Multi-Agent Reinforcement Learning

Deep Reinforcement Learning

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer
Slides by: Leonard Hinckeldey

Multi-Agent Reinforcement Learning: Foundations and Modern Approaches

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer

Published by MIT Press
(print version will be released on 17 December 2024)

This lecture is based on *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches* by Stefano V. Albrecht, Filippos Christianos and Lukas Schäfer

The book can be downloaded for free at www.marl-book.com.

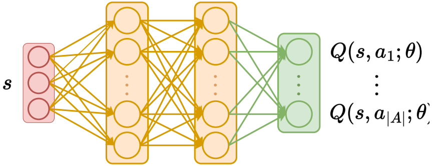
Lecture Outline

- Deep-Q learning
- Moving target problem
- Addressing correlations in consecutive experiences
- Policy gradient algorithms
- Concurrent training

Deep Q-Learning

Deep Q-Learning

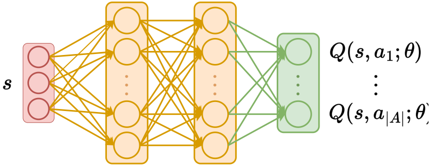
For **deep** Q-learning, we use a neural network to approximate the Q function.



- To train this we could define a loss function $\mathcal{L}(\theta) = \left(y^t - Q(s^t, a^t; \theta)\right)$
- But unlike supervised learning, we are not given y^t beforehand

Deep Q-Learning

For **deep** Q-learning, we use a neural network to approximate the Q function.



- To train this we could define a loss function $\mathcal{L}(\theta) = \left(y^t - Q(s^t, a^t; \theta)\right)$
- But unlike supervised learning, we are not given y^t beforehand
- We can use the Q-learning update rule to define our y^t

$$y^t = \begin{cases} r^t & \text{if } s^{t+1} \text{ is terminal} \\ r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta) & \text{otherwise} \end{cases}$$

Naive Deep Q-Learning Pseudo-Code

Algorithm Deep Q-learning

Initialize value network Q with random parameters θ
for every episode **do**
    for time step $t = 0, 1, 2, \ldots$ **do**
        Observe current state s^t
        With probability ϵ: choose random action $a^t \in A$
        Otherwise: choose $a^t \in \arg\max_a Q(s^t, a; \theta)$
        Apply action a^t; observe reward r^t and next state s^{t+1}
        if s^{t+1} is terminal **then**
            Target $y^t \leftarrow r^t$
        else
            Target $y^t \leftarrow r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$
        Loss $\mathcal{L}(\theta) \leftarrow \left(y^t - Q(s^t, a^t; \theta)\right)^2$
        Update parameters θ by minimising the loss $\mathcal{L}(\theta)$

Naive Deep Q-Learning Pseudo-Code

Algorithm Deep Q-learning

Initialize value network Q with random parameters θ
for every episode **do**
  for time step $t = 0, 1, 2, \ldots$ **do**
    Observe current state s^t
    With probability ϵ: choose random action $a^t \in A$
    Otherwise: choose $a^t \in \arg\max_a Q(s^t, a; \theta)$
    Apply action a^t; observe reward r^t and next state s^{t+1}
    if s^{t+1} is terminal **then**
      Target $y^t \leftarrow r^t$
    else
      Target $y^t \leftarrow r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$
    Loss $\mathcal{L}(\theta) \leftarrow \left(y^t - Q(s^t, a^t; \theta)\right)^2$
    Update parameters θ by minimising the loss $\mathcal{L}(\theta)$

This naive application of neural networks to RL algorithms suffers from several challenges.

The Moving Target Problem

Problem

The moving target problem arises from the bootstrapped targets:

$$y^t = r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$$

The Moving Target Problem

Problem

The moving target problem arises from the bootstrapped targets:

$$y^t = r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$$

- Value function with NNs generalize value estimates across inputs
- As targets y^t depend on θ, any update to θ changes the target

$\rightarrow$ This non-stationarity of the targets makes it difficult to learn optimal θ

The Moving Target Problem

Problem

The moving target problem arises from the bootstrapped targets:

$$y^t = r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$$

- Value function with NNs generalize value estimates across inputs
- As targets y^t depend on θ, any update to θ changes the target

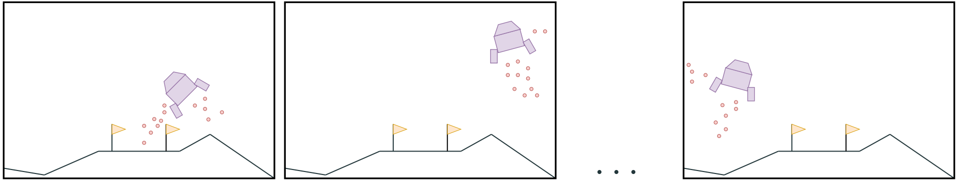
$\rightarrow$ This non-stationarity of the targets makes it difficult to learn optimal θ

