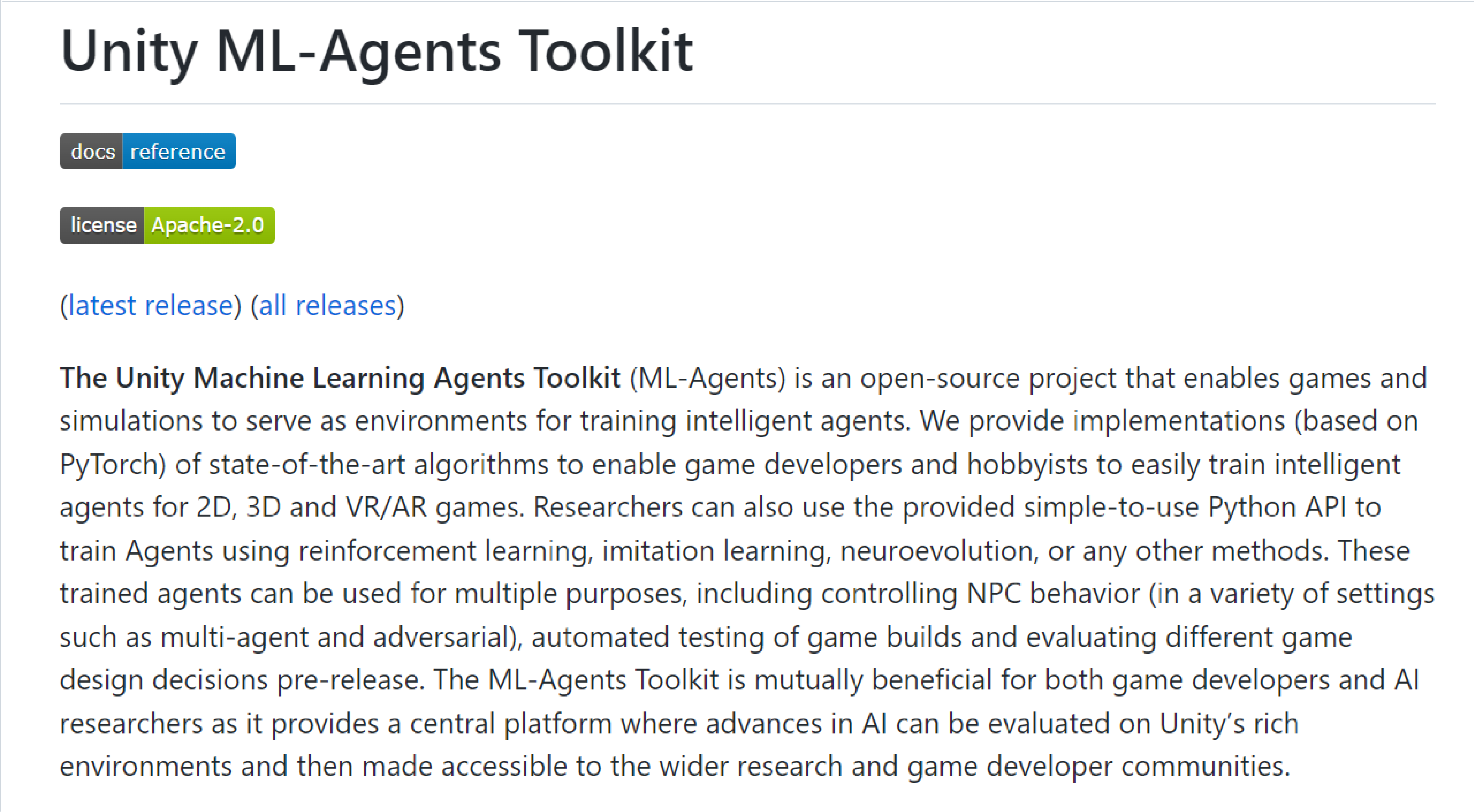
Go to ML Agent Github page

- Google search: Unity ML agent github

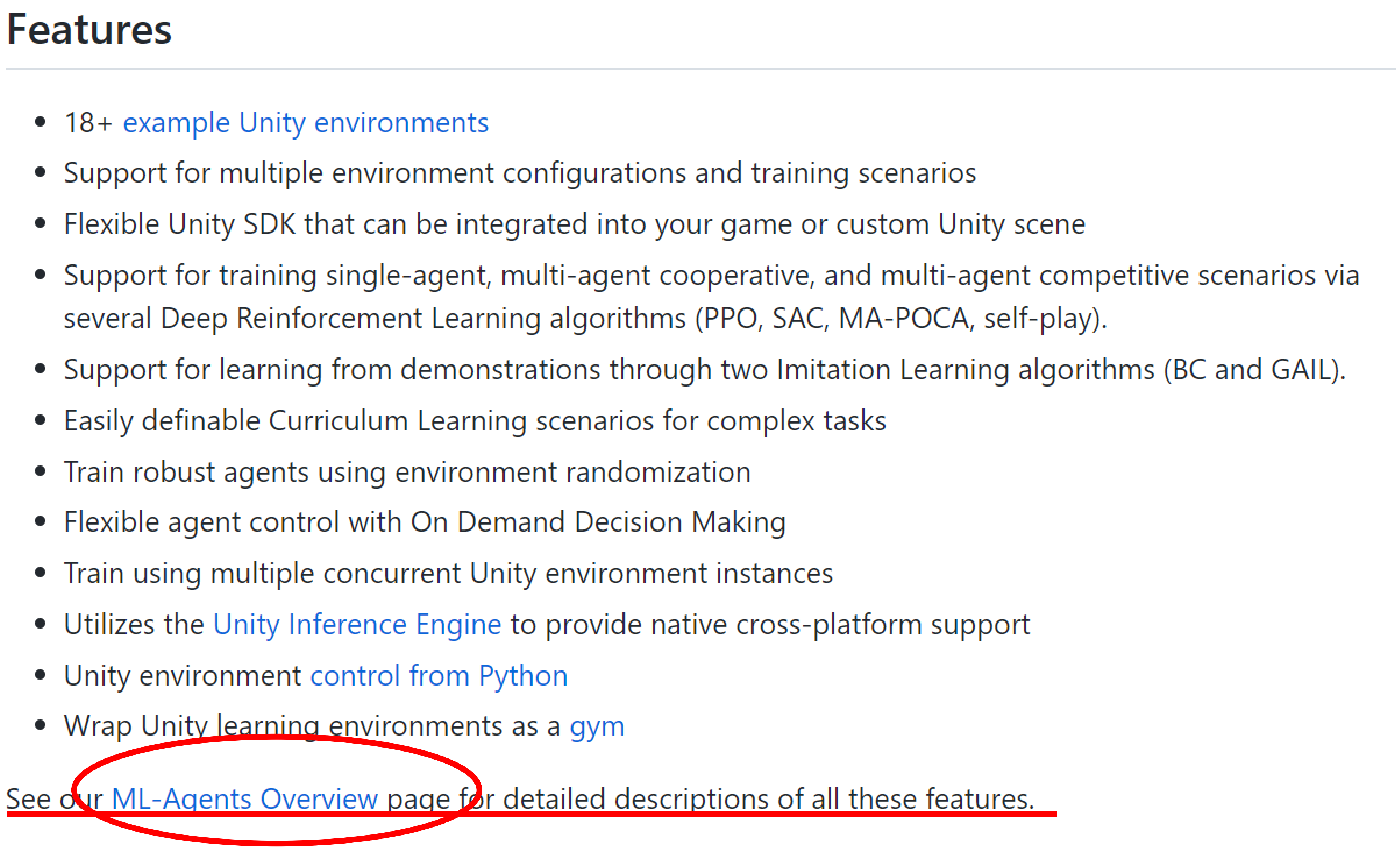


Unity ML-Agents Toolkit

(latest release) (all releases)

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. We provide implementations (based on PyTorch) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. Researchers can also use the provided simple-to-use Python API to train Agents using reinforcement learning, imitation learning, neuroevolution, or any other methods. These trained agents can be used for multiple purposes, including controlling NPC behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release. The ML-Agents Toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity's rich environments and then made accessible to the wider research and game developer communities.

Open ML Agent overview page

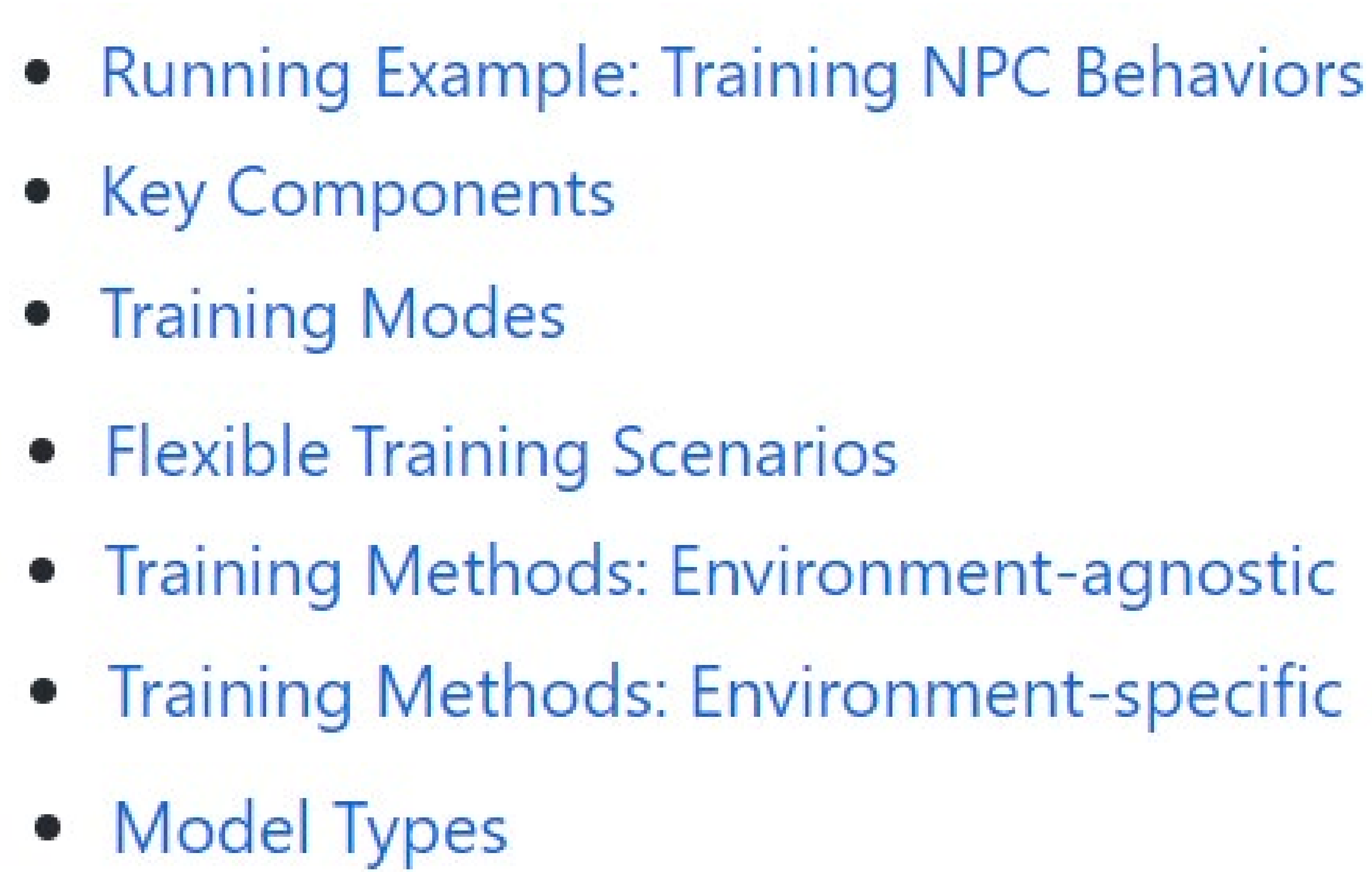


Features

- 18+ example Unity environments
- Support for multiple environment configurations and training scenarios
- Flexible Unity SDK that can be integrated into your game or custom Unity scene
- 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).
- Support for learning from demonstrations through two Imitation Learning algorithms (BC and GAIL).
- Easily definable Curriculum Learning scenarios for complex tasks
- Train robust agents using environment randomization
- Flexible agent control with On Demand Decision Making
- Train using multiple concurrent Unity environment instances
- Utilizes the Unity Inference Engine to provide native cross-platform support
- Unity environment control from Python
- Wrap Unity learning environments as a gym

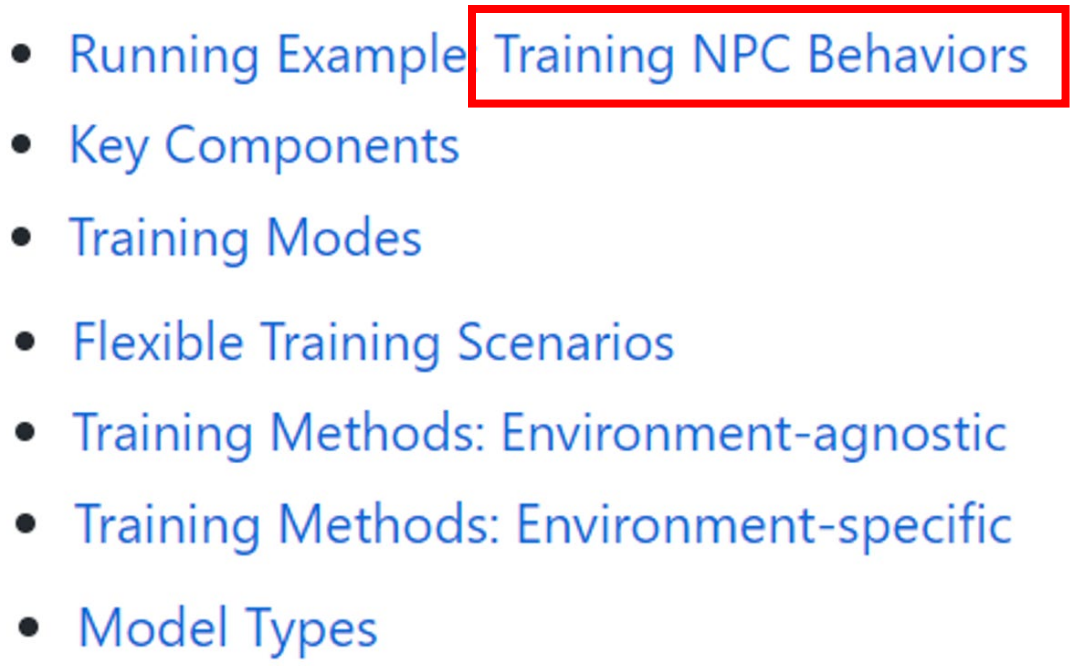
See our ML-Agents Overview page for detailed descriptions of all these features.

