Playing Atari with Deep Reinforcement Learning

Part 2:

Policy Gradients, A2C, A3C, PPO

By Gurbych Oleksandr

Playing Atari with Deep Reinforcement Learning

Part 2:

Policy Gradients, A2C, A3C, PPO

By Gurbych Oleksandr

Intro
Policy Gradients

DQN Drawbacks

DQN's are comparatively simple and efficient but…

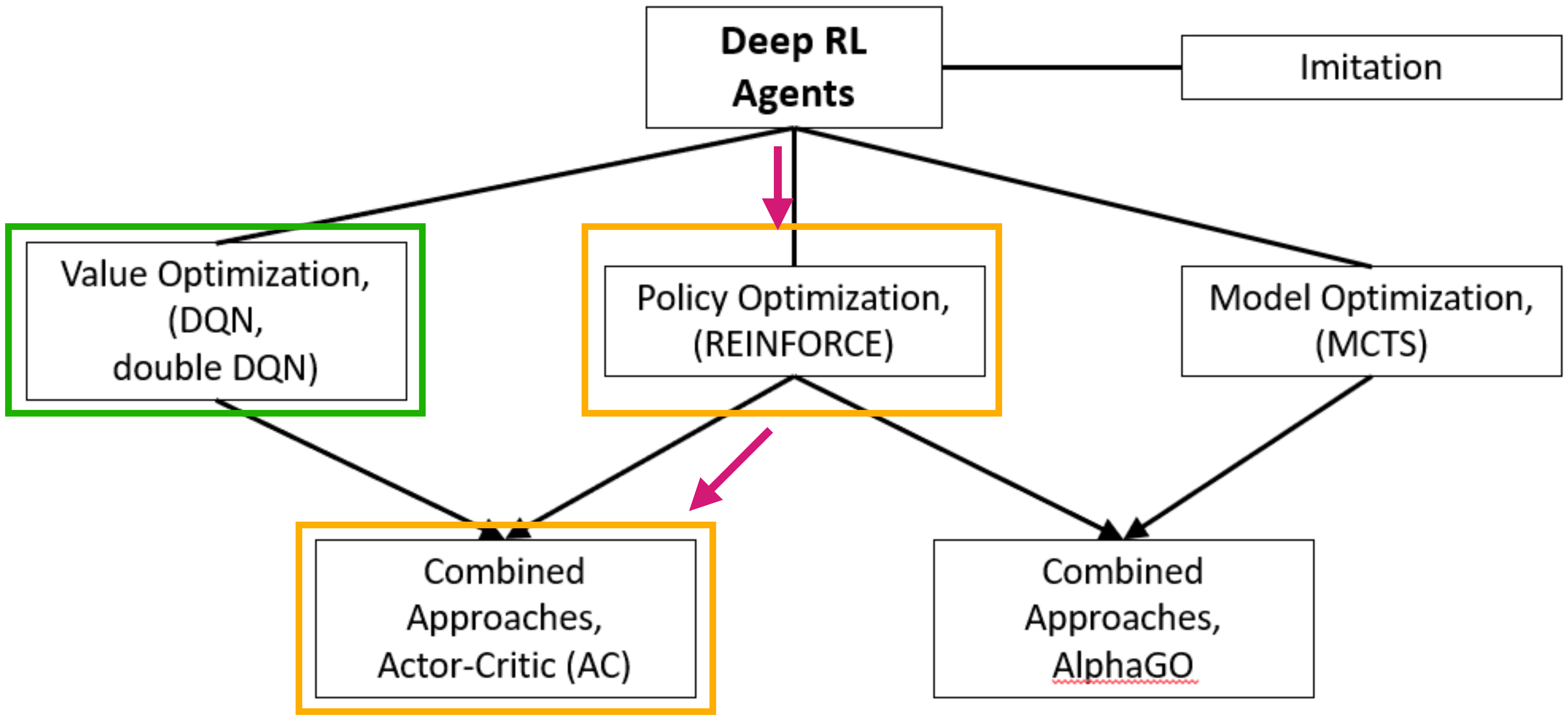
- Significant oscillations while training. This is because the choice of action may change dramatically for an arbitrarily small change in the estimated action values.

- Suppose the possible number of state-action pairs is relatively large in a given environment. In that case, the Q-function can become highly complicated, so it becomes intractable to estimate the optimal Q-value.

- Even in situations where finding Q is computationally tractable, DQN's are not great at exploring relative to some other approaches, so a DQN may not work correctly.

Other Types of Deep RL Agents

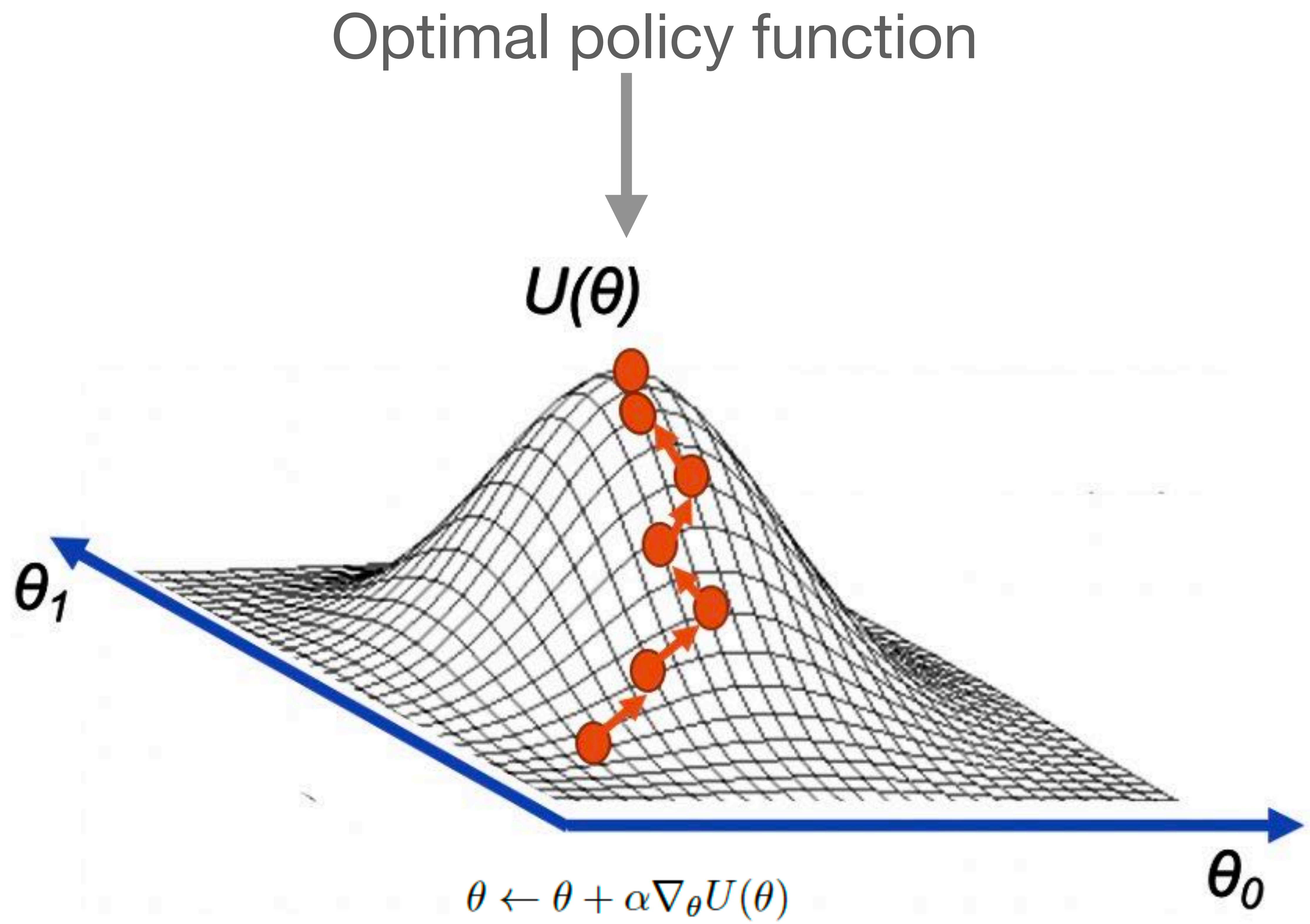
DQN's are comparatively simple and efficient but…



Value Optimisation vs Policy Optimisation

…implements the classic "agent-environment loop"

