SOCCER: Self-Optimizing Competitive Control via Efficient Reinforcement

Atul Shatavart Nadig [* 1] **Cheuk Hei Chu** [* 1]

1. Problem

We consider the soccer-like online game of Pen Football (Scratch, 2016) - a 2 player game where the agents compete to kick the ball into the opposing goal. Each agent can:

- Move left or right, which changes the acceleration in the corresponding direction,
- Jump (with double jump allowed),
- Interact physically with the ball under gravity and according to life-like physical laws.

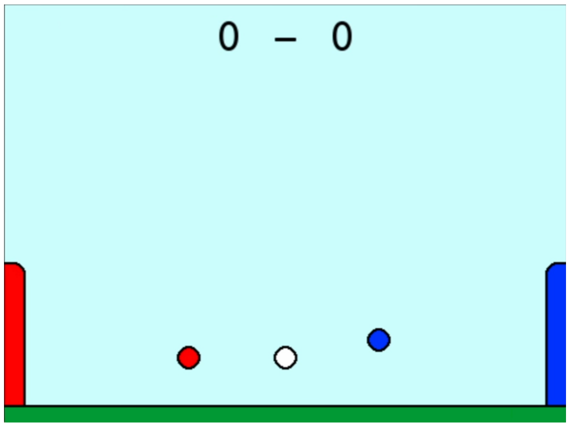

Figure 1. Example game state

Readers can try out the game, hosted on Github, at `https://cheukhei-chu.github.io/pen-football-rl/`

2. Challenges

The game is a mix of many techniques- making it challenging to learn. Some of the challenges are:

- Continuous dynamics with frequent nonlinear interactions.
- Sparse rewards as they are only due to scoring of goals
- Long-term credit assignment, as the effects of a set of actions may not be seen until much later.
- Accurate control- even with the second player not taken into account, the timing of the player needs to be accurate in order to make contact with the ball, making it a challenging control problem as well.

[1]MIT. Correspondence to: C. H. Chu <ch_chu@mit.edu>, A. Nadig <atul567@mit.edu>.

3. Problem setup

3.1. Implementation

The core game logic, which simulates the physics and rules of Pen Football, was developed using the Pygame library. This implementation manages the state transitions of the two-player game, including ball movement, collisions, and scoring at 30 frames per second. To facilitate reinforcement learning, a game-playing interface was developed as a custom environment adhering to the Gymnasium API standard.

- Observation Space: position (x, y) and velocity (v_x, v_y) of both players and the ball, for a total of 12 dimensions, each normalized to within $[-1, 1]$.
- Action Space: For each of the Left, Right, and Up keys, the agent may choose between clicking and not clicking, for a total of 8 possible combinations.
- Game modes: The environment supports a `MultiAgent` mode, where two agents play the game against each other, and a `drill` mode, where a single agent can be trained on drills involving only itself and the ball.

The models for the agents are implemented as neural networks in `Pytorch`. The networks generally take in the concatenated and normalized observation space vector as input and outputs three logits via three heads, one for each key. Models tested are on the order of around 10^5 parameters, hence inference and backpropagation is significantly faster than the game rendering speed of 30 fps, enabling fast training on a single CPU. Model checkpoints are then saved at a regular basis as `.pth` files. Specific model architecture will be discussed in Section 4.

3.2. Mathematical formulation

We model Pen Football as a deterministic, fully-observable, two-player, zero-sum Markov game. Below we state the Markov game tuple and the optimization objective used during training.

Markov game. Let $\mathcal{S}$ denote the state space and $\mathcal{A}^i$ the action space of player $i \in \{1, 2\}$. We define the game by the tuple

$$(\mathcal{S}, \mathcal{A}^1, \mathcal{A}^2, f, r^1, r^2, \gamma, \tau),$$

where

- $\mathcal{S} = \mathbb{R}^{12}$ is the (continuous) state space. A state $s_t \in \mathcal{S}$ is the concatenation of the two players' (x, y) positions and (v_x, v_y) velocities and the ball's (x, y) position and (v_x, v_y), in that order, for a total of 12 real-valued components. In practice each component is normalized to lie in $[-1, 1]$ before being passed to the agent.

- Each player's action space is $\mathcal{A}^i = \{0, 1\}^3$. We write an action as a binary vector

$$a_t^i = (\text{left}_t^i,\ \text{right}_t^i,\ \text{jump}_t^i),$$

where each component is chosen independently at each timestep. A jump action corresponds to an instantaneous vertical impulse; the environment enforces a maximal two-jump constraint (double-jump) before the agent next contacts the ground.

