DeUmbra

RLlib for Deep Hierarchical Multiagent Reinforcement Learning

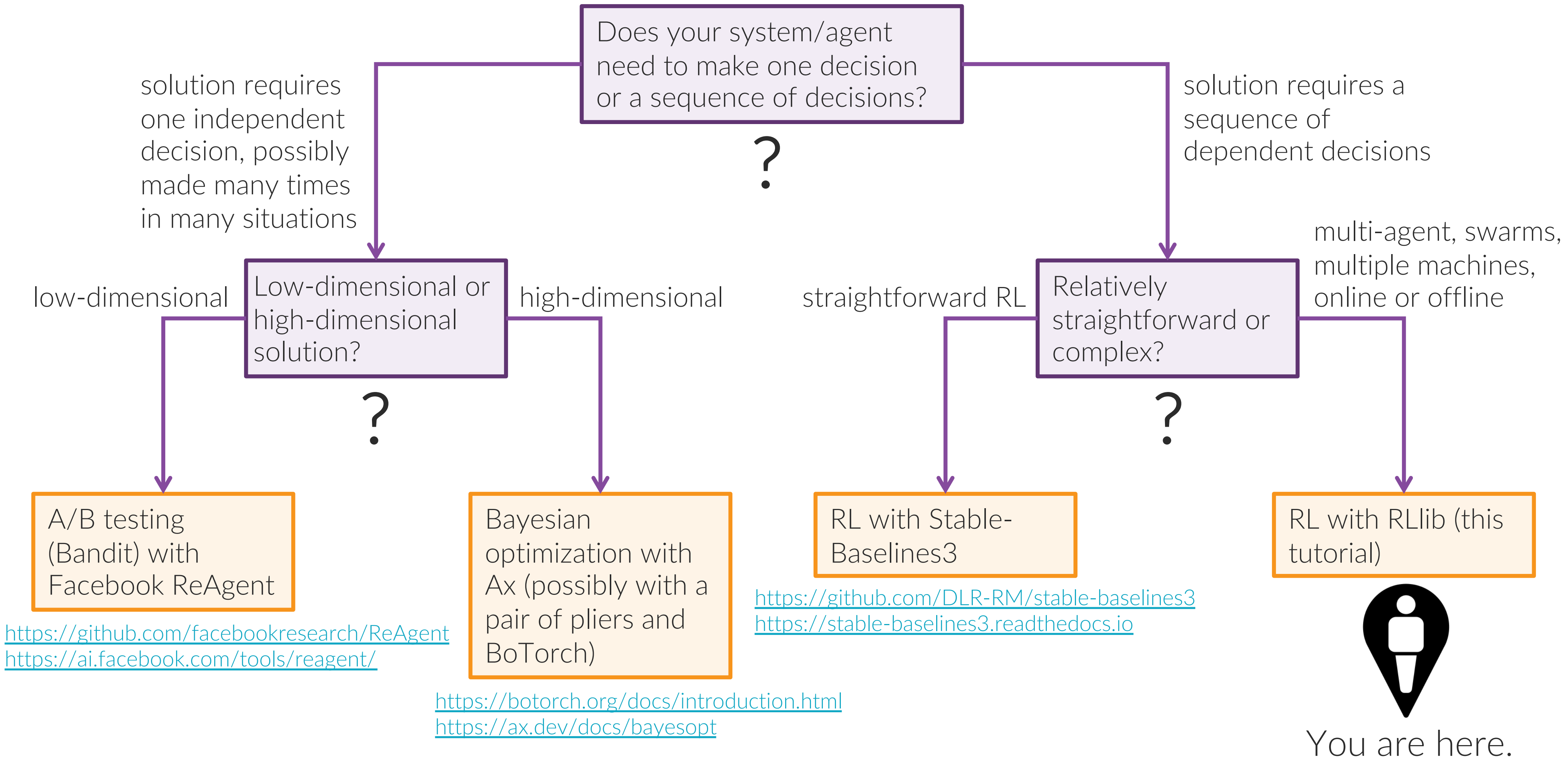
Jonathan Mugan, PhD
jmugan@deumbra.com
@jmugan

March 24, 2022

Goal of this tutorial

- Reinforcement learning (RL) is an effective method for solving problems that require agents to learn the best way to act in complex environments.
- RLlib is a powerful tool for applying reinforcement learning to problems where there are multiple agents or when agents must take on multiple roles.
- There are many of resources for learning about RLlib from a theoretical or academic perspective, but there is a lack of materials for learning how to use RLlib to solve your own practical problems.
- This tutorial helps to fill that gap.

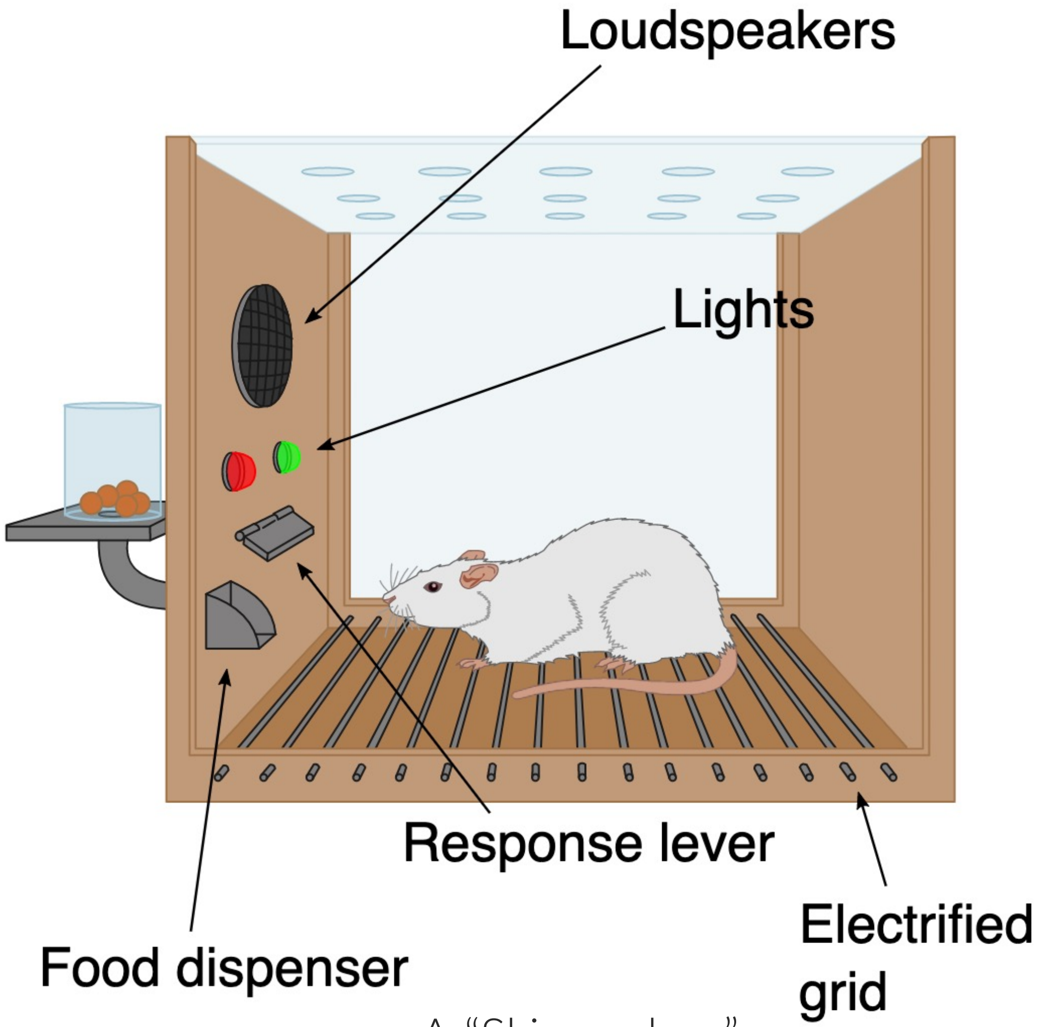
Use the Best Solution for Your Problem



RL is a gradual stamping in of behavior

Reinforcement learning: the first 100 years

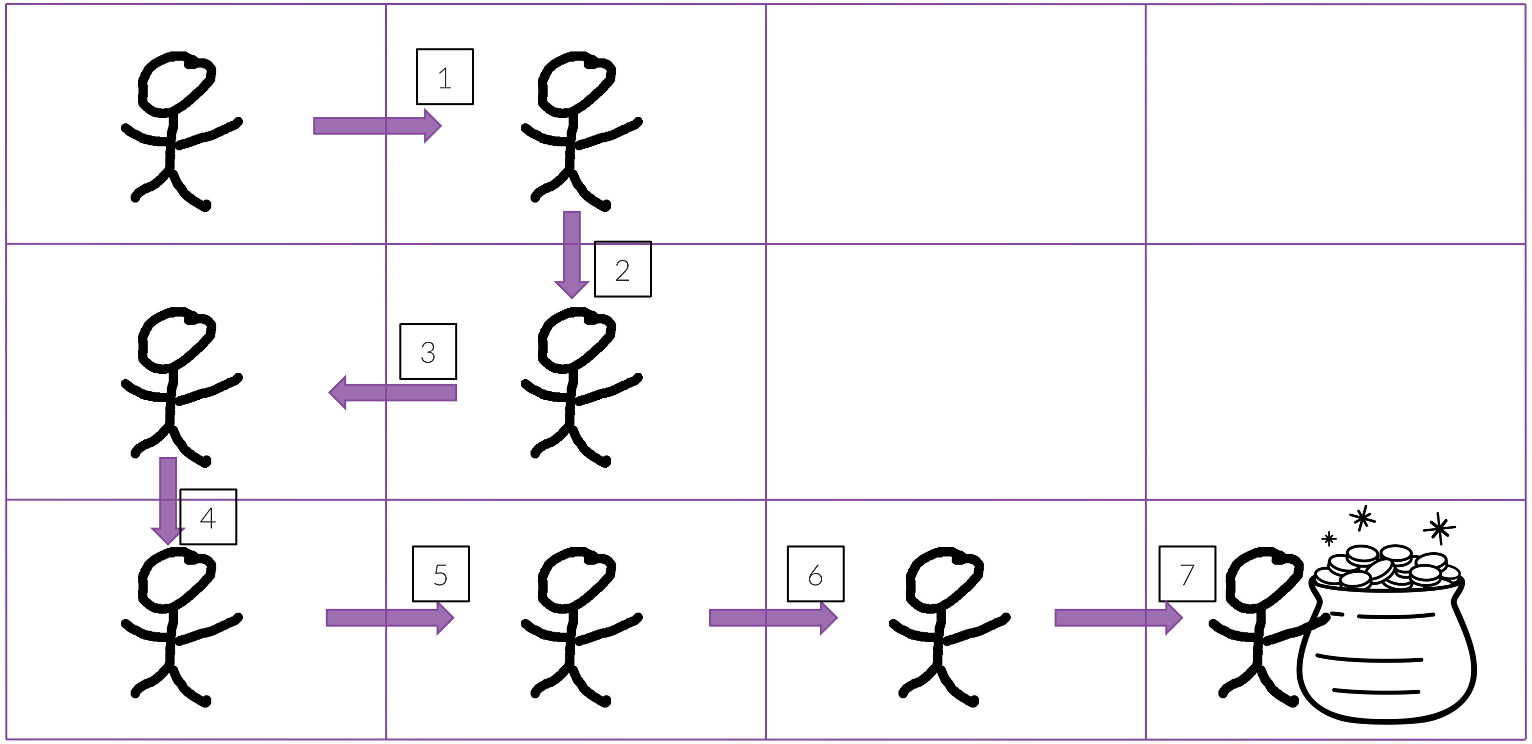
- Some behaviors arise more from a gradual stamping in [Thorndike, 1898].
- Became the study of Behaviorism [Skinner, 1953] (see Skinner box on the right).
- Formulated into artificial intelligence as Reinforcement Learning [Sutton and Barto, 1998].



