Group work content

Research on A Reinforcement Learning Application for Football game Environment

Introduction

Recent progress in the field of reinforcement learning has been accelerated by virtual learning environments such as video games, where novel algorithms and ideas can be quickly tested in a safe and reproducible manner. We introduce the Google Research Football Environment, a new reinforcement learning environment where agents are trained to play football in an advanced, physics-based 3D simulator. The resulting environment is challenging, easy to use and customize, and it is available under a permissive open-source license. In addition, it provides support for multiplayer and multi-agent experiments. We propose three full-game scenarios of varying difficulty with the Football Benchmarks and report baseline results for three commonly used reinforcement algorithms (IMPALA, PPO, and Ape-X DQN). We also provide a diverse set of simpler scenarios with the Football Academy and showcase several promising research directions.
While a variety of reinforcement learning environments exist, they often come with a few drawbacks for research, which we discuss in detail in the next section. For example, they may either be too easy to solve for state-of-theart algorithms or require access to large amounts of computational resources. At the same time, they may either be (near-)deterministic or there may even be a known model of the environment (such as in Go or Chess). Similarly, many learning environments are inherently single player by only modeling the interaction of an agent with a fixed environment or they focus on a single aspect of reinforcement learning such as continuous control or safety. Finally, learning environments may have restrictive licenses or depend on closed source binaries. This highlights the need for a RL environment that is not only challenging from a learning standpoint and customizable in terms of difficulty but also accessible for research both in terms of licensing and in terms of required computational resources. Moreover, such an environment should ideally provide the tools to a variety of current reinforcement learning research topics such as the impact of stochasticity, self-play, multi-agent setups and model-based reinforcement learning, while also requiring smart decisions, tactics, and strategies at multiple levels of abstraction.

Football Engine

The Football Environment is based on the Football Engine, an advanced football simulator built around a heavily customized version of the publicly available GameplayFootball simulator. The engine simulates a complete football game, and includes the most common football aspects, such as goals, fouls, corners, penalty kicks, or offsides.

Supported Football Rules.

The engine implements a full football game under standard rules, with 11 players on each team. These include goal kicks, side kicks, corner kicks, both yellow and red cards, offsides, handballs and penalty kicks. The length of the game is measured in terms of the number of frames, and the default duration of a full game is 3000 (10 frames per second for 5 minutes). The length of the game, initial number and position of players can also be edited in customized scenarios (see Football Academy below). Players on a team have different statistics[1], such as speed or accuracy and get tired over time.

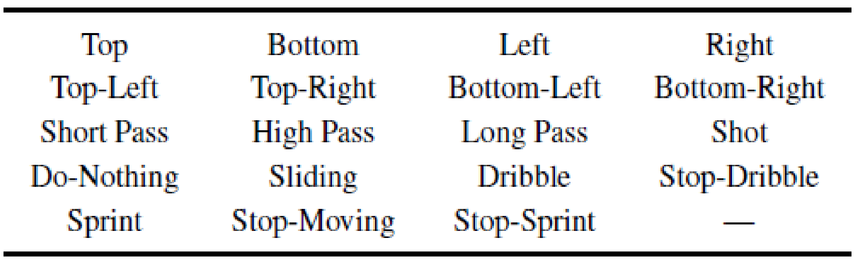
State & Observations.

We define as state the complete set of data that is returned by the environment after actions are performed. On the other hand, we define as observation or representation any transformation of the state that is provided as input to the control algorithms. The definition of the state contains information such as the ball position and possession, coordinates of all players, the active player, the game state (tiredness levels of players, yellow cards, score, etc) and the current pixel frame. We propose three different representations. Two of them (pixels and SMM) can be stacked across multiple consecutive time-steps (for instance, to determine the ball direction), or unstacked, that is, corresponding to the current time-step only. Researchers can easily define their own representations based on the environment state by creating wrappers similar to the ones used for the observations below. Pixels. The representation consists of a 1280 _ 720 RGB image corresponding to the rendered screen. This includes both the scoreboard and a small map in the bottom middle part of the frame from which the position of all players can be inferred in principle. Super Mini Map. The SMM representation consists of four 72_96 matrices encoding information about the home team, the away team, the ball, and the active player respectively. The encoding is binary, representing whether there is a player or ball in the corresponding coordinate. Floats. The floats representation provides a compact encoding and consists of a 115-dimensional vector summarizing many aspects of the game, such as players coordinates, ball possession and direction, active player, or game mode.

Actions. The actions available to an individual agent (player) are displayed in Table 1. They include standard move actions (in 8 directions), and different ways to kick the ball (short and long passes, shooting, and high passes that can’t be easily intercepted along the way). Also, players can sprint (which affects their level of tiredness), try to intercept the ball with a slide ackle or dribble if they posses the ball. We experimented with an action to switch the active player in defense (otherwise, the player with the ball must be active). However, we observed that policies tended to exploit this action to return control to built-in AI behaviors for non-active players, and we decided to remove it from the action set. We do not implement randomized sticky actions. Instead, once executed, moving and sprinting actions are sticky and continue until an explicit stop action is performed (Stop-Moving and Stop-Sprint respectively).