- The deterministic transition function $f : \mathcal{S} \times \mathcal{A}^1 \times \mathcal{A}^2 \to \mathcal{S}$ implements the game physics and contact resolution (gravity, collisions, restitution, and action-induced accelerations/impulses). Thus

$$s_{t+1} = f(s_t, a_t^1, a_t^2).$$

Jump availability and jump-count constraints are handled deterministically inside f (i.e., whether a jump command produces an impulse is a deterministic function of the current state and recent contact history as tracked by the physics).

- The reward functions $r^i : \mathcal{S} \times \mathcal{A}^1 \times \mathcal{A}^2 \to \mathbb{R}$ are zero for all non-goal timesteps and encode scoring at the moment a goal is observed. If player i scores at time t (the ball position lies in the opponent's goal region), then

$$r_t^i = +1, \qquad r_t^j = -1 \quad (j \neq i),$$

otherwise $r_t^1 = r_t^2 = 0$. This makes the game zero-sum: $r_t^1 = -r_t^2$ for every t.

- $\gamma \in (0, 1)$ is the discount factor used by the learning algorithm; in our experiments $\gamma = 0.99$.

- The episode termination function τ ends an episode when a goal is scored or when a fixed horizon $T_{\max} = 1800$ timesteps is reached, whichever occurs first.

Policies and objective. Each player i uses a stochastic policy $\pi_{\theta^i}(\, a^i \mid s)$ parameterized by θ^i. Because the three action bits are chosen independently we factor the per-player policy as

$$\pi_{\theta^i}(a^i \mid s) = \prod_{k \in \{\text{L,R,J}\}} \pi_{\theta^i}^k(a_k^i \mid s),$$

where each π^k is a Bernoulli distribution (represented in practice by a logit output). During training we maximize the standard discounted return for the learner against the (sampled) opponent:

$$J(\theta^i) \;=\; \mathbb{E}_{\tau \sim \pi_{\theta^i}, \pi_{\text{opp}}}\left[\sum_{t=0}^{T} \gamma^t r_t^i\right],$$

where the expectation is over trajectories generated by the learner policy π_{θ^i} and the opponent policy π_{opp} (drawn from the league or self-play pool). In the zero-sum setting improving $J(\theta^i)$ implicitly decreases the opponent's expected return.

Remarks about implementation.

- The environment is deterministic; all stochasticity arises only from the agents' policies during training (exploration) and from opponent sampling in the league.
- The agent network outputs three independent logits, one per action bit; these logits parameterize the Bernoulli factors in π_{θ^i}.
- For experiments that use reward shaping (see Section 4.3), the shaped reward $\tilde{r}_t$ is treated as the environment reward in the objective above.
- The entire codebase is hosted on Github at `https://github.com/cheukhei-chu/pen-football-rl`.

3.3. Evaluation

Initially, before we obtained models that can meaningfully play the game, evaluation was mostly done by hand where we manually play against the model.

Later on, to develop a quantifiable measure of how good a model is at the game, we chose to implement an ELO system, similar to that of chess (Fenner et al., 2011). If two models of ratings R_A, R_B play a game with A winning, then the expected score that A gets is

$$E_a = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}}$$

Using this, we can make the rating adjustment of

$$R_A \leftarrow R_A + K \cdot (W_A - E_A)$$

where W_A is the indicator variable that is 1 if A wins, and K is a constant, usually taken to be $1/16$, which we adopt. The implementation is as follows:

- A small group of diverse models is chosen as the initial pool and made to play a large number of games with each other until they stabilize.
- New models are incrementally added to the pool and begin by playing many games against the stable models from the initial pool.
- A hard-coded baseline model that waits and shoots the ball is maintained at a rating of 1500 to prevent deflation of ratings.

Models saved at every checkpoint are evaluated via the above scheme. This allows us to evaluate the improvement of a model through its training.

4. Experimental results

4.1. REINFORCE vs PPO

Our initial experimental approach utilized the REINFORCE algorithm (Williams, 1992). The policy network was implemented as a standard two-layer Multilayer Perceptron (MLP). Training with this configuration proved insufficient: the resulting policy rarely exhibited fundamental game mechanics, frequently failing to achieve the basic objective of kicking the ball and demonstrating an inability to score.