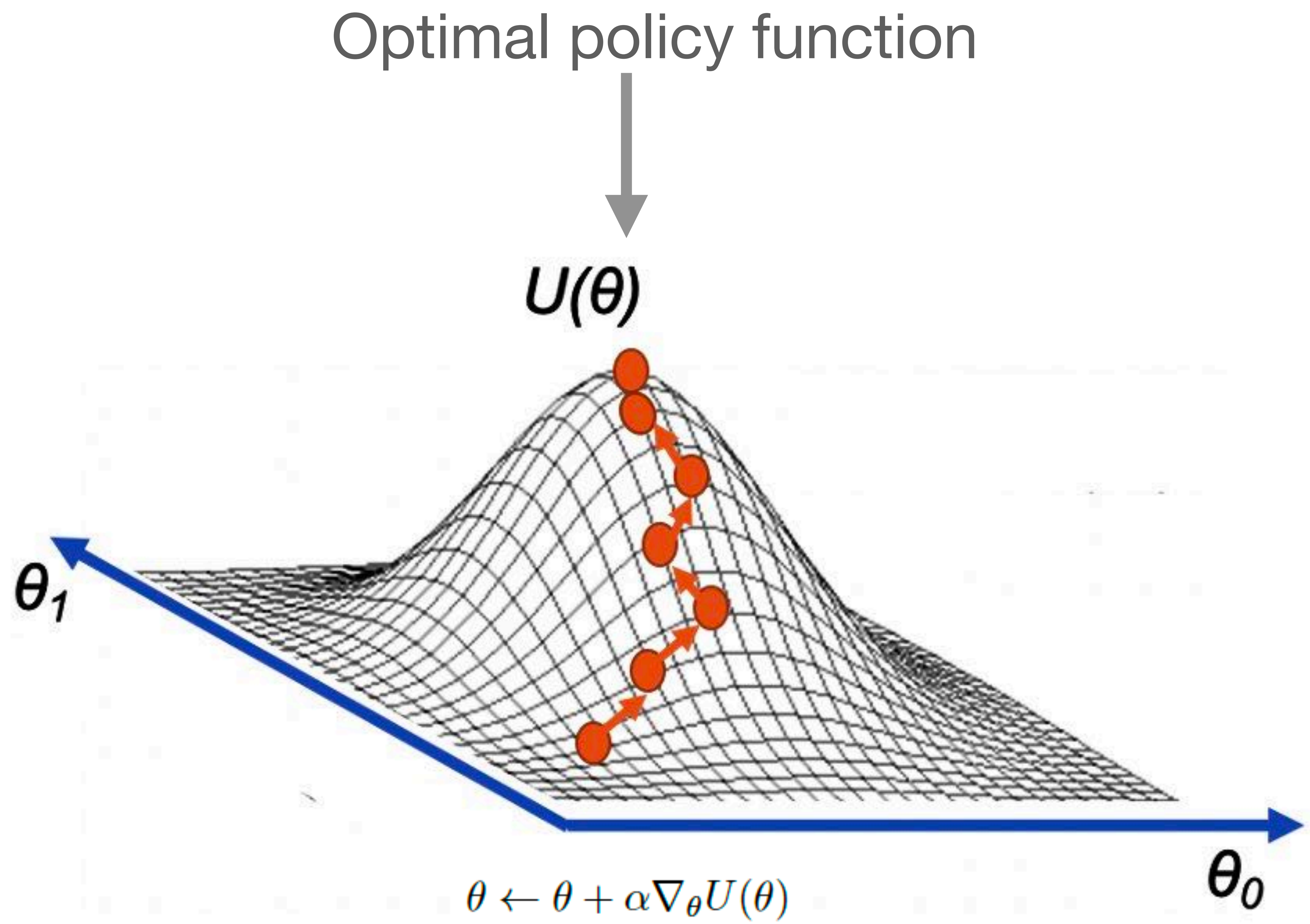
- A policy function **π** maps the state space **S** to the action space **A**
- DQN's "policy" **π** is learned indirectly - by picking the (s,a) pairs with the largest Q-value - choosing "the best action"
- Policy gradient (PG) algorithms **perform gradient ascent on π directly** and learn the policy **π** directly



Value Optimisation vs Policy Optimisation

…implements the classic "agent-environment loop"

- PG algorithms **converge faster** and robuster than value optimisation algorithms such as DQN
- PGs are more **effective in large action spaces** or using continuous actions.
- Deep Q-learning assigns a score (maximum expected future reward) for every possible action, within each time step, given the current state. But what if we have endless possibilities for action?



Policy Gradient

…fit states to actions directly

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Algorithm parameter: step size $\alpha > 0)$
Initialize the policy parameter θ at random

(1) Use the policy π_θ to collect a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$

(2) Estimate the Return for trajectory τ: $R(\tau) = (G_0, G_1, ..., G_H)$
where G_k is the expected return for transition k:

$$G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$$

(3) Use the trajectory τ to estimate the gradient $\nabla_\theta U(\theta)$

$$\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

(4) Update the weights θ of the policy

$$\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$$

(5) Loop over steps 1-5 until not converged

$$\boxed{\Delta J(Q) = E_\tau\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t) G_t\right]}$$

Policy function
(differentiable)

Policy Gradient

Drawbacks

$$\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t) G_t\right]$$

Policy function

- The PG algorithm updates the policy using Monte Carlo (i.e., taking random samples) =>
  each training trajectory can be very different (1) =>
  high variability in log probs and cumulative reward =>
  noisy gradients => unstable learning => non-optimal policy distribution learned

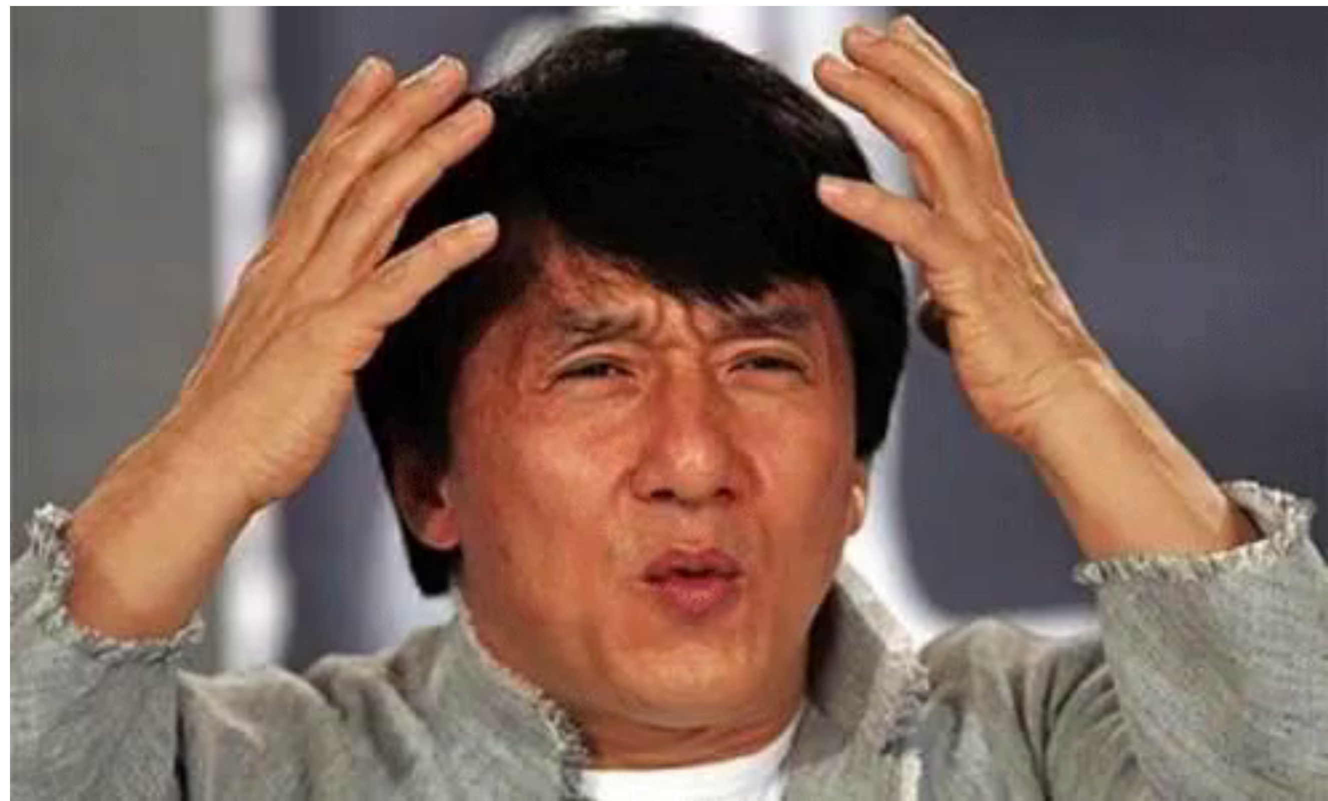
- (1) => trajectories with with cumulative reward 0 => PG algorithm doesn't improve there

