Intro
Human vs AI

Atari Games: Human VS AI

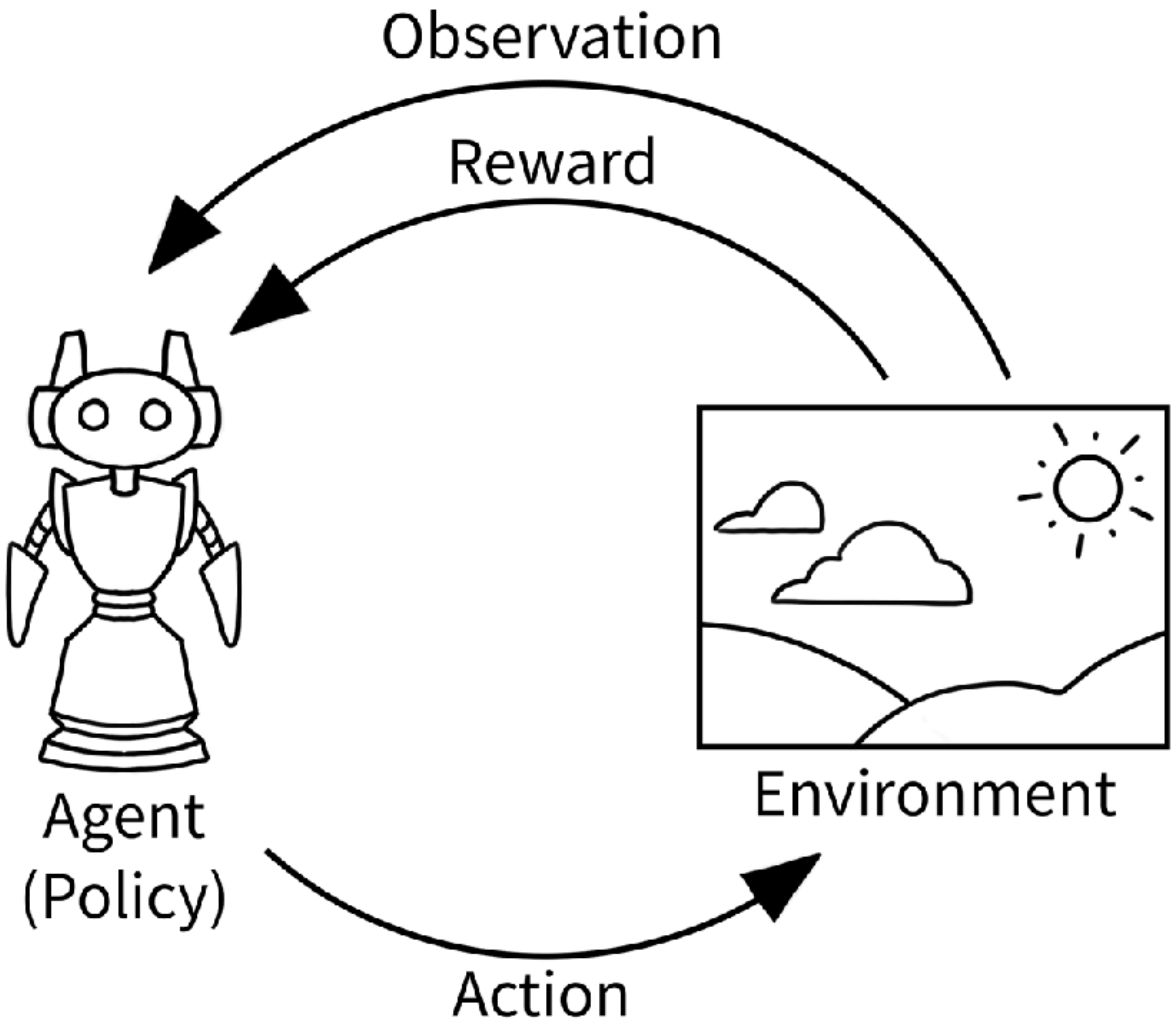
The leaderboard: https://github.com/cshenton/atari-leaderboard

Game	Top Human Score	Top Machine Score	Best	Best Machine	Learning Type	Notes
Alien	103583	9491	Human	Rainbow	Q-gradient	
Amidar	71529	5131	Human	Rainbow	Q-gradient	
Assault	8647	14497	Machine	A3C	Policy-gradient	
Asterix	1000000	428200	Human	Rainbow	Q-gradient	
Asteroids	57340	5093	Human	A3C	Policy-gradient	*
Atlantis	10604840	2311815	Human	PPO	Policy-gradient	
Bank Heist	45899	1611	Human	Dueling DDQN	Q-gradient	
Battlezone	98000	62010	Human	Rainbow	Q-gradient	
Beamrider	52866	26172	Human	Prioritized DDQN	Q-gradient	1B
Berzerk	1057940	2545	Human	Rainbow	Q-gradient	
Bowling	279	135	Human	HyperNEAT	Genetic Policy	J

Section 1
Gym Environment

Gym Env

…implements the classic “agent-environment loop”



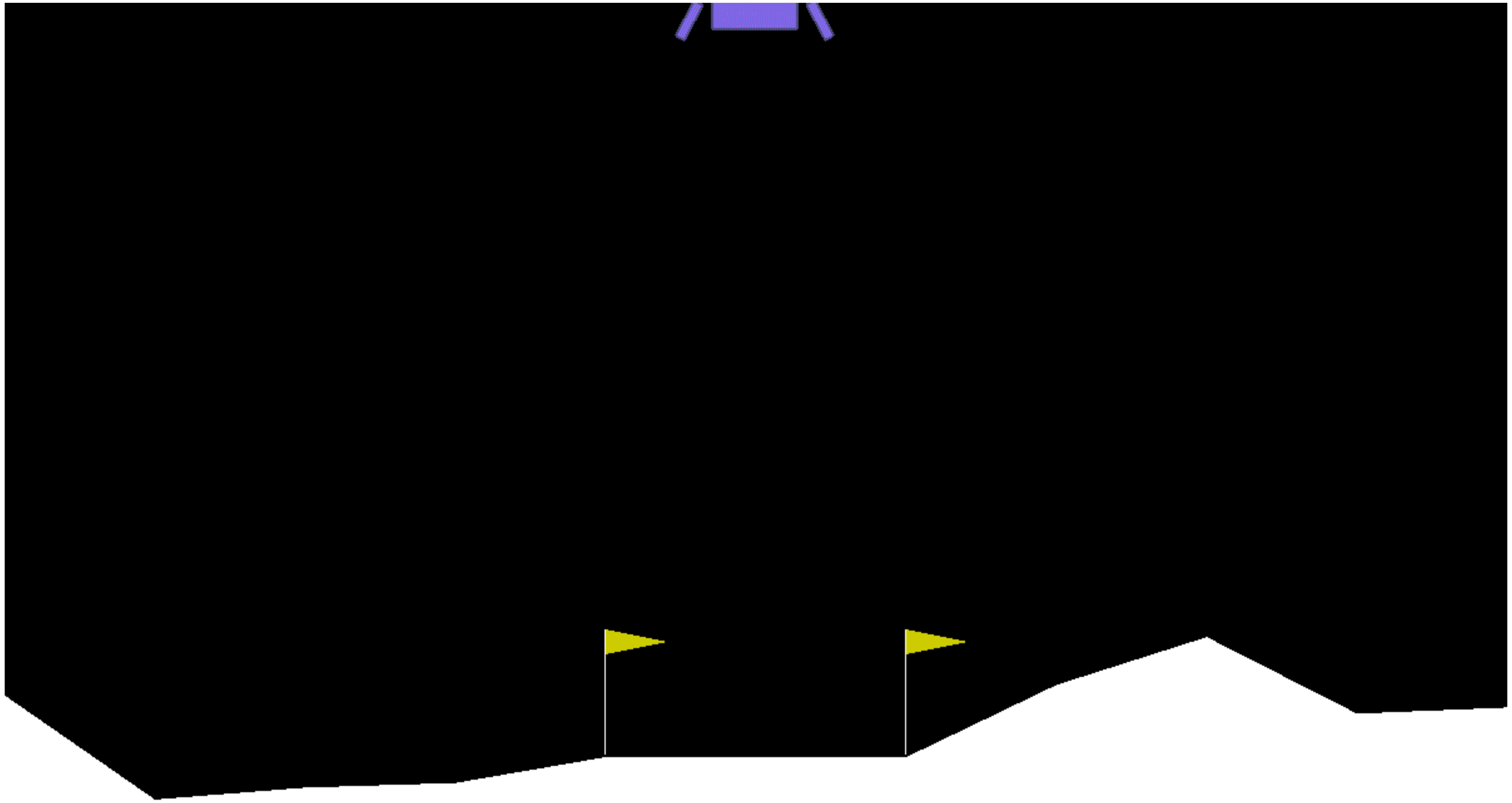
Gym Env

…is a standard API for reinforcement learning…

```python
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
    action = policy(observation)  # User-defined policy function
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
env.close()
```

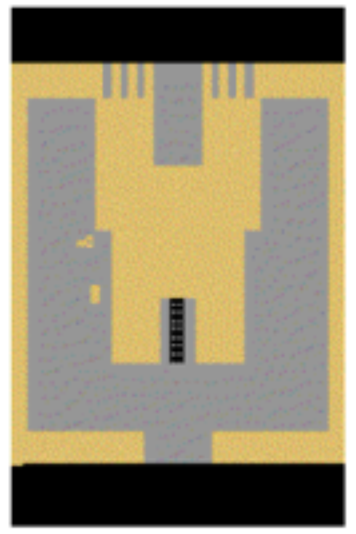
Gym Env

...is a standard API for reinforcement learning...