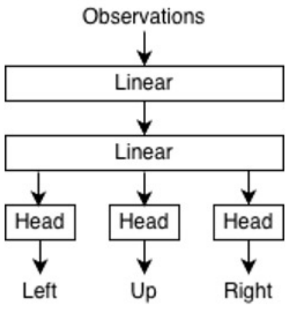

Figure 2. REINFORCE Architecture

This outcome indicated a critical issue of reward sparsity. Despite designing an environment where even minor successes, such as kicking the ball, were rewarded, the overall objective function remained too challenging for a pure policy gradient method without baseline variance reduction.

To address this, we transitioned to the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017), a state-of-the-art actor-critic method. This required a modification to the network architecture - we introduced a value head to the existing two-layer MLP. Crucially, the policy and value functions share all latent layers of the network. This design allows the network to learn a single set of features applicable to both action selection and state-value estimation tasks.

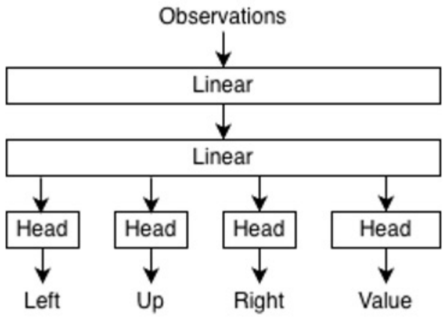

Figure 3. PPO Architecture

The adoption of the PPO framework yielded a significant improvement in training stability and policy performance. The agents demonstrated a sufficient level of competence, reliably executing goal-scoring maneuvers. This breakthrough allowed us to begin quantitative evaluation of policies using the ELO rating system due to the models' ability to consistently win points within the league-training environment.

4.2. League System

In a self-play environment, the method by which an opponent is selected for the current agent is critical for mitigating the problem of a non-stationary environment and ensuring continuous skill improvement. The core of our training progression lies in the evolution of this opponent sampling strategy.

Initially, we implemented a baseline opponent selection strategy based on the recent training history. A parameter C was defined, and the opponent was uniformly sampled from the C most recent model checkpoints saved at fixed episodic intervals. While this approach introduced a minimal degree of opponent variation, preventing the agent from overfitting to a single stationary policy, it was found to be inefficient and unstable. The uniform random sampling failed to strategically challenge the agent, often leading to wasted training episodes against opponents that were either significantly weaker or trivially similar to the current policy.

To achieve more robust and accelerated learning, we adopted an advanced Adaptive League Training strategy, inspired by the framework used in TStarBot-X (Han et al., 2020), a core component of the AlphaStar system (Vinyals et al., 2019).

This optimized system replaces uniform sampling with a prioritized selection mechanism designed to balance exploitation and exploration within the opponent league. The system tracks the current model's win/loss record against various previously saved checkpoints in the league, and opponent sampling is prioritized towards policies against which the current agent has a low win rate). By focusing training against ”hard-to-beat” opponents, the agent is continuously forced out of local optima. This adaptive methodology effectively counters policy plateauing and promotes a faster acquisition of high-level competencies. As demonstrated in Figure 4, the adaptive league system results in higher ELO ratings achieved with significantly fewer training episodes, validating the efficacy of the prioritized sampling technique.

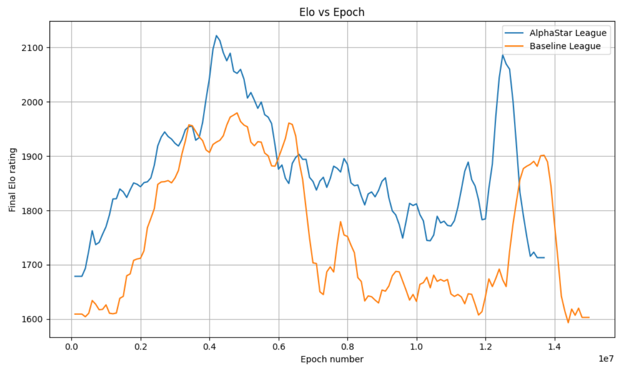

Figure 4. Performance of AlphaStar League against Baseline. Delta variances: 5983.12 (AlphaStar) vs 10617.98 (Baseline)