Policy Gradient

Drawbacks

$$\boxed{\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t) G_t\right]} \longleftarrow \text{Policy function}$$

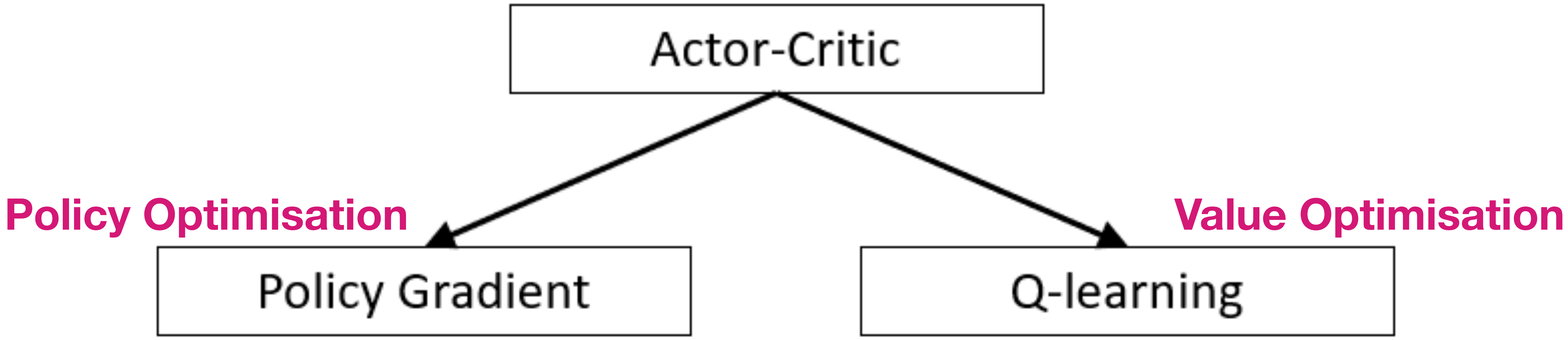
- **Instable**
- **Slow convergence**



Section 2
Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C)

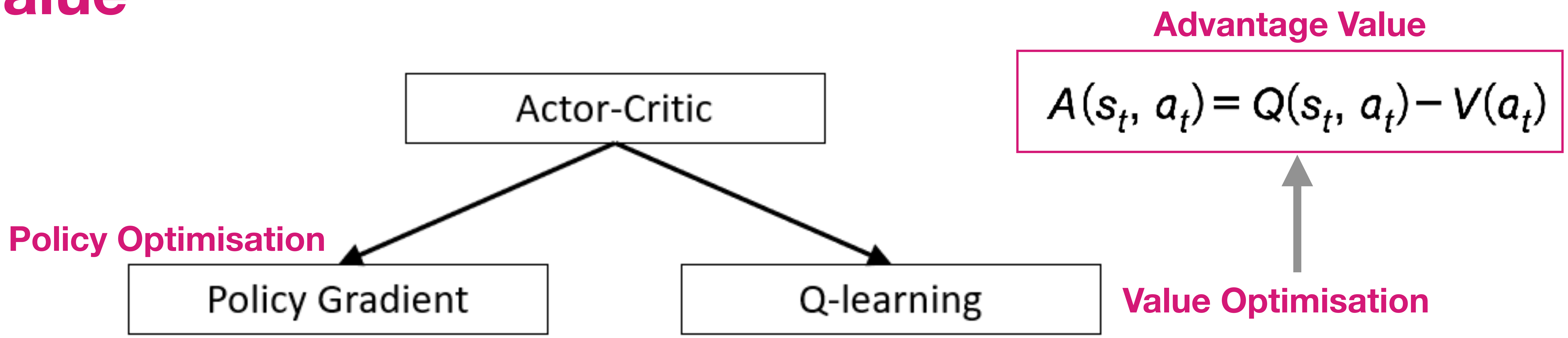
…implements the classic "agent-environment loop"



- **Actor**: a PG algorithm that decides on an action to take;
- **Critic**: Q-learning algorithm that critiques the action that the Actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay.
- **A2C can solve a broader range of problems** than DQN
- **A2C has a lower variance in performance** relative to a pure PG
- Sampling inefficient

Advantage Actor-Critic (A2C)

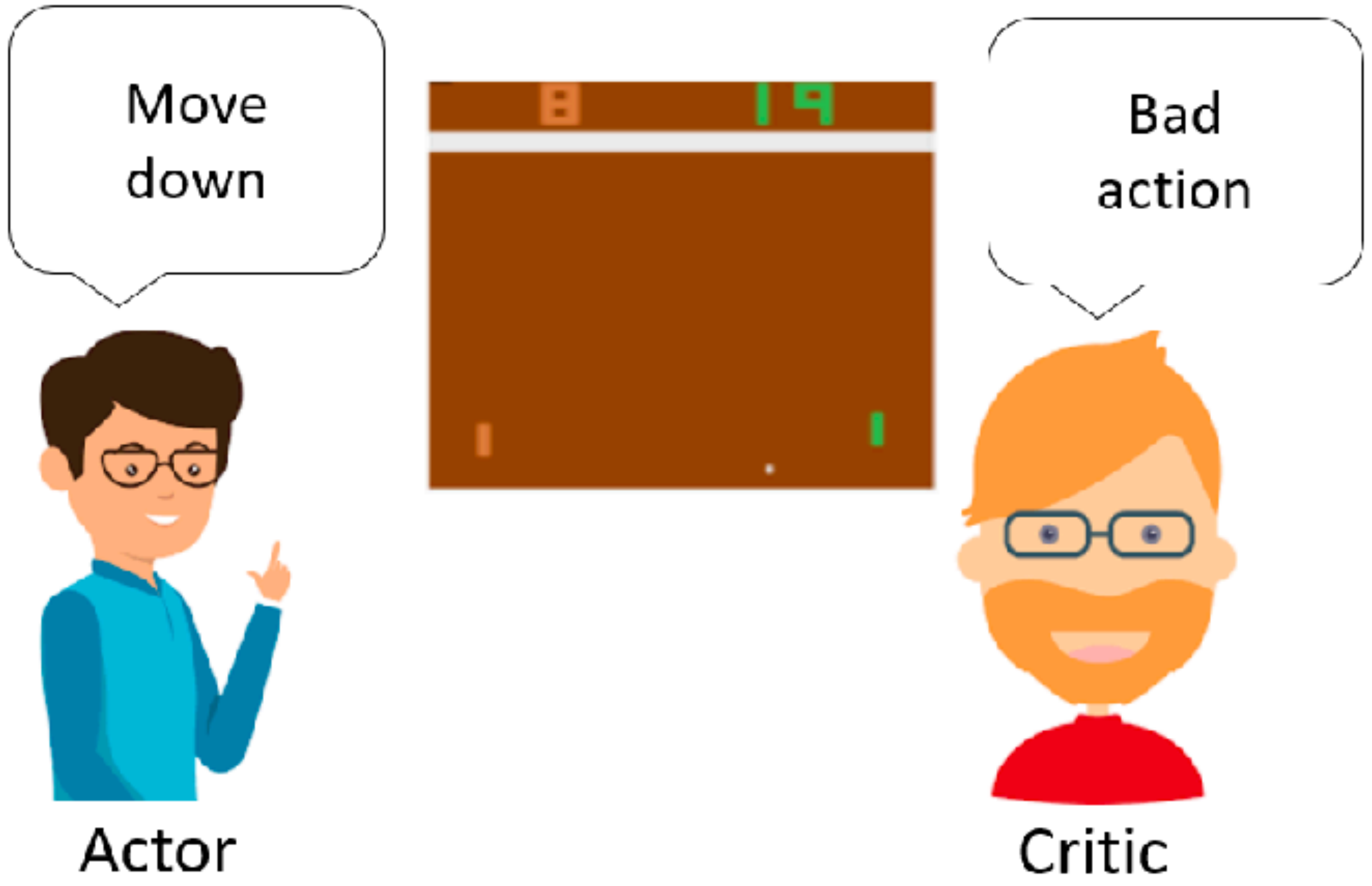
Advantage value



- **Actor**: a PG algorithm that decides on an action to take;
- **Critic**: Q-learning algorithm that critiques the action that the Actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay.
- **how better it is to take a specific action than the average general action at the given state?**