Rewards. The Football Engine includes two reward functions that can be used out-of-the-box: SCORING and CHECKPOINT. It also allows researchers to add custom reward functions using wrappers which can be used to investigate reward shaping approaches. CORING corresponds to the natural reward where each eam obtains a +1 reward when scoring a goal, and a reward when conceding one to the opposing team. The SCORING reward can be hard to observe during the initial stages of training, as it may require a long sequence of consecutive events: overcoming the defense of a potentially strong opponent, and scoring against a keeper. CHECKPOINT is a (shaped) reward that specifically addresses the sparsity of SCORING by encoding the domain knowledge that scoring is aided by advancing across the pitch: It augments the SCORING reward with an additional auxiliary reward contribution for moving the ball close to the opponent's goal in a controlled fashion. More specifically, we divide the opponent's field in 10 checkpoint regions according to the Euclidean distance to the opponent goal. Then, the first time the agent's team possesses the ball in each of the checkpoint regions, the agent obtains an additional reward of +0:1. This extra reward can be up to +1, i.e., the same as scoring a single goal. Any non-collected checkpoint reward is also added when scoring in order to avoid penalizing agents that do not go through all the checkpoints before scoring (i.e., by shooting from outside a checkpoint region). Finally, checkpoint rewards are only given once per episode.

Table 1: Action Set



Top	Bottom	Left	Right
Top-Left	Top-Right	Bottom-Left	Bottom-Right
Short Pass	High Pass	Long Pass	Shot
Do-Nothing	Sliding	Dribble	Stop-Dribble
Sprint	Stop-Moving	Stop-Sprint	—

Experimental Setup

Training agents for the Football Benchmarks can be challenging. To allow researchers to quickly iterate on new research ideas, we also provide the Football Academy: a diverse set of scenarios of varying difficulty. These 11 scenarios (see Figure 5 for a selection) include several variations where a single player has to score against an empty goal (Empty Goal Close, Empty Goal, Run to Score), a number of setups where the controlled team has to break a specific defensive line formation (Run to Score with Keeper, Pass and Shoot with

Keeper, 3 vs 1 with Keeper, Run, Pass and Shoot with Keeper) as well as some standard situations commonly found in football games (Corner, Easy Counter-Attack, Hard Counter-Attack). For a detailed description, we refer to the Appendix. Using a simple API, researchers can also easily define their own scenarios and train agents to solve them.
Here are Examples of Football Academy scenarios.

(a) Empty Goal Close

(b) Run to Score

(c) 11 vs 11 with Lazy Opponents

(d) 3 vs 1 with Keeper

(e) Pass and Shoot

(f) Easy Counter-attack

Experimental Results

Multiplayer Experiments

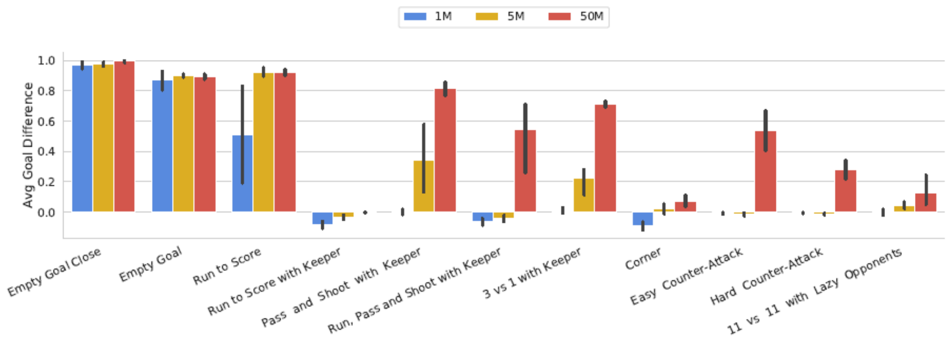
The Football Environment provides a way to train against different opponents, such as built-in AI or other trained agents. Note this allows, for instance, for self-play schemes. When a policy is trained against a fixed opponent, it may exploit its particular weaknesses and, thus, it may not generalize well to other adversaries.We conducted an experiment to showcase this in which a first model A was trained against a built-in AI agent on the standard 11 vs 11 medium scenario. Then, another agent B was trained against a frozen version of agent A on the same scenario. While B managed to beat A consistently, its performance against built-in AI was poor. The numerical results showing this lack of transitivity across the agents are presented in Table.

Table 2: Average goal difference ± one standard deviation across 5 repetitions of the experiment.

A vs built-in AI	4.25 ± 1.72
B vs A	11.93 ± 2.19
B vs built-in AI	-0.27 ± 0.33

Multi-Agent Experiments

The environment also allows for controlling several players from one team simultaneously, as in multi-agent reinforcement earning. We conducted experiments in this setup with the 3 versus 1 with Keeper scenario from Football Academy. We varied the number of players that the policy controls from 1 to 3, and trained with Impala.
Average Goal Difference on Football Academy for IMPALA with SCORING reward.



Average Goal Difference on Football Academy for PPO with SCORING reward.

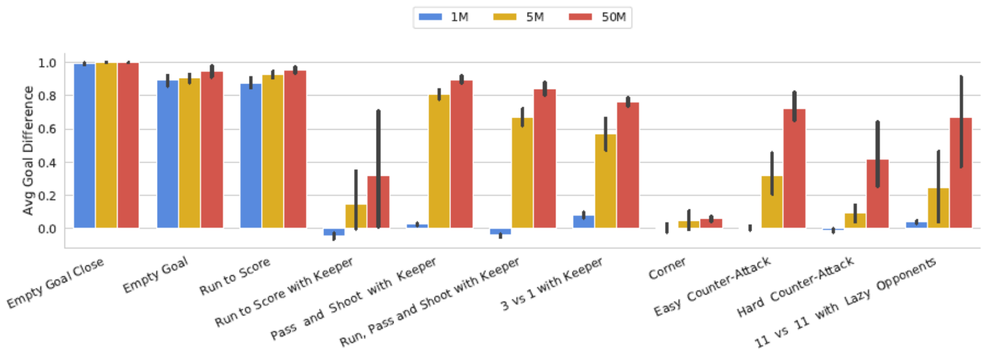


Table 3: Scores achieved by the policy controlling 1, 2 or 3 players respectively, after 5M and 50M steps of training.