Overview of ML Agent

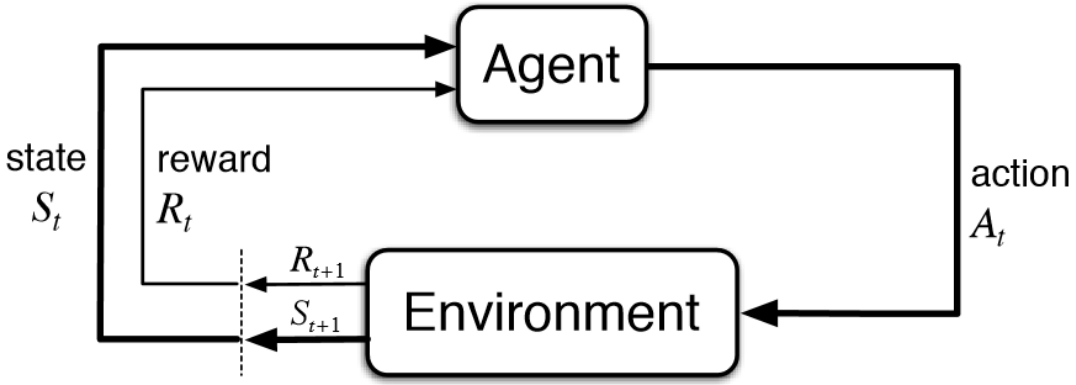


- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

Train an NPC for game and metaverse



- Running Example Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

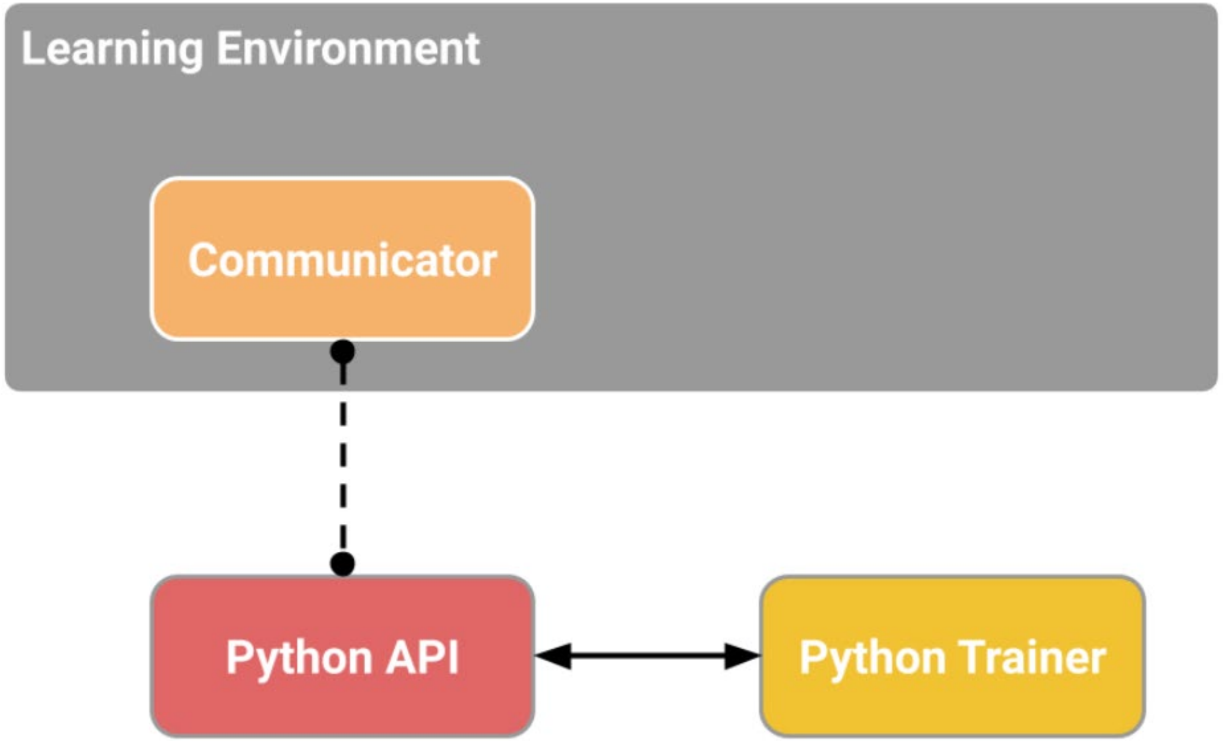


The behavior that the agent learned is called a **policy**, which is essentially a (optimal) mapping from observations to actions.

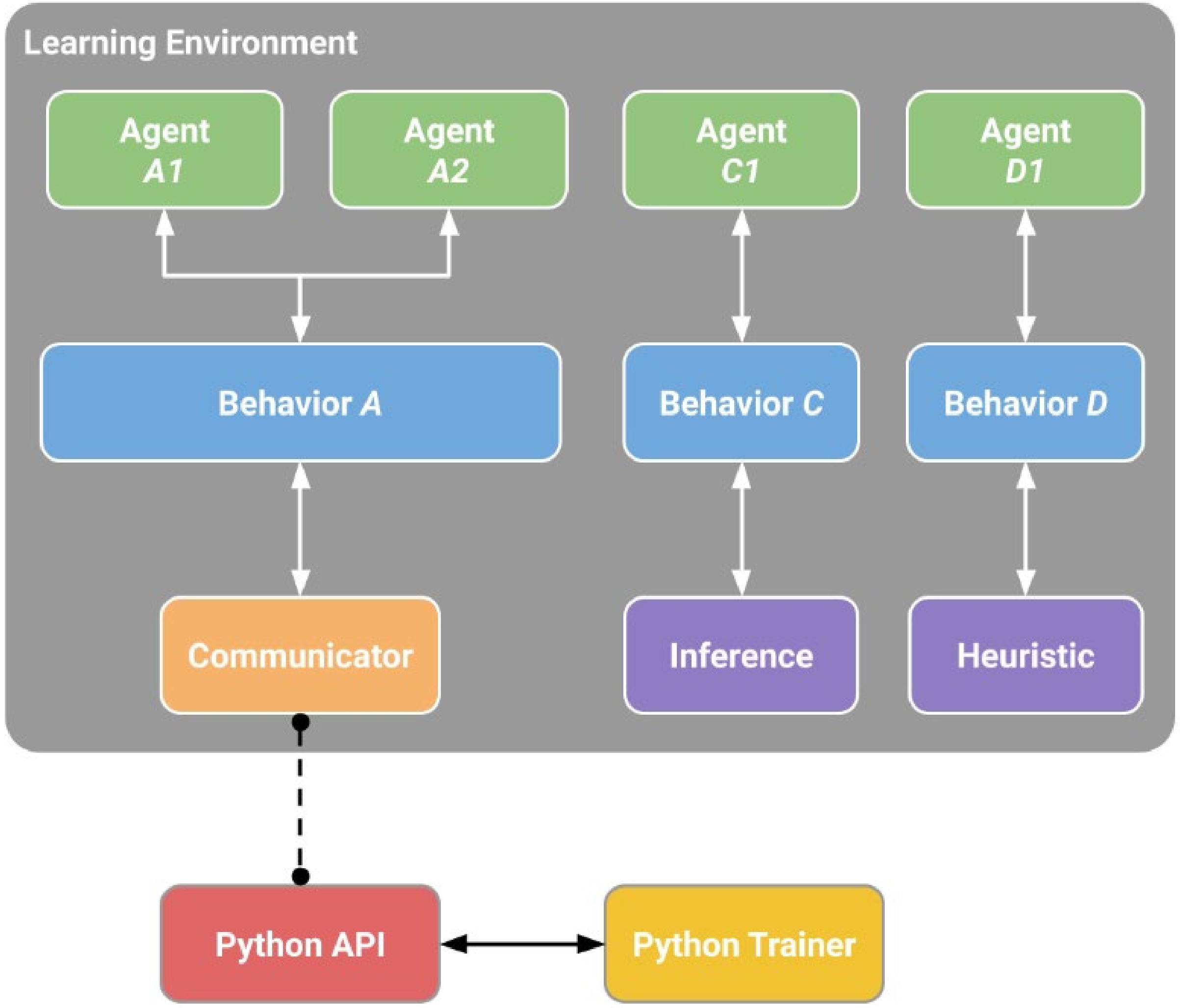
Policies	$a_t \sim \pi_\theta\ (s_t)$
Trajectories	$\tau\ = (s_0, a_0, s_1, a_1, \dots)$
Reward	$r_t\ = R(s_t, a_t, s_{t+1})$
Return	$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$

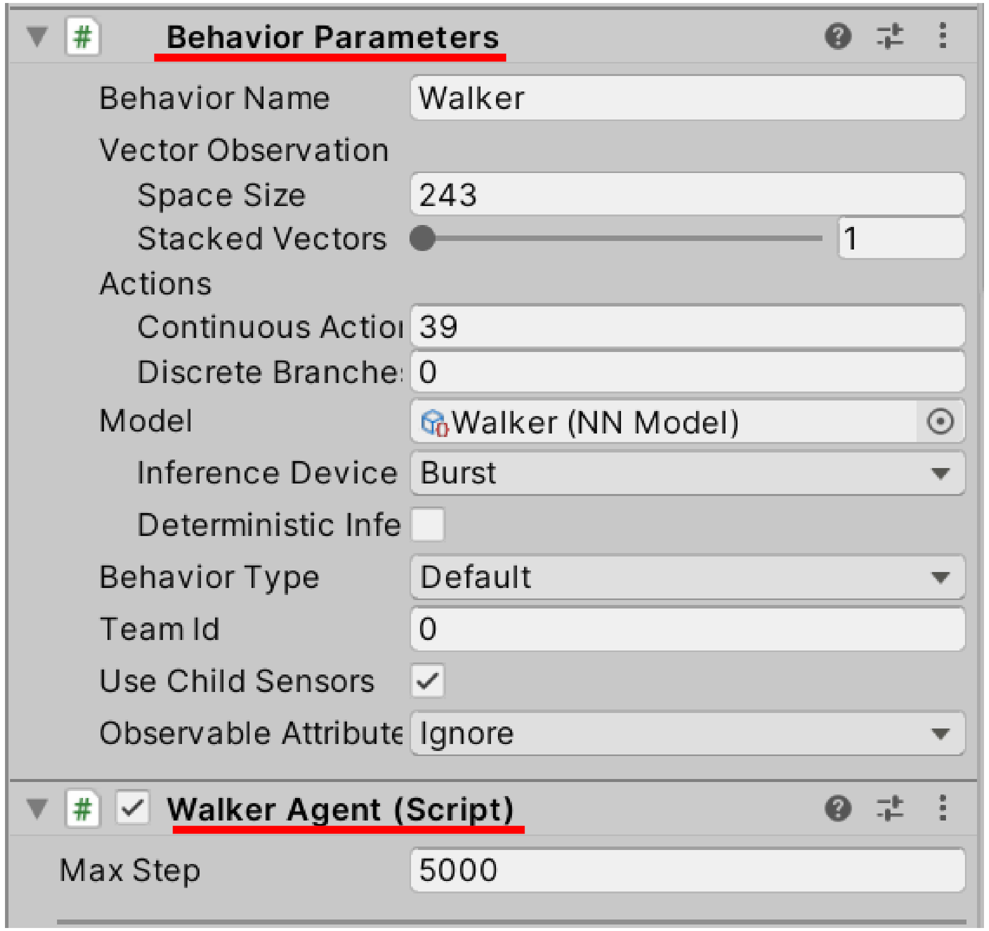
Key components

- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types



Behavior and agent





Behavior Parameters

Behavior Name: Walker

Vector Observation

Space Size: 243

Stacked Vectors: 1

Actions

Continuous Actioı: 39

Discrete Branche: 0

Model: Walker (NN Model)

Inference Device: Burst

Deterministic Infe: □

Behavior Type: Default

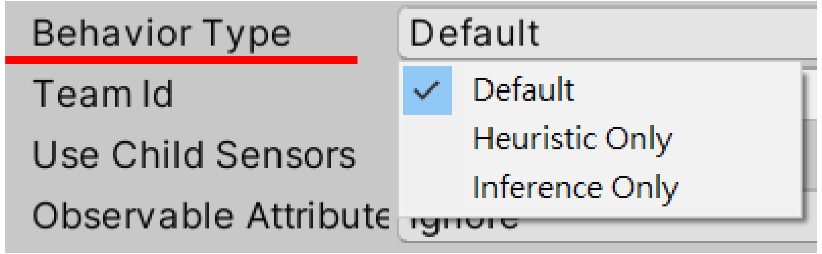
Team Id: 0

Use Child Sensors: ☑

Observable Attribute: Ignore

Walker Agent (Script)