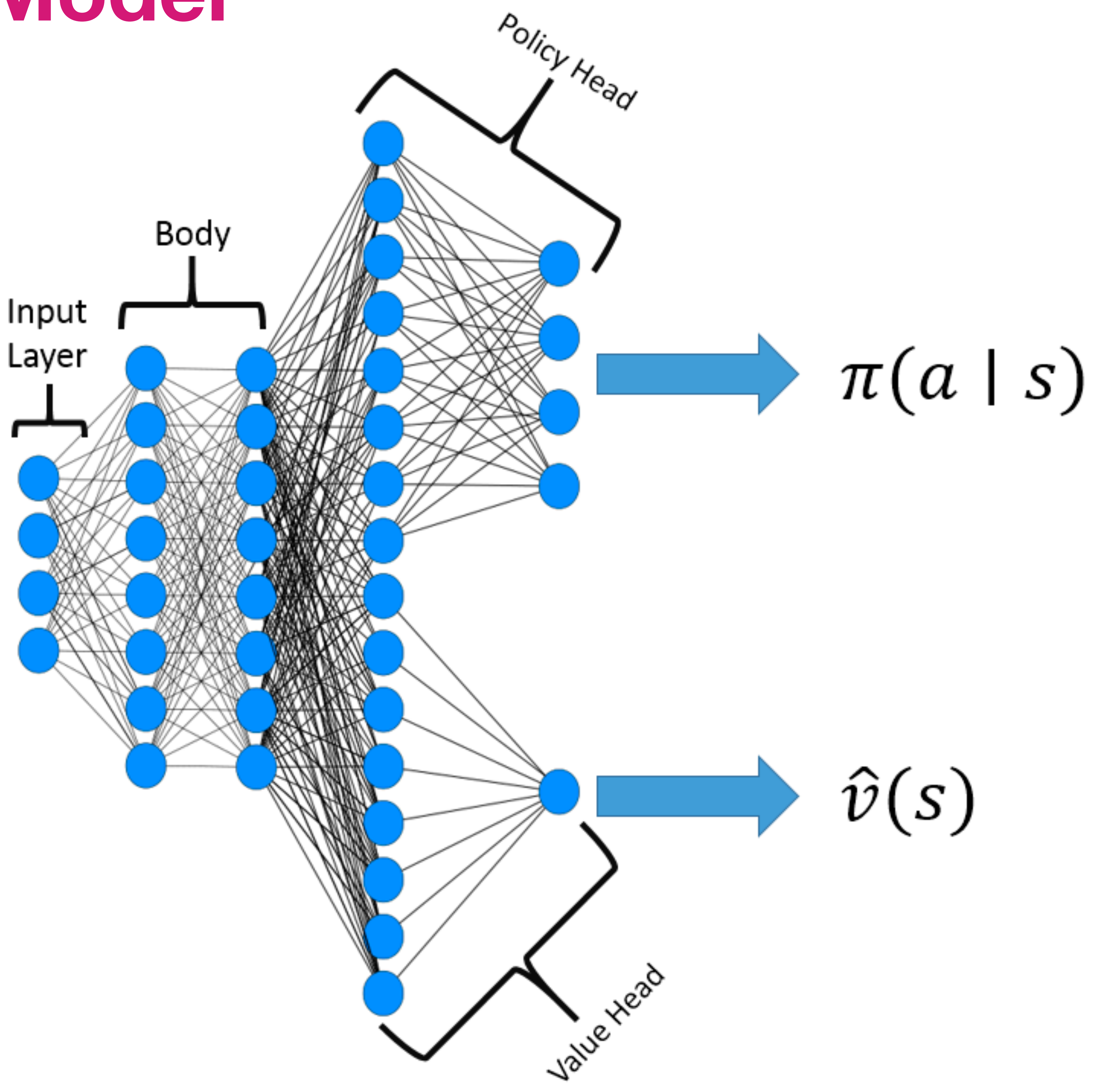
A2C

…fit states to actions directly (actor) + evaluate advantage of a new state (critic)



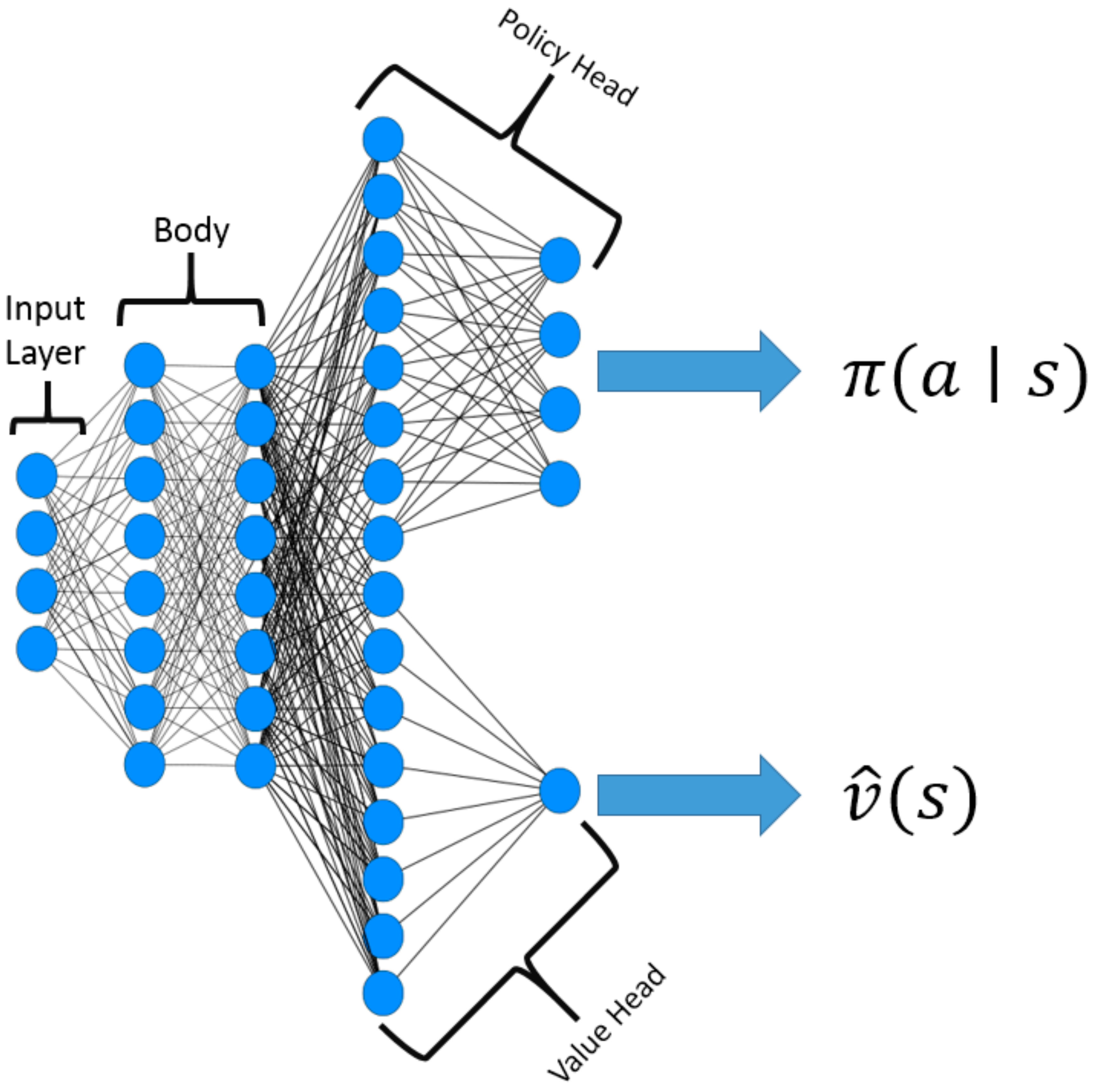
A2C

…fit states to actions directly (actor) + evaluate advantage of a new state (critic) - Model



A2C

…fit states to actions directly (actor) + evaluate advantage of a new state (critic) - Model




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

    X = Flatten(input_shape=input_shape)(X_input)

    X = Dense(512, activation="elu", kernel_initializer='he_uniform')(X)

    action = Dense(action_space, activation="softmax", kernel_initializer='he_uniform')(X)
    value = Dense(1, kernel_initializer='he_uniform')(X)

    Actor = Model(inputs = X_input, outputs = action)
    Actor.compile(loss='categorical_crossentropy', optimizer=RMSprop(lr=lr))

    Critic = Model(inputs = X_input, outputs = value)
    Critic.compile(loss='mse', optimizer=RMSprop(lr=lr))

    return Actor, Critic
```

A2C

…fit states to actions directly (actor) + evaluate advantage of a new state (critic) - Training

```
def replay(self):
    # reshape memory to appropriate shape for training
    states = np.vstack(self.states)
    actions = np.vstack(self.actions)