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

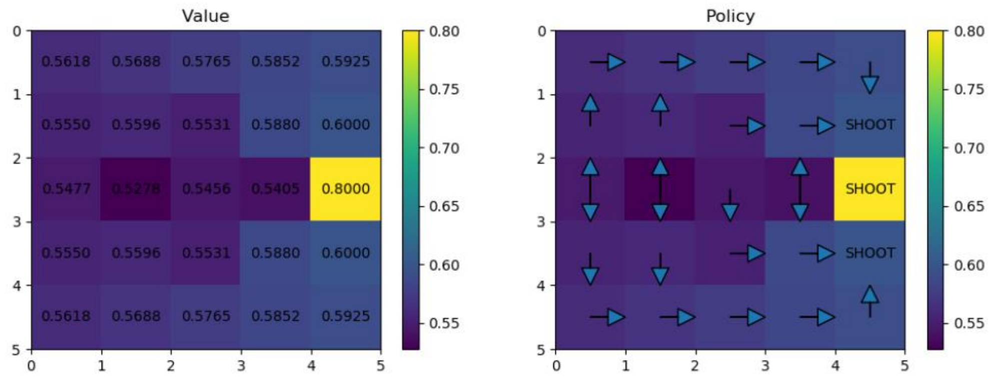
Table 4: Average goal advantages per representation.

Representation	100M steps	500M steps
Floats	2.42 ± 0.46	5.73 ± 0.28
Pixels gray	-0.82 ± 0.26	7.18 ± 0.85
SMM	5.94 ± 0.86	9.75 ± 4.15
SMM stacked	7.62 ± 0.67	12.89 ± 0.51

RL for Football RoboCup 2D Half Field Offense

Simple policy iteration exercise

Based on policy iteration, which is can be expresed in solving for the optimal policy using PPO algorithem in order solve a toy example, The optimal policy and optimal values are



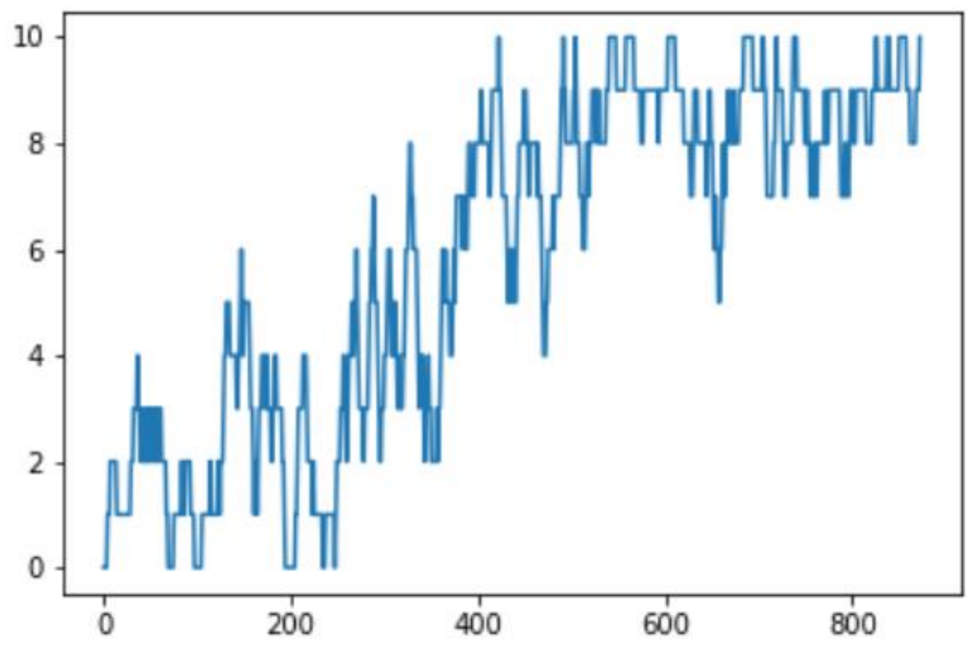
Q-learning algorithms

This version is able to keep very close to the maximum of 10 goals per 10 trials, that means that in 10 trials (i.e. games) it makes close to 10 goals being 10 the maximum achivable value.

