A2C is a special case of PPO by S. Huang et al (May 2022); https://arxiv.org/abs/2205.09123

Annexes

Annex A - Alternative multi-agent environments explored

During the literature review phase of the project, we explored multiple multi-agent environments that could potentially replace Google Research Football. Below are the notes that helped us conclude with Unity SoccerTwos as the environment of choice:

Environment	**Part of/runs as**	**Pros**	**Cons**
SoccerTwos	Unity	+ Similar to gfootball + Can be bundled together with #2 + Manual control possible	- Extra setup & asset exports (can be pre-packaged though)
StrikerVsGoalie	Unity	+ Similar to gfootball + Can be bundled together with #1 as academy scenarios + Manual control possible	- Extra setup & asset exports (can be pre-packaged though)
Slime Volley Gym	Standalone	+ Very light weight	- May have to write a wrapper/edit env for multiple players in one team (currently 1v1)
SimpleTag	PettingZoo - MPE	+ Very light weight + Easily customizable	- A bit too simple - No manual control
Simple World Comm	PettingZoo - MPE	+ Very light weight + Easily customizable + "leadadversary", "forests", "food" help in complex collaborative strategies	- No manual control
Knights Archers Zombies	PettingZoo - Butterfly	+ Very light weight	- Only collaborative out-of-box, may require tweaks to make the zombies playable/learnable - extra work

Derk's Gym	Chromium/Web GL2	+ Inbuilt parallelisms ("arenas") + Docker + Colab version available	- Compute heavy - No headless mode possible

Annex B - Modifying the environment

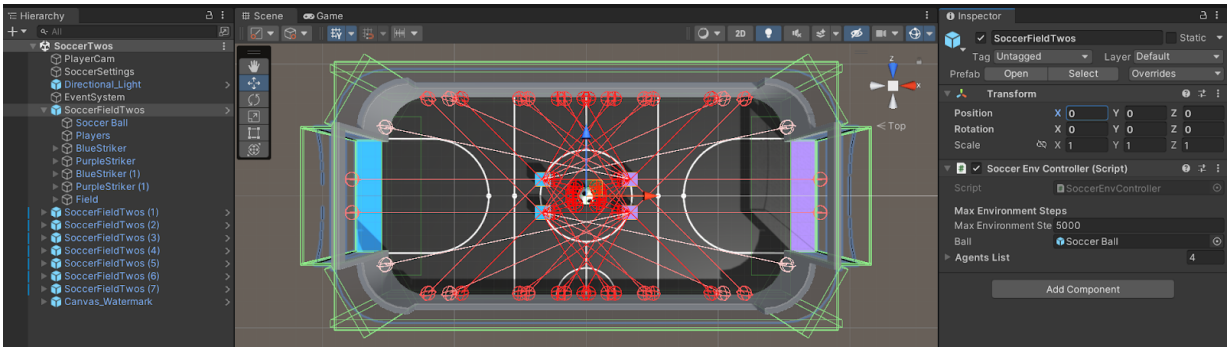


Fig A.1 Unmodified SoccerTwos environment at field-level (replicated 8x)