    # Compute discounted rewards
    discounted_r = self.discount_rewards(self.rewards)

    # Get Critic network predictions
    values = self.Critic.predict(states)[:, 0]
    # Compute advantages
    advantages = discounted_r - values
    # training Actor and Critic networks
    self.Actor.fit(states, actions, sample_weight=advantages, epochs=1, verbose=0)
    self.Critic.fit(states, discounted_r, epochs=1, verbose=0)
    # reset training memory
    self.states, self.actions, self.rewards = [], [], []
```

A2C

…fit states to actions directly (actor) + evaluate advantage of a new state (critic) - Ray config

```python
# Run with:
# rllib train file cartpole_a2c.py \
#     --stop={'timesteps_total': 50000, 'episode_reward_mean': 200}"
from ray.rllib.algorithms.a2c import A2CConfig


config = (
    A2CConfig()
    .environment("CartPole-v1")
    .training(lr=0.001, train_batch_size=20)
    .framework("tf")
    .rollouts(num_rollout_workers=0)
)
```

A2C

Summary

- **A2C learns the policy π directly** (policy head) + accounts for **advantage of a specific action at the given state** (value head)
- **Converge faster** than DQN and PG
- **More stable** than DQN and PG
- **Effective in large action spaces** or using continuous actions.
- Single-threaded (

Section 3
Asynchronous Advantage Actor-Critic (A3C)

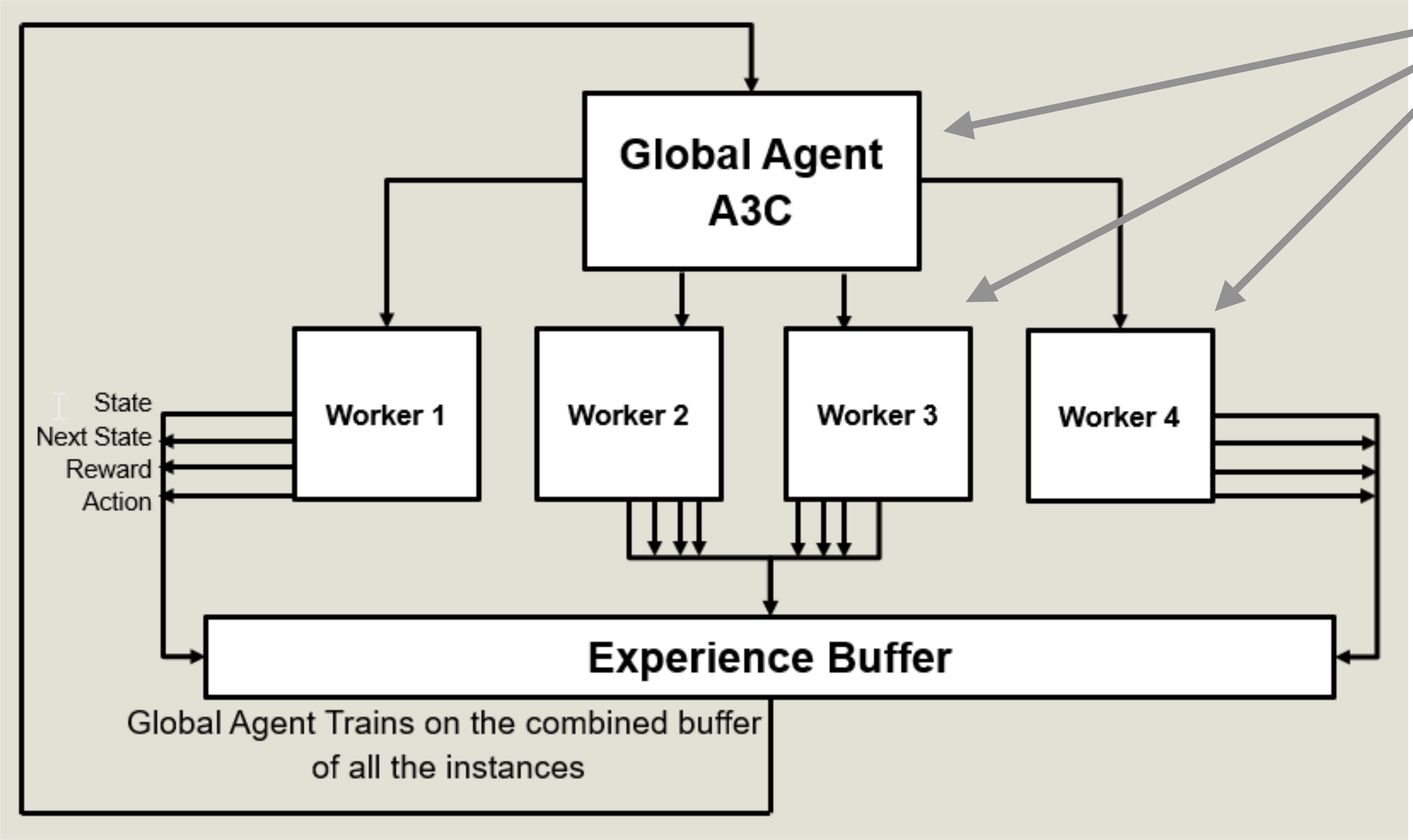
A3C

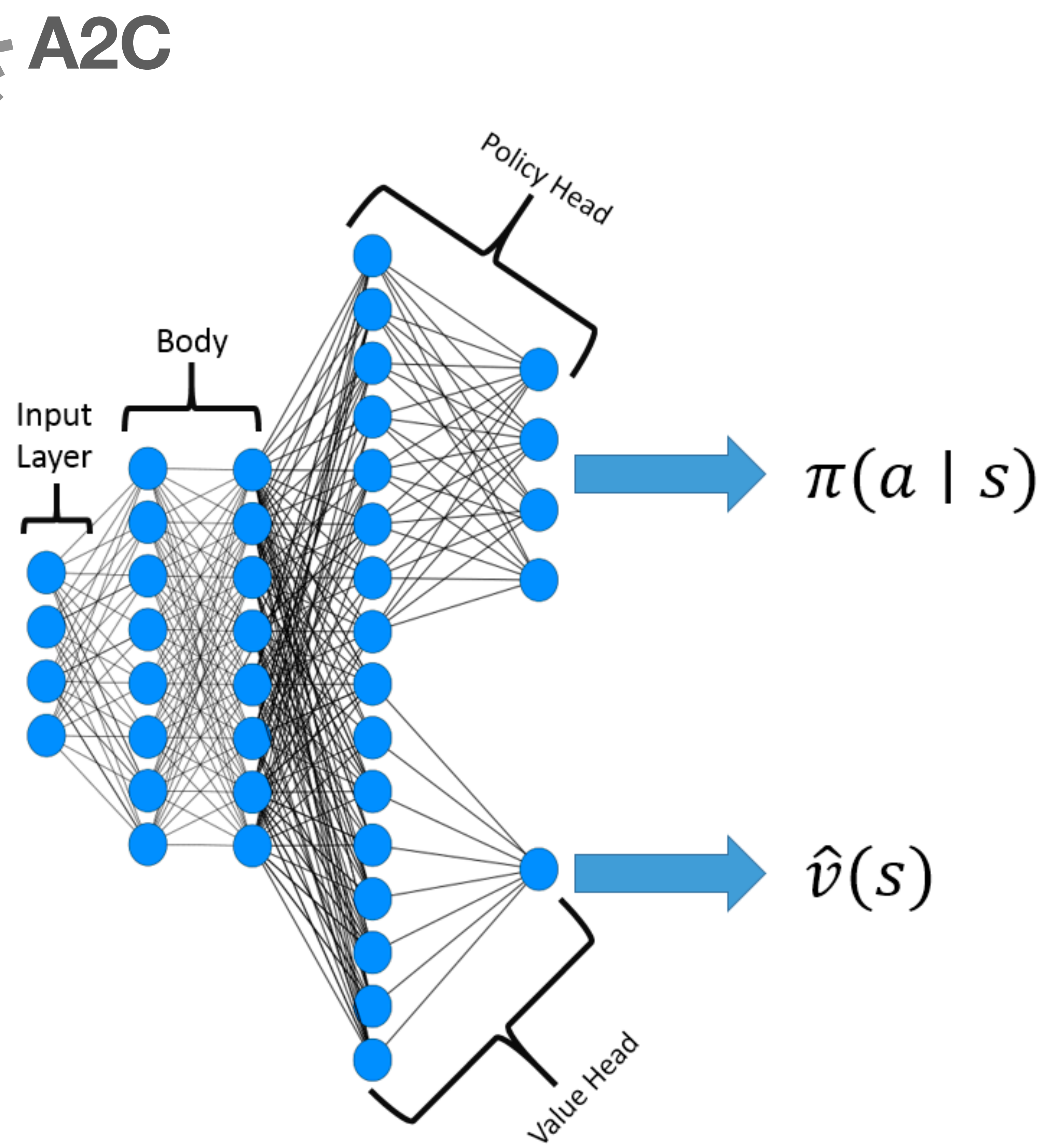
What's in the name?

- **Asynchronous**: multiple worker agents are trained in parallel, each with their environment.
  => **train faster** as more workers are training in parallel
  => **diverse training experience** as each worker's experience is independent