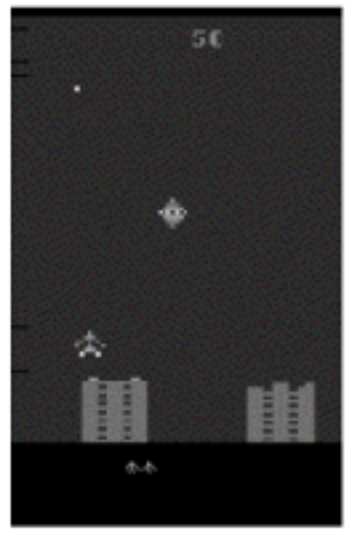


Gym Env

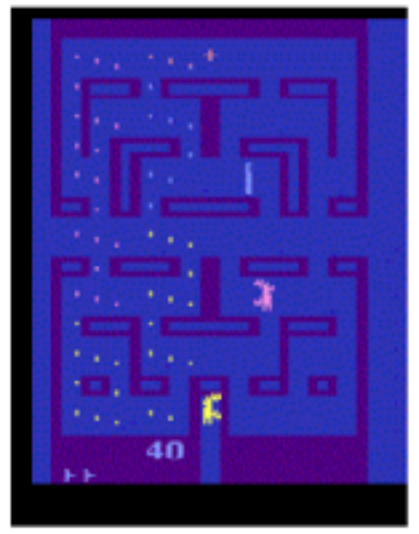
...and a diverse collection of reference environments