Max Step: 5000



Behavior Type: Default

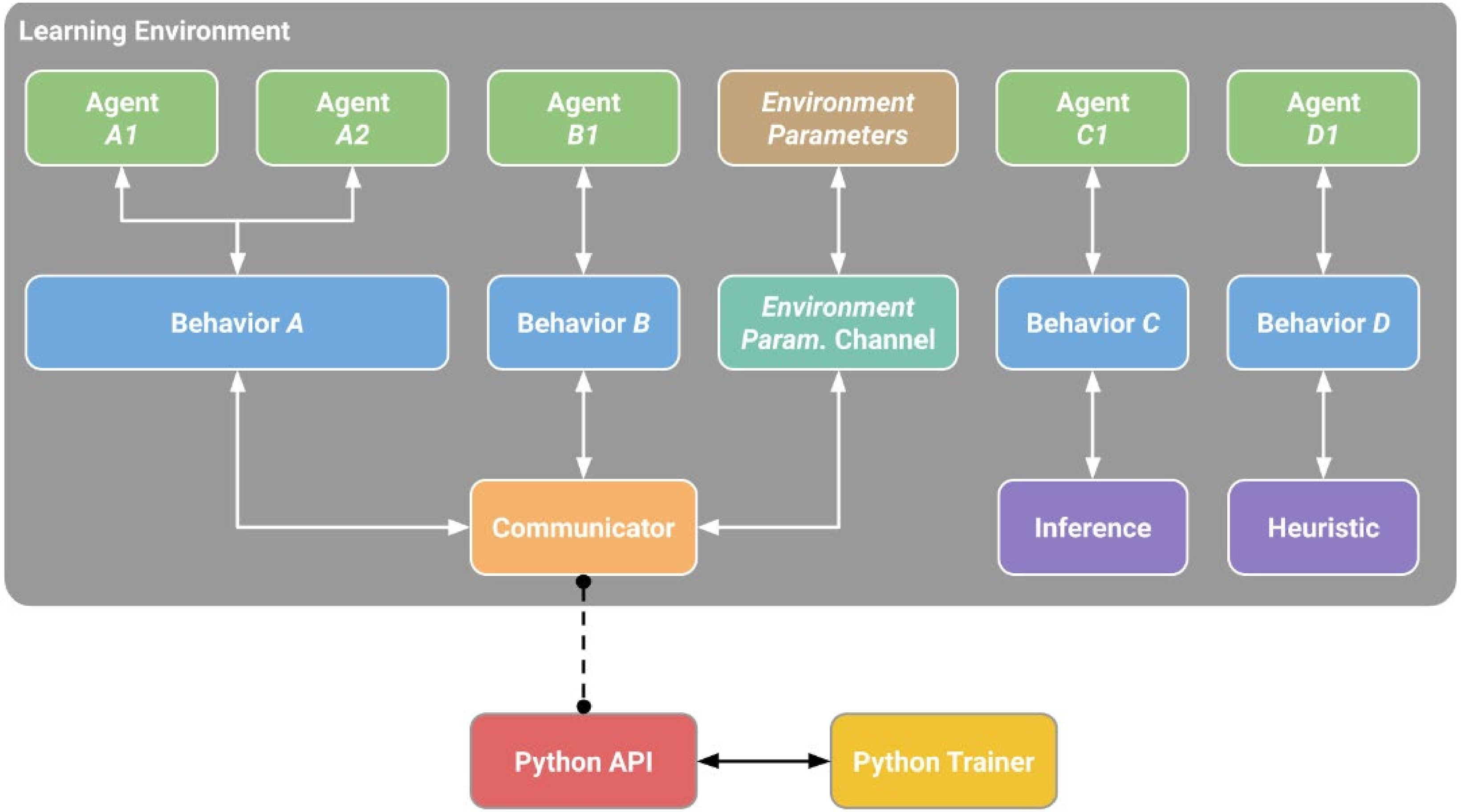
Team Id

Use Child Sensors

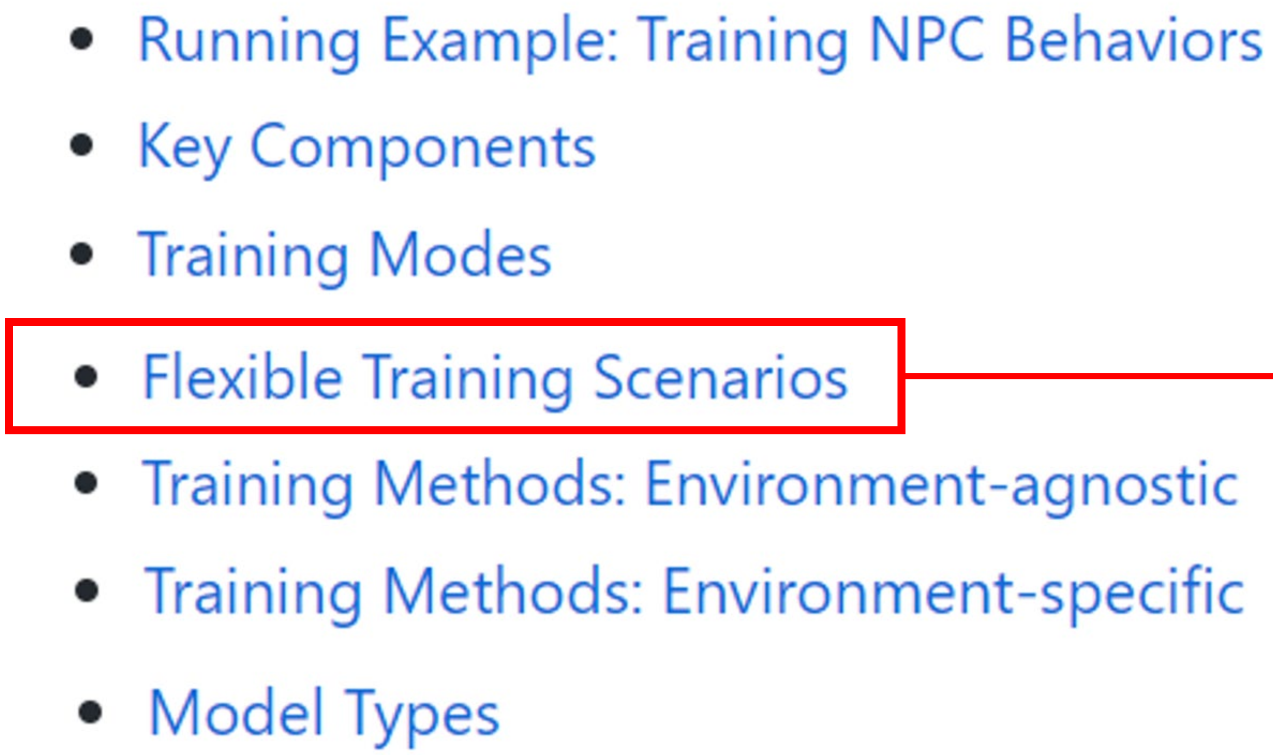
Observable Attribute

ML agent architecture

- An environment can have multiple agents and multiple behaviors.
- We can exchange data between Unity and Python outside of the machine learning loop through Side Channels.



Training scenarios



- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

1. Single agent
2. Simultaneous Single-Agent
3. Adversarial Self-Play
4. Cooperative Multi-Agent
5. Competitive Multi-Agent
6. Ecosystem

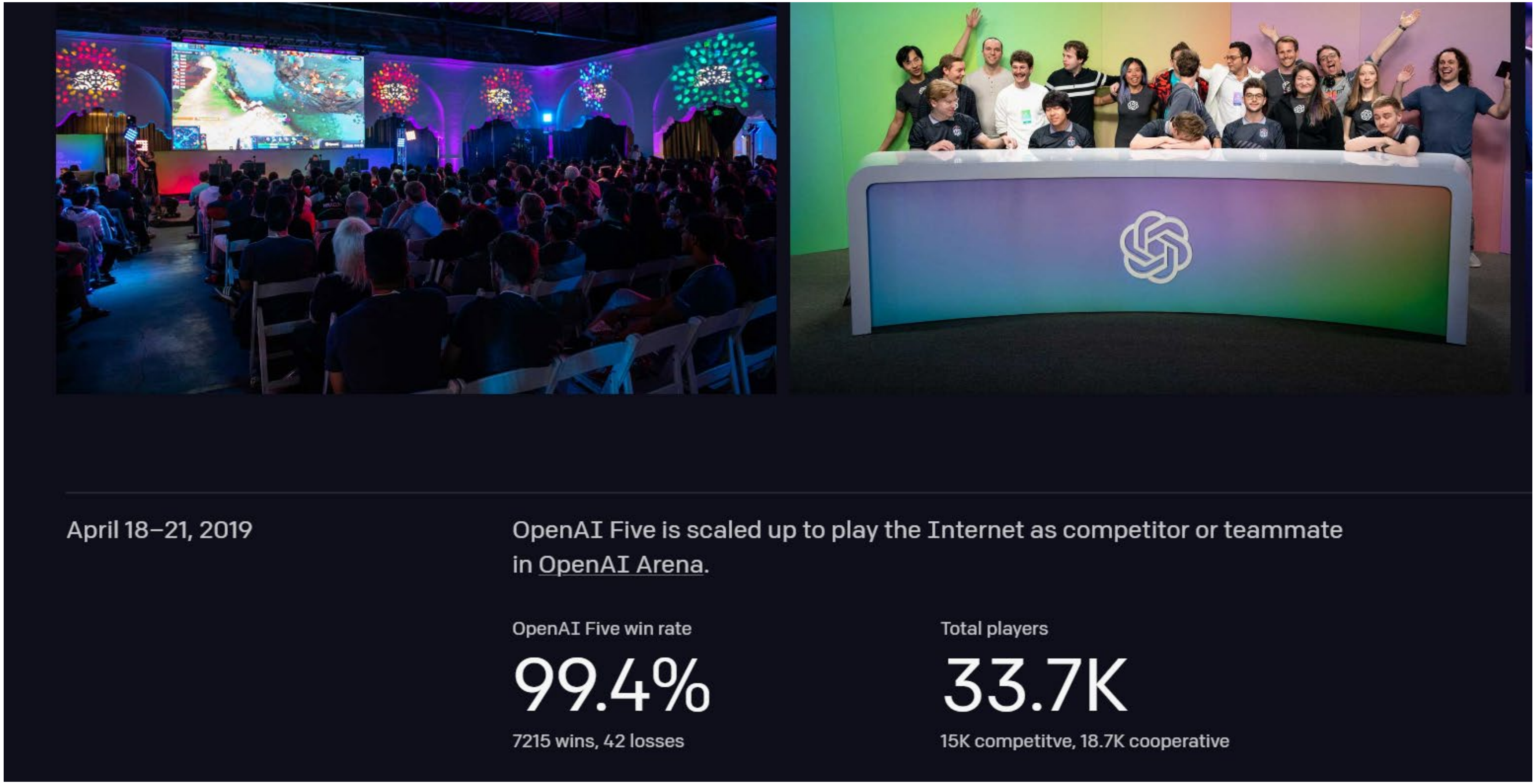
Single agent

- Single-Agent. A single agent, with its own reward signal. The traditional way of training an agent.
- Simultaneous Single-Agent. Multiple independent agents with independent reward signals with same Behavior Parameters. A parallelized version of the traditional training scenario, which can speed-up and stabilize the training process. Helpful when you have multiple versions of the same character in an environment who should learn similar behaviors. An example might be training a dozen robot-arms to each open a door simultaneously.

Adversarial self-play

- Adversarial Self-Play. Two interacting agents with inverse reward signals. In two-player games, adversarial self-play can allow an agent to become increasingly more skilled, while always having the perfectly matched opponent: itself. This was the strategy employed when training AlphaGo, and more recently used by OpenAI to train a human-beating 1-vs-1 Dota 2 agent.

OpenAI Dota 2 agent



April 18–21, 2019

OpenAI Five is scaled up to play the Internet as competitor or teammate in OpenAI Arena.

OpenAI Five win rate

99.4%

7215 wins, 42 losses

Total players

33.7K

15K competitve, 18.7K cooperative

https://openai.com/five/

Cooperative multi-agent

- Cooperative Multi-Agent. Multiple interacting agents with a shared reward signal with same or different Behavior Parameters. In this scenario, all agents must work together to accomplish a task that cannot be done alone. Examples include environments where each agent only has access to partial information, which needs to be shared in order to accomplish the task or collaboratively solve a puzzle.

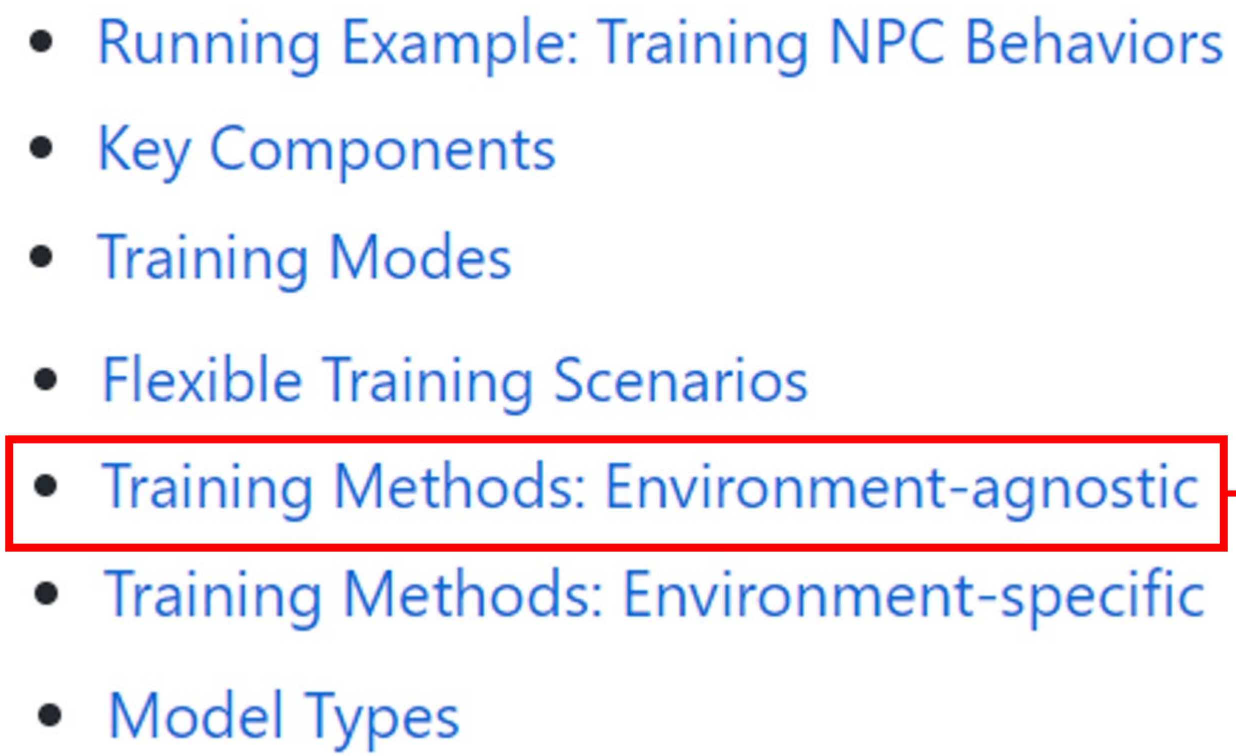
Competitive multi-agent

- Competitive Multi-Agent. Multiple interacting agents with inverse reward signals with same or different Behavior Parameters. In this scenario, agents must compete with one another to either win a competition, or obtain some limited set of resources. All team sports fall into this scenario.