- **Advantage**: a metric to judge how good its actions were and how they turned out.
  => **measure the advantage of an action** at time step t, following the policy π
  => **focus** on where the network's predictions were lacking
- **Actor-Critic**: the algorithm's architecture shares layers between the policy and value function.

A3C

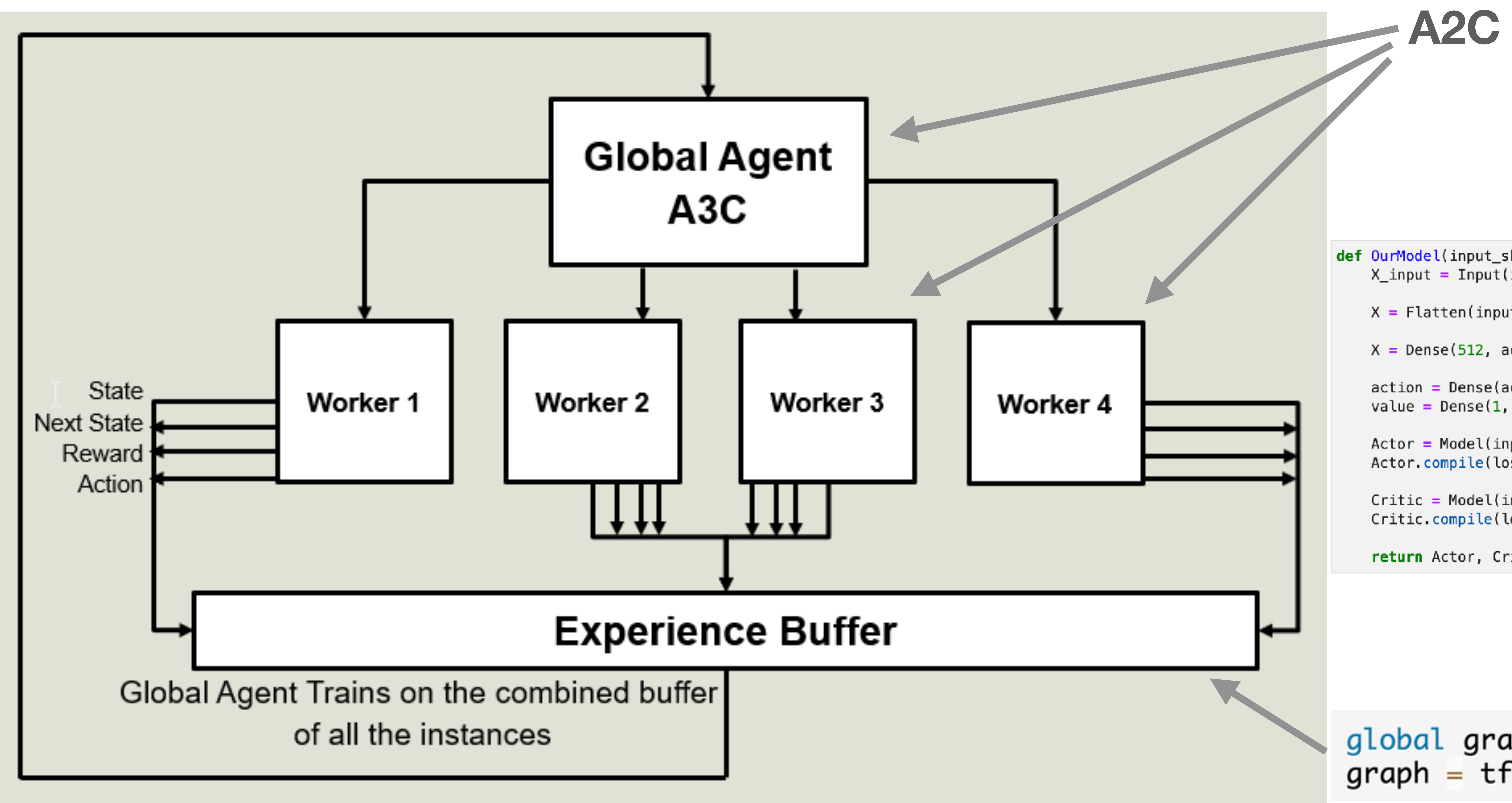
…is the asynchronous training technology + A2C neural nets





A3C

…is the asynchronous training technology + A2C neural nets

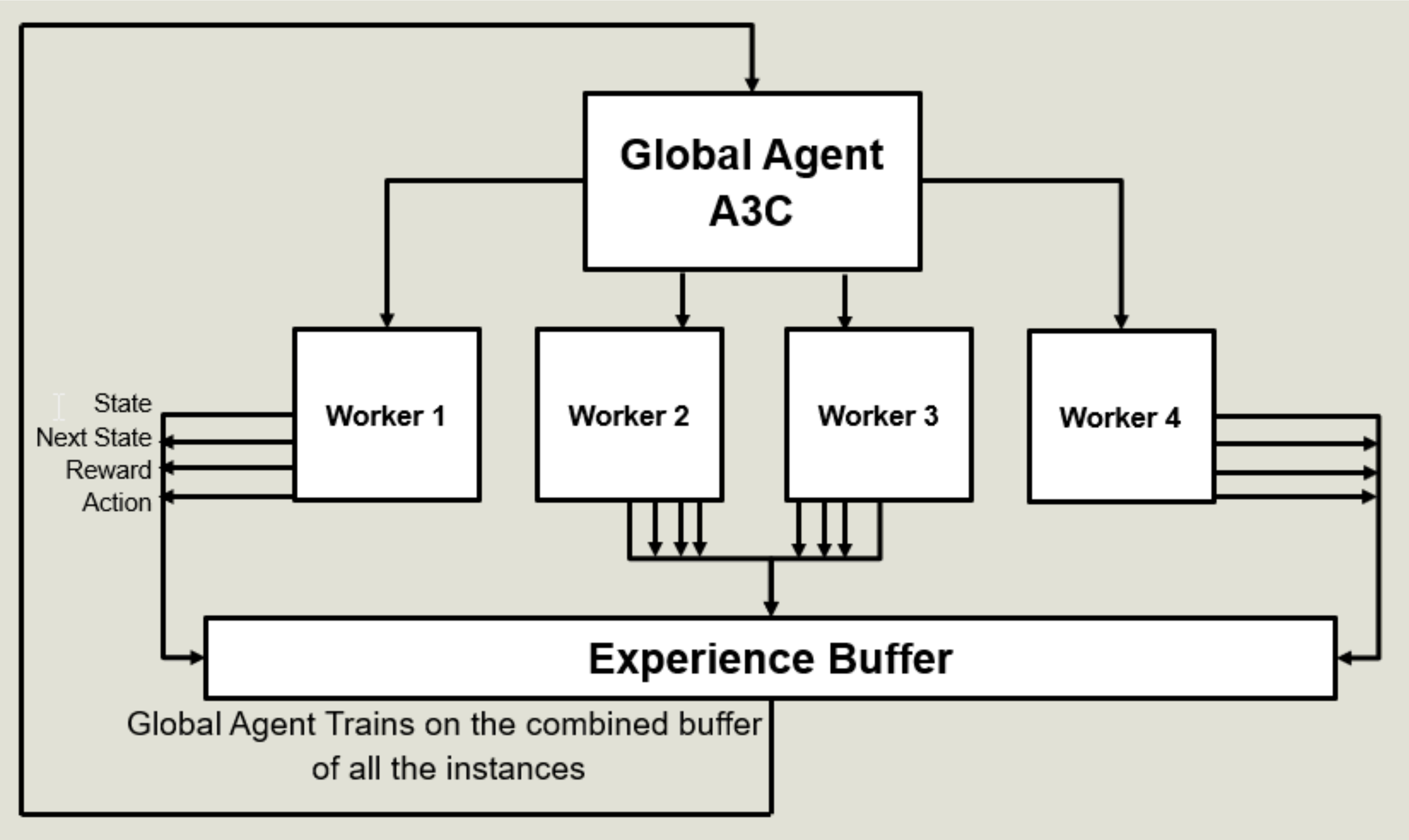



```python
def OurModel(input_shape, action_space, lr):
    X_input = Input(input_shape)

    X = Flatten(input_shape=input_shape)(X_input)

    X = Dense(512, activation="elu", kernel_initializer='he_uniform')(X)

    action = Dense(action_space, activation="softmax", kernel_initializer='he_uniform')(X)
    value = Dense(1, kernel_initializer='he_uniform')(X)

    Actor = Model(inputs = X_input, outputs = action)
    Actor.compile(loss='categorical_crossentropy', optimizer=RMSprop(lr=lr))

    Critic = Model(inputs = X_input, outputs = value)
    Critic.compile(loss='mse', optimizer=RMSprop(lr=lr))

    return Actor, Critic
```

```python
global graph
graph = tf.get_default_graph()
```

A3C

How it works?



- Each ***Worker*** trains independently in its own environment
- Each ***Worker*** at the end of each episode writes its local Experience Buffer to a global ***Experience Buffer***
- The ***Global Agent*** then trains on the global ***Experience Buffer***
- The workers then copy the weights from ***Global Agent***
- ***…***repeat until ***Global Agent*** converges

A3C

How it works?