A “Skinner box”

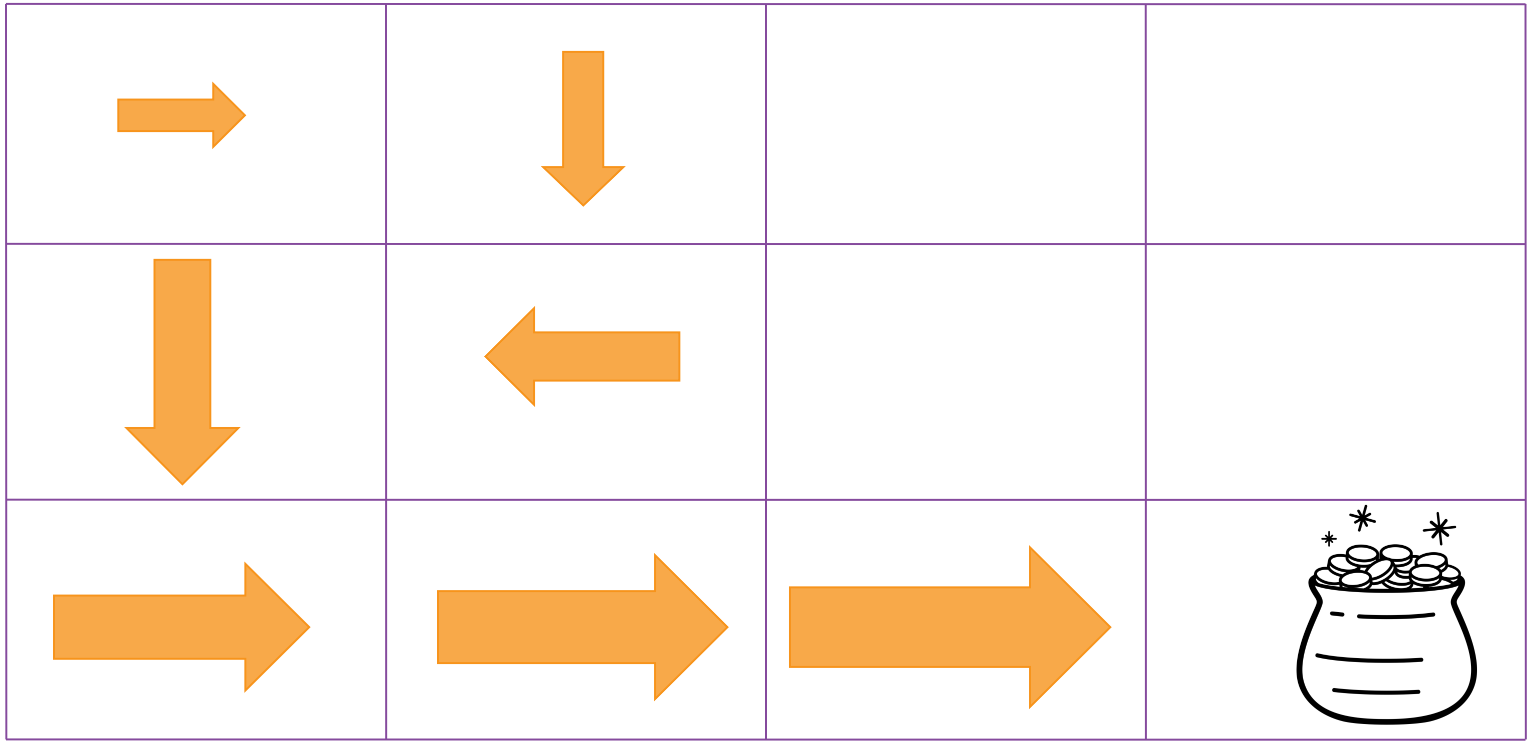
By Original: AndreasJS Vector: Pixelsquid - This file was derived from: Skinner box scheme 01.png: by AndreasJS, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=99322433

RL in a nutshell: begin with random exploration



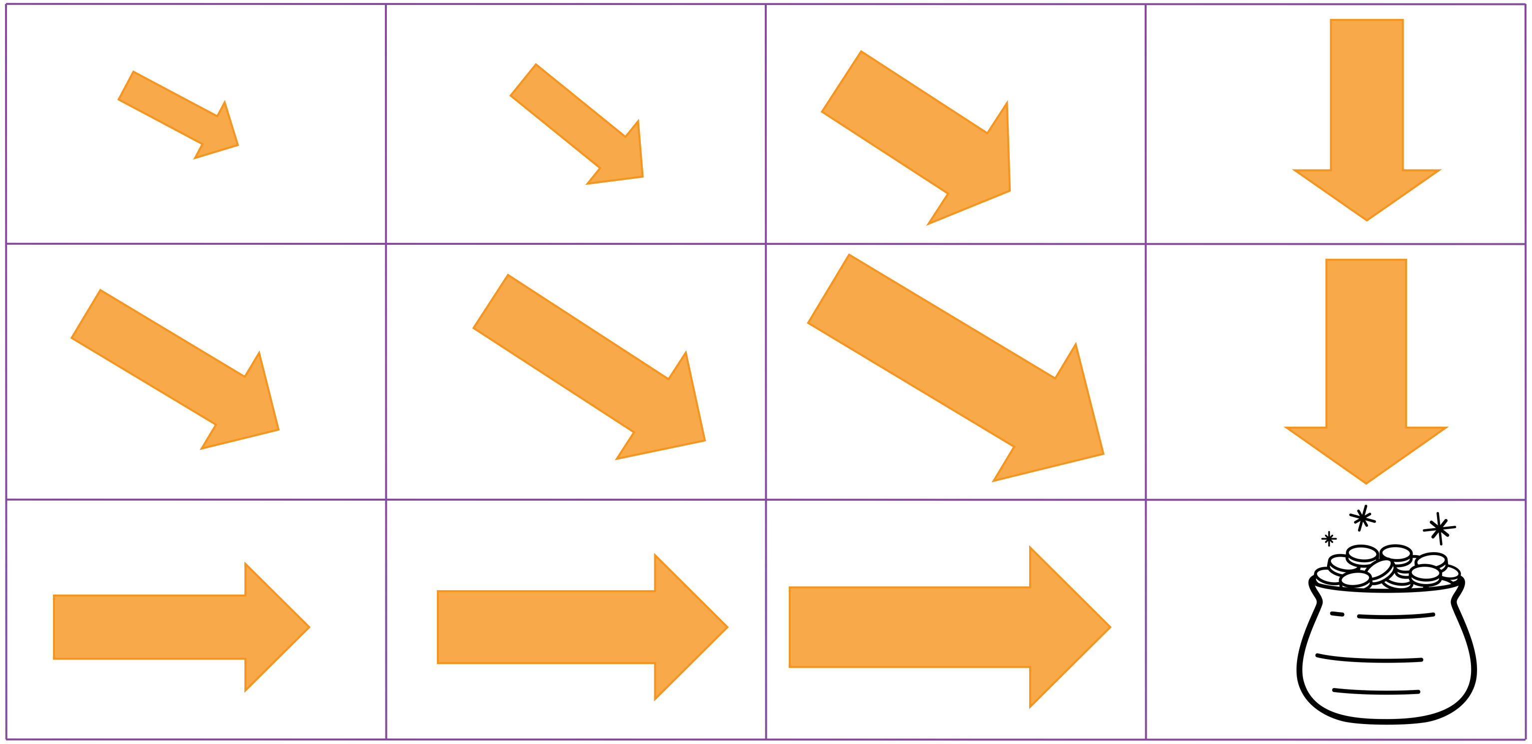
In reinforcement learning, the agent often begins by randomly exploring until it reaches its goal.

RL in a nutshell: remember what got you there



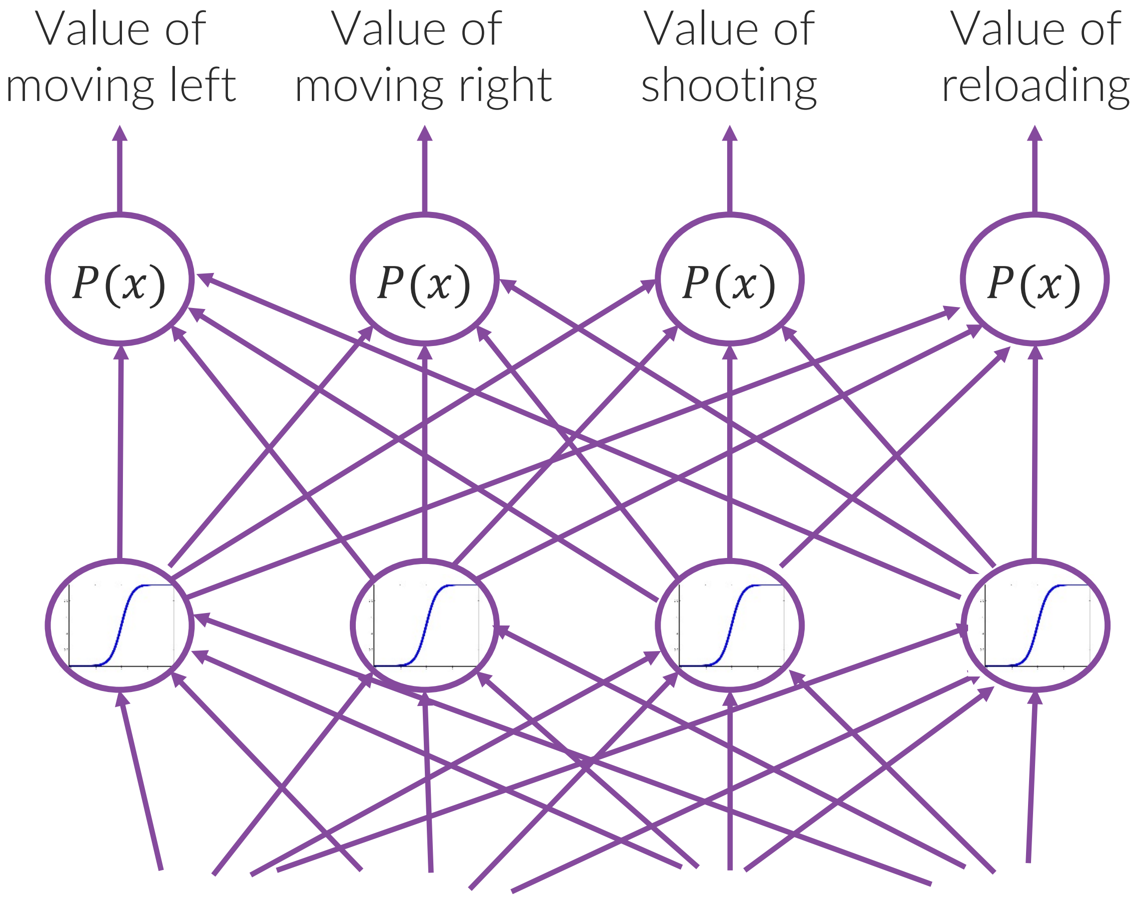
- When it reaches the goal, credit is propagated back to its previous states.
- The agent learns the function $Q^{\pi}(s,a)$, which gives the cumulative expected discounted reward of being in state s and taking action a and acting according to policy π thereafter.