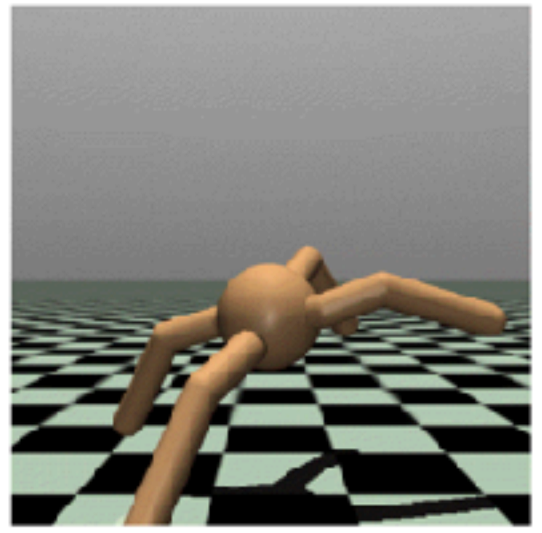
Adventure



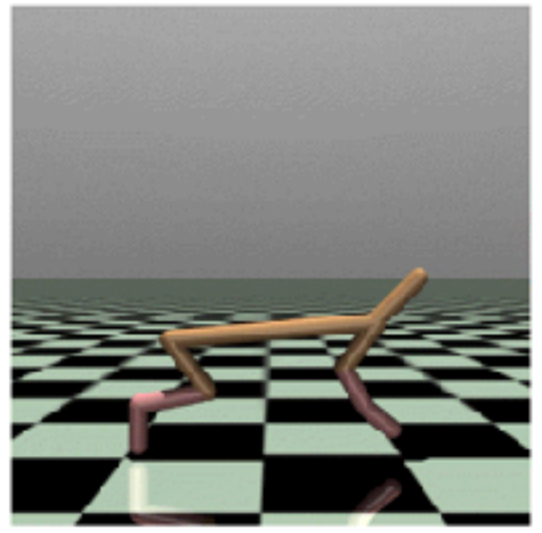
Air Raid



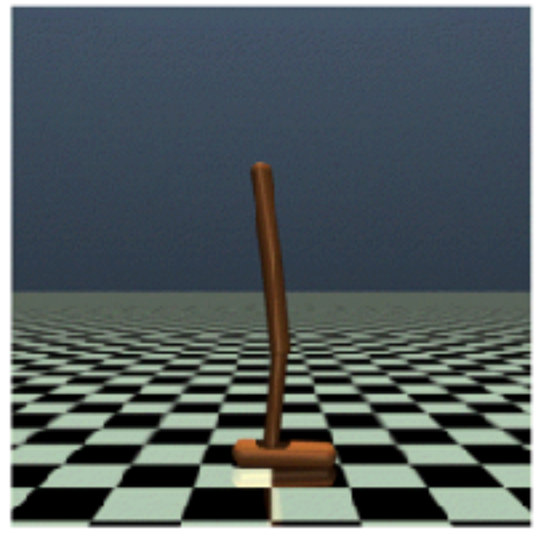
Alien



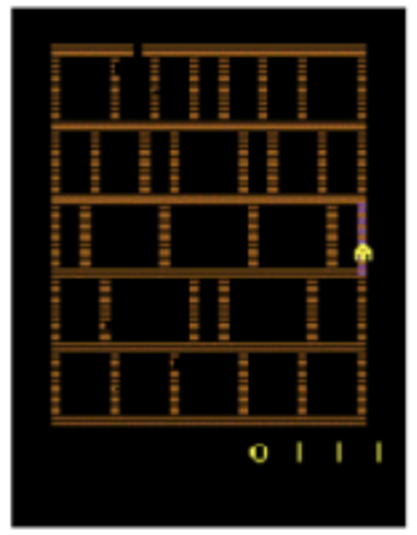
Ant



Half Cheetah

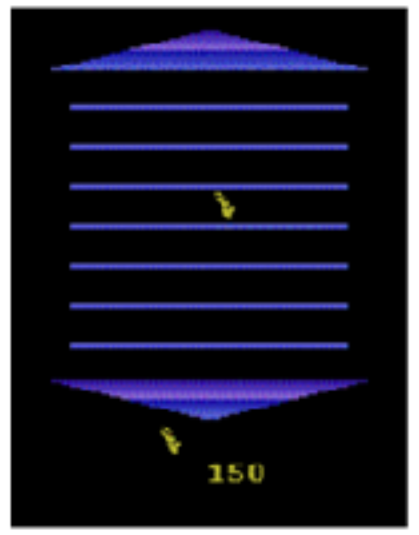


Hopper



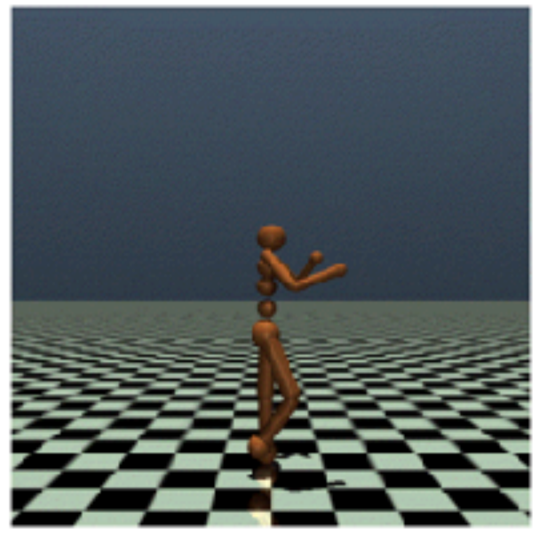
Amidar

Assault

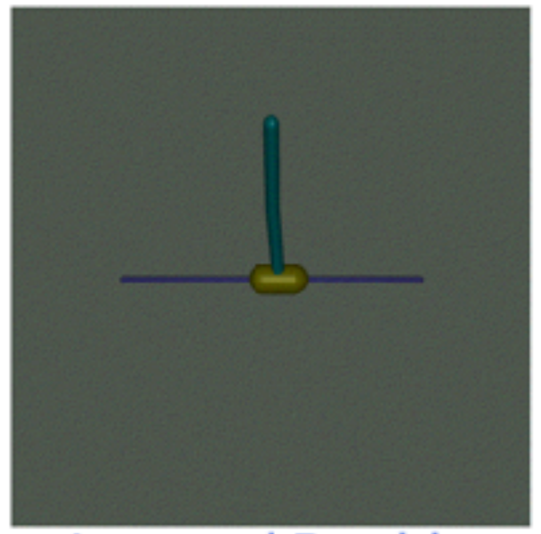


Asterix

Humanoid Standup



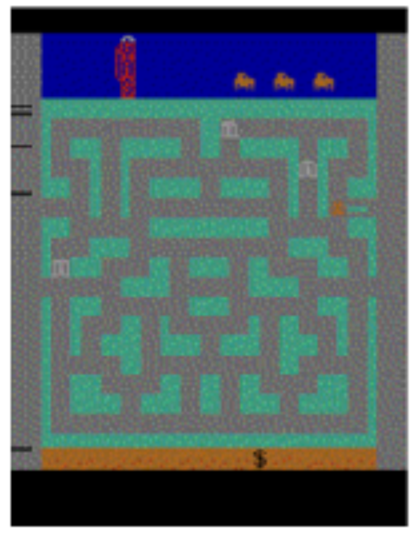
Humanoid



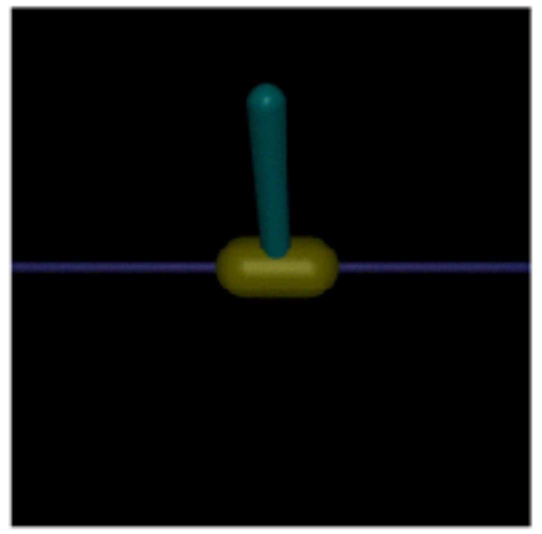
Inverted Double Pendulum

Asteroids

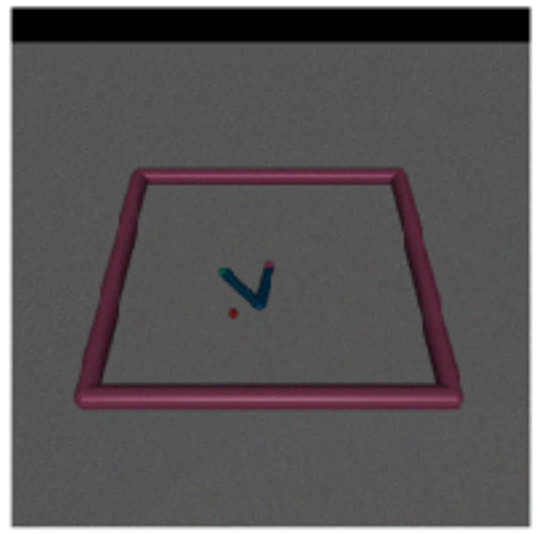
Atlantis



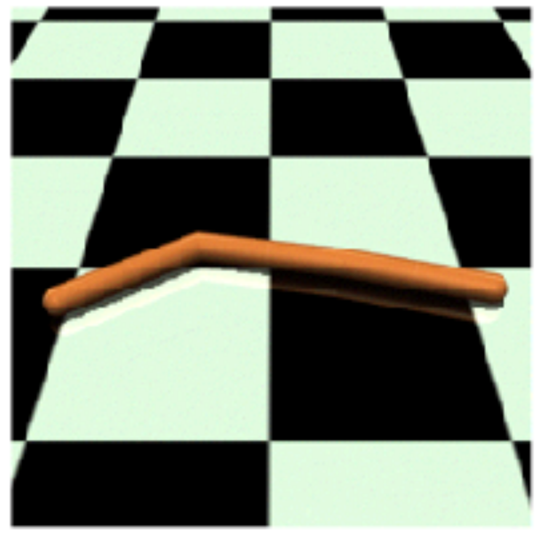
Bank Heist



Inverted Pendulum



Reacher



Swimmer

Gym Env
Demo-1

Section 2
Ray RLlib

RAY RLlib

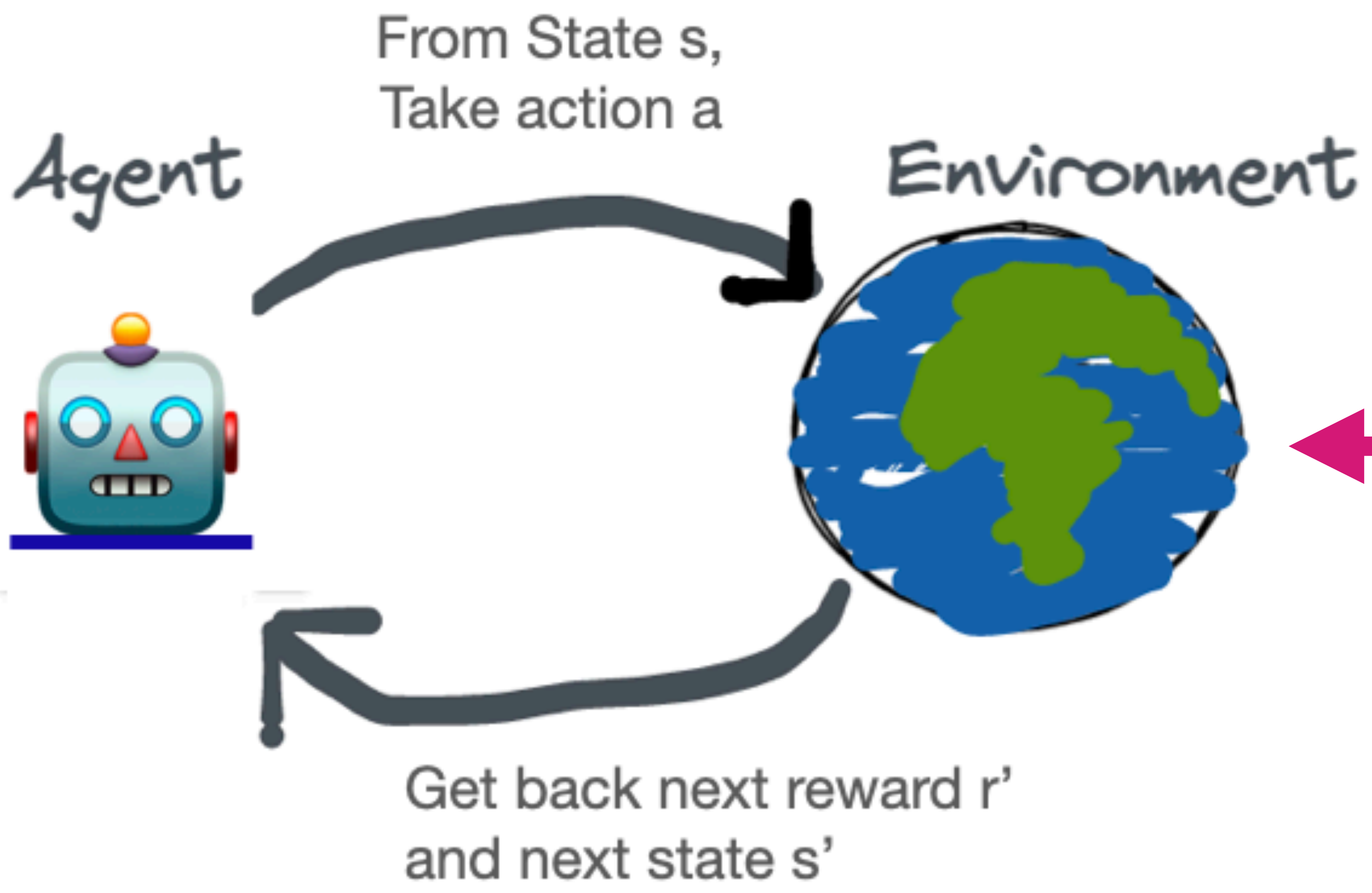
Industry-Grade Reinforcement Learning

RAY RLlib

Components

Policy

Algorithm



Environment

Trajectories

RAY RLlib

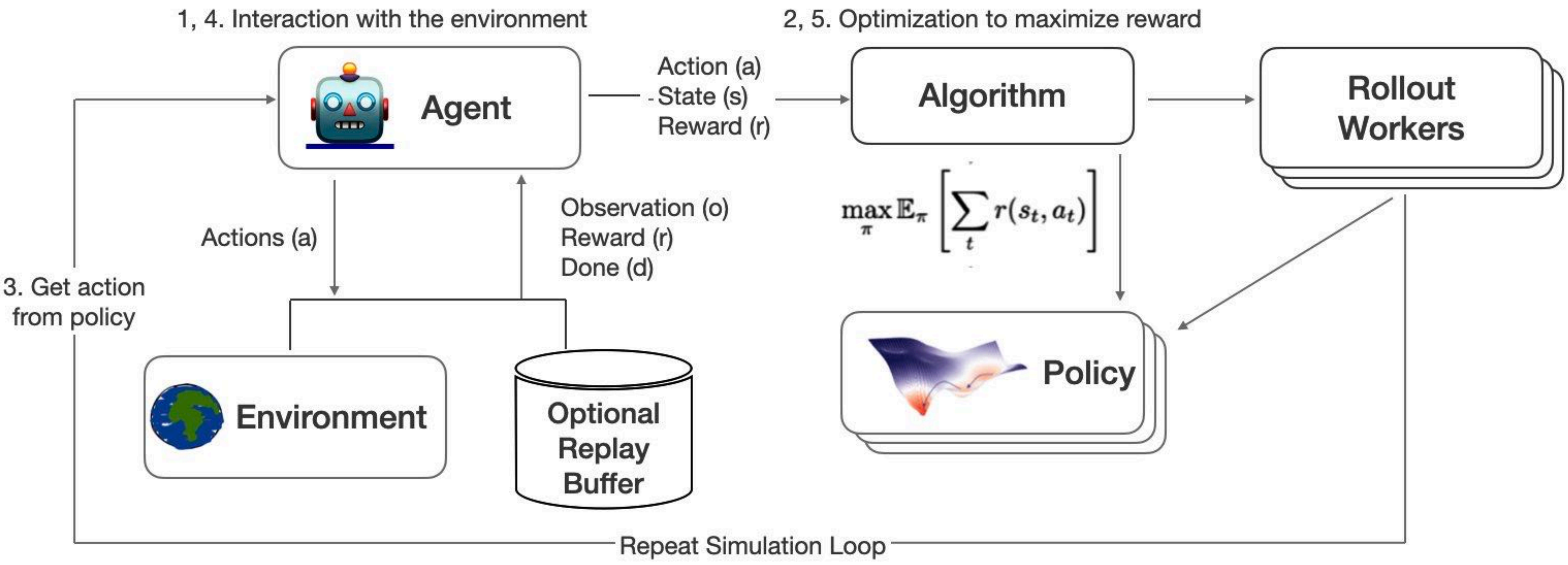
Environment

- All possible actions (**action space**)
- **A** complete description of the environment, nothing hidden (**state space**)
- An observation by the agent of certain parts of the state (**observation space**)
- **Reward**, which is the only feedback the agent receives per action

  The model that tries to maximize the expected sum over all future rewards is called a **policy**. The policy is a function mapping the environment's observations to an action to take, usually written **π** (s(t)) -> a(t).

RAY RLlib

A diagram of the RL iterative learning process



RAY RLlib

Industry-Grade Reinforcement Learning

RAY