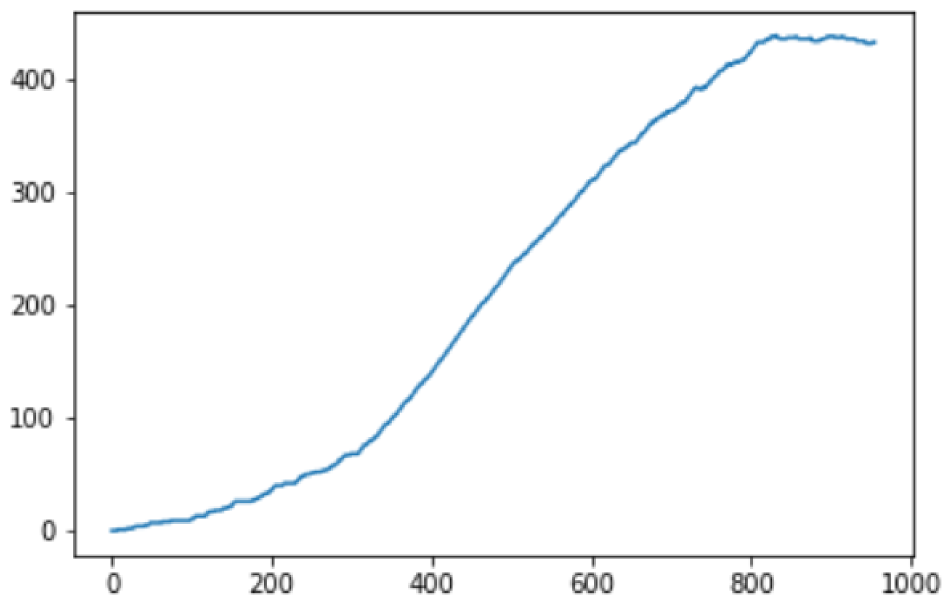
The parameters are

- Learning Rate: 0.95, with a Linear Decay Ratio of 400 episodes (that means that after 400 episodes the learning rate will be at its minimum). The Decay ratio does not decrease during its first 100 trials to improve exploration
- Minimum Learning Rate: 0.01
- Epsilon: 0.5 with the same Linear Decay as the learning rate.
- Minimum epsilon: 0.01 to avoid being stuck at certain positions for longer than 100 steps

Plot of the last 10 goals in training time:



Plot of the last 500 goals in training time: it will always inscrease up to the episode 500, from there on it can decrease but not very fast. **The maximum is reached around 430 goals out of 500 trials**



its performance is

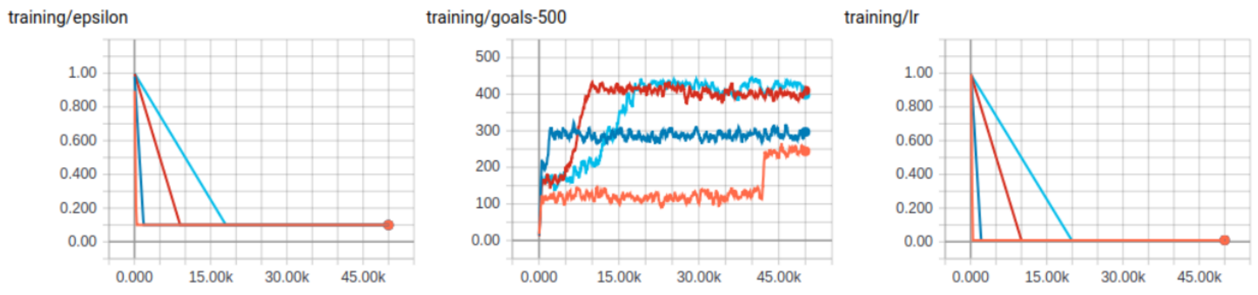
metric	value
TotalFrames	466007
Avg Frames Per Trial	466.0
Avg Frames per Goal	479.7
Trials	1000
Goals	652
Defense Captured	1
Balls out of Bounds	311

metric	value
Out of time	36

Independent Q-Learning (IQL)

In this case there are two agents, the only way to score is if one player marks the enemy (there is only one defensing player) while the other shoots. for IQL the agents are not allowed to comunicate but still have to learn to act together.

The results are similar to those of QL. The agents manage to score around 410 goals out of 500 trials after 20k episodes. Here are the results varying the epsilon and lr scheduler.



My WORK

Research on the detail of TPPO and A3C

Trust Region Policy Optimization

Consider an infinite-horizon discounted Markov decision process (MDP), defined by the tuple (S, A, P, r, ρ0, γ), where S is a finite set of states, A is a finite set of actions, P : S × A × S → R is the transition probability distribution, r : S → R is the reward function, ρ0 : S → R is the distribution of the initial state s0, and γ ∈ (0, 1) is the discount factor.

Let π denote a stochastic policy π : S × A → [0, 1], and let η(π) denote its expected discounted reward:

$$\eta(\pi) = \mathbb{E}_{s_0,a_0,\ldots}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t)\right], \text{ where}$$
$$s_0 \sim \rho_0(s_0),\ a_t \sim \pi(a_t|s_t),\ s_{t+1} \sim P(s_{t+1}|s_t,a_t).$$

We will use the following standard definitions of the stateaction value function Qπ, the value function Vπ, and the advantage function Aπ:

$$Q_\pi(s_t,a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\ldots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l})\right],$$
$$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},\ldots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l})\right],$$
$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s), \text{ where}$$
$$a_t \sim \pi(a_t|s_t), s_{t+1} \sim P(s_{t+1}|s_t,a_t) \text{ for } t \geq 0.$$

The following useful identity expresses the expected return of another policy π˜ in terms of the advantage over π, accumulated over timesteps

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{s_0,a_0,\cdots\sim\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t,a_t)\right] \qquad (1)$$

where the notation Es0,a0,···∼π˜ [. . .] indicates that actions are sampled at ∼ π˜(·|st). Let ρπ be the (unnormalized) discounted visitation frequencies

$$\rho_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \ldots,$$

where s0 ∼ ρ0 and the actions are chosen according to π. We can rewrite Equation (1) with a sum over states instead of timesteps:

$$\begin{aligned}\eta(\tilde{\pi}) &= \eta(\pi) + \sum_{t=0}^{\infty}\sum_{s} P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s)\gamma^t A_\pi(s,a) \\ &= \eta(\pi) + \sum_{s}\sum_{t=0}^{\infty} \gamma^t P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) A_\pi(s,a) \\ &= \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s,a). \qquad (2)\end{aligned}$$