Ecosystem

- Ecosystem. Multiple interacting agents with independent reward signals with same or different Behavior Parameters. This scenario can be thought of as creating a small world in which animals with different goals all interact, such as a savanna in which there might be zebras, elephants and giraffes, or an autonomous driving simulation within an urban environment.

Training methods: Environment-agnostic (不知的)



- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

1. Reward signals
2. Deep RL
3. Imitation learning

Use reward signals in environment-agnostic (不知的)

- Extrinsic: represents the rewards defined in your environment, and is enabled by default
- Intrinsic
  - GAIL: intrinsic reward signal from imitation learning
  - Curiosity: encourages exploration in sparse-reward environments
  - RND: encourages exploration in sparse-reward environments

Reinforcement learning algorithms

ML-Agents provide an implementation of two reinforcement learning algorithms.

- Proximal Policy Optimization (PPO)
- Soft Actor-Critic (SAC)

The default algorithm is PPO. This is a method that has been shown to be more general purpose and stable than many other RL algorithms..

PPO vs SAC

- In contrast with PPO, SAC is off-policy, which means it can learn from experiences collected at any time during the past. As experiences are collected, they are placed in an experience replay buffer and randomly drawn during training. This makes SAC significantly more sample-efficient, often requiring 5-10 times less samples to learn the same task as PPO. However, SAC tends to require more model updates. SAC is a good choice for heavier or slower environments (about 0.1 seconds per step or more). SAC is also a "maximum entropy" algorithm, and enables exploration in an intrinsic way.

Sparse reward

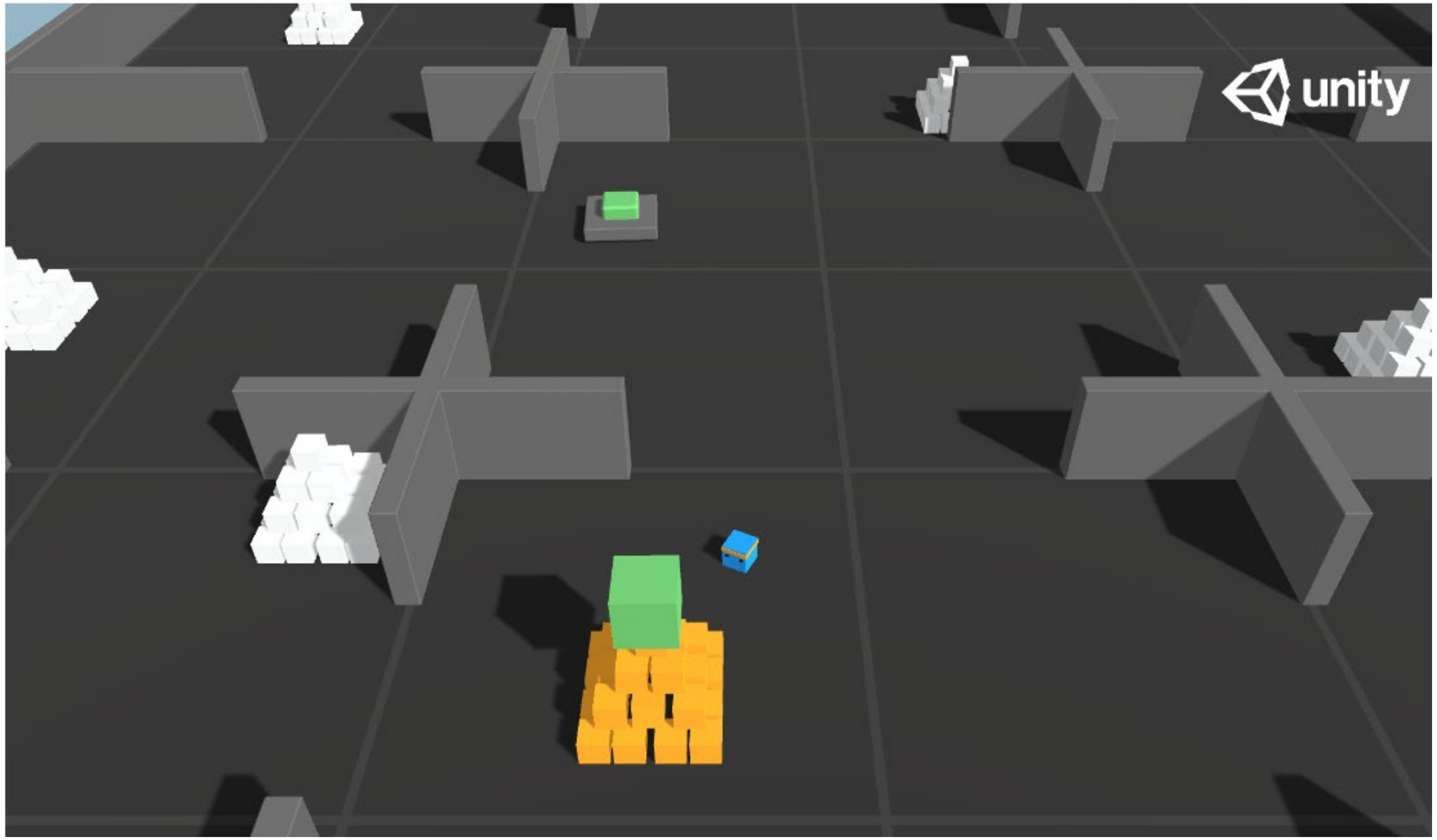
In environments where the agent receives rare or infrequent rewards (i.e. sparse-reward), an agent may never receive a reward signal on which to bootstrap its training process.

- Curiosity
- Random Network Distillation (RND)

Class practice

Pyramids

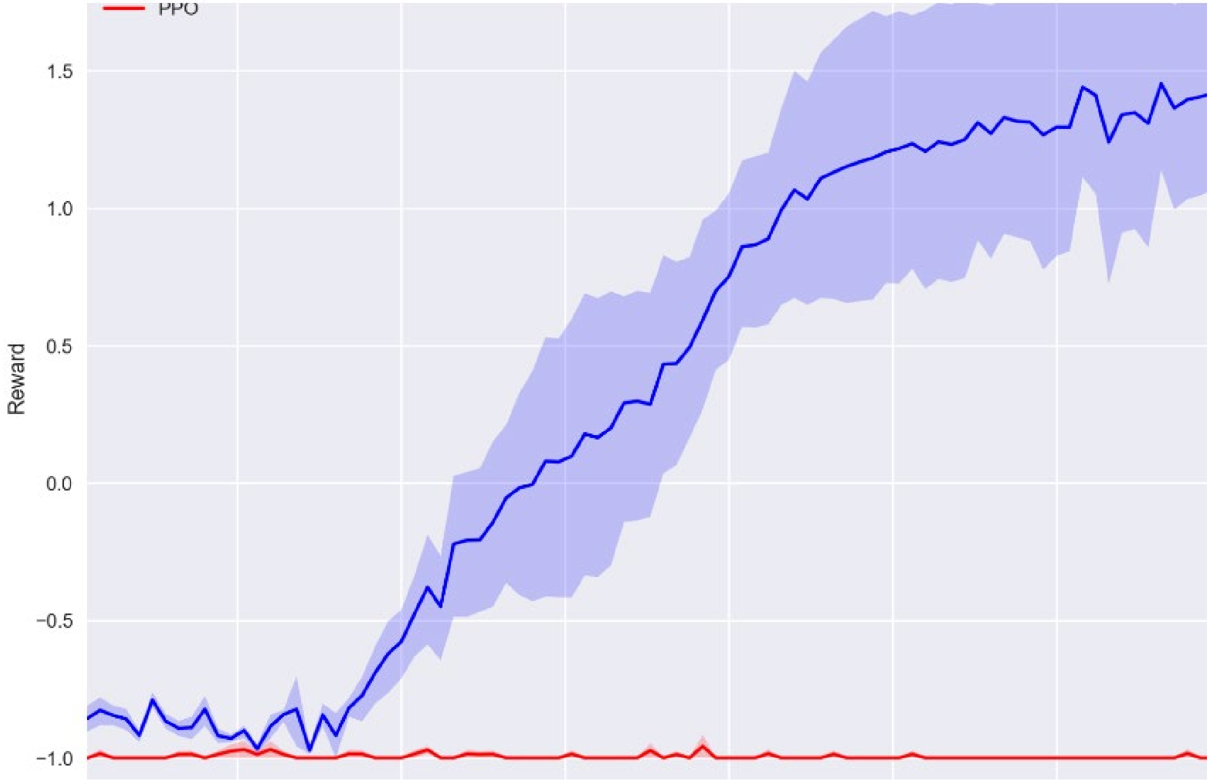
- The agent needs to press a button to spawn a pyramid, then navigate to the pyramid, knock it over, and move to the gold brick at the top.



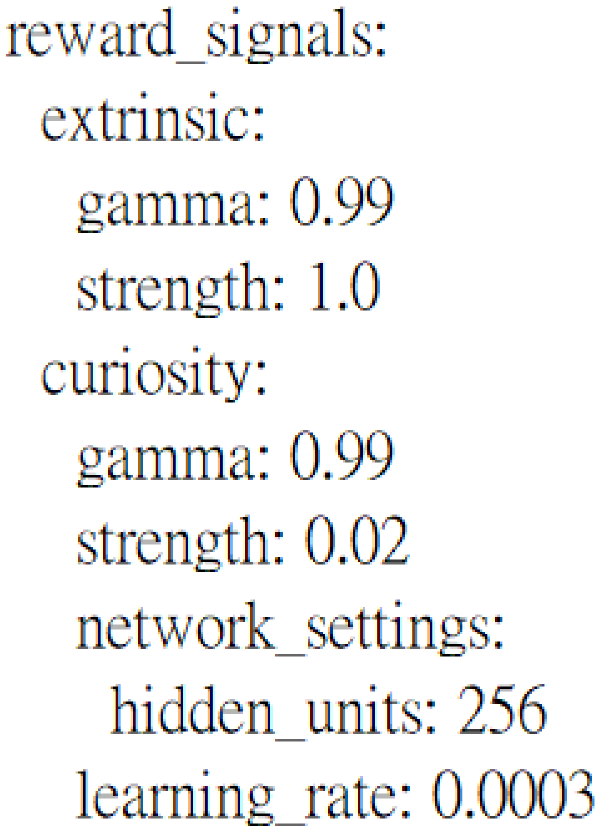
Curiosity

- It trains two networks: 1) an inverse model, which takes s_t and s_{t+1}, and encodes them, and uses the encoding to predict a_t, and 2) a forward model, which takes the encoded s_t and a_t, and predicts the encoded s_{t+1}.
- The loss of the forward model (the difference between s_{t+1} and $\hat{s}_{t+1}$) is used as the intrinsic reward, so the more surprised the model is, the larger the reward will be..

Curiosity



Pyramids.yaml



```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
  curiosity:
    gamma: 0.99
    strength: 0.02
    network_settings:
      hidden_units: 256
    learning_rate: 0.0003
```

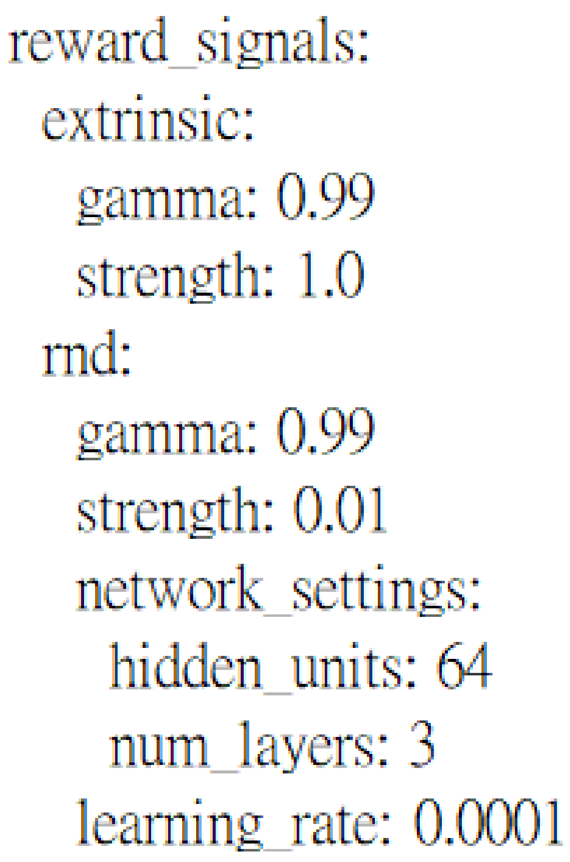
https://blog.unity.com/technology/solving-sparse-reward-tasks-with-curiosity

RND

- RND uses two networks: 1) The first is a network with fixed random weights that takes observations as inputs and generates an encoding, 2) The second is a network with similar architecture that is trained to predict the outputs of the first network and uses the observations the Agent collects as training data.
- The loss of the trained model is used as intrinsic reward. The more an Agent visits a state, the more accurate the predictions and the lower the rewards which encourages the Agent to explore new states with higher prediction errors.

