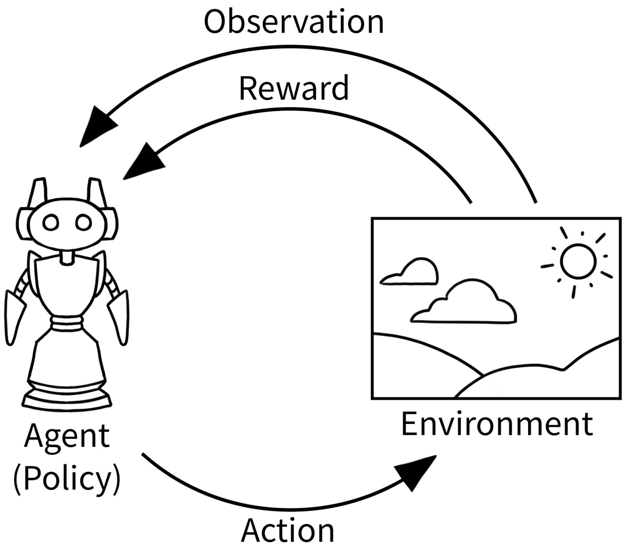
PettingZoo and Related Works

What's Reinforcement Learning?



- Machine learning for "Optimal Control"
  - Computer games
  - Robots
  - Stock trading
  - Elevator power management
  - Etc.

A Brief History of Modern Reinforcement Learning

- "Learning many Atari games is a very good milestone for reinforcement learning"
  - https://gym.openai.com/envs/#atari
  - Arcade Learning Environment
    - Graphical
    - Moderately challenging to humans
    - Clear reward
    - Diverse and unique
- DeepMind did this with the DQN
- Elon Musk thought the robot apocalypse was upon us, founded OpenAI

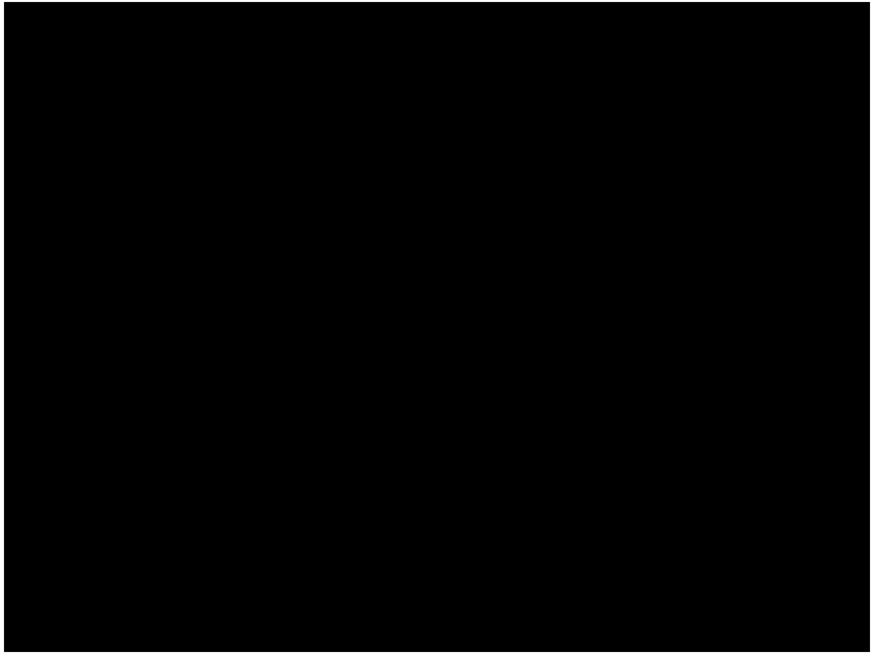
The History of Gym (The Before Times in RL)

- Environments and learning code could not be easily interchanged
- Doing anything in RL was a serious engineering effort
  - Language mismatches
  - APIs non CS beginners couldn’t figure out
  - Mostly only done at DeepMind or OpenAI type outfits
  - Environments were dumped in repos and unmaintained
- Gym made RL accessible to university grade researchers