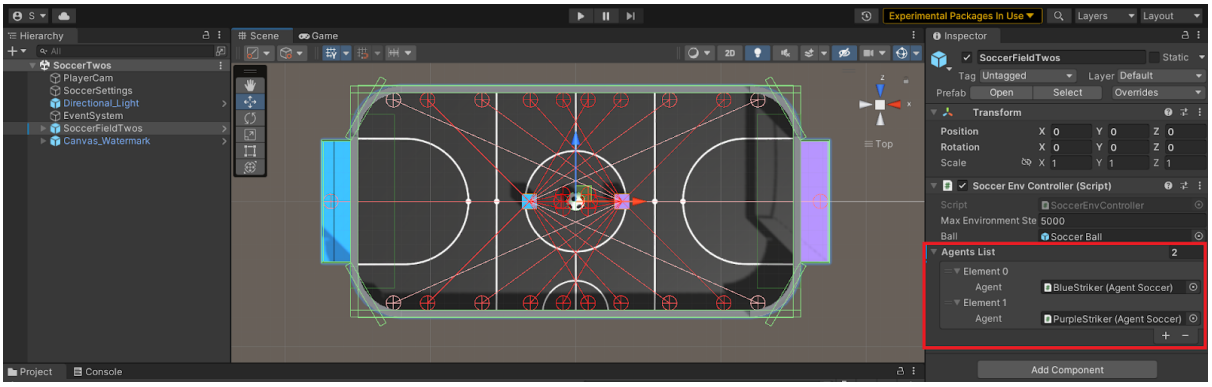


Fig A.2 Stage 1 environment field-level changes

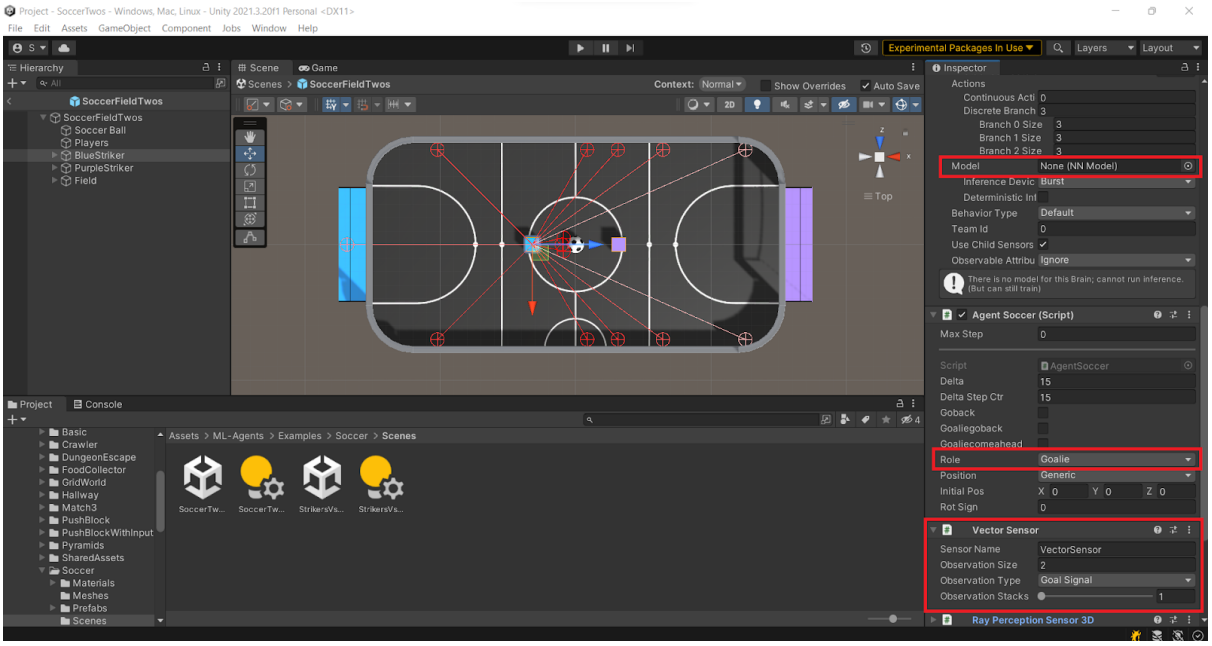


Fig A.3 Behavior parameters and properties of trainable blue player in Stage 1 environment