RLlib

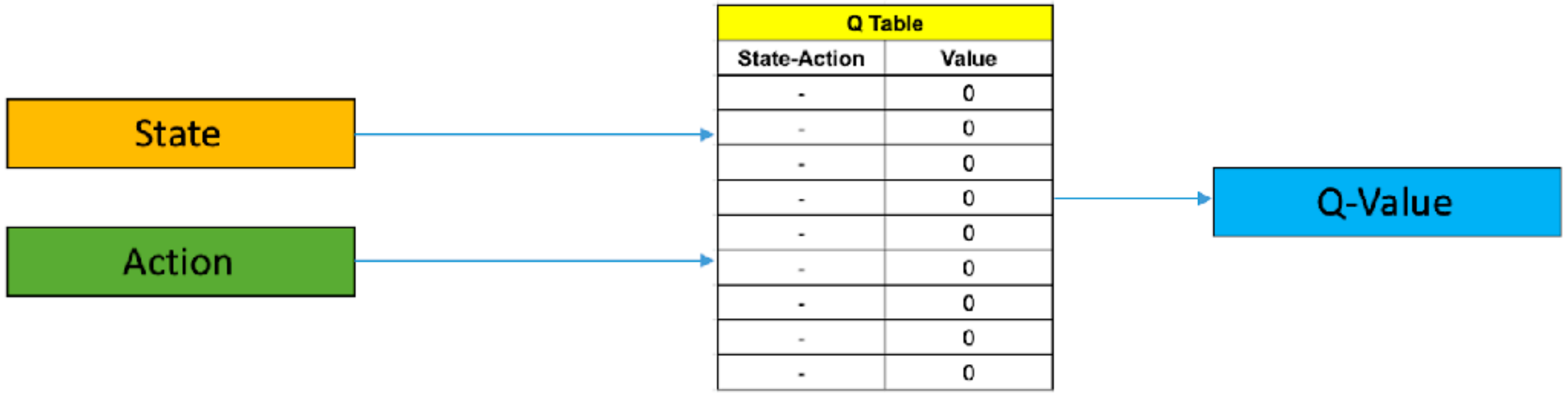
```python
# Import the RL algorithm (Algorithm) we would like to use.
from ray.rllib.algorithms.ppo import PPO

# Configure the algorithm.
config = {
    # Environment (RLlib understands openAI gym registered strings).
    "env": "Taxi-v3",
    # Use 2 environment workers (aka "rollout workers") that parallelly
    # collect samples from their own environment clone(s).
    "num_workers": 2,
    # Change this to "framework: torch", if you are using PyTorch.
    # Also, use "framework: tf2" for tf2.x eager execution.
    "framework": "tf",
    # Tweak the default model provided automatically by RLlib,
    # given the environment's observation- and action spaces.
    "model": {
        "fcnet_hiddens": [64, 64],
        "fcnet_activation": "relu",
    },
    # Set up a separate evaluation worker set for the
    # `algo.evaluate()` call after training (see below).
    "evaluation_num_workers": 1,
    # Only for evaluation runs, render the env.
    "evaluation_config": {
        "render_env": True,
    },
}

# Create our RLlib Trainer.
algo = PPO(config=config)

# Run it for n training iterations. A training iteration includes
# parallel sample collection by the environment workers as well as
# loss calculation on the collected batch and a model update.
for _ in range(3):
    print(algo.train())

# Evaluate the trained Trainer (and render each timestep to the shell's
# output).
algo.evaluate()
```

Ray RLlib
Demo-2

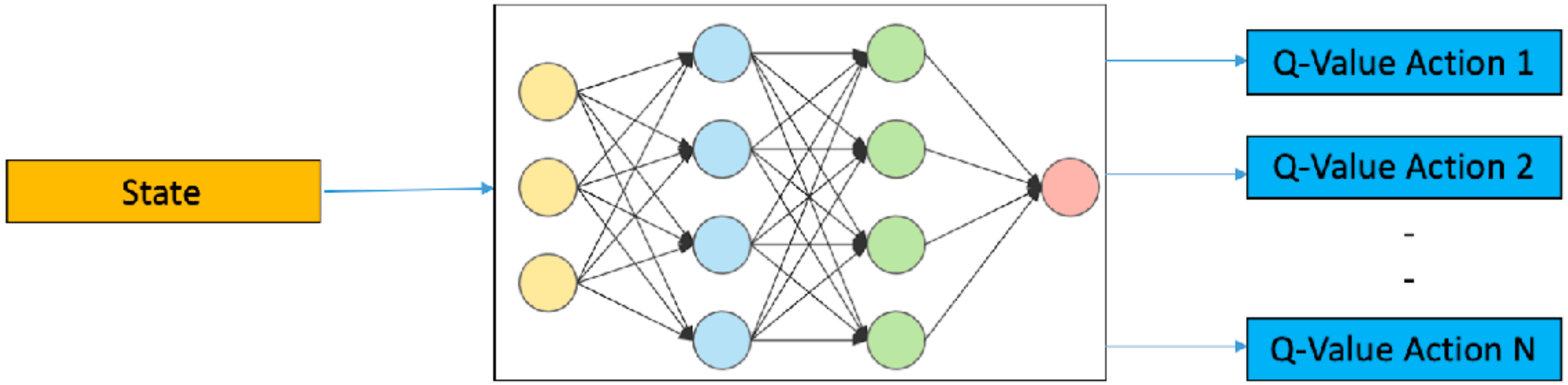
Section 3
DQN, Double DQN, APEX-DQN, Rainbow