4.3. Reward Shaping

To facilitate the initial discovery of goal-scoring policies and stabilize training, we attempted reward shaping by introducing a set of auxiliary, extrinsic rewards based on low-level in-game heuristics. The shaped reward function, $\tilde{r}$, was constructed as a weighted linear combination of the extrinsic components and the terminal goal-scoring reward:

$$\tilde{r} = w_S S + w_K K + w_M M - w_J J$$

where the variables are constructed as follows:

- **Score** (S) - which tells us if the model has scored a goal.
- **Kick** (K) - which tells us if the model kicks the ball.
- **Move** (M) - which tells us if the model moves in the direction of the ball. We found this variable to be useful as it encourages the model to try out trajectories where it is closer to the ball and has a better chance of scoring, at least initially.
- **Jump** (J) - which tells us if the model has jumped. In the initial runs, we observed that trained models seemed to jump a lot - since it has an opportunity to jump at each timestep, even with a low probability, it is likely to end up jumping needlessly over a short period of time. Hence this penalty is introduced to discourage this behavior, which looks unnatural and also strategically suboptimal as it makes the player less ready to block shots.

Experimental results suggested taking $w_S = 100$, $w_k = 10$, $w_M = 0.1$, $w_J = 1$. The introduction of this shaped reward function resulted in a noticeable improvement in training efficiency and policy quality. As demonstrated in Figure 5, the agents trained with $\tilde{r}$ achieved higher peak ELO ratings and reached competence thresholds in fewer training steps compared to the baseline reward function ($r = S$).

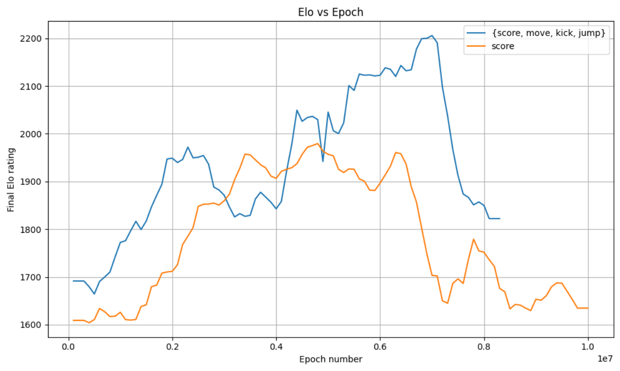

Figure 5. Performance of reward shaping (blue) against sparse rewards (orange)

4.4. Model Architecture - Failures and Insights

As observed from trial games by trained models, we identified that the difficulty of the game lies in the fact that it involves strategy and high-level planning, together with fine control involving the game mechanics. Drawing inspiration from existing literature in robotics involving hierarchical neural networks, such as the FuN architecture (Vezhnevets et al., 2017), we design and test out a similar model architecture. As illustrated in the diagram below, the network consists of a "manager" which devises a plan from the observations, and a "worker" which implements the plan based on its current state and that of the ball. The idea is that the big-picture planning and fine control components of game play can be addressed by separate network components.

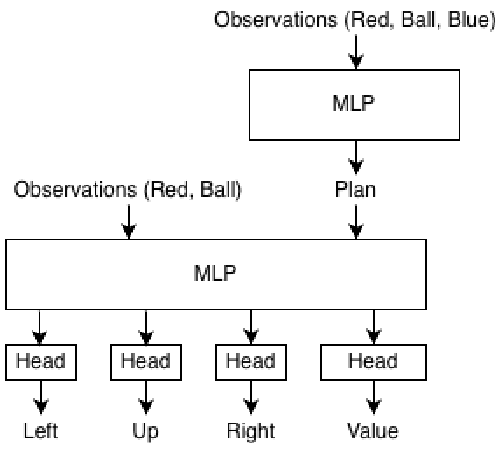

Figure 6. Feudal Architecture

4.4.1. Self-Play

When we trained with the Feudal architecture instead of the 2-layer regular MLP, via all the optimizations described in Sections 4.1-4.3, and the same reward backpropagated through the three decision heads, no signficant changes in performance in terms of ELO are observed beyond the normal stochasticity of training. Both achieved similar peak performances by ELO rating, yet in human play the authors have felt that the model with the Feudal architecture is slightly harder to play against. However this is within boundaries of human preference and such a difference is not seen in our ELO rating system.