RL in a nutshell: learn a policy for behavior



Eventually, the agent learns the value of being in each state and taking each action and can therefore always do the best thing in each state. This behavior is then represented as a policy.

Modern RL uses deep learning



RLlib automatically constructs deep learning models behind the scenes based on your configuration.

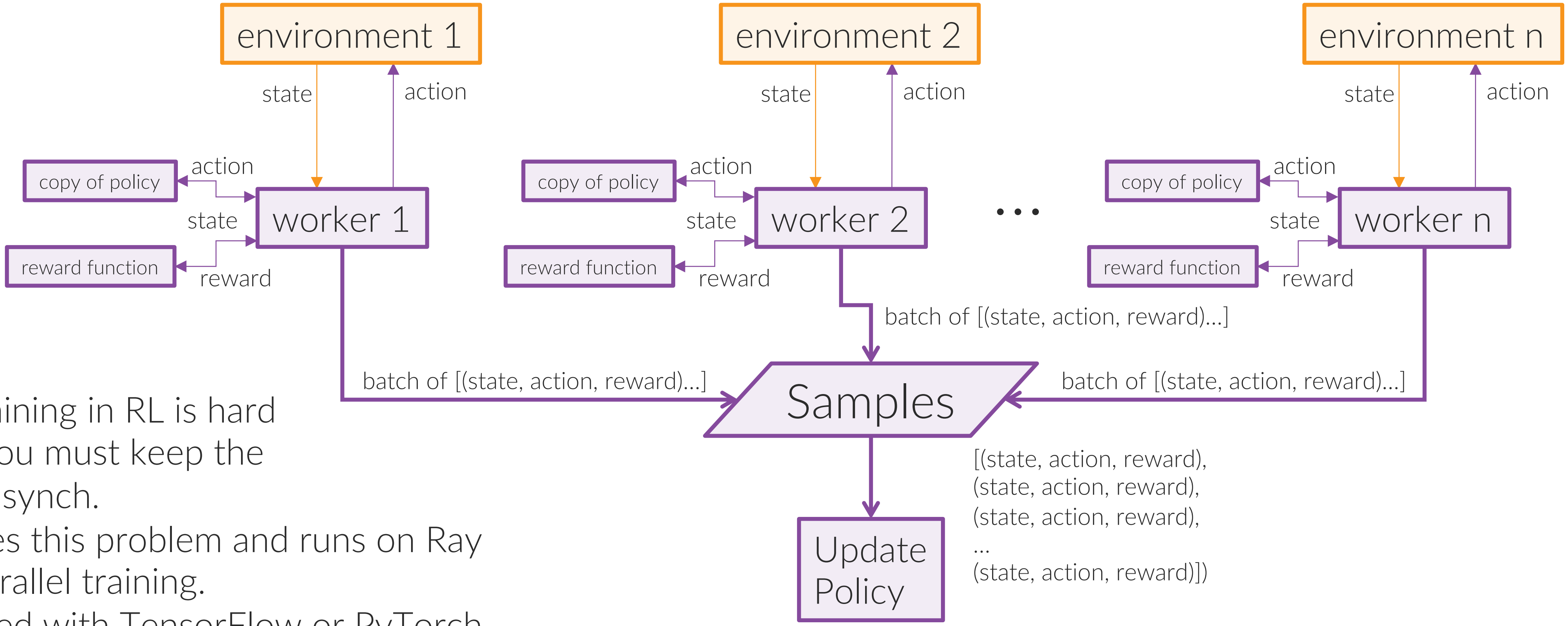
This tutorial fucuses on RLlib.

If you want to learn more about reinforcement learning in general, some great resources are

- http://www.incompleteideas.net/book/RLbook2020.pdf
- https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html

RLlib solves the problem of distributed RL

"RLlib: Abstractions for Distributed Reinforcement Learning" https://arxiv.org/pdf/1712.09381.pdf

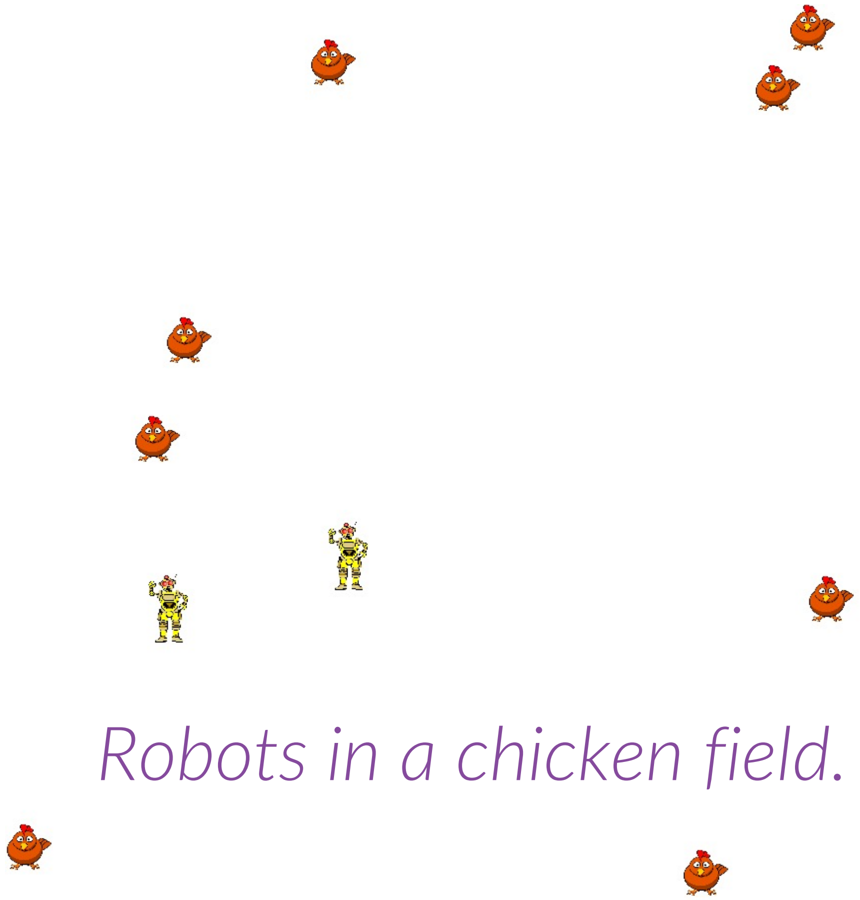


- Parallel training in RL is hard because you must keep the policies in synch.
- RLlib solves this problem and runs on Ray for fast parallel training.
- Can be used with TensorFlow or PyTorch.

Arguably, the most exciting part of RLlib is the abstractions for hierarchical multiagent reinforcement learning, and that is what we will focus on.

Simple, Custom, Multi-Agent, Hierarchical Environment

We created this custom environment to show you how to adapt RLlib to your problem. The custom environment keeps us from glossing over any necessary details for using RLlib in your system. This custom environment is silly but entails the necessary complexity.



Robots in a chicken field.

Scenario: Two robots in a chicken yard. The robots want to capture (move to) the chickens that are most like them.

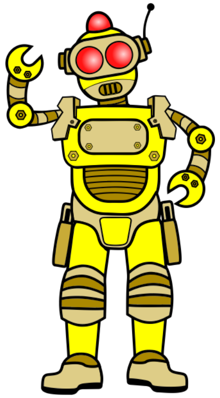
- Each robot and chicken has a personality based on the OCEAN model https://en.wikipedia.org/wiki/Big_Five_personality_traits

Reward: getting close to a chicken that is like the robot.

- Punished at each time step
- Reward when it reaches the chicken is the dot product of its personality with that of the chicken.
  - E.g., neurotic robots are rewarded for finding neurotic chickens

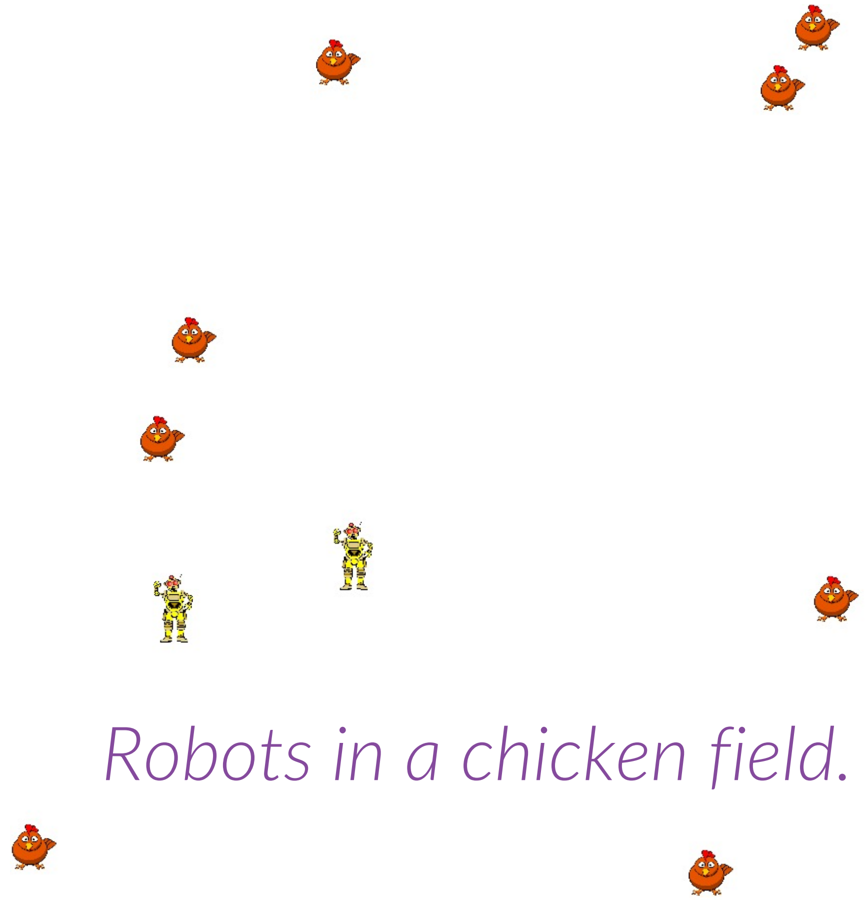
Multiagent and Hierarchical

- There are two robots
- Each robot uses two policies
  - High-level policy: pick a chicken
  - Low-level policy: make way to chicken



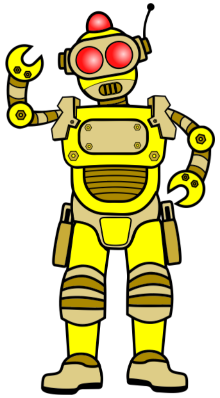
Simple, Custom, Multi-Agent, Hierarchical Environment

We created this custom environment to show you how to adapt RLlib to your problem. The custom environment keeps us from glossing over any necessary details for using RLlib in your system. This custom environment is silly but entails the necessary complexity.



Robots in a chicken field.