DQN = Deep Q-Network (1st gen)

Predicts Q-values (expected future rewards) for state-action pairs



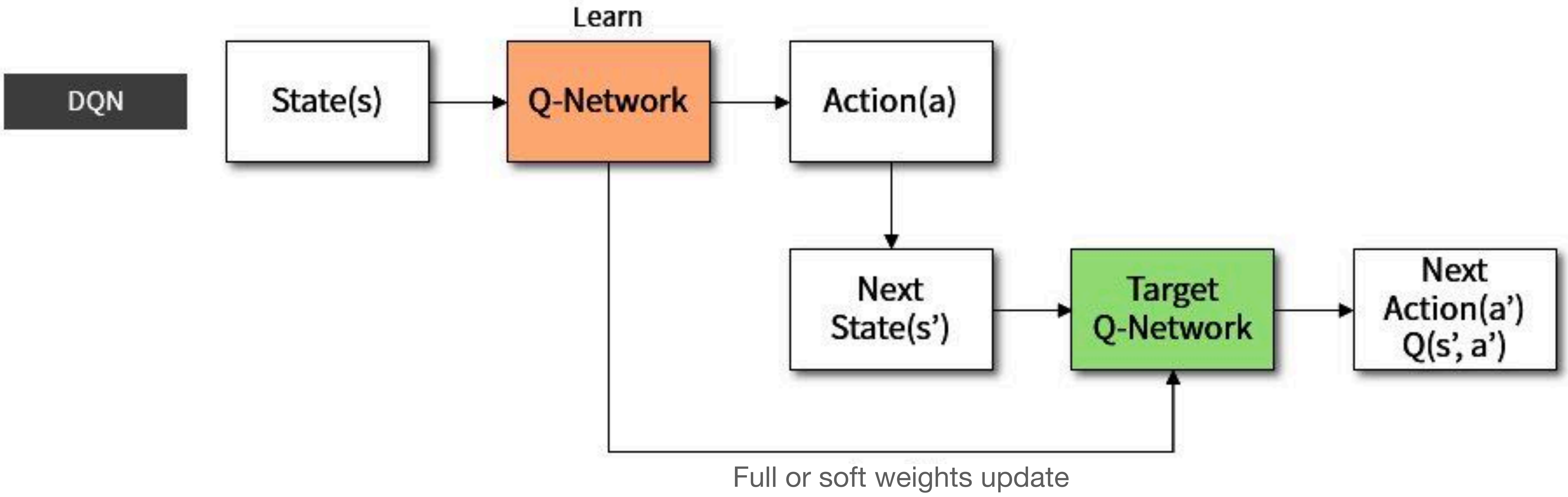
Q Learning



Deep Q Learning

DQN = Deep Q-Network

Training (one network)

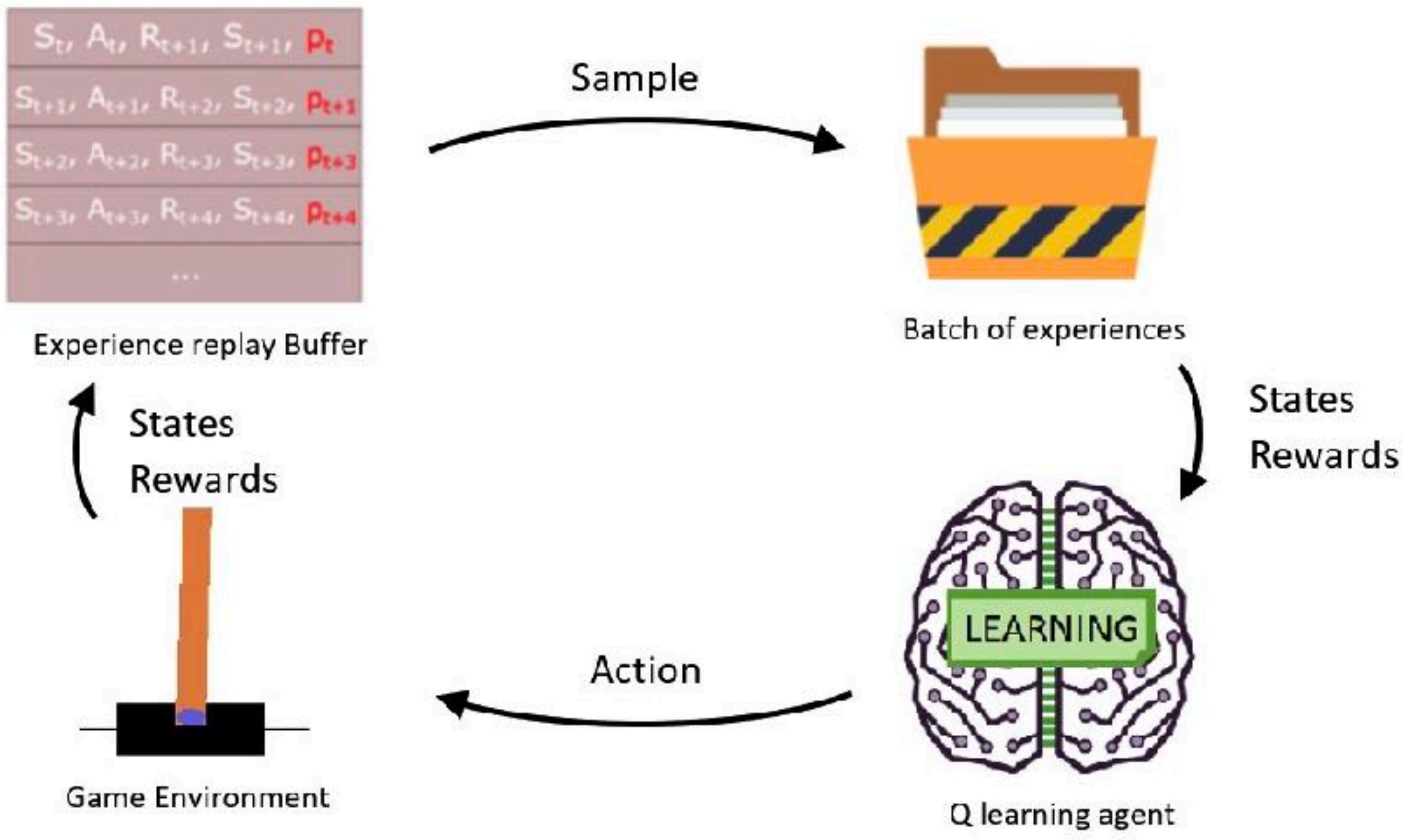


$$\boxed{Q(s,a) = Q(s,a) + \alpha(R + \gamma \max Q(s',a') - Q(s,a))}$$

Q-Network - selects & evaluates actions

DQN = Deep Q-Network

Experience replay




```python
def replay(self):
    if len(self.memory) < self.train_start:
        return
    # Randomly sample minibatch from the memory
    minibatch = random.sample(self.memory, min(len(self.memory), self.batch_size))

    state = np.zeros((self.batch_size, self.state_size))
    next_state = np.zeros((self.batch_size, self.state_size))
    action, reward, done = [], [], []

    # do this before prediction
    # for speedup, this could be done on the tensor level
    # but easier to understand using a loop
    for i in range(self.batch_size):
        state[i] = minibatch[i][0]
        action.append(minibatch[i][1])
        reward.append(minibatch[i][2])
        next_state[i] = minibatch[i][3]
        done.append(minibatch[i][4])

    # do batch prediction to save speed
    target = self.model.predict(state)
    target_next = self.model.predict(next_state)

    for i in range(self.batch_size):
        # correction on the Q value for the action used
        if done[i]:
            target[i][action[i]] = reward[i]
        else:
            # Standard - DQN
            # DQN chooses the max Q value among next actions
            # selection and evaluation of action is on the target Q Network
            # Q_max = max_a' Q_target(s', a')
            target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next[i]))

    # Train the Neural Network with batches
    self.model.fit(state, target, batch_size=self.batch_size, verbose=0)
```