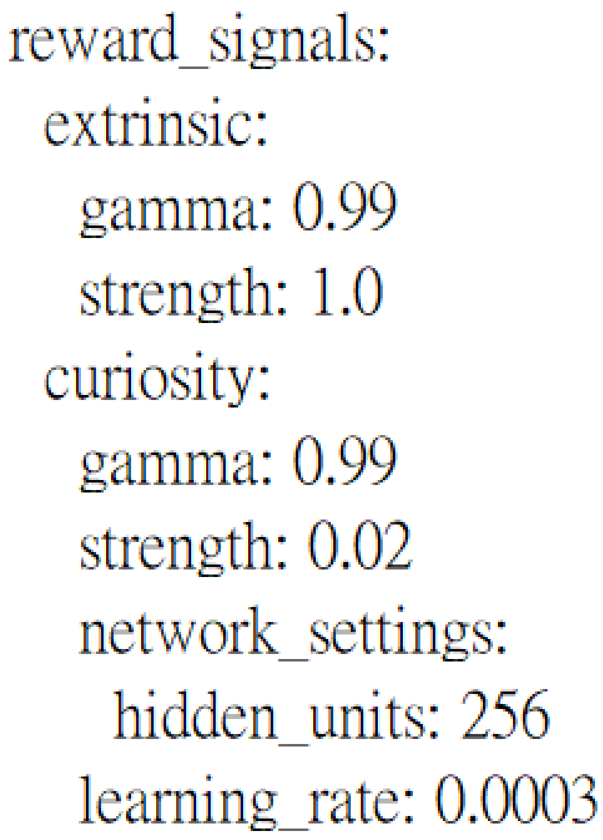
RDN

PyramidsRDN.yaml



```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
  rnd:
    gamma: 0.99
    strength: 0.01
    network_settings:
      hidden_units: 64
      num_layers: 3
    learning_rate: 0.0001
```

Pyramids.yaml



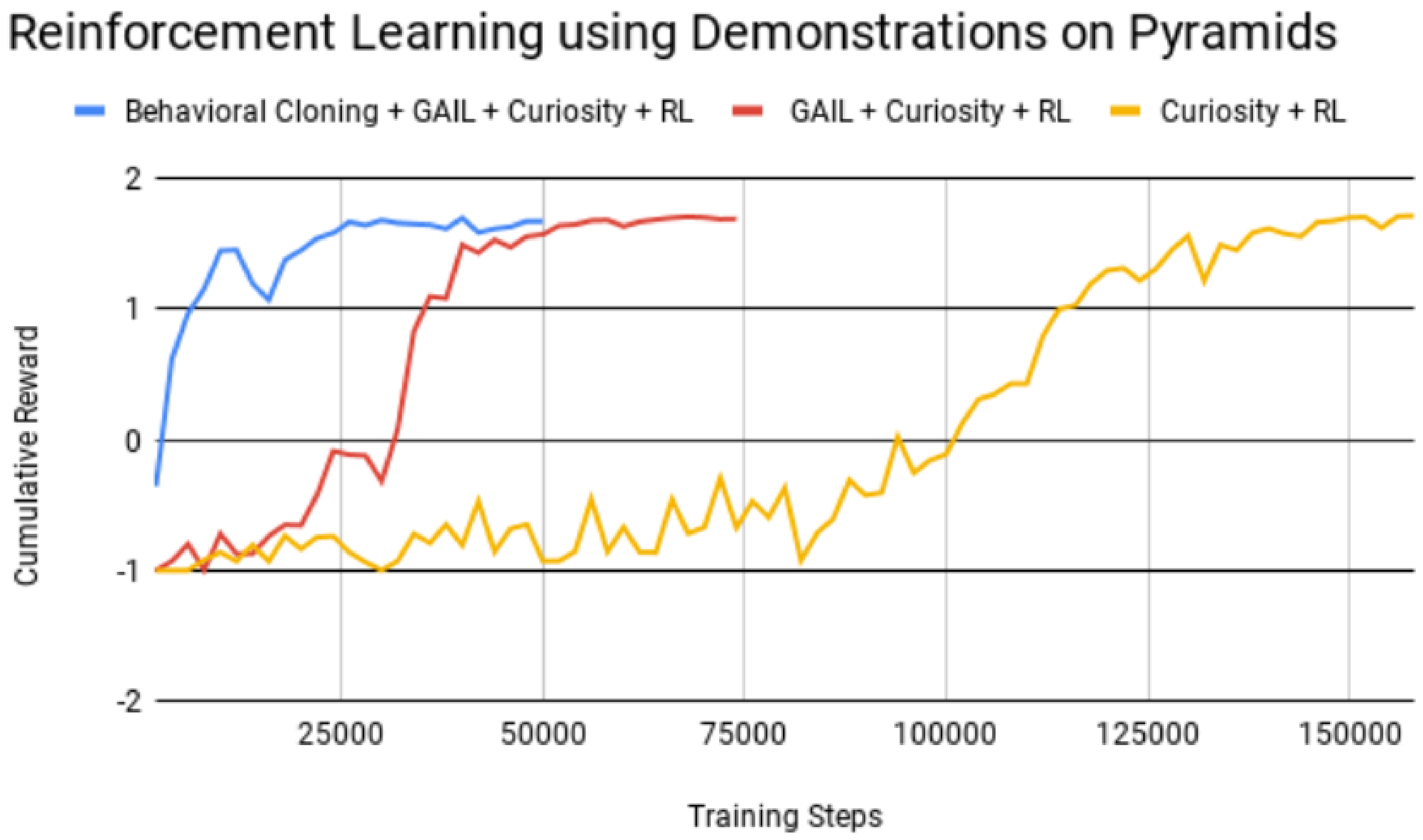
```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
  curiosity:
    gamma: 0.99
    strength: 0.02
    network_settings:
      hidden_units: 256
    learning_rate: 0.0003
```

Imitation learning

It is often more intuitive to simply demonstrate the behavior we want an agent to perform, rather than attempting to have it learn via trial-and-error methods..

- GAIL (Generative Adversarial Imitation Learning)
- Behavioral Cloning (BC)

Imitation learning

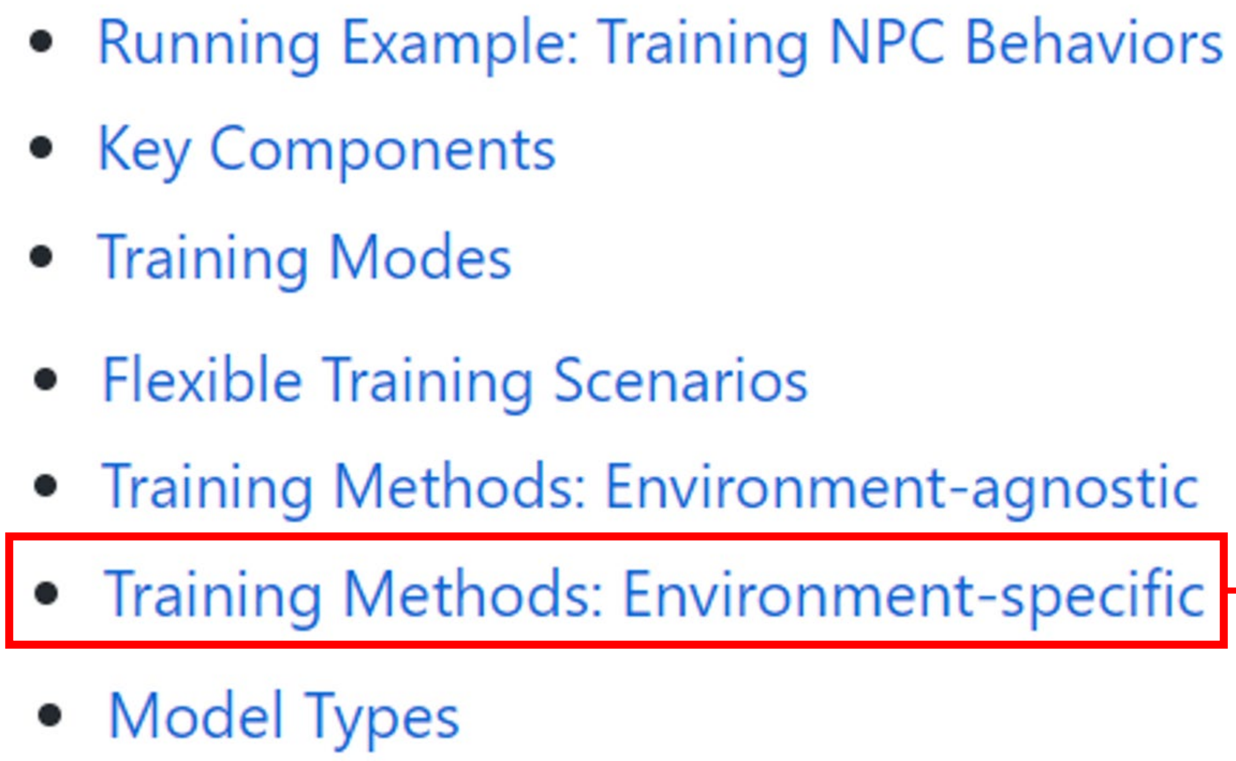


Summary of training methods

To summarize, we provide 3 training methods: BC, GAIL and RL (PPO or SAC) that can be used independently or together:

- BC can be used on its own or as a pre-training step before GAIL and/or RL
- GAIL can be used with or without extrinsic rewards
- RL can be used on its own (either PPO or SAC) or in conjunction with BC and/or GAIL.

Training methods: Environment-specific



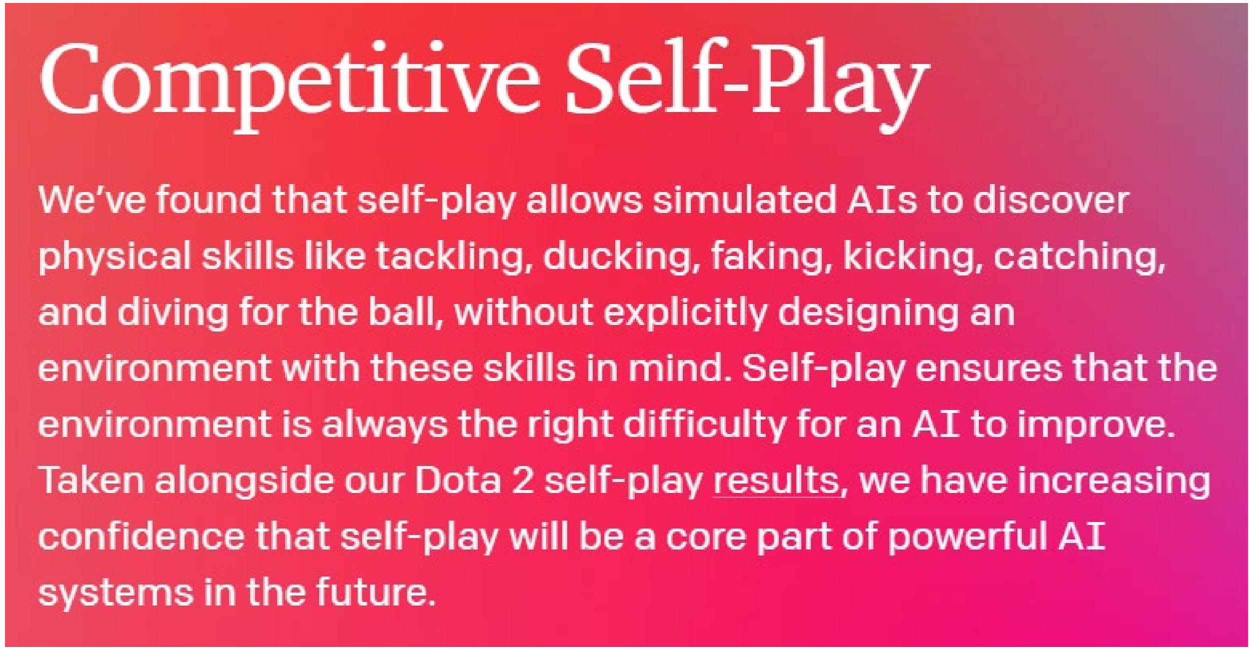
- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

1. Competitive Multi-Agent Environments with Self-Play
2. Cooperative Multi-Agent Environments with MA-POCA
3. Curriculum Learning
4. Environment Parameter Randomization

Competitive Self-Play

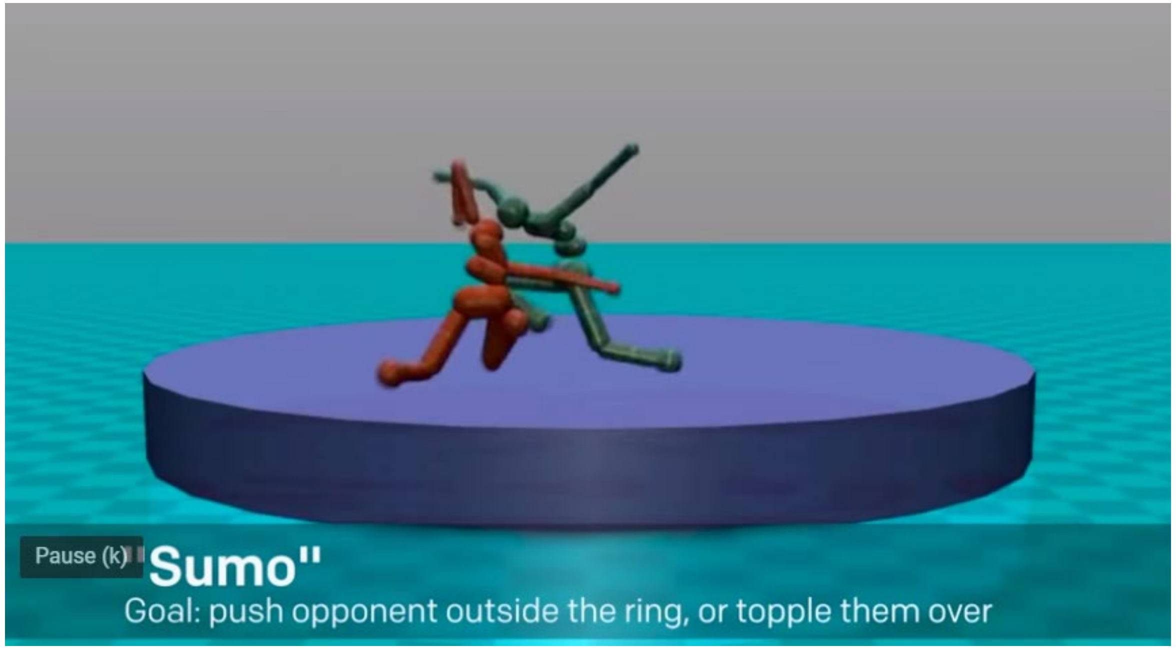
- ML-Agents provides the functionality to train both symmetric and asymmetric adversarial games with Self-Play.
- A symmetric game is one in which opposing agents are equal in form, function and objective (e.g., Tennis and Soccer). In reinforcement learning, this means both agents have the same observation and actions and learn from the same reward function and so they can share the same policy.
- In asymmetric games (e.g. Hide and Seek), agents do not always have the same observation or actions and so sharing policy networks is not necessarily ideal.