4.4.2. Curriculum Learning

The Feudal architecture also allows for curriculum learning through the use of repetitive, single player drills. Drills are simulated situations designed to train a particular type of play, with the second player non-existent in these episodes. This is done via feeding learnt embeddings of drills as plans into the "worker" component of the model, hence training the "worker" to refine techniques such as shooting and blocking. In an attempt to emulate plans the "manager" in the model might make, we created a variety of drills. Each situation comes with a unique reward function - representing the goal of the drill, which is guaranteed to be achievable via design of the starting scenarios. Some of the drills are:

- **Blocking** - the player and ball are initialized in a random position filtered to ensure that the ball is in danger of going into the goal and that the player is within range of being able to stop it. The rewards are only for having prevented a goal.
- **Shooting** - the player and ball are initialized in random positions, the rewards are for being able to score a goal, with a constant negative reward to encourage scoring as quickly as possible.
- **Shooting targets** - the player and ball are initialized in random positions, and a target location on the opposing goal is chosen. Rewards are now the same as the shooting drill but with the scoring penalty adjusted to be dependent on distance from target location.

Unfortunately, while singular models trained on singular drills can learn to perform quite well on that specific task (e.g. shooting), we were unable to observe significant improvement on general play using this method, with training collapsing usually after a certain threshold. This is suspected to be due to the difficulty in learning a good embedding space which encodes all drills in a suitable manner for the "worker" to interpret.

4.4.3. Goal Embedding and Intrinsic Rewards

Reflecting on the failure of curriculum learning implemented as described above, the authors attempted to replicate the goal representation and separation of instrinsic and extrinsic rewards exactly as in FuN (Vezhnevets et al., 2017).

- Goal representation: a latent embedding z_t of size 16 of the game state is learnt, and the goal g_t, also of size 16, is the target value of $z_{t+c} - z_t$ for some horizon length c for the worker to achieve.
- Instrinsic rewards: instead of the "extrinsic" rewards from the environment, only instrinsic rewards backpropagate through the worker. Instrinsic rewards is defined to be the cosine similarity between the actual realized value of $z_{t+c} - z_t$ and the original goal g_t.

Upon training, this policy framework initially showed promise as it demonstrated an ability to differentiate between different "regimes" of game state - one of the saved checkpoint models early in training followed roughly the simple heuristic of trying to go towards and shoot the ball if the opponent is on the wrong side of the ball, and stay put otherwise. This awareness of game state in such a clear cut manner was not demonstrated in other previous models obtained. However, training performance plateaued and the model was not able to obtain performance comparable to the best models from previous experiments. One possible reason for this is insufficient diverse data points for the model, only via self-play, to learn both the implementation of techniques in every scenario as well as the decision making as to which plan to adopt.

5. Conclusion and Future Work

By choosing a suitable reinforcement learning algorithm and implementing a league system and reward shaping via heuristics, we are able to train a policy that achieves, and somewhat exceeds, human performance. This policy is hosted online at `https://cheukhei-chu.github.io/pen-football-rl/`.

Empirical trials of humans playing against said policy show that it would take humans upwards of a few hours of game play in order to have a slight chance of beating it. This is significant since the model took downwards of an hour to train from scratch, on a simple laptop with a CPU.

However, this is still far from the theoretical optimal policy and has weaknesses that, although hard to target, can be exploited. Hence, some areas of future work include:

- Trying to solve weaknesses of the model via targeted curriculum learning
- Finding better model architectures or utilizing scaling law to obtain maximally performant models via the same setups
- Designing a league system that incorporates players of diverse play styles and lets them learn from each other most efficiently
- Tackle the challenge of online adaptive strategization when a policy plays against another several points in a row

References

Fenner, T., Levene, M., and Loizou, G. A discrete evolutionary model for chess players' ratings. *arXiv preprint arXiv:1103.1530*, 2011. URL `https://arxiv.org/abs/1103.1530`.

Han, L., Xiong, J., Sun, P., Sun, X., Fang, M., Guo, Q., Chen, Q., Shi, T., Yu, H., Wu, X., and Zhang, Z. Tstarbot-x: An open-sourced and comprehensive study for efficient league training in starcraft ii full game. *arXiv preprint arXiv:2011.13729*, 2020. URL `https://arxiv.org/abs/2011.13729`.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017. URL `https://arxiv.org/abs/1707.06347`.

Scratch. Pen football, 2016. URL `https://scratch.mit.edu/projects/103746364/`. [Accessed 10-12-2025].

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning, 2017. URL `https://arxiv.org/abs/1703.01161`.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., and et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL `https://doi.org/10.1007/BF00992696`.