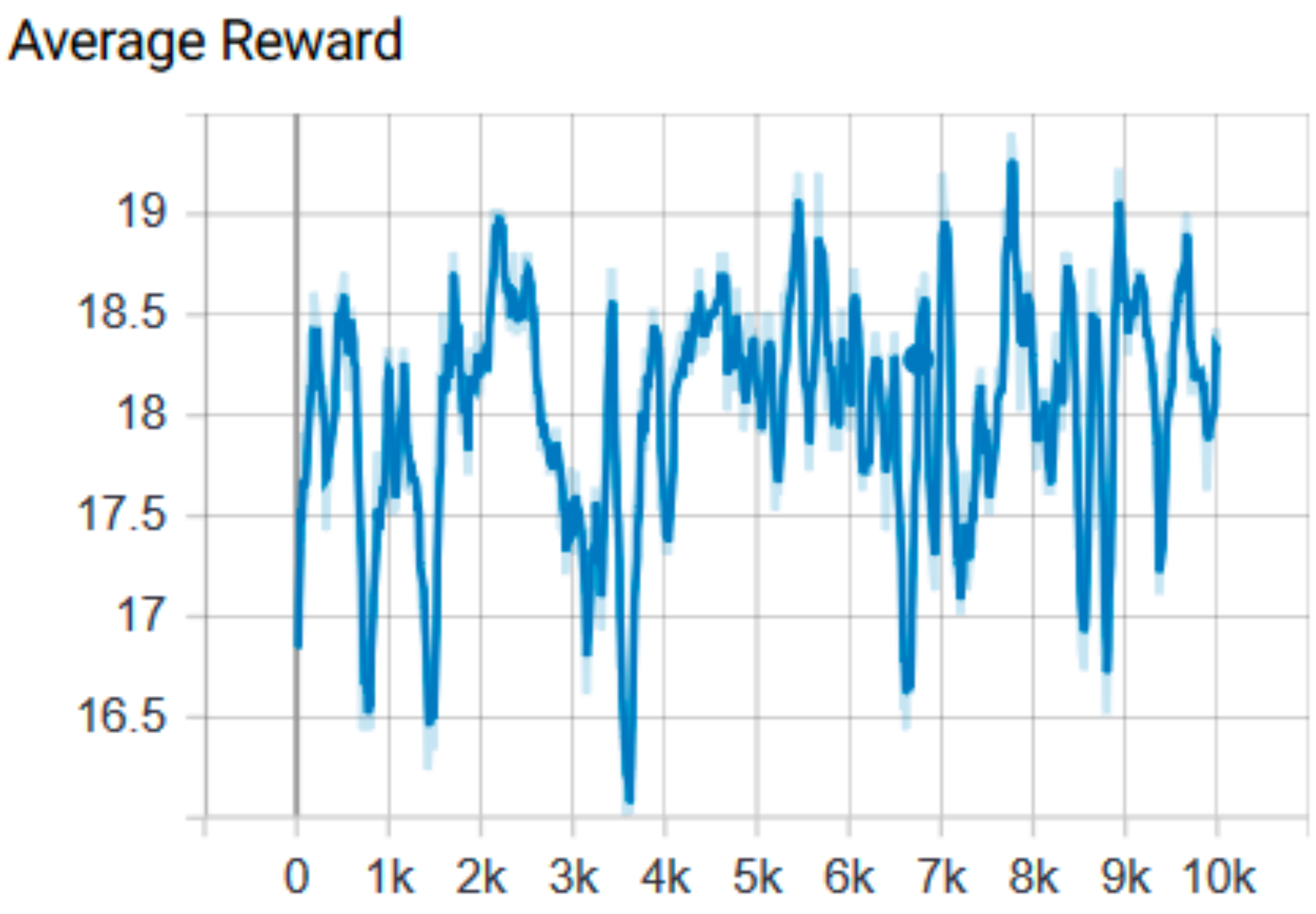
DQN = Deep Q-Network

Training (one network) + Experience Replay

```
trainer = env.train([None, against])
observations = trainer.reset()
while not done:

    # take an action and store outcome
    action = TrainNet.get_action(observations, epsilon)
    prev_observations = observations
    observations, reward, done, _ = env.step(action)

    # Adding experience into buffer
    exp = {'s': prev_observations, 'a': action, 'r': reward,
           's2': observations, 'done': done}
    TrainNet.add_experience(exp)

    # Train the training model by using experiences in buffer
    # and the target model
    TrainNet.train(TargetNet)

    iter += 1
    if iter % copy_step == 0:
        # Update the weights of the target model after
        # reaching copy  interval
        TargetNet.copy_weights(TrainNet)

    return reward
```

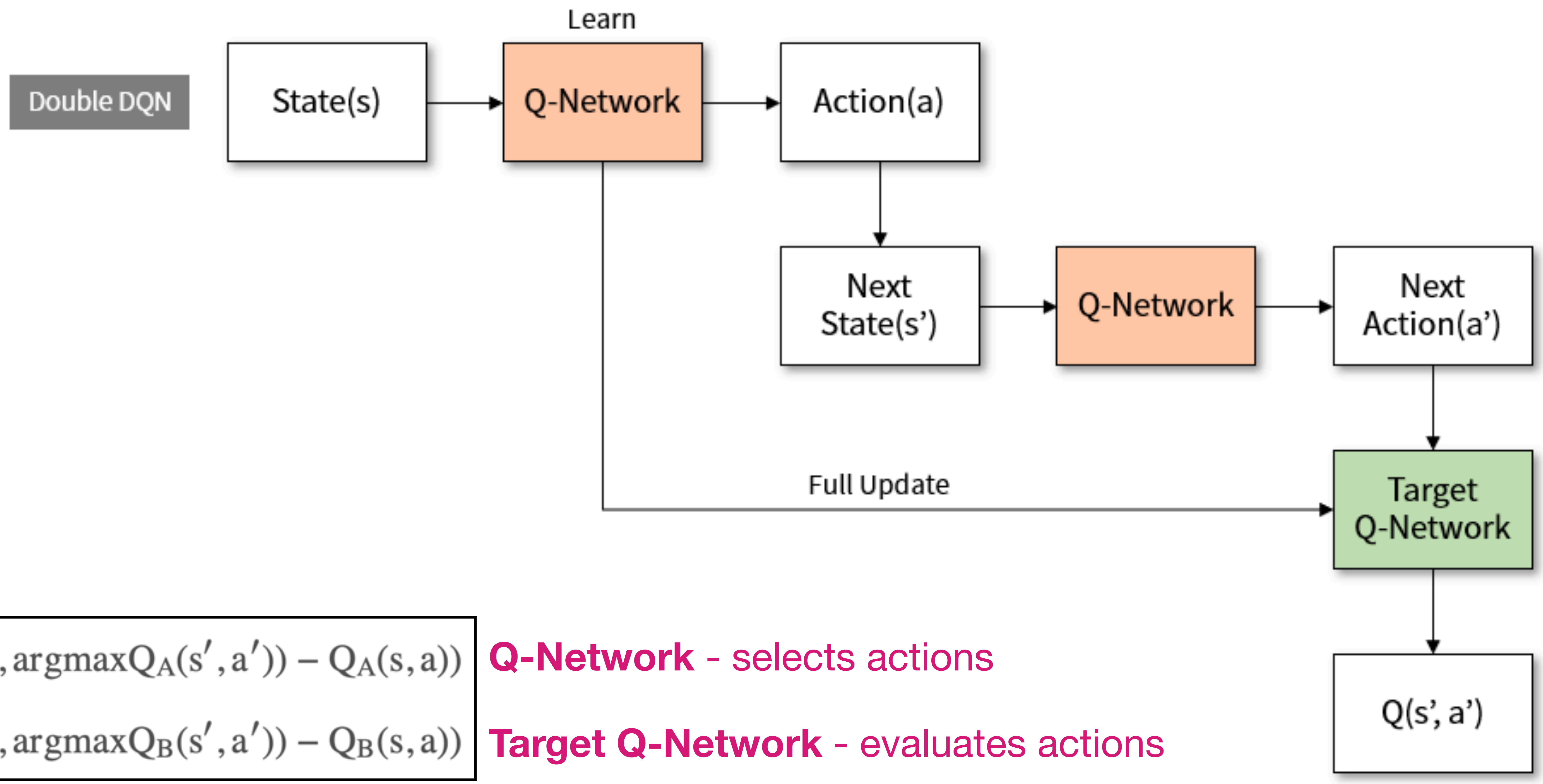
DQN = Deep Q-Network

Training (one network) + Experience Replay Buffer



Double DQN (2nd gen)