This equation implies that any policy update π → π˜ that has a nonnegative expected advantage at every state s, i.e., P a π˜(a|s)Aπ(s, a) ≥ 0, is guaranteed to increase the policy performance η, or leave it constant in the case that the expected advantage is zero everywhere. This implies the classic result that the update performed by exact policy iteration, which uses the deterministic policy
π˜(s) = arg maxa Aπ(s, a), improves the policy if there is at least one state-action pair with a positive advantage value and nonzero state visitation probability, otherwise the algorithm has converged to the optimal policy. However, in the approximate setting, it will typically be unavoidable, due to estimation and approximation error, that there will be some states s for which the expected advantage is negative, that is, P a π˜(a|s)Aπ(s, a) < 0. The complex dependency of ρπ˜(s) on π˜ makes Equation (2) difficult to optimize directly. Instead, we introduce the following local approximation to η:

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_\pi(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s,a). \qquad (3)$$

Note that Lπ uses the visitation frequency ρπ rather than ρπ˜, ignoring changes in state visitation density due to changes in the policy. However, if we have a parameterized policy πθ, where πθ(a|s) is a differentiable function of the parameter vector θ, then Lπ matches η to first order (see Kakade & Langford (2002)). That is, for any parameter value θ0,

$$\begin{aligned} L_{\pi_{\theta_0}}(\pi_{\theta_0}) &= \eta(\pi_{\theta_0}), \\ \nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)\big|_{\theta=\theta_0} &= \nabla_\theta \eta(\pi_\theta)\big|_{\theta=\theta_0}. \qquad (4)\end{aligned}$$

Equation (4) implies that a sufficiently small step πθ0 → π˜ that improves Lπθold will also improve η, but does not give us any guidance on how big of a step to take
To address this issue, Kakade & Langford (2002) proposed a policy updating scheme called conservative policy iteration, for which they could provide explicit lower bounds on the improvement of η. To define the conservative policy iteration update, let πold denote the current policy, and let π 0 = arg maxπ0 Lπold (π 0). The new policy πnew was defined to be the following mixture:

$$\pi_{\text{new}}(a|s) = (1-\alpha)\pi_{\text{old}}(a|s) + \alpha\pi'(a|s). \qquad (5)$$

Optimization of Parameterized Policies

we considered the policy optimization problem independently of the parameterization of π and under the assumption that the policy can be evaluated at all states. We now describe how to derive a practical algorithm from these theoretical foundations, under finite sample counts and arbitrary parameterizations.
Since we consider parameterized policies πθ(a|s) with parameter vector θ, we will overload our previous notation to use functions of θ rather than π, e.g. η(θ) := η(πθ), Lθ(˜θ) := Lπθ (πθ˜), and DKL(θ k ˜θ) := DKL(πθ k πθ˜). We will use θold to denote the previous policy parameters that we want to improve upon.
The preceding section showed that η(θ) ≥ Lθold (θ) − CDmax KL (θold, θ), with equality at θ = θold. Thus, by performing the following maximization, we are guaranteed to improve the true objective η:

$$\underset{\theta}{\text{maximize}}\left[L_{\theta_{\text{old}}}(\theta) - C D_{\text{KL}}^{\max}(\theta_{\text{old}}, \theta)\right].$$

In practice, if we used the penalty coefficient C recommended by the theory above, the step sizes would be very small. One way to take larger steps in a robust way is to use a constraint on the KL divergence between the new policy and the old policy, i.e., a trust region constraint:

$$\begin{aligned}&\underset{\theta}{\text{maximize}}\, L_{\theta_{\text{old}}}(\theta) \\ &\text{subject to } D_{\text{KL}}^{\max}(\theta_{\text{old}}, \theta) \le \delta.\end{aligned} \tag{11}$$

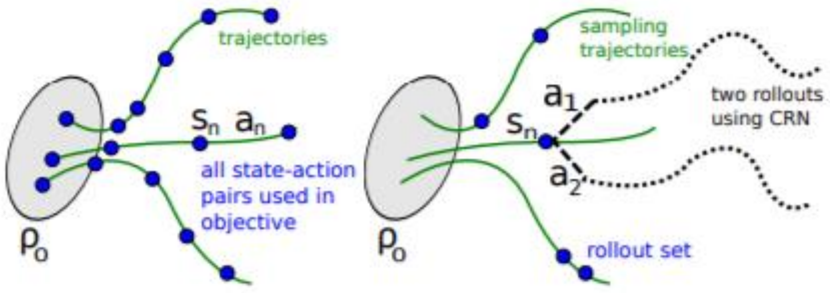
This problem imposes a constraint that the KL divergence is bounded at every point in the state space. While it is motivated by the theory, this problem is impractical to solve due to the large number of constraints. Instead, we can use a heuristic approximation which considers the average KL divergence:

$$\overline{D}_{\text{KL}}^{\rho}(\theta_1, \theta_2) := \mathbb{E}_{s\sim\rho}\left[D_{\text{KL}}(\pi_{\theta_1}(\cdot|s) \,\|\, \pi_{\theta_2}(\cdot|s))\right].$$

We therefore propose solving the following optimization problem to generate a policy update:

$$\begin{aligned}&\underset{\theta}{\text{maximize}}\, L_{\theta_{\text{old}}}(\theta) \\ &\text{subject to } \overline{D}_{\text{KL}}^{\rho_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \le \delta.\end{aligned} \tag{12}$$

Similar policy updates have been proposed in prior work (Bagnell & Schneider, 2003; Peters & Schaal, 2008b; Peters et al., 2010), and we compare our approach to prior methods in Section 7 and in the experiments in Section 8. Our experiments also show that this type of constrained update has similar empirical performance to the maximum KL divergence constraint in Equation (11).