High-level Observation (State) Space
- Set of chickens
- Each chicken measured with the OCEAN model, so is a vector
- Each chicken also has a location, x, y
- Each robot also has an OCEAN vector



Low-level Observation (State) Space
- robot position x, y
- chicken position x, y

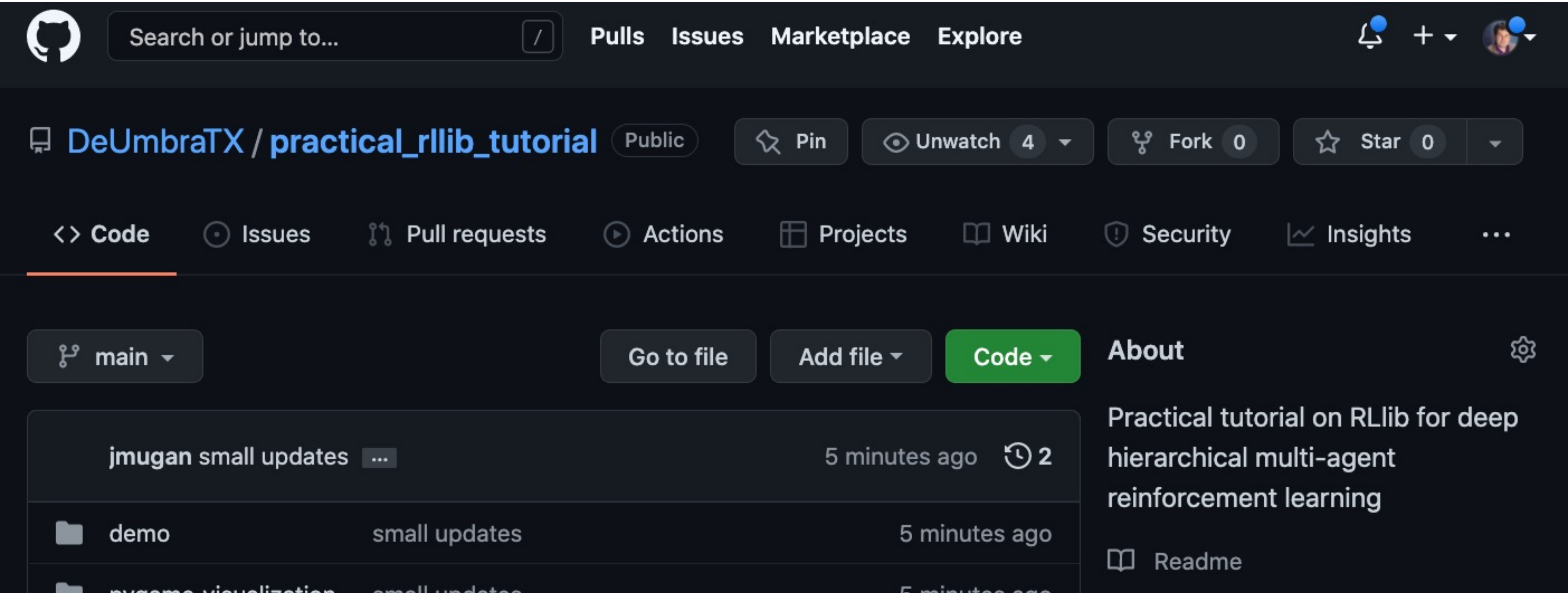
High-level Action Space
- Which chicken to choose

Low-level Action space
- Go in 8 directions: N, NE, E, SE, S, SW, W, and NW

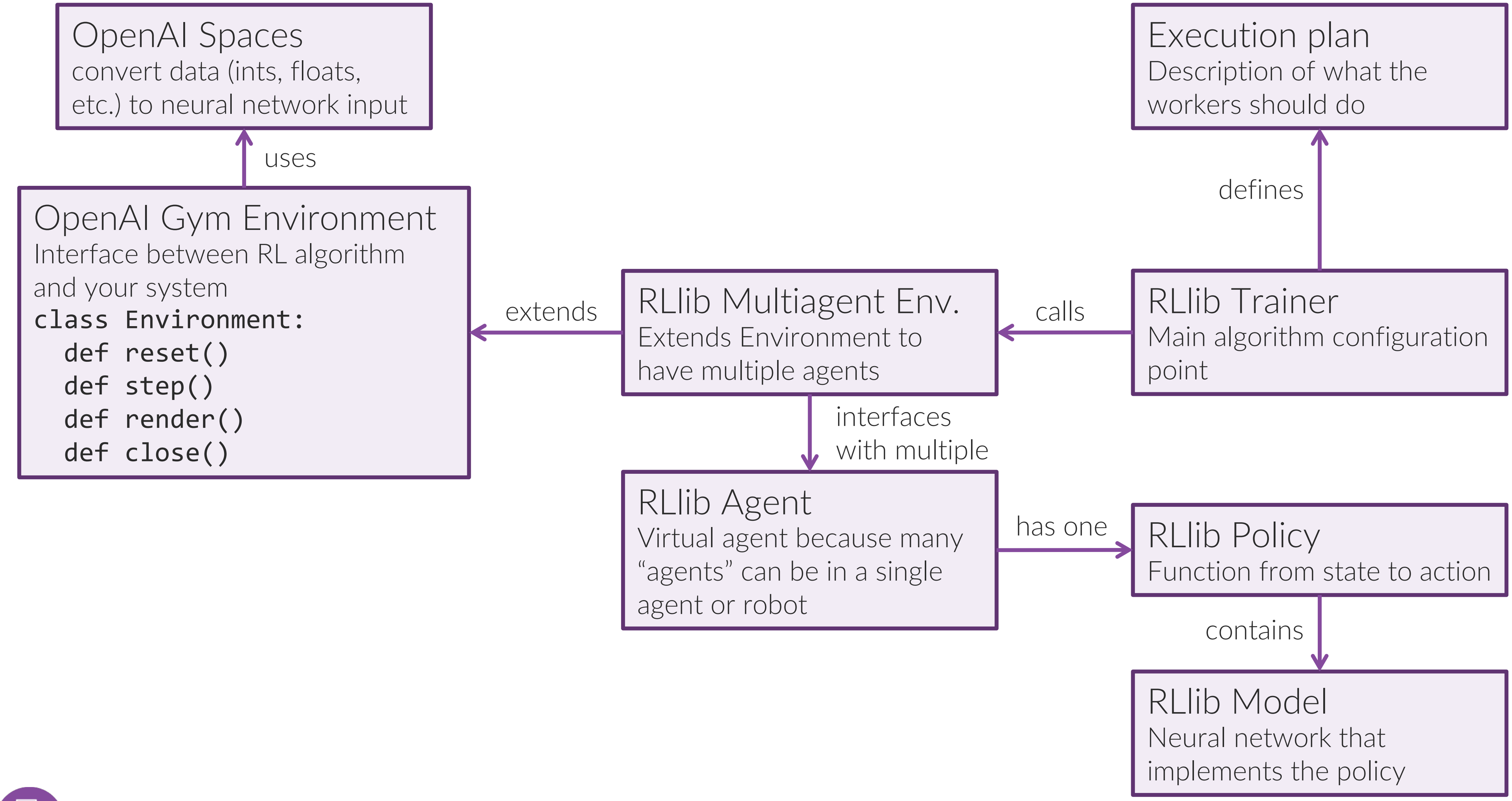
Episode End
- When each robot gets within a distance threshold of a chicken

Code is online

https://github.com/DeUmbraTX/practical_rllib_tutorial



Main RLlib Abstractions



Open AI Spaces

OpenAI Spaces
convert data (ints, floats, etc.) to neural network input

Makes it easy to format your domain for neural networks.

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_openai_spaces.py

```python
robot_position_space = spaces.Box(
            shape=(2,),
            dtype=np.float,
            low=0.0,
            high=1.0)

robot_ocean_space = spaces.Box(
            shape=(NUM_OCEAN_FEATURES,),
            dtype=np.float,
            low=0.0,
            high=1.0)


high_level_obs_space = spaces.Dict({
        'chicken_oceans': chicken_ocean_space,
        'chicken_positions': chicken_position_space,
        'robot_ocean': robot_ocean_space,
        'robot_position': robot_position_space,
})


# At the low level, you don't care about ocean, you've already
low_level_obs_space = spaces.Dict({
        'robot_position': robot_position_space,
        'chicken_positions': chicken_position_space,
})
```

Open AI Gym Environment

Provides a uniform API to interface with your system.

OpenAI Gym Environment
Interface between RL algorithm and your system

```
class Environment:
  def reset()
  def step()
  def render()
  def close()
```

extends

RLlib Multiagent Env.
Extends Environment to have multiple agents

```python
# MultiAgentEvent subclass of gym.Env
class YourEnvironment(MultiAgentEnv):
    def __init__(self, config:EnvContext):
        self.config_val = config['my_config_val']
        self.target_system = YourTargetSystem()
        self.visualization = Visualization()
        self.observation_space = None  # is_atari bug


    def reset(self):
        # Start a new chicken yard
        self.target_system.initialize_yard()
        yard = self.target_system.get_yard()

        obs_robot_1_high = {'chicken_oceans': yard.chicken_ocean,
                            'chicken_positions': yard.chicken_positions,
                            'robot_ocean': yard.robot_1_ocean,
                            'robot_position': yard.robot_1_position}
        obs_robot_2_high = {'chicken_oceans': yard.chicken_ocean,
                            'chicken_positions': yard.chicken_positions,
                            'robot_ocean': yard.robot_2_ocean,
                            'robot_position': yard.robot_2_position}

        # Because only high-level robots returned, only their policies will be called
        return {'robot_1_high': obs_robot_1_high, 'robot_2_high': obs_robot_2_high}
```

Main method is `step`, which has 3 parts:

1. Get actions from RLlib policies and execute in your environment
2. Get state of your environment after taking actions
3. Pass relevant information to RLlib

```python
def step(self, action_dict):
    obs, rew, done, info = {}, {}, {}, {}

    #**********************************************************************
    # Part 1: get actions from RLlib policies and execute in your environment
    # **********************************************************************
```

