

Balancing Player Experience without Players

—
Case mo.co

Markus Ojala
2025-10-08





SUPERCELL

A BRIEF HISTORY OF SUPERCELL

2010

BEGINS AS
A STARTUP
IN ESPOO,
FINLAND



2012



2014



2016



2018



2024



2025



mo.co

- Multiplayer PvE
- Whack monsters, collect loot and XP with friends or randos



Motivation: Why Reinforcement Learning?

The current use case for the bots in mo.co is **Game Balancing**.

Massive-Scale Testing: Train thousands of AI bots to play the game to test new gear, content, and features.

Exploit Discovery: Quickly find unintended strategies or "exploits" that break the game's balance.
Speed: We need results fast – preferably within hours or by the next day – to keep up with development.

No Data, No Problem: Imitation learning wasn't an option due to the lack of gameplay data. RL provided a path forward.

RL for Balancing in Action

Effective Workflow: For each content update, we run balancing tests for 1-2 weeks.

Rapid Feedback Loop:

1. Get results back in hours to days.
2. The design team adjusts gear & enemy stats and level difficulty.
3. Rinse and repeat.

Fight

Exploit?



D

D

Implementation: The AI Simulator

- **Headless Game Client:** We use a separate multithreaded parallel build of the game client with rendering disabled.
 - Enables O(1000) parallel envs per GPU machine
 - Parallelism is essential for scaling RL effectively
- **Full Game Access:** We can directly access game code (C++) to define precise observations and rewards based on any in-game event.
- **Python Bindings:** `pybind11` connects the C++ simulator to our Python-based training environment.

The Training Framework

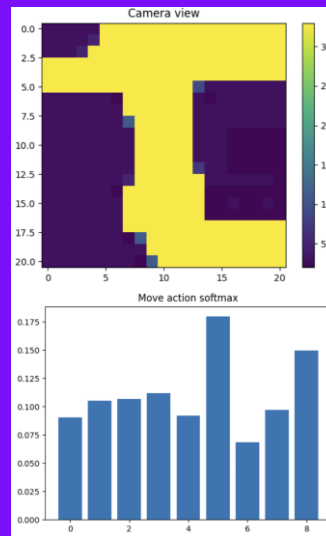
Simple Setup:

```
>>> pip install ai_simulator
```

Pseudocode:

```
import ai_simulator
...
obs, reward = ai_simulator.step(action)
action = agent(obs)
...
```

Small in-House RL Library: We currently use a custom in-house library built on PyTorch. Exploring commercial/open-source options like Ray RLLib is also on the radar.



```
obs = TensorDict({'player': tensor([[[[ 1., 1., 0., ..., 0.],
> special variables
> function variables
  requires_grad = False
> 'player' = tensor([[[[ 1., 1., 0., ..., 0., -1., 0.],
> 'player_cat' = tensor([[[[ 2401, 2419, 1300001, ...,
> 'player_button_masks' = tensor([[[[ True, True, True, Tru
> 'player_team' = tensor([[[[ 0.0000, 0.0000, 0.0000],
> 'team' = tensor([[[[ 1., 0., 1., ..., 1., 0., 0.],
> 'team_cat' = tensor([[[[ 2401, 2419, 1300001, ...,
> 'player_enemy' = tensor([[[[ 3.5853e+00, 2.8786e+00, 3.685
> 'enemy' = tensor([[[[ 0., 0., 1.],
> 'player_enemy_cat' = tensor([[[[2401, 2413, 2401, 2413],
> 'players.position' = tensor([[[[63.5000, 99.5000],
> 'characters.position' = tensor([[[[63.5000, 99.5000],
> 'characters.rotation' = tensor([[[[0., 0., 0., ..., nan, na
> 'characters.health' = tensor([[[[ 1700., 1700., 1800.,
> 'characters.max_health' = tensor([[[[ 1700., 1700., 1800
> 'characters.type' = tensor([[[[800000, 800000, 800000, ...,
```

Details: Model & Algorithm

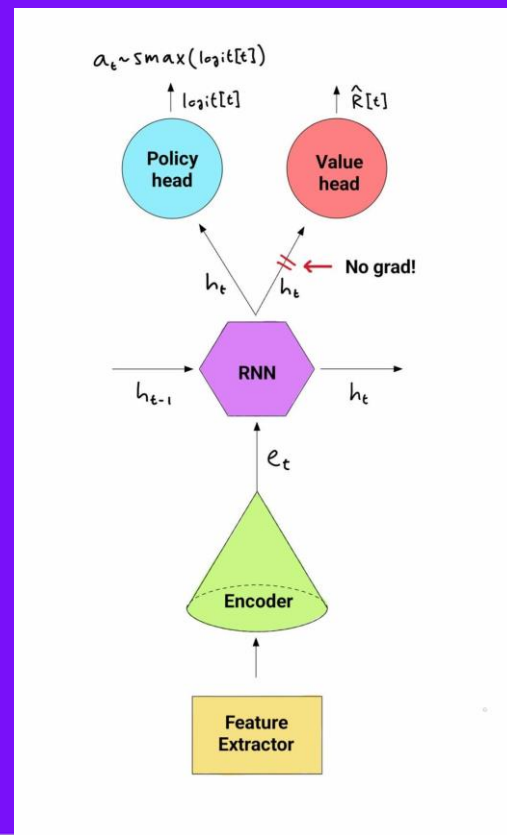
Model Architecture:

- Encoder
- RNN (GRU)
- Policy head and *detached* value head

Algorithm: Proximal Policy Optimization (PPO)

$$\mathbb{E}_{s, a \sim \pi_{\text{old}}} \left\{ \frac{\nabla \pi(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(s, a) \right\}$$

How it works: Collect experience for T steps (rollout), then update the policy using ~ 5 (sequential) minibatches.