```
def train(self, n_threads):
    self.env.close()
    # Instantiate one environment per thread
    envs = [gym.make(self.env_name) for i in range(n_threads)]

    # Create threads
    threads = [threading.Thread(
            target=self.train_threading,
            daemon=True,
            args=(self,
                envs[i],
                i)) for i in range(n_threads)]

    for t in threads:
        time.sleep(2)
        t.start()
```

Training (multi-threaded)

```
def replay(self, states, actions, rewards):
    # reshape memory to appropriate shape for training
    states = np.vstack(states)
    actions = np.vstack(actions)

    # Compute discounted rewards
    discounted_r = self.discount_rewards(rewards)

    # Get Critic network predictions
    value = self.Critic.predict(states)[:, 0]
    # Compute advantages
    advantages = discounted_r - value
    # training Actor and Critic networks
    self.Actor.fit(states, actions, sample_weight=advantages, epochs=1, verbose=0)
    self.Critic.fit(states, discounted_r, epochs=1, verbose=0)
```

Training on Experience Buffer

Experience Buffer

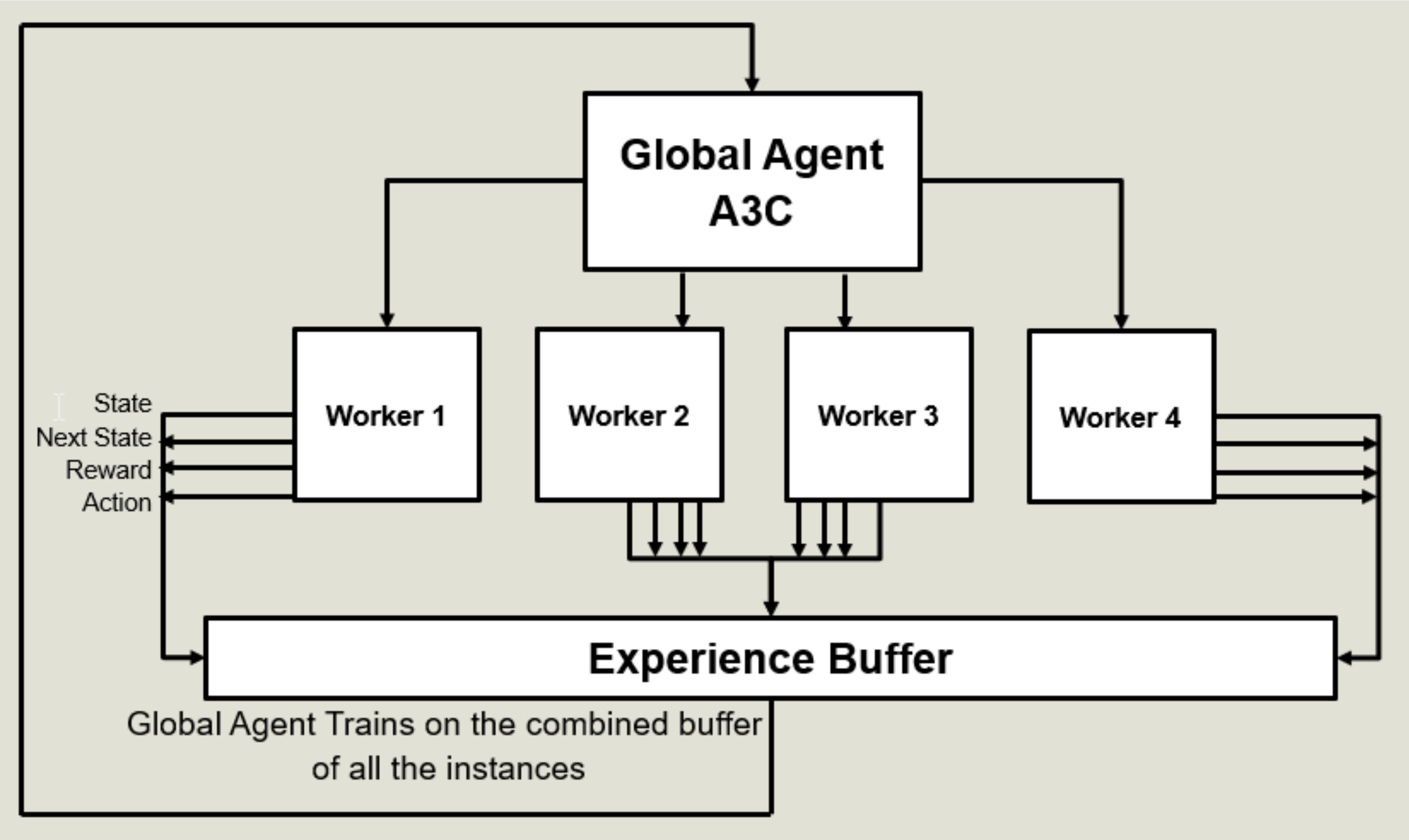
```
global graph
graph = tf.get_default_graph()
```

One worker's environment (single thread)

```
def train_threading(self, agent, env, thread):
    global graph
    with graph.as_default():
        while self.episode < self.EPISODES:
            # Reset episode
            score, done, SAVING = 0, False, ''
            state = self.reset(env)
            # Instantiate or reset games memory
            states, actions, rewards = [], [], []
            while not done:
                action = agent.act(state)
                next_state, reward, done, _ = self.step(action, env, state)

                states.append(state)
                action_onehot = np.zeros([self.action_size])
                action_onehot[action] = 1
                actions.append(action_onehot)
                rewards.append(reward)

                score += reward
                state = next_state

            self.lock.acquire()
            self.replay(states, actions, rewards)
            self.lock.release()
```

A3C

What is the point?



- Faster training since workers running in parallel.
- Distributed / federated training

A3C

Ray config (.yaml & .py)

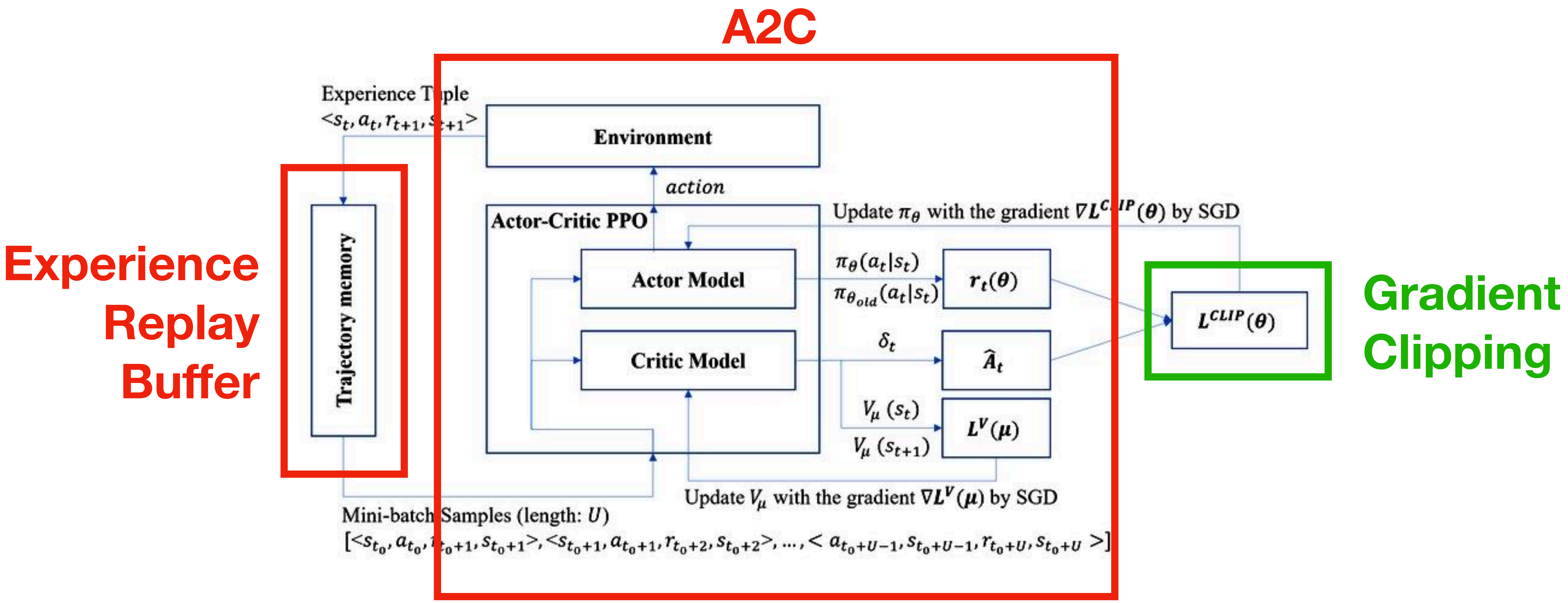
```yaml
pong-a3c:
    env: PongDeterministic-v4
    run: A3C
    config:
        # Works for both torch and tf.
        framework: tf
        num_workers: 16
        rollout_fragment_length: 20
        vf_loss_coeff: 0.5
        entropy_coeff: 0.01
        gamma: 0.99
        grad_clip: 40.0
        lambda: 1.0
        lr: 0.0001
        observation_filter: NoFilter
        preprocessor_pref: rllib
        model:
            use_lstm: true
            conv_activation: elu
            dim: 42
            grayscale: true
            zero_mean: false
            # Reduced channel depth and kernel size from default
            conv_filters: [
                [32, [3, 3], 2],
                [32, [3, 3], 2],
                [32, [3, 3], 2],
                [32, [3, 3], 2],
            ]
```