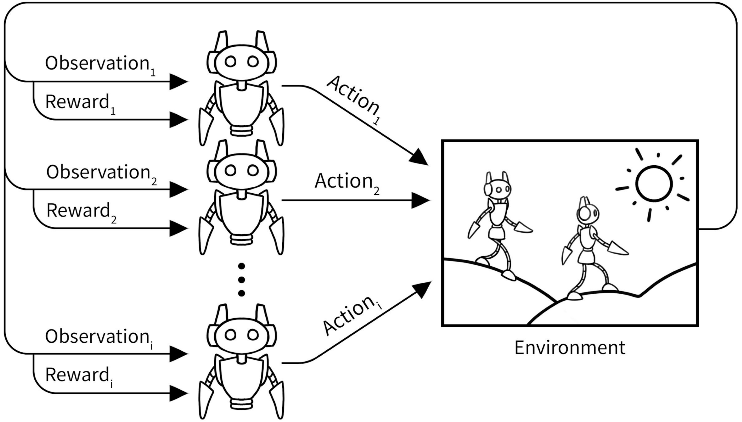
```python
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for _ in range(1000):
    env.render()
    action = policy(observation)
    observation, reward, done, info = env.step(action)
env.close()
```

What if you want to learn chess or StarCraft 2?

What if you want to coordinate Amazon warehouse robots?



Multi-Agent Reinforcement Learning (MARL)



All multi-agent system can, though not always should, be modeled as a Partially Observable Stochastic Game ("POSG"):

- $\mathcal{S}$ is the set of possible *states*.
- N is the *number of agents*. The *set of agents* is $[N]$.
- $\mathcal{A}_i$ is the set of possible *actions* for agent i.
- $P\colon \mathcal{S} \times \prod_{i \in [N]} \mathcal{A}_i \times \mathcal{S} \to [0, 1]$ is the (stochastic) *transition function*.
- $R_i\colon \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_N \times \mathcal{S} \to \mathbb{R}$ is the *reward function* for agent i.
- Ω_i is the set of possible *observations* for agent i.
- $O_i\colon \mathcal{A}_i \times \mathcal{S} \times \Omega_i \to [0, 1]$ is the *observation function*.

PettingZoo

- Multi-Agent RL is in a similar absolute mess, there should be Gym
- Two specific problems need to be solved
    - Standardized API
    - Large set of functioning environments compliant with it

Existing MARL APIs

```python
import gym
from ray.rllib.examples.env.multi_agent
    import MultiAgentCartPole
env = MultiAgentCartPole()
observations = env.reset()
for _ in range(1000):
    env.render()
    actions = policies(agents, observation)
    observations, rewards, dones,
        infos = env.step(actions)
env.close()
```

RLlib (POSG based)

-What's a POSG
-{'Agent1':reward1…}
-Standard, original PettingZoo API
-Weird to model turn based games
-Other conceptual problems

```python
import pyspiel
game = pyspiel.load_game("kuhn_poker")
state = game.new_initial_state()
while not state.is_terminal():
    legal_actions = state.legal_actions()
    if state.is_chance_node():
        outs, probs = zip(*state.chance_outcomes())
        actions = np.random.choice(outs, p=probs, size=3)
    else:
        actions = policies(agents, observation)
    state.apply_actions(actions)
```

OpenSpiel (EFG based)

-Tree based
-Only intended for classic card/board games

Gym Hacks

POSGs are Frequently a Bad Model of MARL Environments

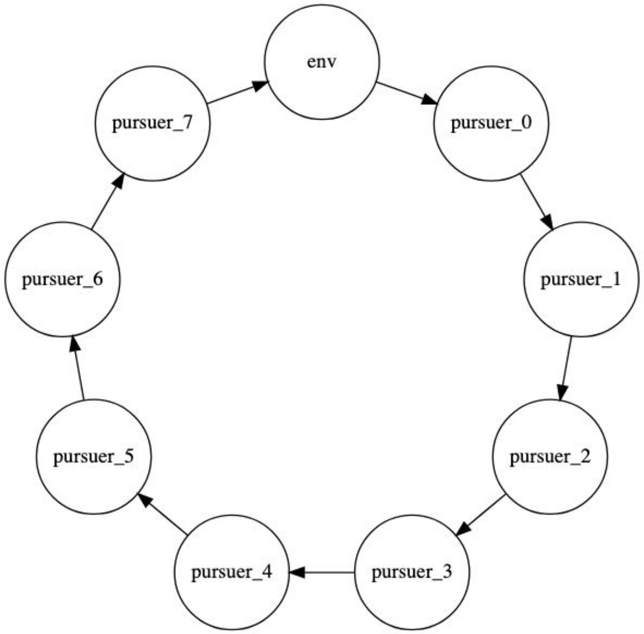
Modeling computer games as POSGs doesn't make conceptual sense

- Imperfect tie breaking in competitive snake
  - http://doublesnakegame.com/
  - What if the snakes hit each other or eat the food at the same time?
- Updating is always sequential unless you do crazy parallelization stuff
- Writing code that assumes you do is an excellent way to get race conditions
  - Open source social sequential delima game race condition
  - https://github.com/eugenevinitsky/sequential_social_dilemma_games
- The AEC games removes this opportunity for incredibly subtle bugs and better aligns with games in practice