Competitive Self-Play



Competitive Self-Play

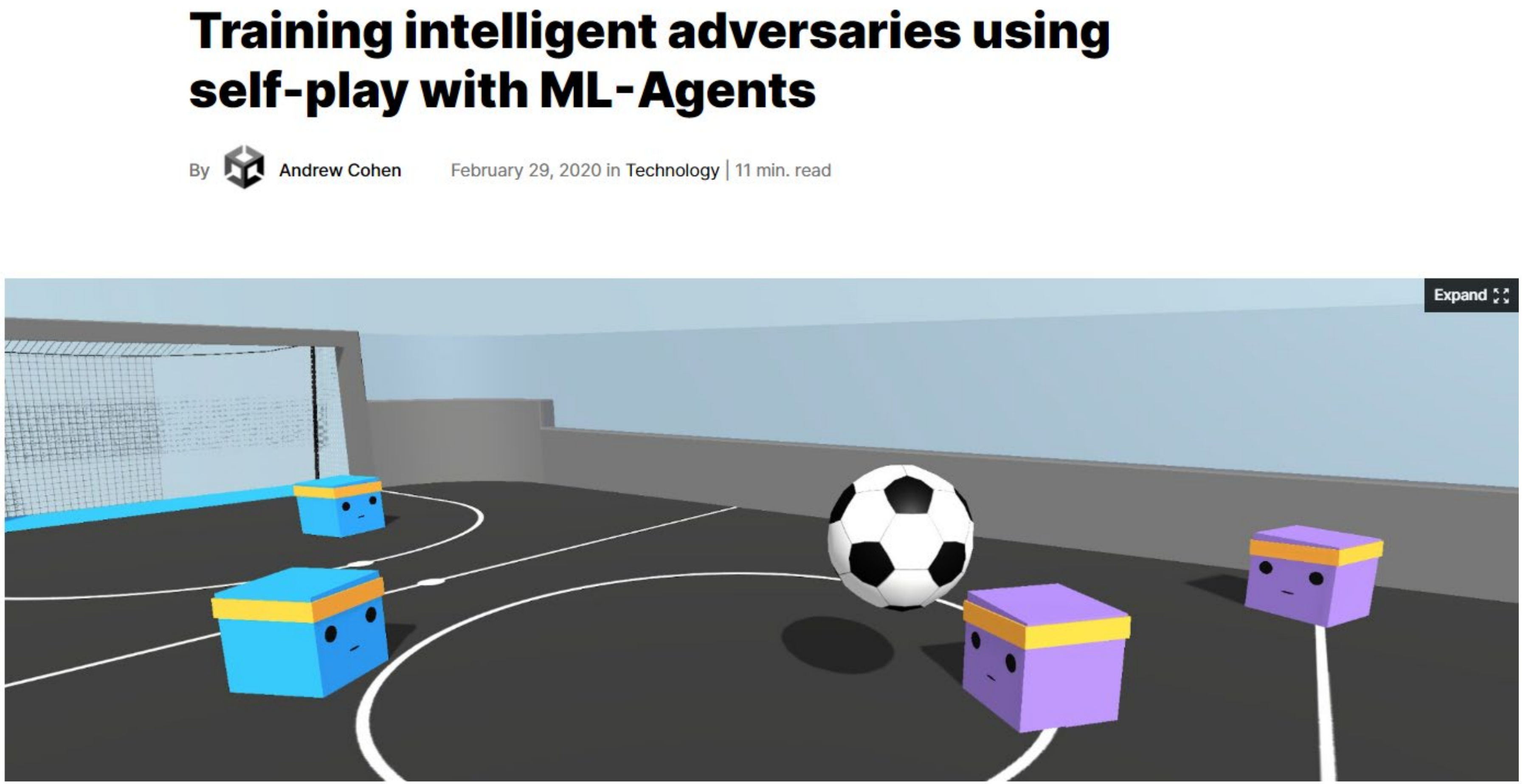
We've found that self-play allows simulated AIs to discover physical skills like tackling, ducking, faking, kicking, catching, and diving for the ball, without explicitly designing an environment with these skills in mind. Self-play ensures that the environment is always the right difficulty for an AI to improve. Taken alongside our Dota 2 self-play results, we have increasing confidence that self-play will be a core part of powerful AI systems in the future.



Goal: push opponent outside the ring, or topple them over

https://openai.com/blog/competitive-self-play/

Competitive Self-Play



Training intelligent adversaries using self-play with ML-Agents

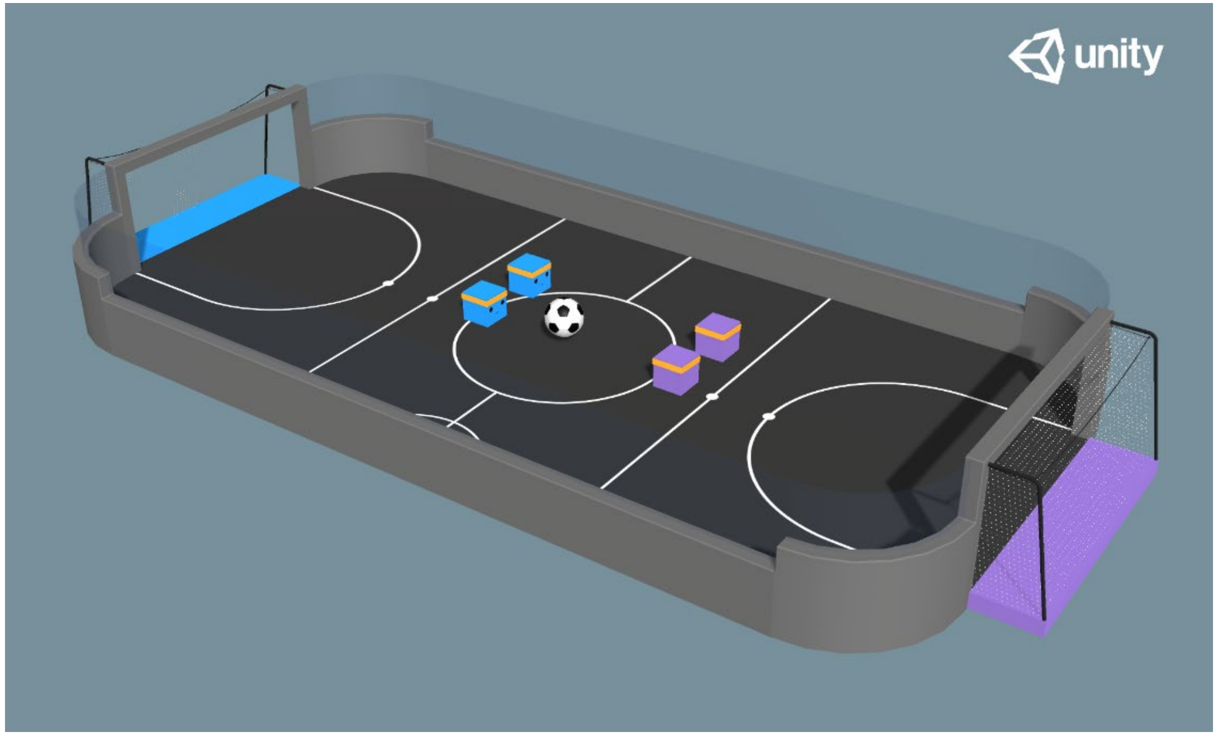
By Andrew Cohen February 29, 2020 in Technology | 11 min. read

https://blog.unity.com/technology/training-intelligent-adversaries-using-self-play-with-ml-agents

Class practice

Soccer two

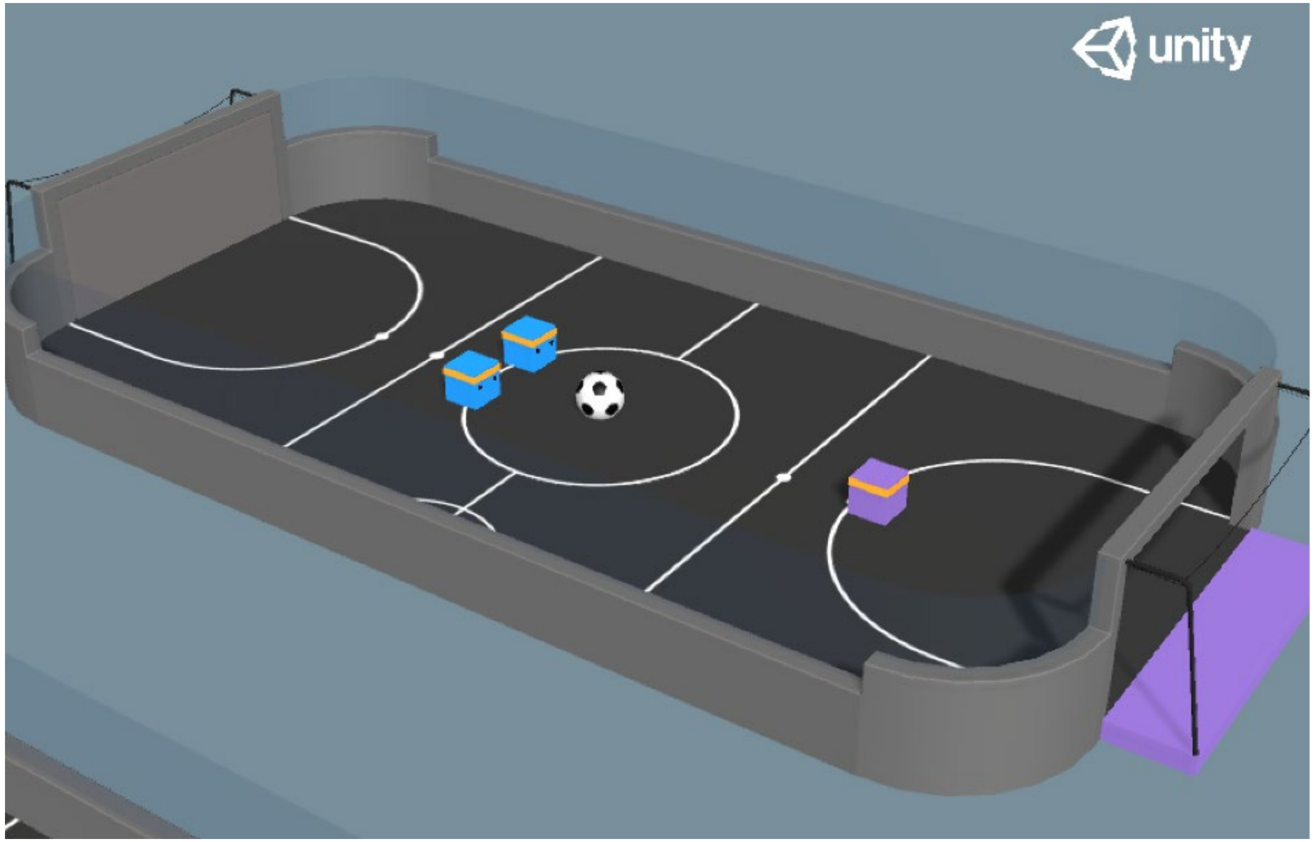
- Four agents compete in a 2 vs 2 toy soccer game. Get the ball into the opponent's goal while preventing the ball from entering own goal..



Class practice

Strikers Vs. Goalie

- Two agents compete in a 2 vs 1 soccer variant. Striker wants to get the ball into the opponent's goal. Goalie keeps the ball out of the goal.



Cooperative Environments with MA-POCA

- MA-POCA (MultiAgent POsthumous Credit Assignment) is a novel multi-agent trainer that trains a centralized critic, a neural network that acts as a "coach" for a whole group of agents.
- You can then give rewards to the team as a whole, and the agents will learn how best to contribute to achieving that reward.
- Agents can also be given rewards individually, and the team will work together to help the individual achieve those goals.