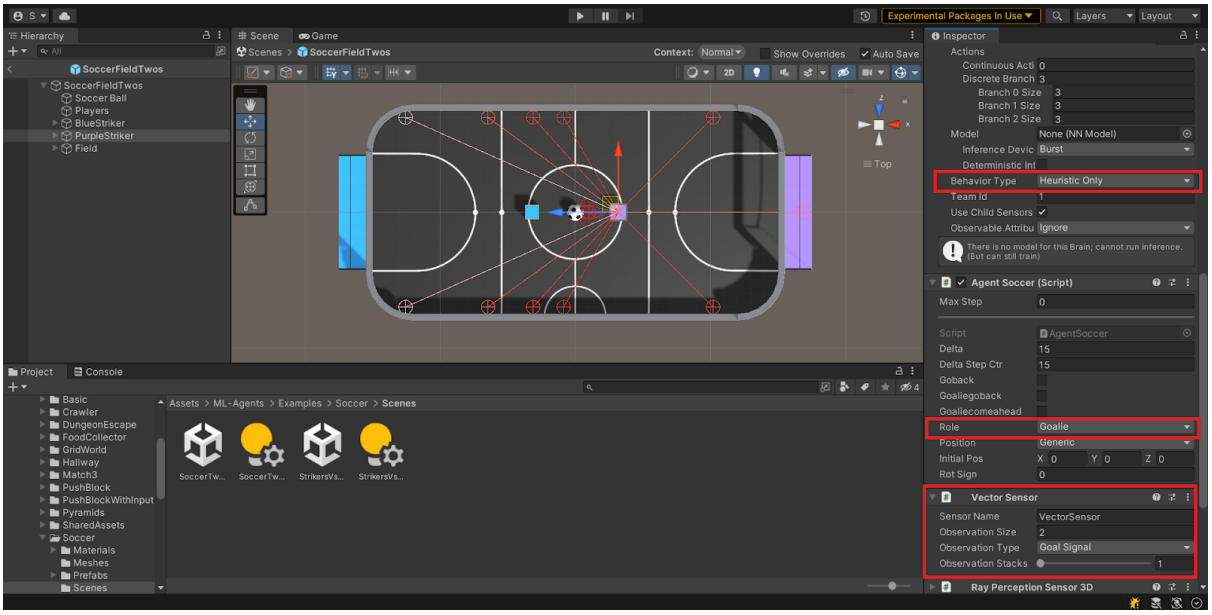


Fig A.4 Behavior parameters and properties of heuristic purple player in Stage 1 environment

Annex C - Environment and library changes for QMIX

The *rl-lib* implementation of QMIX we used for this project requires all the agents in an environment to be grouped. Such groups are treated like a single agent with a Tuple action and observation space, which are leveraged by QMIX for centralized training but decentralized execution.

In our case, such agent grouping is done by logically separating the teams of blue and purple players into 2 groups - and QMIX will only learn one policy for the 2 blue agents. The group name “team_0” is now treated as a (new) agent name - which has the original observation space and action space replicated 2x:

```
def groupedenv(args):
    o = Tuple([Box(-1.0, 1.0, (264,)), Box(-1.0, 1.0, (72,)),])
    a = Discrete(27)
    obs_space = Tuple([o, o])
    act_space = Tuple([a, a])
    grouping = {'team_0': ['SoccerTwos?team=0?agent_id=1',
'SoccerTwos?team=0?agent_id=3'],
                'team_1': ['SoccerTwos?team=1?agent_id=0',
'SoccerTwos?team=1?agent_id=2'], }
    sa_env = Unity3DEnvQMix(file_name=asset_file_name,
                            episode_horizon=3000,
                            no_graphics=NO_GRAPHICS_MODE,
                            side_channels=[customChannel])
    grouped_env = sa_env.with_agent_groups(groups=grouping, obs_space=obs_space,
act_space=act_space)
    return grouped_env
```

Secondly, the implementation only supports Discrete action spaces, however in our case, we had a MultiDiscrete(3,3,3) action space. The workaround to this was to use a flattened Discrete(27) space on the Python side of the algorithm and use a mapping (in *AgentSoccer.cs*) on the Unity side when the actions result in the movement of the player/agent.

Annex D - Goal Signals in Unity ML-Agents

Agents have the ability to gather data that will be used as "goal signals". The policy of the agent is conditional on a goal signal, thus if the goal (also referred to in this report as *role*) changes, the policy (i.e., the mapping from observations to actions) will also change. Do note that every observation has *some* impact on the Agent's policy, i.e. this is true for all observations. Yet, we can increase the significance of this conditioning for the agent by explicitly terming it a “goal signal”. In Unity Editor, this is done by adding a *VectorSensorComponent* or a *CameraSensorComponent* to the Agent and choosing *Goal Signal* as the *Observation Type*:

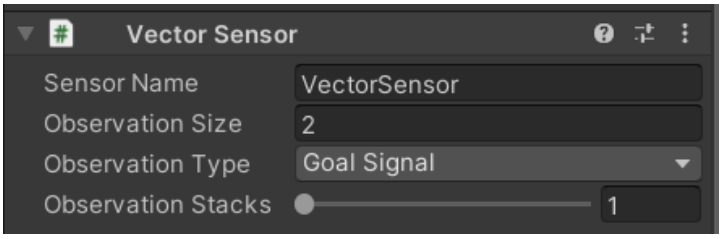


Fig B.1 VectorSensorComponent used in the SoccerTwos environment for the additional dependence of policy on goals (which, in our case, is essentially the role as a goalie or striker)