Agents

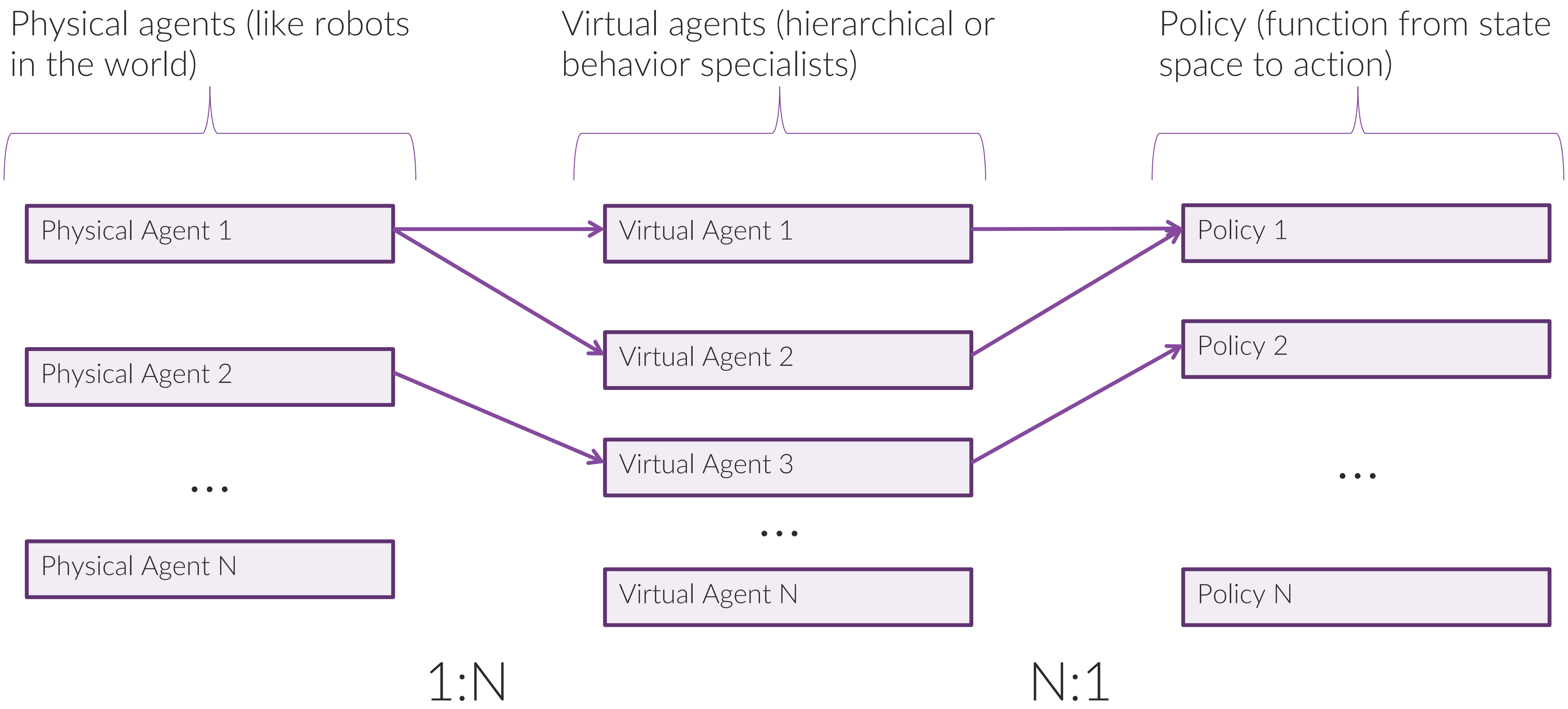
The RLlib `MultiAgentEnv` class enables you to dynamically say which of your agents are acting at which timesteps and you can specify what their observation and reward should be if they are active.

RLlib Multiagent Env.
Extends Environment to have multiple agents

$\downarrow$ interfaces with multiple

RLlib Agent
Virtual agent because many “agents” can be in a single agent or robot

RLlib Makes Multi-Agent Easy

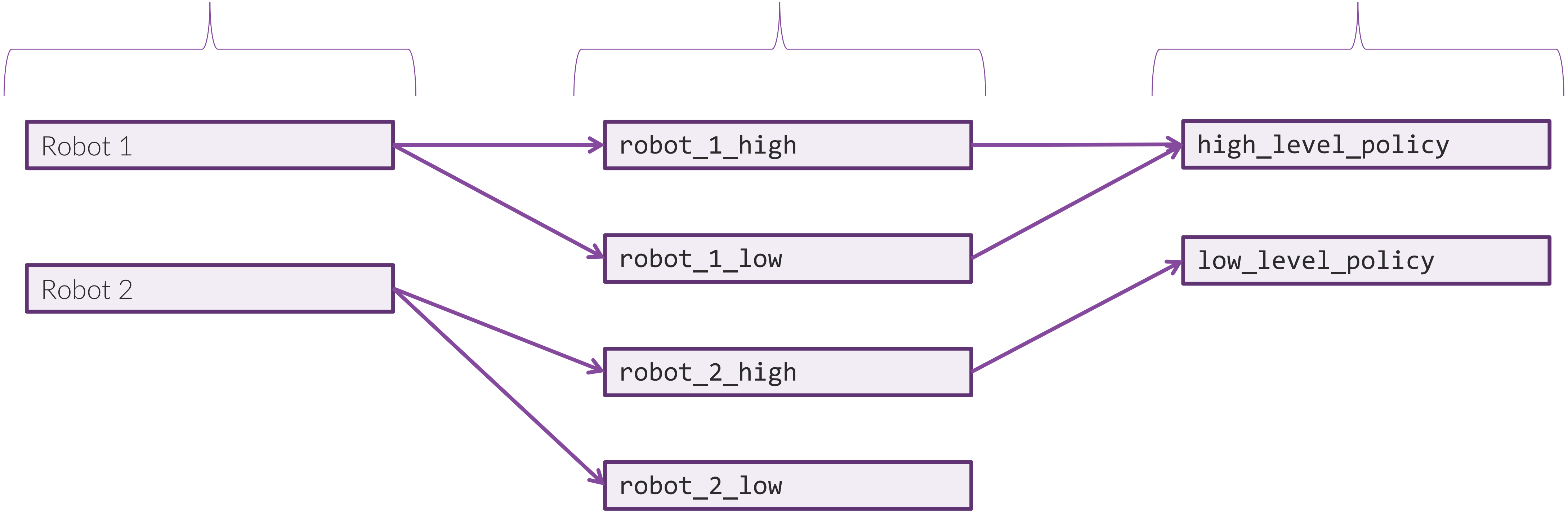


Mapping Agents to Our Environment

Physical agents (like robots in the world)

Virtual agents (hierarchical or behavior specialists)

Policy (function from state space to action)



1:N

N:1

Your System/Simulation

```python
class YourTargetSystem:
    def __init__(self):
        self.chicken_ocean: Optional[np.ndarray, None] = None
        self.chicken_positions: Optional[np.ndarray, None] = None
        self.robot_1 = YourAutonomousAgent()
        self.robot_2 = YourAutonomousAgent()
        self.timestep: Optional[int, None] = None


    def initialize_yard(self) -> None:
        """
        Get a new chicken yard. For each of the NUM_CHICKEN chickens, compute their
        positions randomly and compute the results of phychological testing
        using the OCEAN model.
        """
        self.chicken_positions = np.random.rand(NUM_CHICKENS, 2)
        self.chicken_ocean = np.random.rand(NUM_CHICKENS, NUM_OCEAN_FEATURES)
        # normalize
        for i in range(NUM_CHICKENS):
            self.chicken_ocean[i,:] = self.chicken_ocean[i,:] / np.linalg.norm(self.c
        self.robot_1.initialize()
        self.robot_2.initialize()
        self.timestep = 0
```

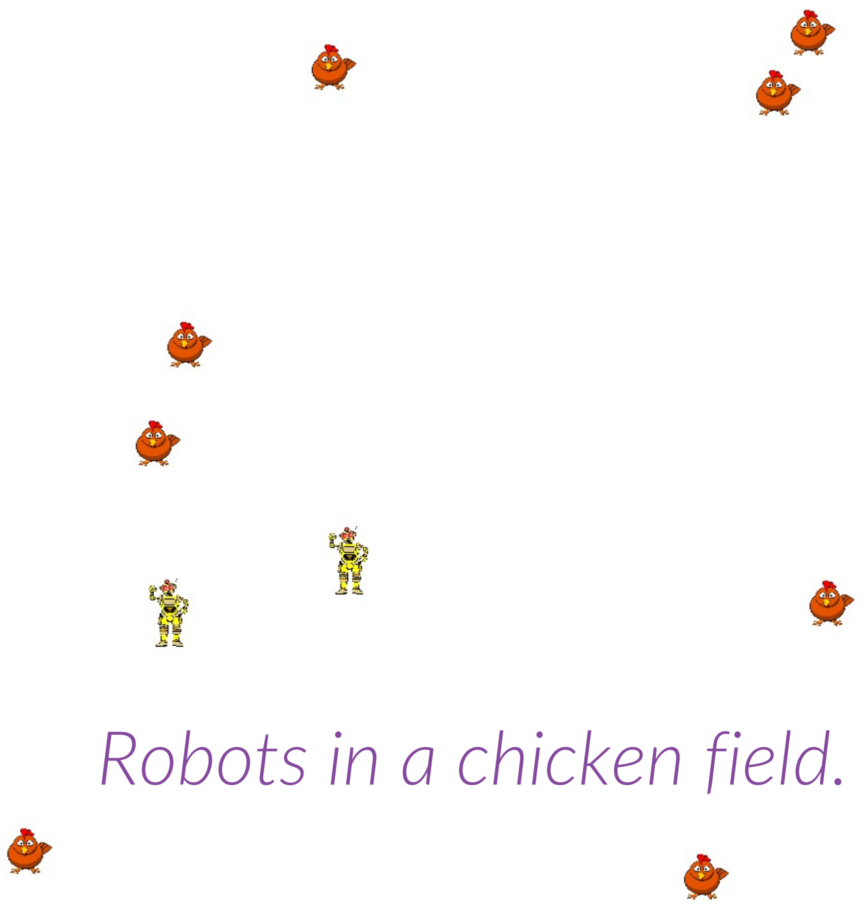
https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_target_system.py

```python
class YourAutonomousAgent:
    def __init__(self):
        self.position: Optional[np.ndarray, None] = None
        self.ocean: Optional[np.ndarray, None] = None
        self.chicken_choice: Optional[int, None] = None

    def initialize(self) -> None:
        """
        Robot shoudl perform introspection using OCEAN model and generate
        a random position.
        """
        self.position = np.random.rand(2,)
        self.ocean = np.random.rand(NUM_OCEAN_FEATURES, )
        self.ocean = self.ocean / np.linalg.norm(self.ocean)

    def get_position(self) -> np.ndarray:
        return self.position

    def get_ocean(self) -> np.ndarray:
        return self.ocean

    def set_chicken_choice(self, chicken: int) -> None:
        self.chicken_choice = chicken
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_autonomous_agent.py

Testing the Environment

Start with a random policy to test that the environment works as expected with your system/simulation.



Robots in a chicken field.

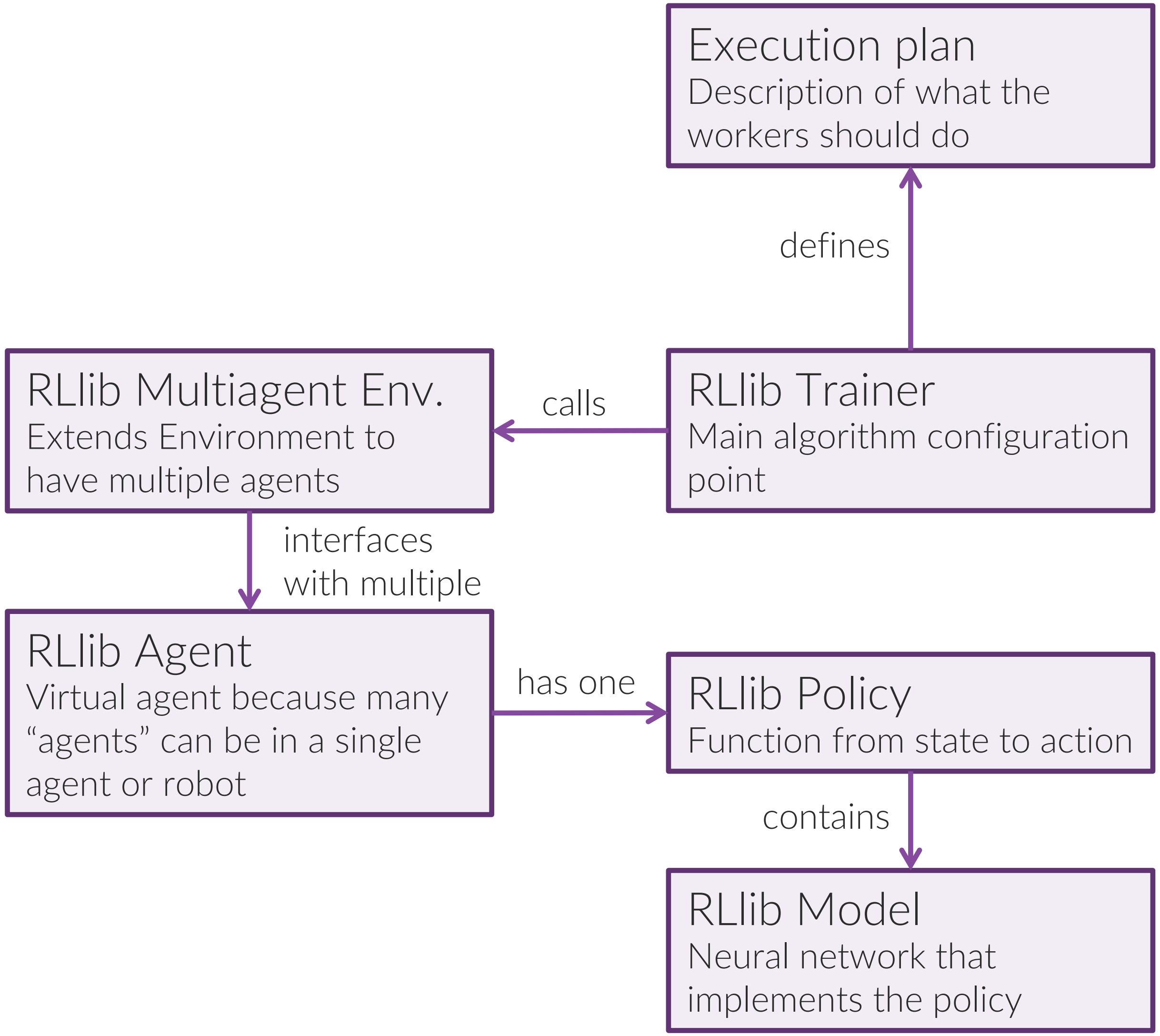
```python
env = YourEnvironment(config)