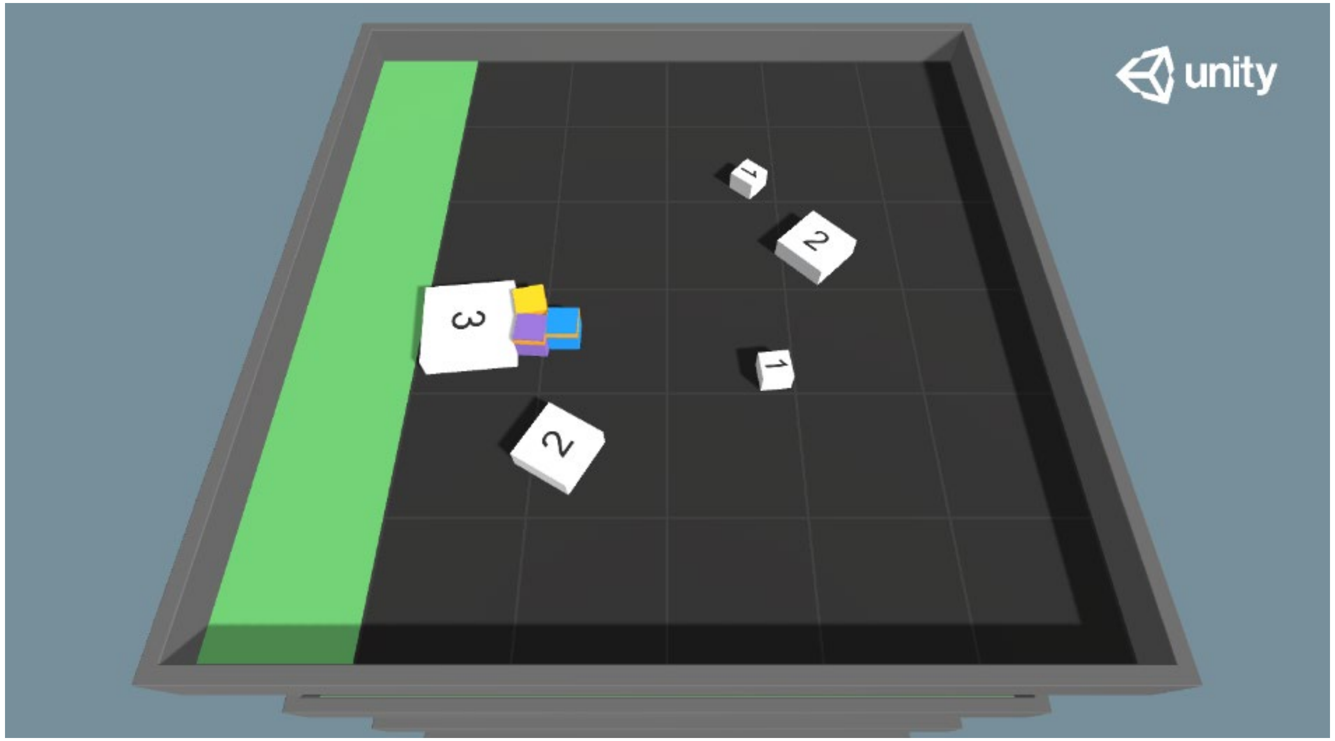
Cooperative Environments with MA-POCA

- During an episode, agents can be added or removed from the group, such as when agents spawn or die in a game.
- If agents are removed mid-episode (e.g., if teammates die or are removed from the game), they will still learn whether their actions contributed to the team winning later, enabling agents to take group-beneficial actions even if they result in the individual being removed from the game (i.e., self-sacrifice).
- MA-POCA can also be combined with self-play to train teams of agents to play against each other.

Class practice

Cooperative Push Block

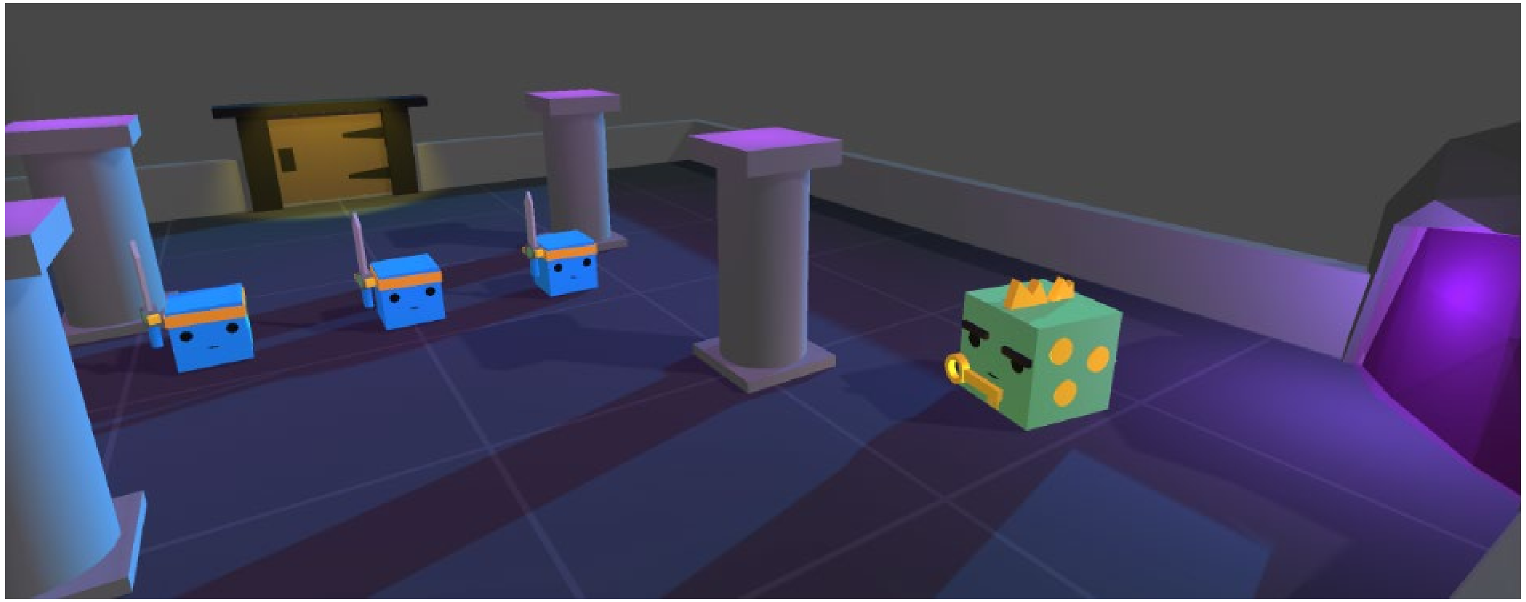
- Push all blocks into the goal. Small blocks can be pushed by one agents and are worth +1 value, medium blocks require two agents to push in and are worth +2, and large blocks require all 3 agents to push and are worth +3.



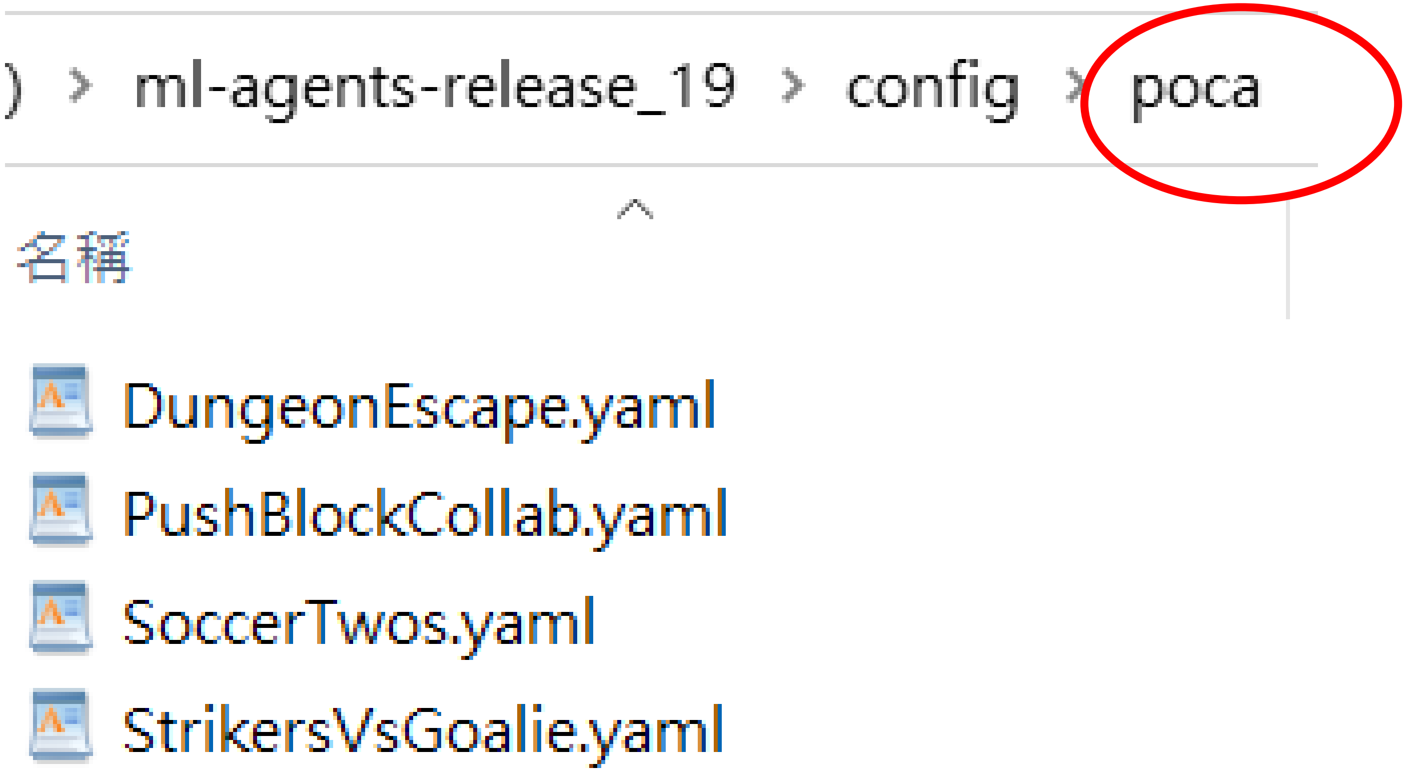
Class practice

Dungeon Escape

- Agents are trapped in a dungeon with a dragon, and must work together to escape. To retrieve the key, one of the agents must find and slay the dragon, sacrificing itself to do so. The dragon will drop a key for the others to use. The other agents can then pick up this key and unlock the dungeon door. If the agents take too long, the dragon will escape through a portal and the environment resets.



Class practice



) › ml-agents-release_19 › config › poca

DungeonEscape.yaml

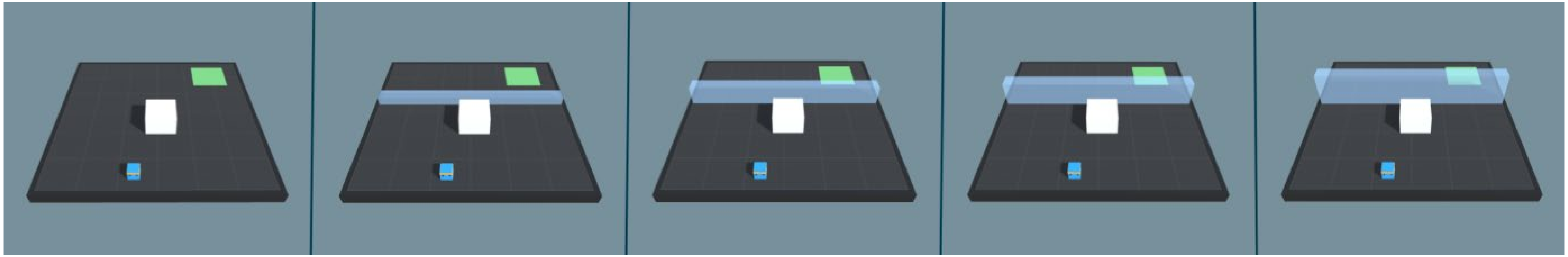
PushBlockCollab.yaml

SoccerTwos.yaml

StrikersVsGoalie.yaml

Curriculum Learning

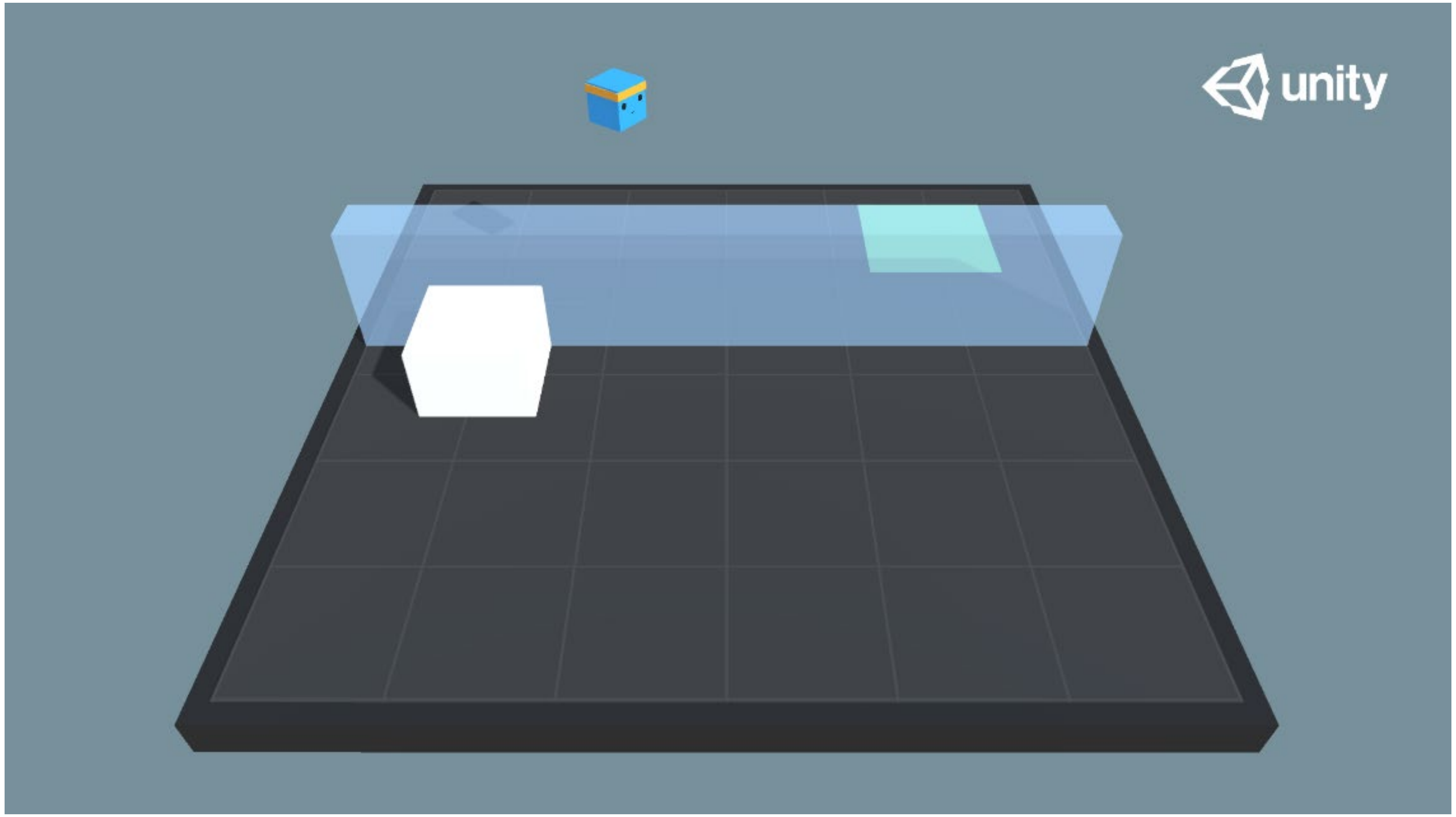
- Curriculum learning is a way of training a machine learning model where more difficult aspects of a problem are gradually introduced in such a way that the model is always optimally challenged.