```python
# Run with:
# rllib train file cartpole_a3c.py \
#     --stop={'timesteps_total': 20000, 'episode_reward_mean': 150}"
from ray.rllib.algorithms.a3c import A3CConfig


config = (
    A3CConfig()
    .training(gamma=0.95)
    .environment("CartPole-v1")
    .framework("tf")
    .rollouts(num_rollout_workers=0)
)
```

Section 4
Proximal Policy Optimisation (PPO)

PPO

= A2C + policy gradient clipping



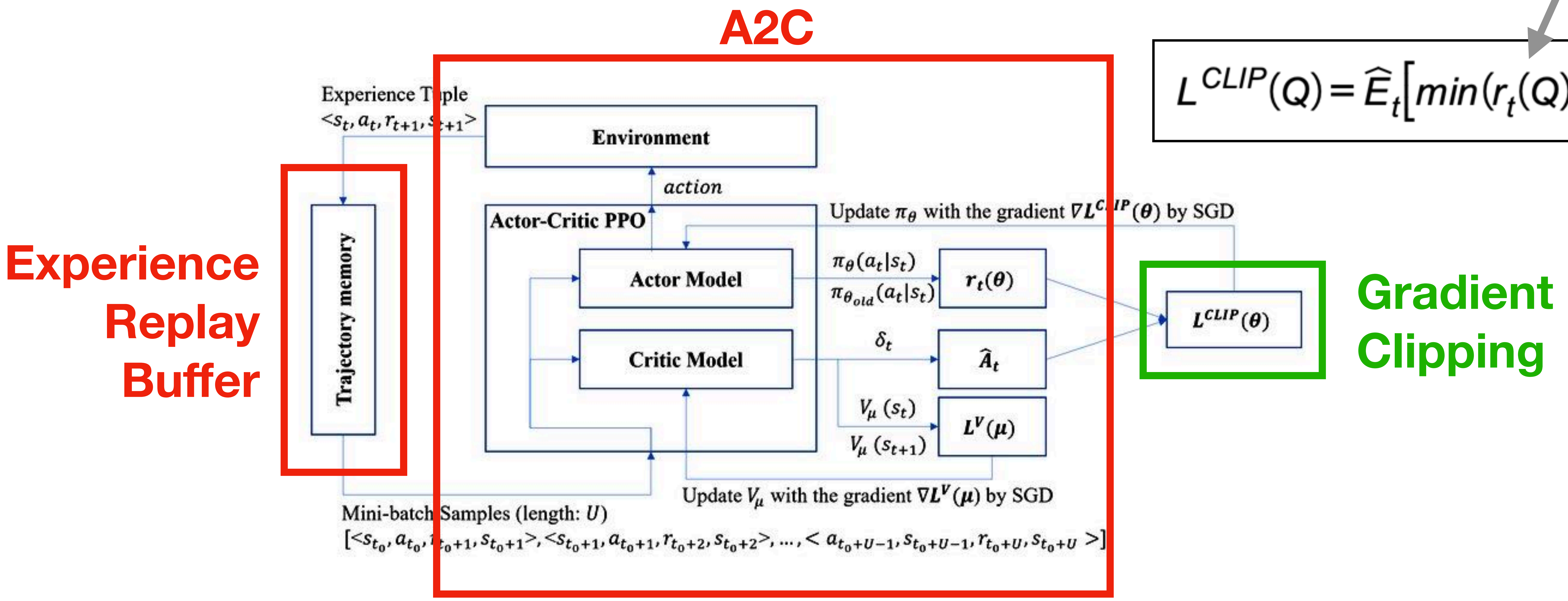
PPO

= A2C + policy gradient clipping

Surrogate policy function

$$r_t(Q) = \frac{\pi_Q(a_t|s_t)}{\pi_{Q_{old}}(a_t|s_t)}$$

$$L^{CLIP}(Q) = \widehat{E}_t\left[min(r_t(Q)\widehat{A}_t,\ clip(r_t(Q),\ \underline{1-\epsilon\ ,\ 1+\epsilon})\widehat{A}_t)\right]$$



[1−∈,1+∈]

Clipped range

Advantage (value gain)

PPO

Model architecture

```
def OurModel(input_shape, action_space, lr):
    X_input = Input(input_shape)

    X = Flatten(input_shape=input_shape)(X_input)

    X = Dense(512, activation="elu", kernel_initializer='he_uniform')(X)

    action = Dense(action_space, activation="softmax", kernel_initializer='he_uniform')(X)
    value = Dense(1, activation='linear', kernel_initializer='he_uniform')(X)

    def ppo_loss(y_true, y_pred):
        # Defined in https://arxiv.org/abs/1707.06347
        advantages, prediction_picks, actions = \
            y_true[:, :1], y_true[:, 1:1+action_space], y_true[:, 1+action_space:]
        LOSS_CLIPPING = 0.2
        ENTROPY_LOSS = 5e-3

        prob = y_pred * actions
        old_prob = actions * prediction_picks
        r = prob/(old_prob + 1e-10)
        p1 = r * advantages
        p2 = K.clip(r, min_value=1 - LOSS_CLIPPING, max_value=1 + LOSS_CLIPPING) * advantages
        loss =  -K.mean(K.minimum(p1, p2) + ENTROPY_LOSS * -(prob * K.log(prob + 1e-10)))

        return loss

    Actor = Model(inputs = X_input, outputs = action)
    Actor.compile(loss=ppo_loss, optimizer=RMSprop(lr=lr))

    Critic = Model(inputs = X_input, outputs = value)
    Critic.compile(loss='mse', optimizer=RMSprop(lr=lr))

    return Actor, Critic
```

$$L^{CLIP}(Q) = \widehat{E}_t\left[min(r_t(Q)\widehat{A}_t,\ clip(r_t(Q), 1-\epsilon, 1+\epsilon)\widehat{A}_t)\right]$$

PPO

Ray config (.yaml)

```yaml
# $ python train.py -f tuned_configs/pong-ppo.yaml
#
pong-ppo:
    env: PongNoFrameskip-v4
    run: PPO
    config:
        # Works for both torch and tf.
        framework: tf
        lambda: 0.95
        kl_coeff: 0.5
        clip_rewards: True
        clip_param: 0.1
        vf_clip_param: 10.0
        entropy_coeff: 0.01
        train_batch_size: 5000
        rollout_fragment_length: 20
        sgd_minibatch_size: 500
        num_sgd_iter: 10
        num_workers: 32
        num_envs_per_worker: 5
        batch_mode: truncate_episodes
        observation_filter: NoFilter
        num_gpus: 1
        model:
            dim: 42
            vf_share_layers: true
```

Homework

Assault + Gym Env + Ray RLlib

- **Train & tune A2C on Assault**
  Tuned examples: https://github.com/ray-project/ray/tree/master/rllib/tuned_examples/a2c
- **Train & tune A3C on Assault**
  Tuned examples: https://github.com/ray-project/ray/tree/master/rllib/tuned_examples/a3c
- **Train & tune PPO on Assault**
  Tuned examples: https://github.com/ray-project/ray/tree/master/rllib/tuned_examples/ppo