The associated code snippet to collect the observations and add it programmatically to each agent's policy network (in *AgentSoccer.cs*) is the following:

```
public override void CollectObservations(VectorSensor sensor)
{
    //The first arg is the selection index; the second is the
    //number of possibilities
    System.Array values = System.Enum.GetValues(typeof(Role));
    if (m_GoalSensor is object)
    {
        int goalNum = (int)CurrentGoal;
        m_GoalSensor.GetSensor().AddOneHotObservation(goalNum, values.Length);
    }
}
```

Annex E - Side Channels in Unity ML-Agents

Bundled with ML-Agents toolkit, Unity provides a concept known as *Side Channels* - which refers to additional communication channels that can be used to send information between the Unity Engine and external scripts or processes. These channels allow for more flexible and customizable interactions between the Unity environment and the machine learning algorithms running outside of it. Several out-of-the-box built-in Side Channels are provided to be used for monitoring the performance of the agents during training, receiving reward signals from external scripts, or controlling the behavior of the environment.

In addition, ML-Agents also allows users to create custom side channels to send and receive data specific to one's own use cases and to extend the functionality of ML-Agents and integrate it with other tools and systems. Two mirrored classes are required for the same, one in Unity (by extending *SideChannel* from the namespace *Unity.MLAgents.SideChannels*) and another in Python by extending the Python class *mlagents_envs.side_channel.side_channel.SideChannel*. Then, within the C# project on the Unity side, the channel object is registered with RegisterSideChannel(SideChannel) and UnregisterSideChannel(SideChannel). A similar is also required on the Python side while registering the environment using *tune*:

```
customChannel = CustomChannel(args)
...
tune.register_env("Stage1SoccerTwosRR-XS", lambda c:
Unity3DEnv(file_name=asset_file_name, episode_horizon=3000,
no_graphics=NO_GRAPHICS_MODE, side_channels=[customChannel]),)
```

Annex F - WandB Sweeps & chosen configuration

A2C sweep can be found at https://wandb.ai/pfunk/unity-football-cm3/sweeps/qdyttj1g. PPO sweep can be found at https://wandb.ai/pfunk/unity-football-cm3/sweeps/ysfm47b9.

With the sweep results, the hyperparameters used in the curriculum stages are as follows:

Parameter	A2C (lunar-sweep-17)	PPO (quiet-sweep-7)
lambda	0.87	0.95
gamma	0.80	0.99
lr	0.00005	0.0003
fcnet_layers	[256, 256, 256]	[512, 512]
fcnet_activation	tanh	tanh
use_lstm	False	False
vf_share_layers	False	False

Table D.1 Hyperparameters selected from sweep results

Annex G - NN Architecture for training runs

The architecture used in all the runs of MAPPO, QMIX and CM3 (PPO) is a simple feed-forward network with 2 FC layers of size 512 and *tanh* activation. LSTMs are not used.

Parameter	Value
lambda	0.95
gamma	0.99
lr	0.0003
sgd_minibatch_size	256
train_batch_size	4000
clip_param	0.2

Table E.1 Hyperparameters used for MAPPO/QMIX/CM3 as obtained from "quiet-sweep-7" configuration

This model was determined after playing around with the model architecture determined from default values of *rl-lib* and benchmarked configurations of Google Research Football with the assumption that the complexity of Unity SoccerTwos is less than that of Google Research Football, but since the domain is still very similar, a similar architecture could prove to be a good starting point. Further tuning was done using hyperparameter sweeps in WandB to arrive at said configuration.

Annex H - Source code and WandB runs

All the source code of the project has been committed to the GitHub repository https://github.com/shivanip14/unity-football-cm3.

The directory structure is as follows:

1. */checkpoints* - saved checkpoints during training
2. */captures* - videos and gifs captured during visual interpretation of algorithms
3. */src* - main source directory

a. */assets* - Unity assets for SoccerTwos environment for Windows and Linux
b. */configs* - configuration files for CM3 stages and baselines
c. */scripts* - contains all the scripts for all the algorithms - hyperparameter sweeps, training and play scripts
d. */stats* - statistics files generated after training and play
e. */utils* - utility scripts to log metrics and generate graphs, a wrapper for Unity environment, side channel configuration for SoccerTwos environment
f. */wincounts* - win counts files generated after training and play

The training metrics for both the baselines and CM3, as well as the win rate metrics have been logged in the following WandB projects:

1. Training metrics: https://wandb.ai/patesg/tfm-cm3
2. Win rates: https://wandb.ai/patesg/tfm-cm3-winrate