Class practice

Wall Jump

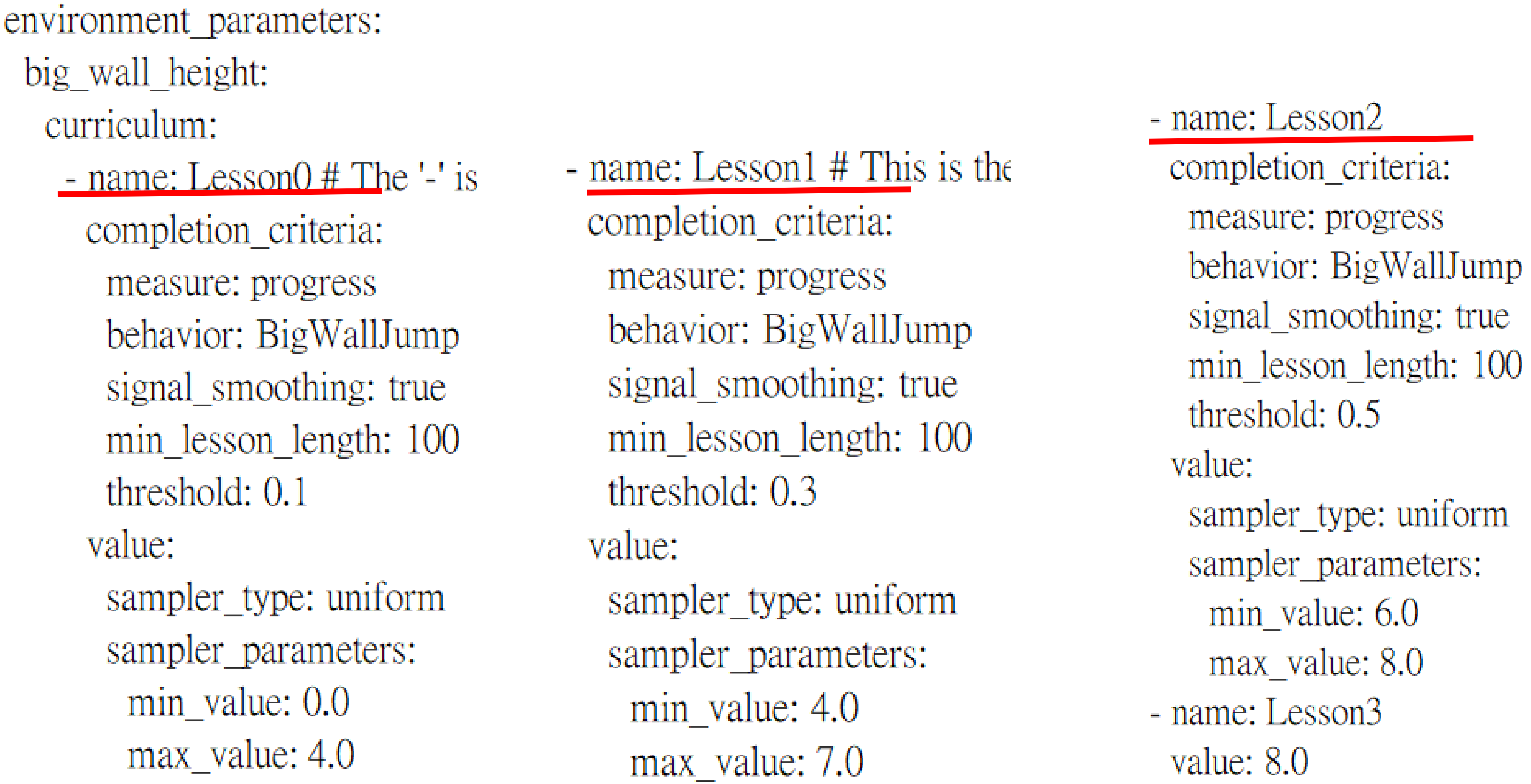
- A platforming environment where the agent can jump over a wall. The agent must use the block to scale the wall and reach the goal.



Class practice

Wall Jump

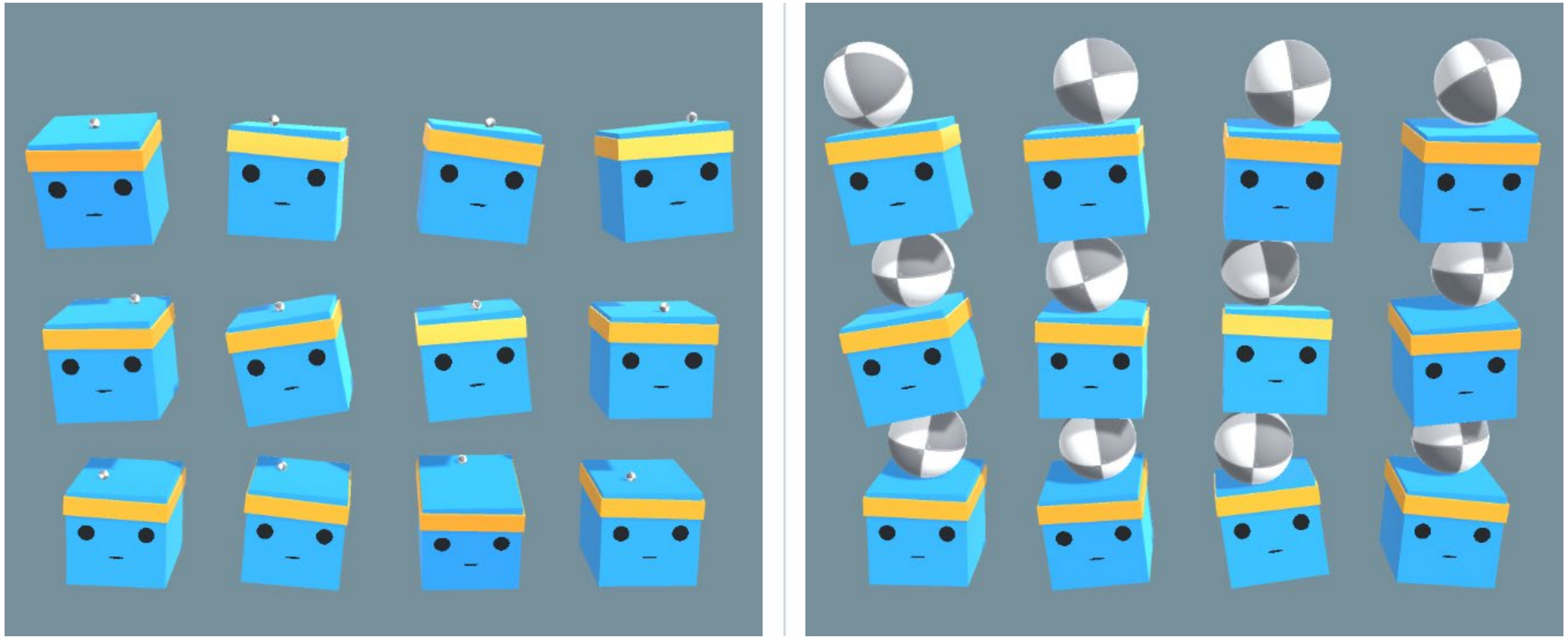
WallJump_curriculum.yaml



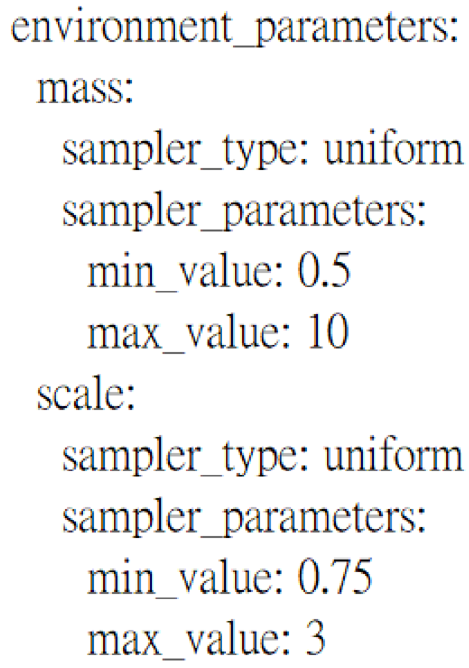
```
environment_parameters:
  big_wall_height:
    curriculum:
      - name: Lesson0 # The '-' is
        completion_criteria:
          measure: progress
          behavior: BigWallJump
          signal_smoothing: true
          min_lesson_length: 100
          threshold: 0.1
        value:
          sampler_type: uniform
          sampler_parameters:
            min_value: 0.0
            max_value: 4.0
      - name: Lesson1 # This is the
        completion_criteria:
          measure: progress
          behavior: BigWallJump
          signal_smoothing: true
          min_lesson_length: 100
          threshold: 0.3
        value:
          sampler_type: uniform
          sampler_parameters:
            min_value: 4.0
            max_value: 7.0
      - name: Lesson2
        completion_criteria:
          measure: progress
          behavior: BigWallJump
          signal_smoothing: true
          min_lesson_length: 100
          threshold: 0.5
        value:
          sampler_type: uniform
          sampler_parameters:
            min_value: 6.0
            max_value: 8.0
      - name: Lesson3
        value: 8.0
```

Environment Parameter Randomization

- An agent trained on a specific environment, may be unable to generalize to any tweaks or variations in the environment (in machine learning this is referred to as overfitting).

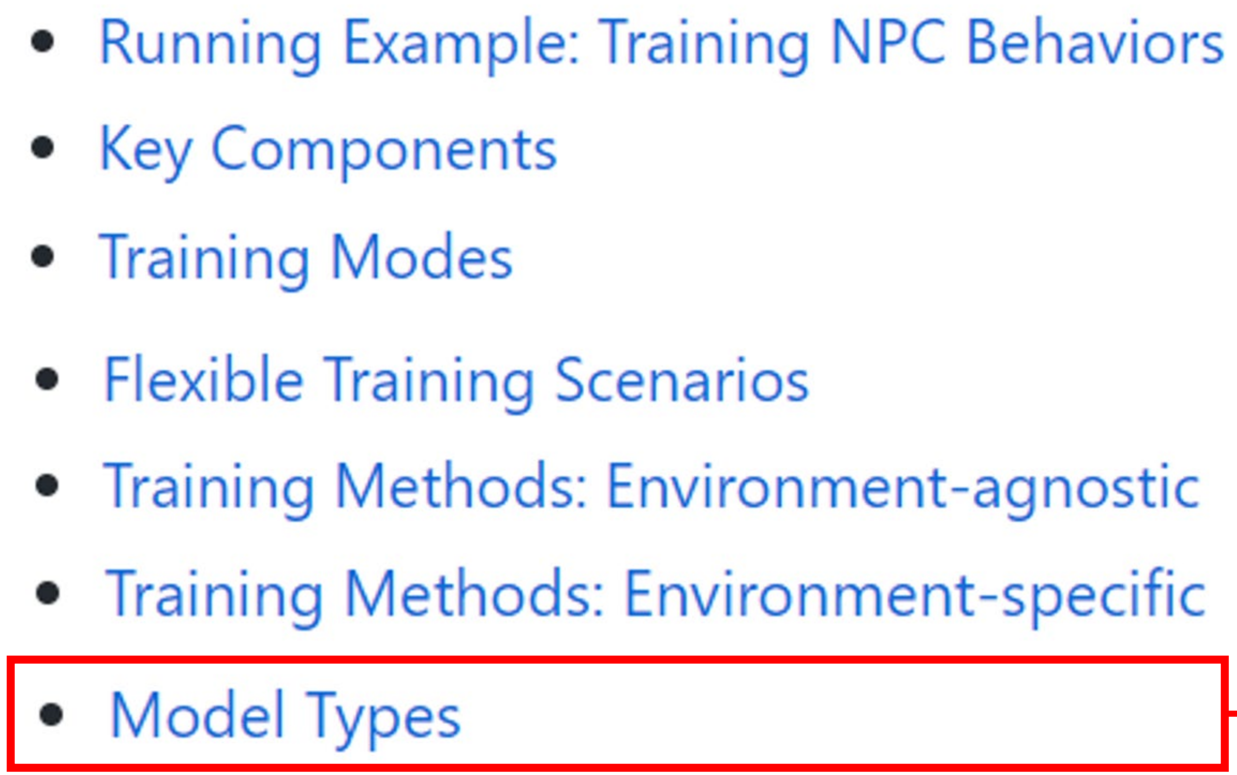


3DBall_randomize.yaml



```yaml
environment_parameters:
  mass:
    sampler_type: uniform
    sampler_parameters:
      min_value: 0.5
      max_value: 10
  scale:
    sampler_type: uniform
    sampler_parameters:
      min_value: 0.75
      max_value: 3
```

Model types



- Running Example: Training NPC Behaviors
- Key Components
- Training Modes
- Flexible Training Scenarios
- Training Methods: Environment-agnostic
- Training Methods: Environment-specific
- Model Types

1. Vector observation
2. Camera images using CNN
3. Variable length observation using Attention
4. Memory-enhanced agents using RNN