env.reset()

action_dict = {'robot_1_high': random.choice(range(NUM_CHICKENS)),
                'robot_2_high': random.choice(range(NUM_CHICKENS))}

obs, rew, done, info = env.step(action_dict)
env.render()

def is_all_done(done: Dict) -> bool:
    for key, val in done.items():
        if not val:
            return False
    return True

while not is_all_done(done):
    action_dict = {}
    assert 'robot_1_low' in obs or 'robot_2_low' in obs
    if 'robot_1_low' in obs and not done['robot_1_low']:
        action_dict['robot_1_low'] = random.choice(range(NUM_DIRECTIONS))
    if 'robot_2_low' in obs and not done['robot_2_low']:
        action_dict['robot_2_low'] = random.choice(range(NUM_DIRECTIONS))
    obs, rew, done, info = env.step(action_dict)
    print("Reward: ", rew)
    env.render()
    time.sleep(.1)
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/demo/demo_your_rllib_env.py

Learn Policies

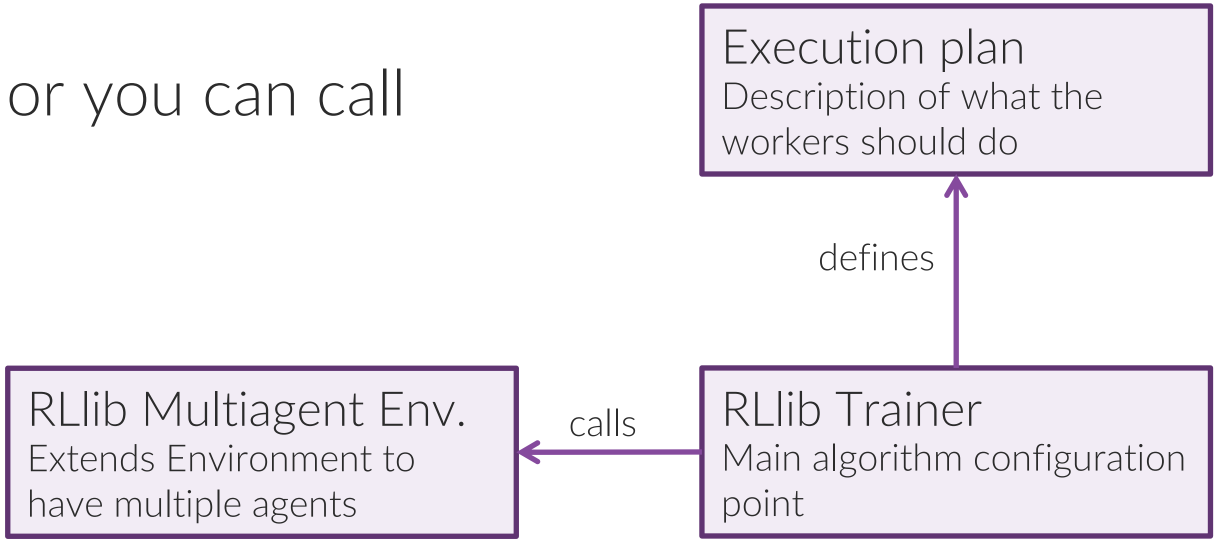
https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_config.py

```python
def policy_map_fn(agent_id: str, _episode=None, _worker=None, **_kwargs) -> str:
    """
    Maps agent_id to policy_id
    """
    if 'high' in agent_id:
        return 'high_level_policy'
    elif 'low' in agent_id:
        return 'low_level_policy'
    else:
        raise RuntimeError(f'Invalid agent_id: {agent_id}')


def get_multiagent_policies() -> Dict[str,PolicySpec]:
    policies: Dict[str,PolicySpec] = {}  # policy_id to policy_spec

    policies['high_level_policy'] = PolicySpec(
                policy_class=None, # use default in trainer
                observation_space=high_level_obs_space,
                action_space=high_level_action_space,
                config={}
    )

    policies['low_level_policy'] = PolicySpec(
        policy_class=None,  # use default in trainer
        observation_space=low_level_obs_space,
        action_space=low_level_action_space,
```



Train your agents

You can use Ray tune to train, or you can call the trainer directly.

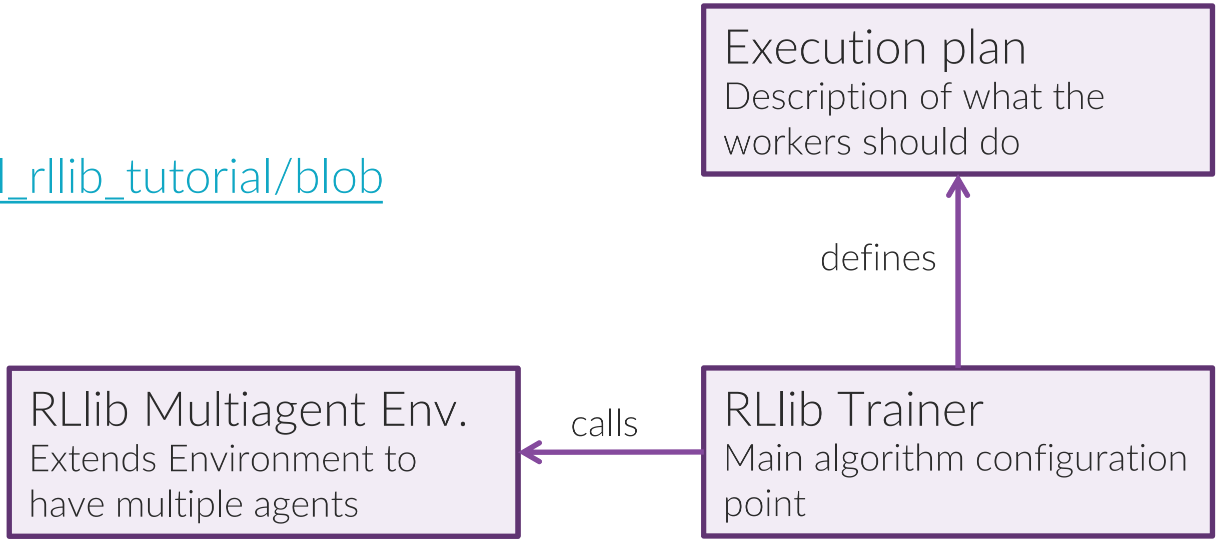



```
RUN_WITH_TUNE = True

# Tune is the system for keeping track of all of the running jobs, originally for
# hyperparameter tuning
if RUN_WITH_TUNE:

    tune.registry.register_trainable("YourTrainer", YourTrainer)
    stop = {
            "training_iteration": 500  # Each iteration is some number of episodes
        }
    results = tune.run("YourTrainer", stop=stop, config=config, verbose=1, checkpoint_freq=10)

    # You can probably just do PPO or DQN but we wanted to show how to customize
    #results = tune.run("PPO", stop=stop, config=config, verbose=1, checkpoint_freq=10)

    # Results at /Users/jmugan/ray_results/YourTrainer

else:
    from your_rllib_environment import YourEnvironment
    trainer = YourTrainer(config, env=YourEnvironment)

    # You can probably just do PPO or DQN but we wanted to show how to customize
    #from ray.rllib.agents.ppo import PPOTrainer
    #trainer = PPOTrainer(config, env=YourEnvironment)

    trainer.train()
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_train.py

You can create a custom trainer

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_trainer.py



A custom trainer allows you to do any operation you want. It can be a little trickly to get it all working together, but there are few limits to customization.

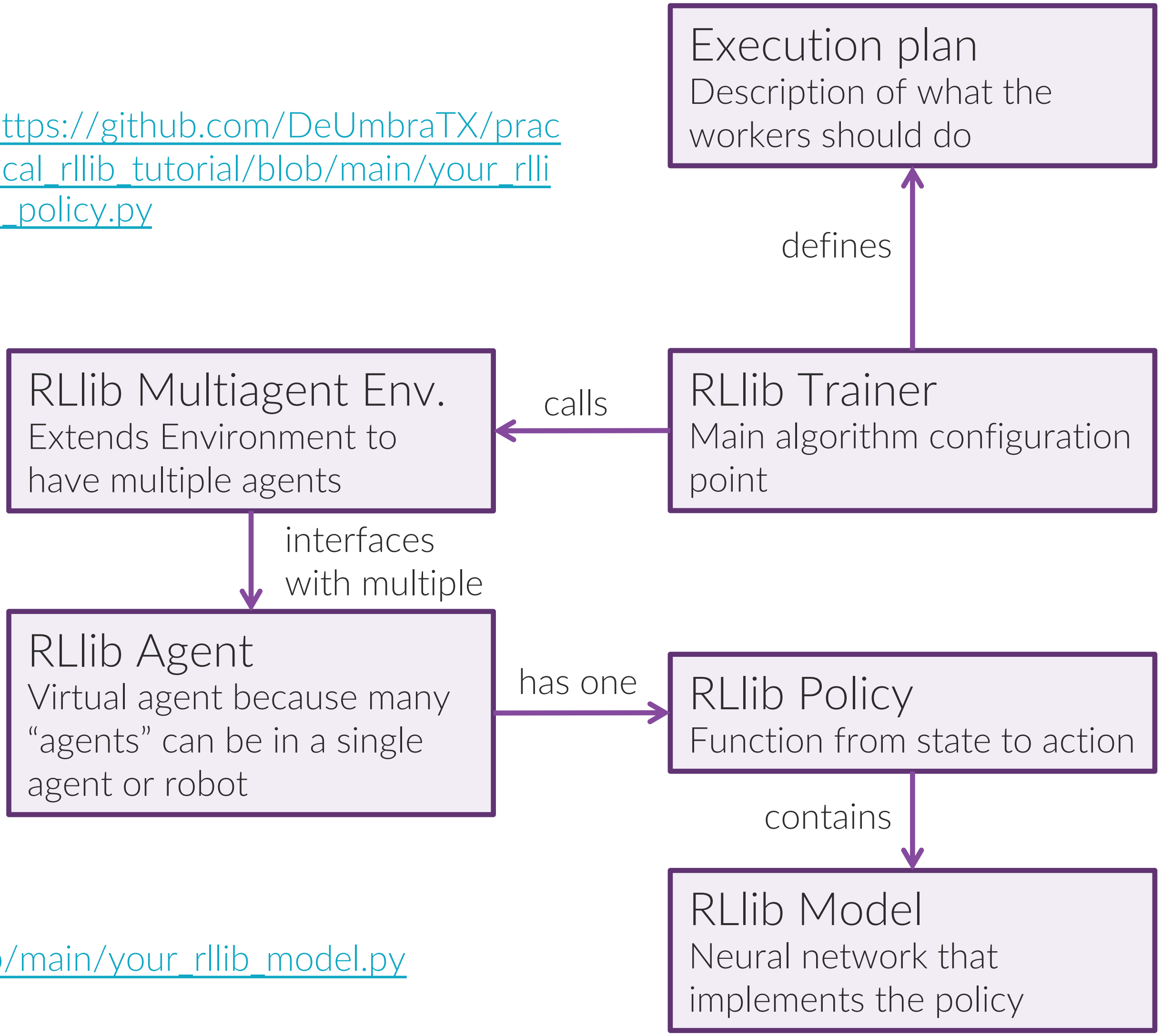
And you can have a custom policy & model