Two DQN networks converge faster, Target network's weights are updated once per N epochs



$$Q_A(s,a) = Q_A(s,a) + \alpha(R + \gamma Q_B(s', \operatorname{argmax} Q_A(s',a')) - Q_A(s,a))$$

$$Q_B(s,a) = Q_B(s,a) + \alpha(R + \gamma Q_A(s', \operatorname{argmax} Q_B(s',a')) - Q_B(s,a))$$

Q-Network - selects actions

Target Q-Network - evaluates actions

Double DQN

Q-Network - selects actions

Target Q-Network - evaluates actions

$$Q_A(s,a) = Q_A(s,a) + \alpha(R + \gamma Q_B(s', \text{argmax} Q_A(s',a')) - Q_A(s,a))$$

$$Q_B(s,a) = Q_B(s,a) + \alpha(R + \gamma Q_A(s', \text{argmax} Q_B(s',a')) - Q_B(s,a))$$

```python
def replay(self):
    if len(self.memory) < self.train_start:
        return
    # Randomly sample minibatch from the memory
    minibatch = random.sample(self.memory, min(self.batch_size, self.batch_size))

    state = np.zeros((self.batch_size, self.state_size))
    next_state = np.zeros((self.batch_size, self.state_size))
    action, reward, done = [], [], []

    # do this before prediction
    # for speedup, this could be done on the tensor level
    # but easier to understand using a loop
    for i in range(self.batch_size):
        state[i] = minibatch[i][0]
        action.append(minibatch[i][1])
        reward.append(minibatch[i][2])
        next_state[i] = minibatch[i][3]
        done.append(minibatch[i][4])

    # do batch prediction to save speed
    target = self.model.predict(state)
    target_next = self.model.predict(next_state)
    target_val = self.target_model.predict(next_state)

    for i in range(len(minibatch)):
        # correction on the Q value for the action used
        if done[i]:
            target[i][action[i]] = reward[i]
        else:
            if self.ddqn: # Double - DQN
                # current Q Network selects the action
                # a'_max = argmax_a' Q(s', a')
                a = np.argmax(target_next[i])
                # target Q Network evaluates the action
                # Q_max = Q_target(s', a'_max)
                target[i][action[i]] = reward[i] + self.gamma * (target_val[i][a])
            else: # Standard - DQN
                # DQN chooses the max Q value among next actions
                # selection and evaluation of action is on the target Q Network
                # Q_max = max_a' Q_target(s', a')
                target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next[i]))

    # Train the Neural Network with batches
    self.model.fit(state, target, batch_size=self.batch_size, verbose=0)
```

Double DQN

Q-Network - selects actions

Target Q-Network - evaluates actions

- The Bellman equation used to calculate the Q values to update the online network follows the equation:

```
value = reward + discount_factor *
target_network.predict(next_state)
[argmax(online_network.predict(next_state))]
```

- The Bellman equation used to calculate the Q value updates in the original DQN is:

```
value = reward + discount_factor *
max(target_network.predict(next_state))
```

Double DQN - Soft Target Update

target_weights = target_weights * (1-TAU) + q_weights * TAU where 0 < TAU < 1

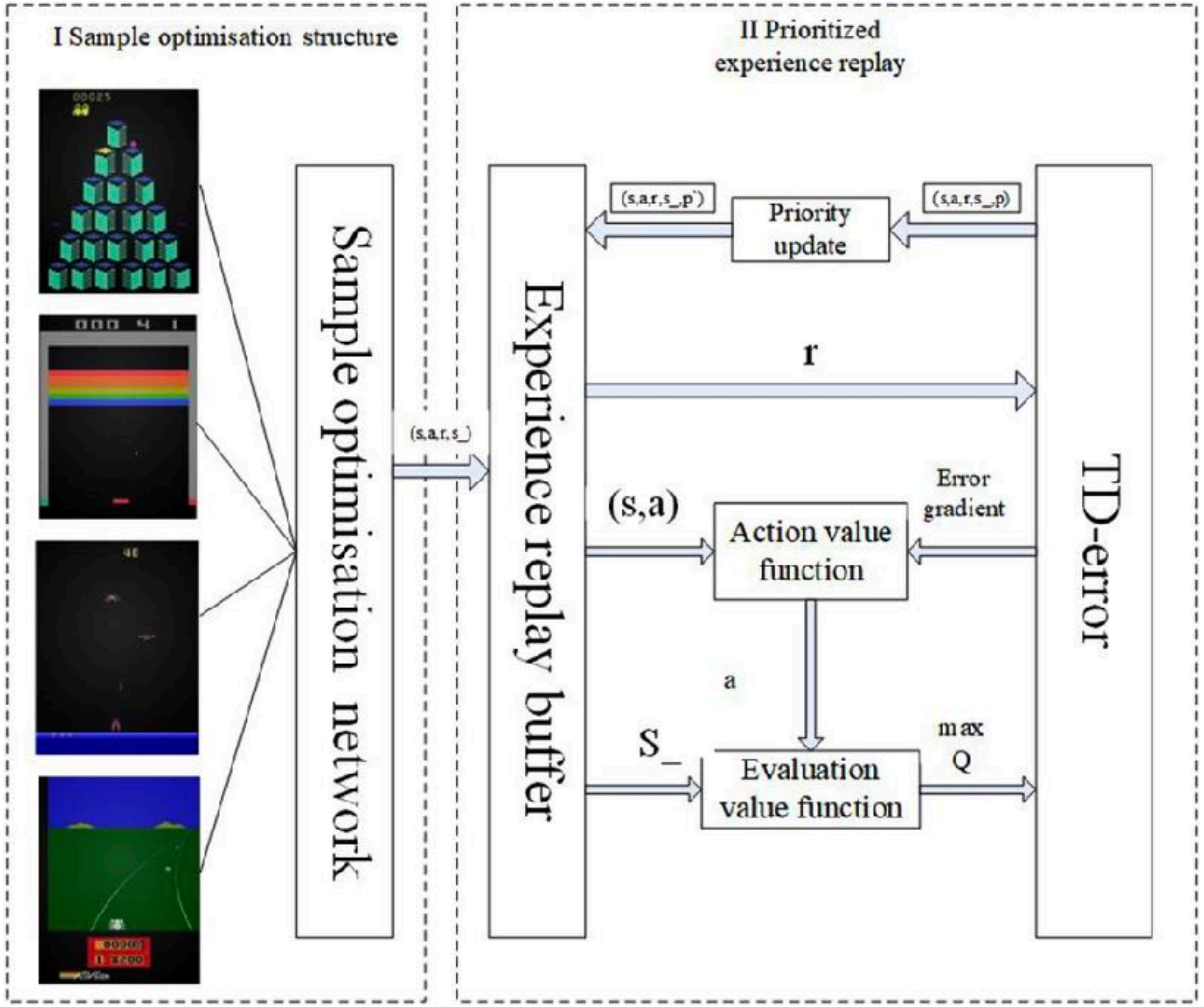
```
def update_target_model(self):
    if not self.Soft_Update and self.ddqn:
        self.target_model.set_weights(self.model.get_weights())
        return
    if self.Soft_Update and self.ddqn:
        q_model_theta = self.model.get_weights()
        target_model_theta = self.target_model.get_weights()
        counter = 0
        for q_weight, target_weight in zip(q_model_theta, target_model_theta):
            target_weight = target_weight * (1-self.TAU) + q_weight * self.TAU
            target_model_theta[counter] = target_weight
            counter += 1
        self.target_model.set_weights(target_model_theta)
```

Prioritised Replay (2nd gen)

For TD-learning.