Why PPO Clipping is So Important

RL updates can be unstable. A single large reward can drastically change the policy, often for the worse.

Example Scenario (no PPO):

1. Agent finally kills a boss and gets a huge reward.
2. The policy update drastically increases the probability of the "killing blow" action.
3. This large update inadvertently lowers the probability of a critical early-game action (e.g. taking the correct path forward).
4. The agent now fails to even reach the boss in the next run.

PPO's Solution: Clipping the loss function limits how much the policy can change in a single update, leading to more stable learning.

Training Strategy: Pre-training & Fine-tuning

Pre-train: A general model is trained on all maps with random gear.

Fine-tune: This general model is then specialized for a specific scene.

Result: Cuts down total training time to just 10-20% of training from scratch.

Gear Optimization: During **fine-tuning**, we use a simple **Genetic Algorithm** to find the optimal gear combinations for the task.

	Slot 1	Slot 2	Slot 3	...	Slot 12
Gear id	13	11	0	...	31

The Core Challenge: Credit Assignment

The Credit Assignment Problem: A battle can last up to 5 minutes. If the only reward is for completing the dungeon, how does the agent know which of the thousands of actions it took were actually helpful?

No-brainer Solution: **Reward Shaping**

- Give a dense structure of smaller, auxiliary rewards to guide the agent

Reward Hierarchy:

1. Staying Alive (small reward)
2. Hitting Monsters
3. Killing Monsters
4. Hitting a Boss
5. Killing a Boss
6. Completing the Dungeon (largest reward)

WORKS GREAT... if the reward scales are just right!

**SUP
ERC
ELL**



The Problem with Reward Shaping: A Case Study

The "Squid Blades" Gear: This item turns the player invisible if they haven't attacked recently.

Optimal (Human) Strategy: All 4 players should sneak past the monsters and go straight to the boss.

The RL Agent's Strategy: The agents couldn't resist the temptation of the "kill monster" rewards along the way, breaking their invisibility and failing the optimal strategy.



Moving Beyond Complex Rewards?

The Problem:

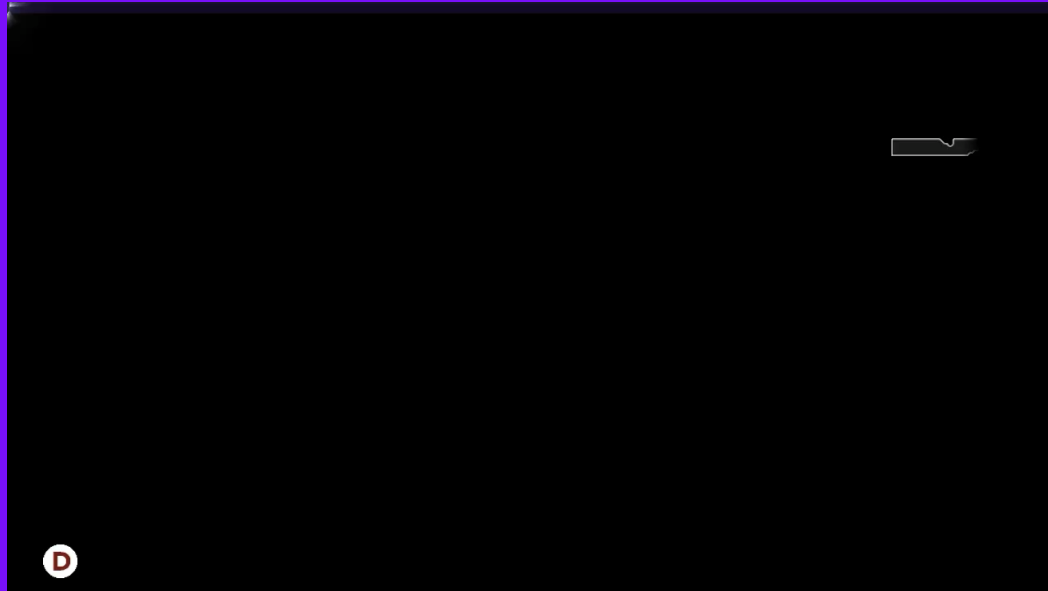
- Balancing numerous rewards is difficult and time-consuming.
- Agents might learn to "farm" auxiliary rewards instead of focusing on the main objective

The Ideal Solution: A minimal reward signal, just for (fast) completion of the game + **exploration**

Experiment: Just use these rewards:

Boss Damage, Boss Kill, Staying Alive

- Using just these in the fine-tuning phase => sometimes learns the squid blade, *but often can't even find the boss...*



The New Challenge: With sparse rewards, **Exploration** becomes the most critical problem to solve. How does the agent find the boss in the first place?

How to Encourage Exploration?

What is Curiosity? How do we encourage an agent to be "curious" and try different things?

Common Techniques:

- Entropy Bonus: Add a loss term that encourages the action probabilities to be less certain (more uniform), preventing the policy from collapsing too early
 - usually annealed to zero over time
 - This is very useful in general, at least in mo.co
- Pseudocounts: Penalize the agent for visiting states it has already seen many times. This encourages it to find new, novel states.
 - Works, but only when not too many (or continuous) states
 - "Boredom paint": make tiles seen in camera more and more boring per visit – works well for geographic exploration

The End

Recap:

- We use RL for rapid game balancing in mo.co
- It enables the game designers to quickly validate new concepts and ideas
- Implementing RL is still a challenge technically
- Exploration journey ongoing!

Thank you for your attention!

Questions?