```python
class YourHighLevelPolicy(TorchPolicy):

    def __init__(self, observation_space, action_space, config):
        your_model = YourModel(observation_space, action_space,
                               num_outputs=1, model_config=config,
                               name='YourModel')
        self.action_space = action_space
        super().__init__(
            observation_space,
            action_space,
            config,
            model = your_model)
        # if you don't pass it a model it will create one automatically
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_policy.py

```python
from ray.rllib.models.torch.torch_modelv2 import TorchModelV2


class YourModel(TorchModelV2):
    pass
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_model.py



You customize a huge configuration dictionary

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/your_rllib_config.py

```python
def policy_map_fn(agent_id: str, _episode=None, _worker=None, **_kwargs) -> str:
    """
    Maps agent_id to policy_id
    """
    if 'high' in agent_id:
        return 'high_level_policy'
    elif 'low' in agent_id:
        return 'low_level_policy'
    else:
        raise RuntimeError(f'Invalid agent_id: {agent_id}')


def get_multiagent_policies() -> Dict[str,PolicySpec]:
    policies: Dict[str,PolicySpec] = {}  # policy_id to policy_spec

    policies['high_level_policy'] = PolicySpec(
                policy_class=None, # use default in trainer
                observation_space=high_level_obs_space,
                action_space=high_level_action_space,
                config={}
    )

    policies['low_level_policy'] = PolicySpec(
        policy_class=None,  # use default in trainer
        observation_space=low_level_obs_space,
        action_space=low_level_action_space,
```

RLlib Trainer
Main algorithm configuration point

Three levels of configuration

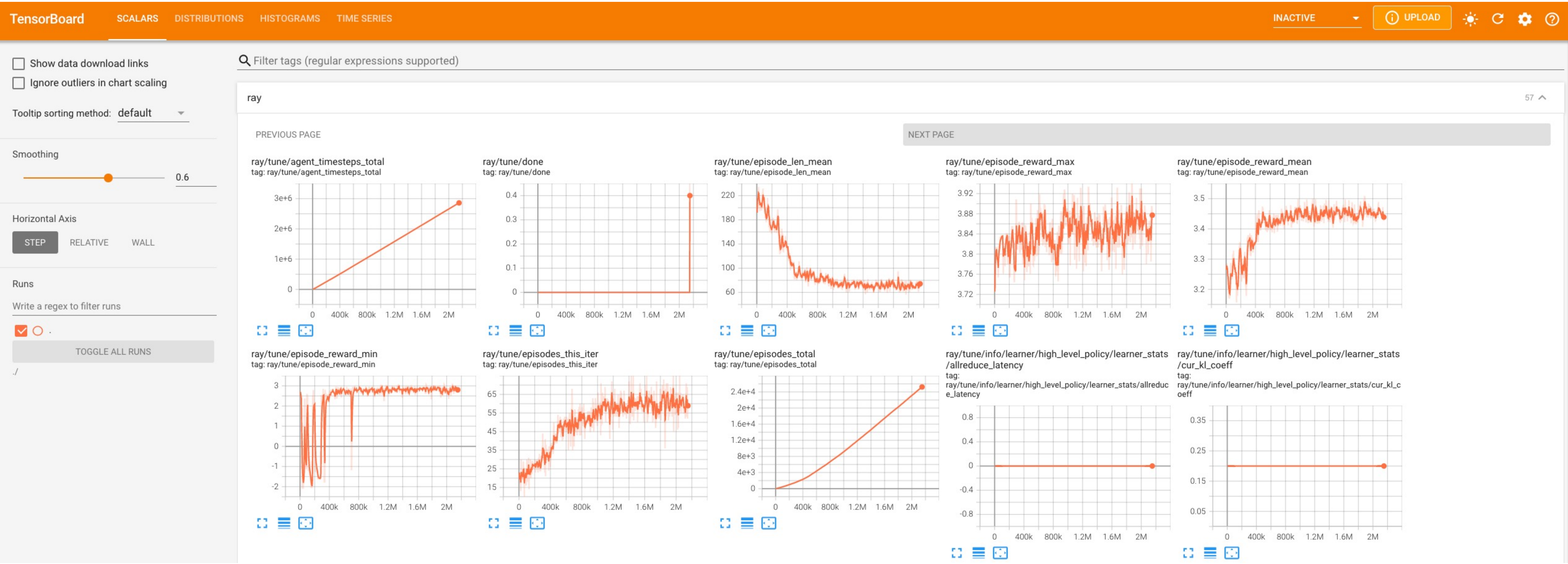
1. Configuration from the trainer you are basing off
   - https://github.com/ray-project/ray/blob/releases/1.10.0/rllib/agents/trainer.py
2. Configuration for your trainer
   - E.g., https://github.com/ray-project/ray/blob/releases/1.10.0/rllib/agents/ppo/ppo.py
3. Mapping of agents to policies
   - Our config above

And now we are back to train and are ready to run

```
# Tune is the system for keeping track of all of the running jobs, originally for
# hyperparameter tuning
if RUN_WITH_TUNE:

    tune.registry.register_trainable("YourTrainer", YourTrainer)
    stop = {
            "training_iteration": 500  # Each iteration is some number of episodes
        }
    results = tune.run("YourTrainer", stop=stop, config=config, verbose=1, checkpoint_freq=10)

    # You can probably just do PPO or DQN but we wanted to show how to customize
    #results = tune.run("PPO", stop=stop, config=config, verbose=1, checkpoint_freq=10)

    # Results at /Users/jmugan/ray_results/YourTrainer

else:
    from your_rllib_environment import YourEnvironment
    trainer = YourTrainer(config, env=YourEnvironment)

    # You can probably just do PPO or DQN but we wanted to show how to customize
    #from ray.rllib.agents.ppo import PPOTrainer
    #trainer = PPOTrainer(config, env=YourEnvironment)