Order of replying updates could help speed up learning. Priority of a tuple s_i-a_i-r_i-s_{i+1} is proportional to TD error

More informative experience replay -> faster training



Duelling DQN (2nd gen)

Why

- TDQN Networks tend to overestimate rewards in noisy environments, leading to non-optimal training outcomes;

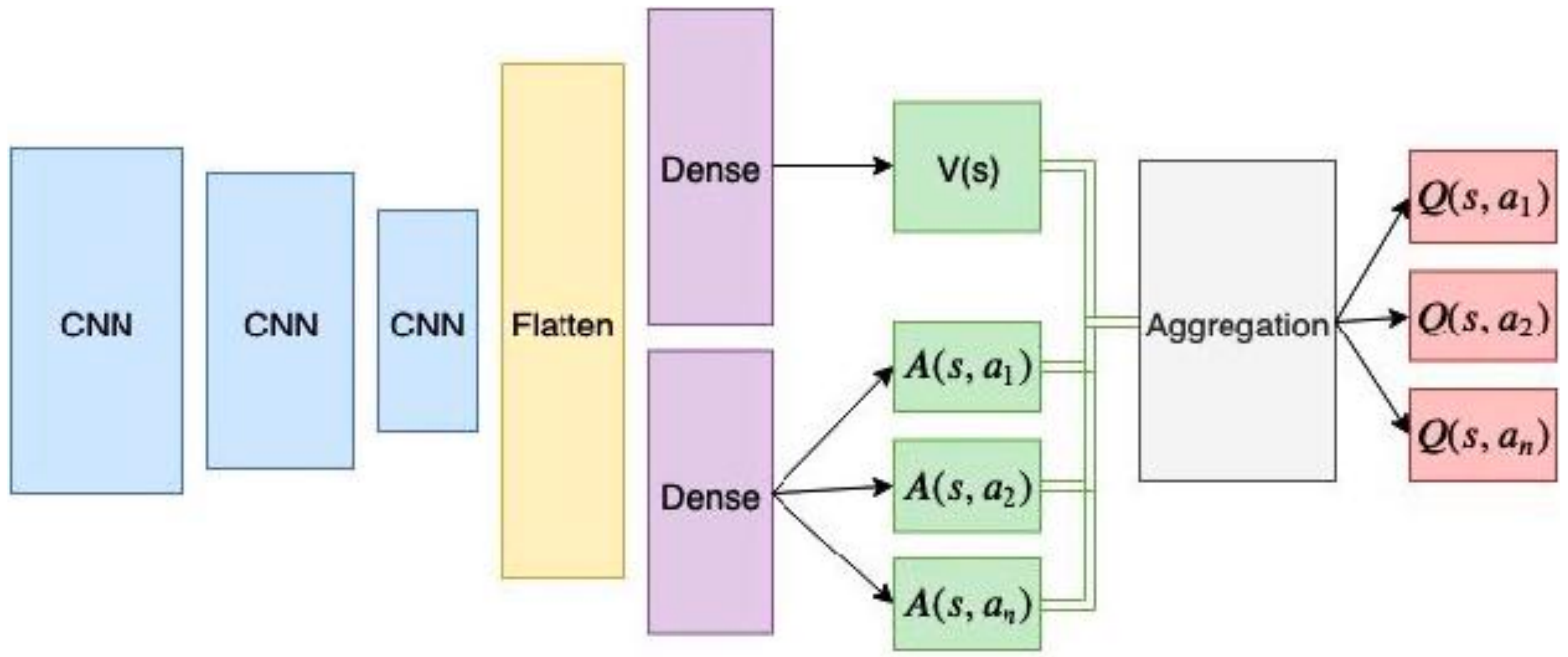
- The moving target problem is that the same network is responsible for choosing and evaluating actions, leading to training instability.

- **Double Dueling DQN**: the evaluation of the Q function implicitly calculates two quantities:

  V(s) – the value of being in state s;
  A(s, a) – the advantage of taking action in state s.

Duelling DQN (2nd gen)

Has two heads - for V(s) and A(s,a) estimation




```python
def OurModel(input_shape, action_space, dueling):
    X_input = Input(input_shape)
    X = X_input

    # 'Dense' is the basic form of a neural network layer
    # Input Layer of state size(4) and Hidden Layer with 512 nodes
    X = Dense(512, input_shape=input_shape, activation="relu", kernel_initializer='he_uniform')(X)

    # Hidden layer with 256 nodes
    X = Dense(256, activation="relu", kernel_initializer='he_uniform')(X)

    # Hidden layer with 64 nodes
    X = Dense(64, activation="relu", kernel_initializer='he_uniform')(X)

    if dueling:
        state_value = Dense(1, kernel_initializer='he_uniform')(X)
        state_value = Lambda(lambda s: K.expand_dims(s[:, 0], -1), output_shape=(action_space,))(state_value)

        action_advantage = Dense(action_space, kernel_initializer='he_uniform')(X)
        action_advantage = Lambda(lambda a: a[:, :] - K.mean(a[:, :], keepdims=True), output_shape=(action_space,))(action_advantage)

        X = Add()([state_value, action_advantage])
    else:
        # Output Layer with # of actions: 2 nodes (left, right)
        X = Dense(action_space, activation="linear", kernel_initializer='he_uniform')(X)

    model = Model(inputs = X_input, outputs = X, name='CartPole Dueling DDQN model')
    model.compile(loss="mean_squared_error", optimizer=RMSprop(lr=0.00025, rho=0.95, epsilon=0.01), metrics=["accuracy"])

    model.summary()
    return model
```

Ape-X =
Distributed Prioritized Experience Replay
+
DQN / Double DQN / Duelling DQN

Ape-X config for Ray RLlib

Example: https://github.com/ray-project/ray/blob/master/rllib/algorithms/apex_dqn/apex_dqn.py

```python
def __init__(self, algo_class=None):
    """Initializes a ApexConfig instance."""
    super().__init__(algo_class=algo_class or ApexDQN)

    # fmt: off
    # __sphinx_doc_begin__
    # APEX-DQN settings overriding DQN ones:
    # .training()
    self.optimizer = merge_dicts(
        DQNConfig().optimizer, {
            "max_weight_sync_delay": 400,
            "num_replay_buffer_shards": 4,
            "debug": False
        })
    self.n_step = 3
    self.train_batch_size = 512
    self.target_network_update_freq = 500000
    self.training_intensity = 1
    # Number of timesteps to collect from rollout workers before we start
    # sampling from replay buffers for learning. Whether we count this in agent
    # steps  or environment steps depends on config["multiagent"]["count_steps_by"].
    self.num_steps_sampled_before_learning_starts = 50000

    self.max_requests_in_flight_per_replay_worker = float("inf")
    self.timeout_s_sampler_manager = 0.0
    self.timeout_s_replay_manager = 0.0
    # APEX-DQN is using a distributed (non local) replay buffer.
    self.replay_buffer_config = {
        "no_local_replay_buffer": True,
        # Specify prioritized replay by supplying a buffer type that supports
        # prioritization
        "type": "MultiAgentPrioritizedReplayBuffer",
        "capacity": 2000000,
```

Rainbow?
= a successful combinations of
DeepRL improvements

Rainbow

Example: https://github.com/ray-project/ray/blob/master/rllib/tuned_examples/dqn/pong-rainbow.yaml

```yaml
pong-deterministic-rainbow:
    env: PongDeterministic-v4
    run: DQN
    stop:
        episode_reward_mean: 20
    config:
        num_atoms: 51
        noisy: True
        gamma: 0.99
        lr: .0001
        hiddens: [512]
        rollout_fragment_length: 4
        train_batch_size: 32
        exploration_config:
          epsilon_timesteps: 2
          final_epsilon: 0.0
        target_network_update_freq: 500
        replay_buffer_config:
          type: MultiAgentPrioritizedReplayBuffer
          prioritized_replay_alpha: 0.5
          capacity: 50000
        num_steps_sampled_before_learning_starts: 10000
        n_step: 3
        gpu: True
        model:
          grayscale: True
          zero_mean: False
          dim: 42
        # we should set compress_observations to True because few machines
        # would be able to contain the replay buffers in memory otherwise
        compress_observations: True
```

Homework

Assault + Gym Env + Ray RLlib

- Train & tune DQN
- Train & tune DQN + Experience Replay
- Train & tune Double DQN + Experience Replay
- Train & tune Double DQN + Prioritised Experience Replay
- Train & tune Double Duelling DQN + Prioritised Experience Replay
- Train & tune Ape-X: Double Duelling DQN + Distributed Prioritised Experience Replay
- Cheatsheet: https://github.com/ray-project/ray/tree/master/rllib/tuned_examples