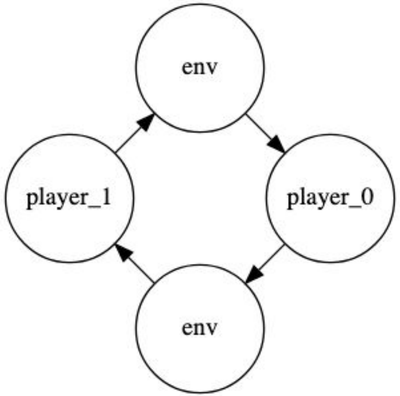
POSGs don’t allow access to reward information that should be available

- Reward is emitted from specific agent or environment actions
- If the game steps sequentially, you can get reward at different time points from each source
  - POSGs smash all this together
- When you view reward from this perspective, there’s often lots of things in it that shouldn’t be there that are very hard to figure out
  - Pursuit environment (2 line order switch, 10x performance improvement)
  - http://ala2017.it.nuigalway.ie/papers/ALA2017_Gupta.pdf

Agent Environment Cycle Diagrams



Pursuit



Chess

Properties of a Good Model

- Sequential
  - Modeling simultaneous games as sequential isn't a huge mess
  - Modeling sequential games as simultaneous is a huge mess
- Make a model that's a sequential POSG
  - Turn the cycle diagrams we were using into a formal model
- Uno (reverse card)
  - The cycle has to be mutable, use a "next agent" function

Agent Environment Cycle ("AEC") Games

- Provably equivalent representation to POSGs
- Basis of PettingZoo API
- Potentially useful for theory as a simpler version of EFGs with RL styled rewards

Formalism of an AEC game:
S is a set of states in an environment, $s \in S$
N is number of agents
Ξ is the set of all agents and the environment agent
$\mathbb{A}_i$ is a set of actions, $a \in A$
$T_i : Sx\mathbb{A}_i \to S$ is the transition function for agents
$P : SxS \to [0, 1]$ is the transition function of the environment
R_i is the reward function for agent i
Ω_i is the set of possible observations for agent i
$O_i : S \times \Omega_i \to [0, 1]$ is the observation function for agent i
$v : S \times \Xi \times \mathbb{A}\Xi \to [0, 1]$ is the next agent function

PettingZoo API

- Simple universal API that's very similar to Gym
    - Same action/observation spaces
- Last and agent_iter
- Robust handling of death and procgen
- https://www.pettingzoo.ml/api
- https://github.com/PettingZoo-Team/PettingZoo

Figure 1: Basic Usage of Gym

```
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for _ in range(1000):
    env.render()
    action = policy(observation)
    observation, reward, done, info = env.step(action)
env.close()
```

Figure 2: Basic Usage of PettingZoo

```
from pettingzoo.butterfly import pistonball_v0
env = pistonball_v0.env()
env.reset()
for agent in env.agent_iter(1000):
    env.render()
    observation, reward, done, info = env.last()
    action = policy(observation, agent)
    env.step(action)
env.close()
```

Advantages of POSGs

- Save ~6 lines of code if ton right
- Slightly faster by reducing number of calls
  - Mainly a problem with >>1000 agents
  - If you’re worried about this and using Python you’re doing it wrong
- Allow inferencing with multiple policies in parallel
  - I’m not aware of any case of it having been done
- PettingZoo envs exposure modular APIs that are wrapped
  - This means you can add your own API
  - We’ve added a parallel wrapper to environments with all applicable APIs

PettingZoo Environments

- https://www.pettingzoo.ml/envs

PettingZoo Quality of Life Improvements

- Environments are configurable by default
- Documentation
- Good error messages
- API compliance tests and recommendations for improvements

SuperSuit

- Preprocessing is almost always done to adapt actions/rewards/observations from an environment to whatever the policy function and learning methods need
- It’s an essential feature of all RL
  - It’s really hard to to a good job
  - It’s really easy to create small bugs
  - Everyone does it themselves
- There should probably be a library for that
- https://github.com/PettingZoo-Team/SuperSuit
  - Supports Gym and PettingZoo
  - Actually good

PettingZoo Integrations

- Stable baselines 2/3 (through SuperSuit)
- RLlib
- Autonomous learning library (dev branch)
- PyMARL (Planned)
- https://towardsdatascience.com/multi-agent-deep-reinforcement-learning-in-15-lines-of-code-using-pettingzoo-e0b963c0820b

Thanks for watching

- Please star the PettingZoo repo: https://github.com/PettingZoo-Team/PettingZoo