Sample-Based Estimation of the Objective and Constraint

We seek to solve the following optimization problem, obtained by expanding Lθold in Equation (12):

$$\underset{\theta}{\text{maximize}} \sum_s \rho_{\theta_\text{old}}(s) \sum_a \pi_\theta(a|s) A_{\theta_\text{old}}(s,a)$$
$$\text{subject to } \overline{D}_\text{KL}^{\rho_{\theta_\text{old}}}(\theta_\text{old}, \theta) \le \delta. \qquad (13)$$

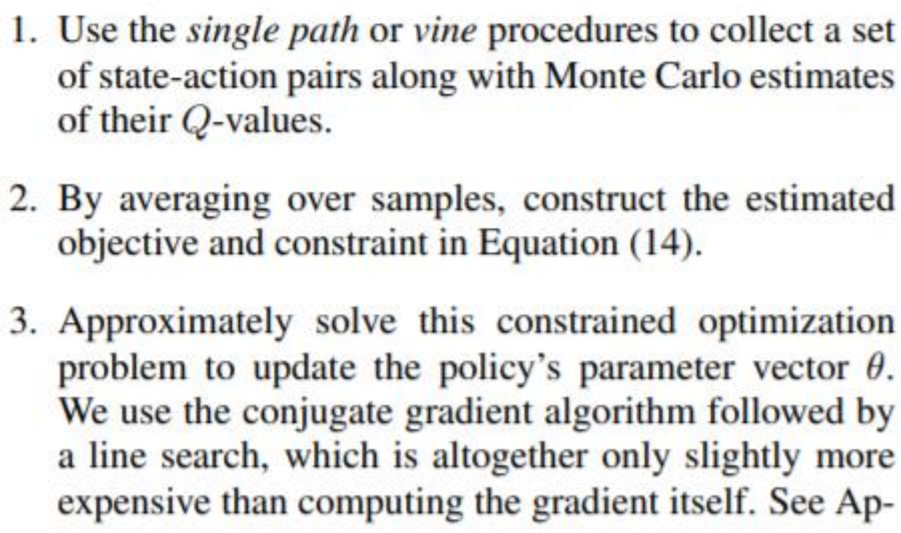
We first replace $\sum_s \rho_{\theta_\text{old}}(s)\,[\dots]$ in the objective by the expectation $\frac{1}{1-\gamma}\mathbb{E}_{s\sim\rho_{\theta_\text{old}}}[\dots]$. Next, we replace the advantage values A_{θ_old} by the Q-values Q_{θ_old} in Equation (13), which only changes the objective by a constant. Last, we replace the sum over the actions by an importance sampling estimator. Using q to denote the sampling distribution, the contribution of a single s_n to the loss function is

$$\sum_a \pi_\theta(a|s_n) A_{\theta_\text{old}}(s_n,a) = \mathbb{E}_{a\sim q}\left[\frac{\pi_\theta(a|s_n)}{q(a|s_n)} A_{\theta_\text{old}}(s_n,a)\right].$$

Our optimization problem in Equation (13) is exactly equivalent to the following one, written in terms of expectations:

$$\underset{\theta}{\text{maximize}}\, \mathbb{E}_{s\sim\rho_{\theta_\text{old}}, a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_\text{old}}(s,a)\right] \qquad (14)$$
$$\text{subject to } \mathbb{E}_{s\sim\rho_{\theta_\text{old}}}\left[D_\text{KL}(\pi_{\theta_\text{old}}(\cdot|s) \,\|\, \pi_\theta(\cdot|s))\right] \le \delta.$$

Practical Algorithm

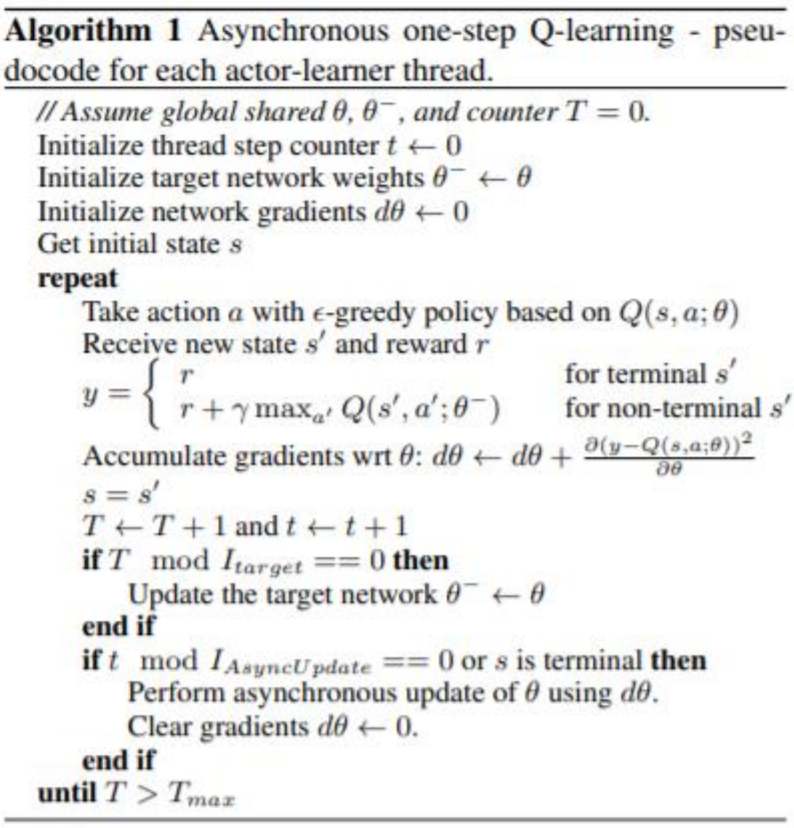


1. Use the *single path* or *vine* procedures to collect a set of state-action pairs along with Monte Carlo estimates of their Q-values.
2. By averaging over samples, construct the estimated objective and constraint in Equation (14).
3. Approximately solve this constrained optimization problem to update the policy's parameter vector θ. We use the conjugate gradient algorithm followed by a line search, which is altogether only slightly more expensive than computing the gradient itself. See Ap-