    trainer.train()
```

https://github.com/DeUmbraTX/deumbra_rllib_tutorial/blob/main/your_rllib_train.py

Creates 57 Graphs in TensorBoard



Hopefully, something is on (for those who remember the 1990s).
https://www.youtube.com/watch?v=YAlDbP4tdqc&ab_channel=BruceSpringsteenVEVO

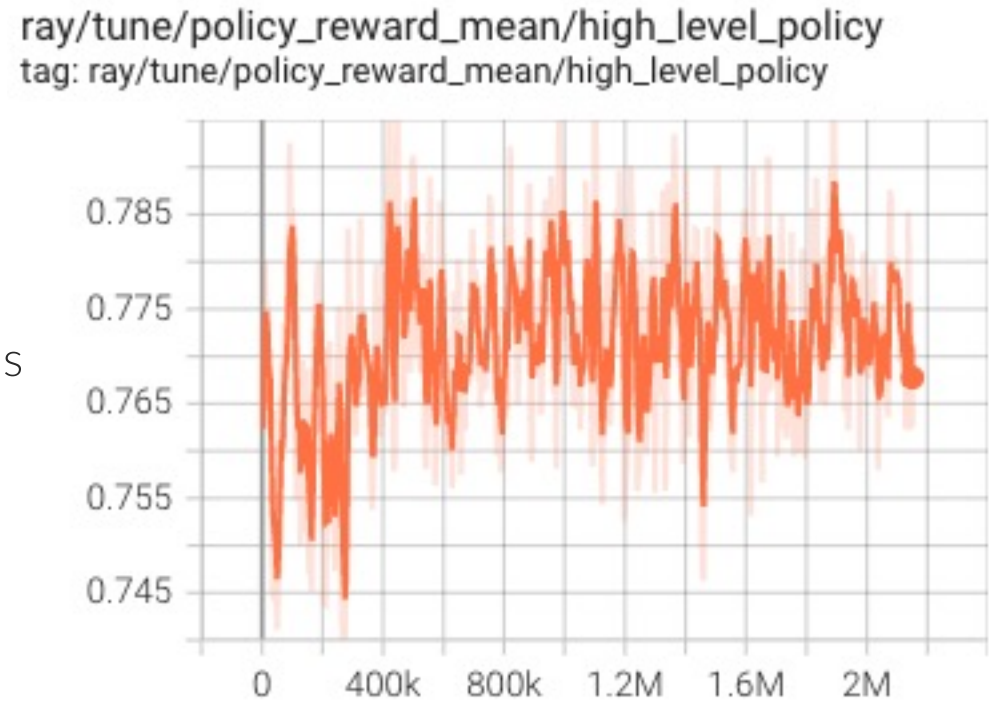
Training Results

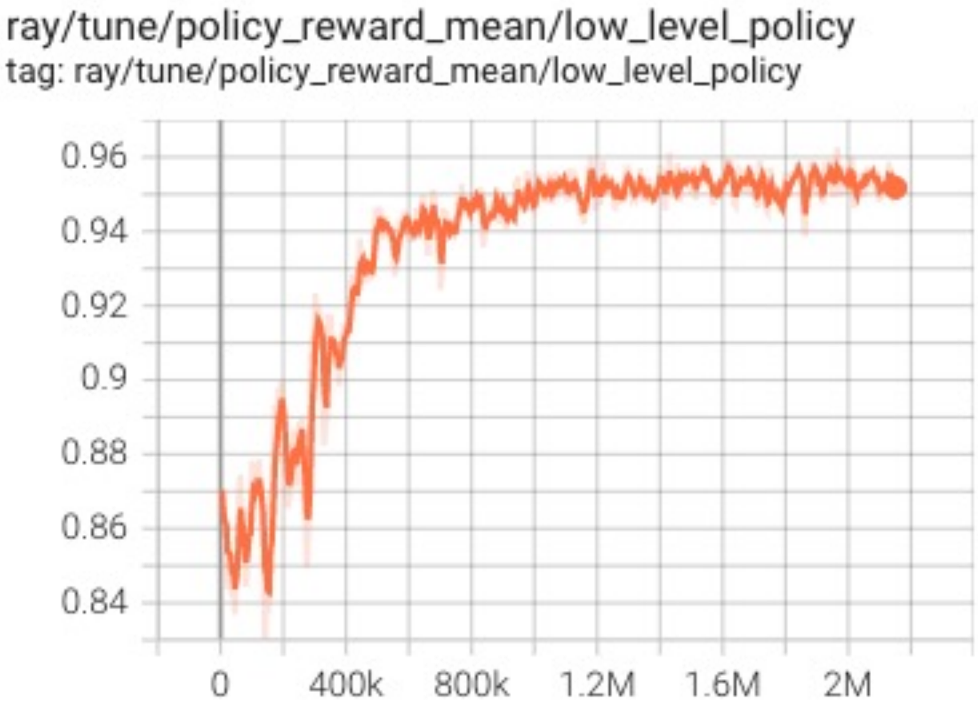
Go to `/Users/you/ray_results`
to find your run and type `tensorboard logdir=./`

- I let it run on my laptop for about an hour. That's a long time for a small environment, but we didn't add any smart features.
- RLlib lets you put this directly on a cluster of machines.

The robots share a high-level policy and a low-level policy. Great for swarm training.

The high-level policy learned at the beginning but there is a lot of randomness (finding the right person is hard).





x-axis is timesteps, y-axis is reward

DeUmbra

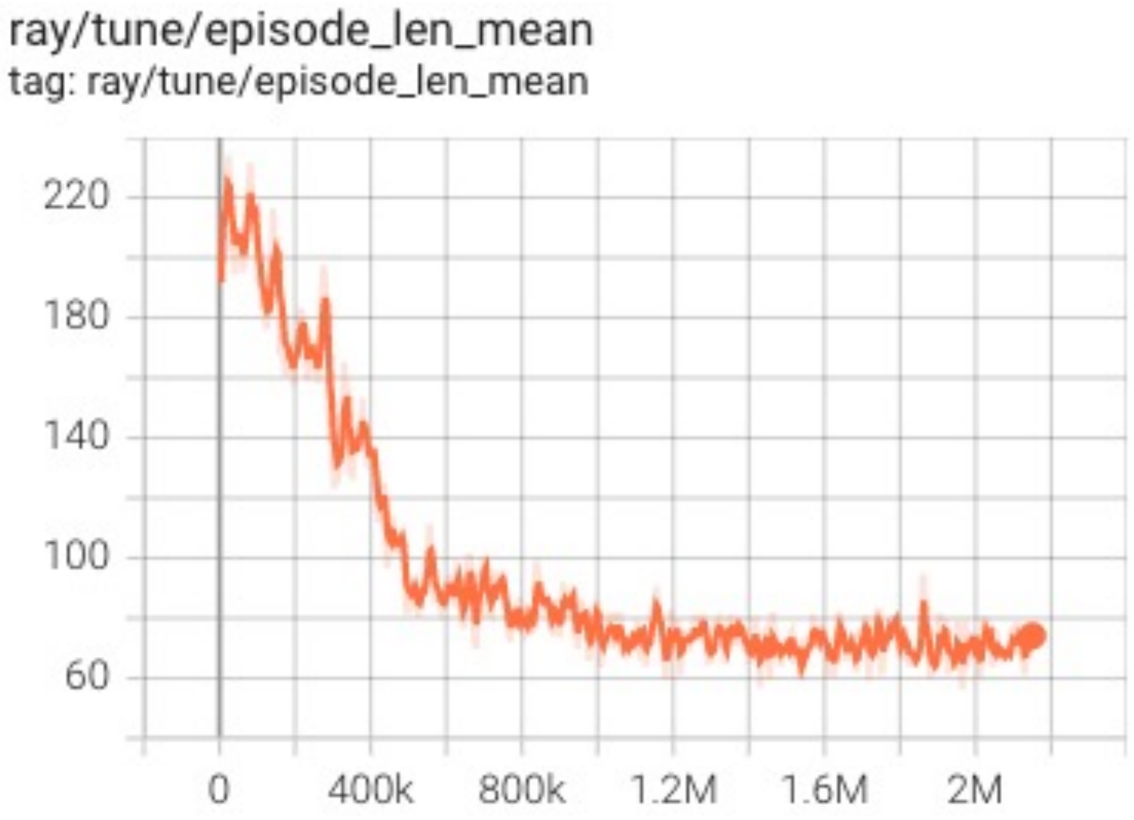
Training Results

- I let it run on my laptop for about an hour. That's a long time for a small environment, but we didn't add any smart features.
- RLlib lets you put this directly on a cluster of machines.

Go to `/Users/you/ray_results`
to find your run and type `tensorboard logdir=./`

Recall that they are punished a small amount on every timestep.

They learn to find chickens increasingly quickly.



x-axis is timesteps, y-axis is time for robots to find a chicken

Loading and using the policies

After the policy is learned,
we can load it up and get the best action for any state

```
# Note that they both use the same policy
robot_1_high_action = trainer.compute_single_action(obs['robot_1_high'], policy_id='high_level_policy')
robot_2_high_action = trainer.compute_single_action(obs['robot_2_high'], policy_id='high_level_policy')

action_dict = {'robot_1_high': robot_1_high_action,
               'robot_2_high': robot_2_high_action}

obs, rew, done, info = env.step(action_dict)

env.render()

def is_all_done(done: Dict) -> bool:
    for key, val in done.items():
        if not val:
            return False
    return True

while not is_all_done(done):
    action_dict = {}
    assert 'robot_1_low' in obs or 'robot_2_low' in obs
    if 'robot_1_low' in obs and not done['robot_1_low']:
        action_dict['robot_1_low'] = trainer.compute_single_action(obs['robot_1_low'], policy_id='low_level_policy')
    if 'robot_2_low' in obs and not done['robot_2_low']:
        action_dict['robot_2_low'] = trainer.compute_single_action(obs['robot_2_low'], policy_id='low_level_policy')
    obs, rew, done, info = env.step(action_dict)
    print("Reward: ", rew)
    env.render()
    #time.sleep(1)
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/demo/demo_after_training.py

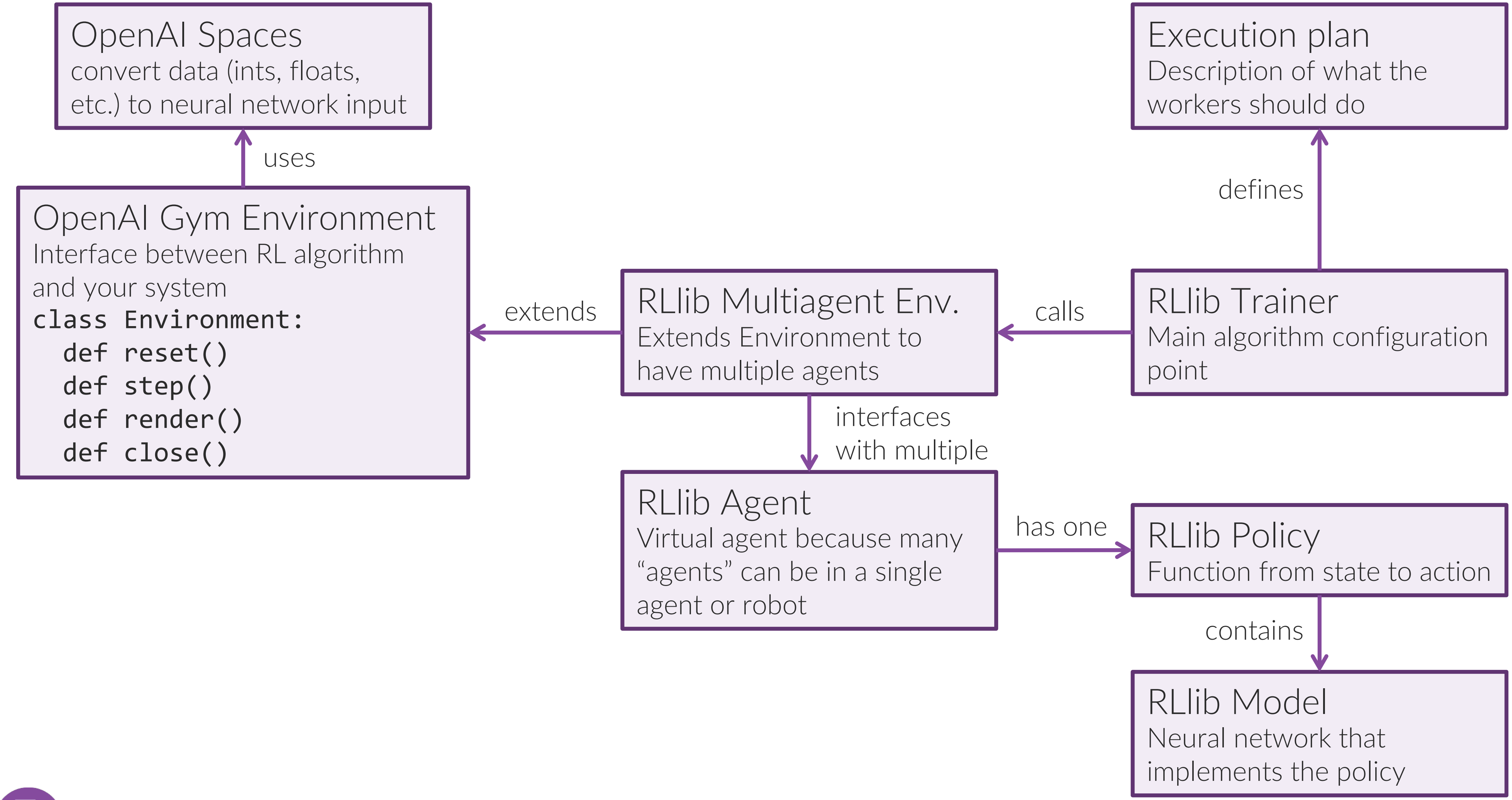
Looking at the policy

You can look at the size of the automatically created neural network.

```
# Change these for your run
run = 'YourTrainer_YourEnvironment_1e97d_00000_0_2022-02-20_13-13-00'
checkpoint = 'checkpoint_000500/checkpoint-500'

restore_point = os.path.join(YOUR_ROOT, run, checkpoint)
trainer.restore(restore_point)

print("************************ high level policy ***************************************")
# Note the output size of 10
policy: PPOTorchPolicy = trainer.get_policy('high_level_policy')
# https://github.com/ray-project/ray/blob/releases/1.10.0/rllib/models/torch/complex_input_net.py
model: ComplexInputNetwork = policy.model
for m in model.variables():
    print(m.shape)

print("************************ low level policy ***************************************")
# Note the output size of 8
policy: PPOTorchPolicy = trainer.get_policy('low_level_policy')
model: ComplexInputNetwork = policy.model
for m in model.variables():
    print(m.shape)
```

https://github.com/DeUmbraTX/practical_rllib_tutorial/blob/main/demo/demo_look_at_policies.py

Now we have seen all the abstractions



Additional RLlib Resources

Paper https://arxiv.org/pdf/1712.09381.pdf
GitHub https://github.com/ray-project/ray/tree/master/rllib
Documentation https://docs.ray.io/en/master/rllib/index.html
Discussion https://discuss.ray.io/top?period=monthly

Parting thoughts

Reinforcement learning is a gradual stamping in of behavior, so you need lots of observations or a simulator.

You also need useful representations, which is what DeUmbra focuses on.

DeUmbra builds representations to make reinforcement learning maximally effective.

DeUmbra

Jonathan Mugan, PhD
@jmugan
jmugan@deumbra.com

6500 River Place Blvd.
Bldg. 3, Suite 120
Austin, TX. 78730