Asynchronous Methods for Deep Reinforcement Learning(A3C)

Asynchronous RL Framework

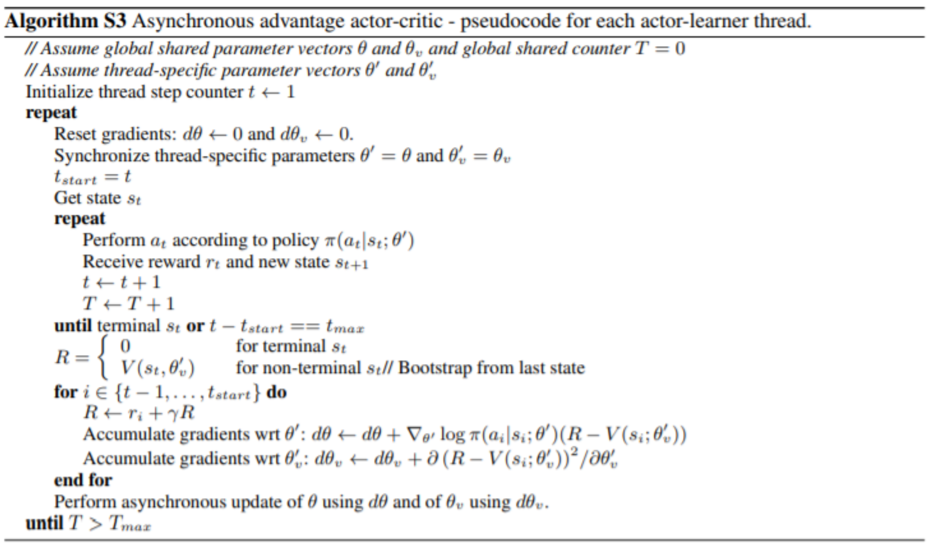
We now present multi-threaded asynchronous variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic. The aim in designing these methods was to find RL algorithms that can train deep neural network policies reliably and without large resource requirements. While the underlying RL methods are quite different, with actor-critic being an on-policy policy search method and Q-learning being an off-policy value-based method, we use two main ideas to make all four algorithms practical given our design goal



Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

```
// Assume global shared $\theta$, $\theta^-$, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$
repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} == 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
```

learners running in parallel are likely to be exploring different parts of the environment. Moreover, one can explicitly use different exploration policies in each actor-learner to maximize this diversity. By running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates. Hence, we do not use a replay memory and rely on parallel actors employing different exploration policies to perform the stabilizing role undertaken by experience replay in the DQN training algorithm. In addition to stabilizing learning, using multiple parallel actor-learners has multiple practical benefits. First, we obtain a reduction in training time that is roughly linear in the number of parallel actor-learners. Second, since we no longer rely on experience replay for stabilizing learning we are able to use on-policy reinforcement learning methods such as Sarsa and actor-critic to train neural networks in a stable way. We now describe our variants of one-step Qlearning, one-step Sarsa, n-step Q-learning and advantage actor-critic.



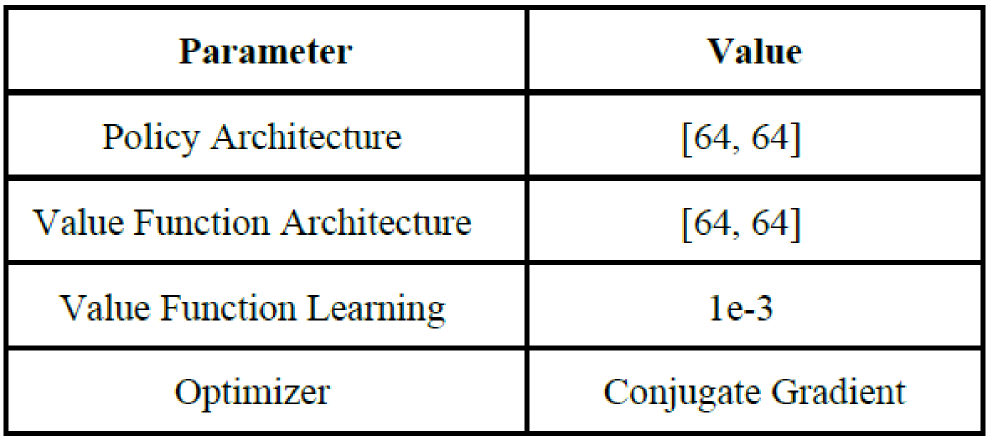
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

Experiment result

Trust Region Policy Optimiztation

In the final baseline model, a TRPO agent was trained with the Reinforcement Learning Replications (rl-replicas) in Pytorch. The model was set up with Policy and ValueFunction. These two core components have their own neural network and optimizer. To run inference, Policy was then initialized with the trained network. The model trained in the default 11 vs 11 games with the number of epochs and steps varied. The running time ranged from 3 hours to 9 hours in Kaggle GPUs. The same reward system as in the Double DQN model was applied, with negative rewards indicating own goals. All the experiments were done without any hyperparameter tuning, as specified below:



Parameter	Value
Policy Architecture	[64, 64]
Value Function Architecture	[64, 64]
Value Function Learning	1e-3
Optimizer	Conjugate Gradient

Table 3. Hyperparameters for the TRPO model

The average episode rewards are shown as below:

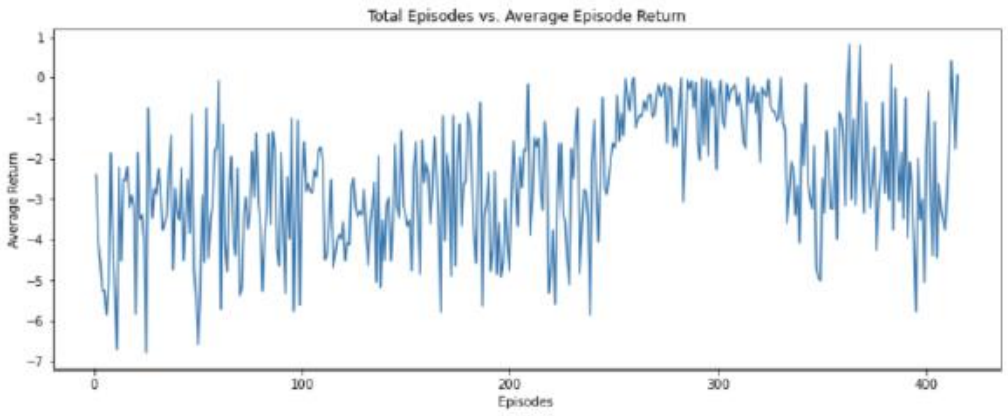


Figure 5. Average episode rewards for the TRPO model

The graph is not entirely conclusive but it does show that the average rewards will probably increase with the number of episodes at a later stage. The algorithm also takes more time to converge as compared to others. Also attached is a video of the best case scenario where the agent has managed to score a goal.

Number of Epochs	**Steps per Epoch**	**Final Score (Goals)**
5	200000	1 - 0
10	50000	0 - 0
20	50000	0 - 0
10	250000	Timeout
20	10000	0 - 0
5	250000	0 - 0

Table 4. Experiment specifications and rewards for the TRPO model

A3C

To run A3C, a docker container was created based on the Nvidia GPU and Google Football

base image. This process was tedious and full of errors. However, after completion the network was able to be trained. There were 64000 time steps.
The designed network for both the actor and the critic had the following layout:

2D Convolution

Batch Norm

Relu

2D Convolution

Batch Norm

Relu

2D Convolution

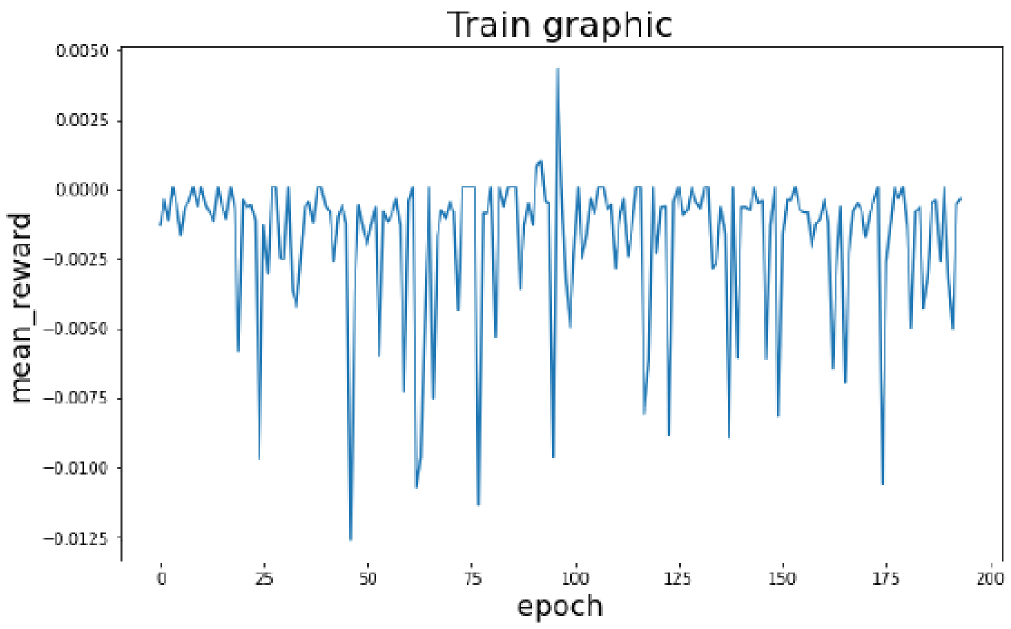
Batch Norm

Relu

Linear

Linear

The first three layers theoretically will parse the input image of the environment, then process that information in such a way that it acquires game state information, and then, compute an action to take in the environment. Since this is A3C, it requires both an actor and a critic model for the algorithm, and thus weights must be computed for two models. Unfortunately, the model created had errors, and was unable to learn from the football environment.



Mean Rewards for the A3C trained model

REFERENCES

1. Google Research. Google Research Football with Manchester City F.C., 2020, https://www.kaggle.com/c/google-football. Accessed 11 Dec. 2020.

2. Google Research. Google Research Football, 2020, https://github.com/google-research/football. Accessed 11 Dec. 2020.

3. Kurach, Karol, et al. "Google research football: A novel reinforcement learning environment." arXiv preprint arXiv:1907.11180 (2019).

4. Depth First Learning, https://www.depthfirstlearning.com/2018/TRPO.
5. Trust Region Policy Optimization, https://spinningup.openai.com/en/latest/algorithms/trpo.html.
6. Asynchronous Methods for Deep Learning
7. Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." arXiv preprint arXiv:1509.06461 (2015).
8. J Schulman, F Wolski, P Dhariwal, A Radford, O Klimov. “Proximal policy optimization algorithms” arXiv preprint arXiv:1707.06347(2017)