Solution

One potential solution is to use a **target network** with parameters $\bar{\theta}$, which are updated less frequently than our Q network's parameters θ

Correlation of Consecutive Experiences

- Most ML algorithms using NNs assume i.i.d data
- In RL, we collect data by interacting with an MDP with $s^{t+1} \sim \mathcal{T}(s^t, a^t) \rightarrow$ **not** i.i.d



Correlation of Consecutive Experiences

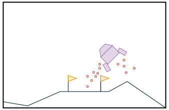
- Most ML algorithms using NNs assume i.i.d data
- In RL, we collect data by interacting with an MDP with $s^{t+1} \sim \mathcal{T}(s^t, a^t) \rightarrow$ **not** i.i.d

Problem

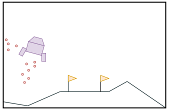
This correlated data can lead to **overfitting** of the value function to recent experiences, and result in **catastophic forgetting** of previously learned estimates.

Correlation of Consecutive Experiences

- Most ML algorithms using NNs assume i.i.d data
- In RL, we collect data by interacting with an MDP with $s^{t+1} \sim \mathcal{T}(s^t, a^t) \rightarrow$ **not** i.i.d



$\ldots$



Problem

This correlated data can lead to **overfitting** of the value function to recent experiences, and result in **catastophic forgetting** of previously learned estimates.

Solution

We train on samples of previous experiences stored in a **replay buffers**.

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

- Tabular value function $\longrightarrow$ neural network
- Moving target problem $\longrightarrow$ target networks
- Correlated experiences $\longrightarrow$ replay buffer

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Target networks:

- Compute target estimates with target network parameters $\bar{\theta}$:

$$y^t \leftarrow r^t + \gamma \max_{a'} Q(s^t + 1, a'; \bar{\theta})$$

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Target networks:

- Compute target estimates with target network parameters $\bar{\theta}$:

$$y^t \leftarrow r^t + \gamma \max_{a'} Q(s^t + 1, a'; \bar{\theta})$$

- Select actions according to the "main" value network $\rightarrow a^t \in \arg\max_a Q(s^t, a; \theta)$

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Target networks:

- Compute target estimates with target network parameters $\bar{\theta}$:

$$y^t \leftarrow r^t + \gamma \max_{a'} Q(s^t + 1, a'; \bar{\theta})$$

- Select actions according to the "main" value network $\rightarrow a^t \in \arg\max_a Q(s^t, a; \theta)$
- Update the "main" value network parameters θ by minimizing the DQL loss $\mathcal{L}(\theta)$

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Target networks:

- Compute target estimates with target network parameters $\bar{\theta}$:

$$y^t \leftarrow r^t + \gamma \max_{a'} Q(s^t + 1, a'; \bar{\theta})$$

- Select actions according to the "main" value network $\rightarrow a^t \in \arg\max_a Q(s^t, a; \theta)$
- Update the "main" value network parameters θ by minimizing the DQL loss $\mathcal{L}(\theta)$
- Update the **target network** parameters in regular intervals $\bar{\theta} \leftarrow \theta$

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Replay buffers:

- Store experience tuples (s^t, a^t, r^t, s^{t+1}) in a replay buffer $\mathcal{D}$

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Replay buffers:

- Store experience tuples (s^t, a^t, r^t, s^{t+1}) in a replay buffer $\mathcal{D}$
- To compute the loss, sample batches of experience tuples (uniformly at random) from the replay buffer $\mathcal{B} \sim \mathcal{U}(\mathcal{D})$

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Replay buffers:

- Store experience tuples (s^t, a^t, r^t, s^{t+1}) in a replay buffer $\mathcal{D}$
- To compute the loss, sample batches of experience tuples (uniformly at random) from the replay buffer $\mathcal{B} \sim \mathcal{U}(\mathcal{D})$
- Random sampling ”breaks” correlations, allowing for a more stable optimization

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Replay buffers:

- Store experience tuples (s^t, a^t, r^t, s^{t+1}) in a replay buffer $\mathcal{D}$
- To compute the loss, sample batches of experience tuples (uniformly at random) from the replay buffer $\mathcal{B} \sim \mathcal{U}(\mathcal{D})$
- Random sampling ”breaks” correlations, allowing for a more stable optimization
- Also reuses experiences during training $\rightarrow$ improved sample efficiency

Deep Q-Networks

Deep Q-networks (DQN) is a foundational deep RL algorithm based on Q-learning.

Replay buffers:

- Store experience tuples (s^t, a^t, r^t, s^{t+1}) in a replay buffer $\mathcal{D}$
- To compute the loss, sample batches of experience tuples (uniformly at random) from the replay buffer $\mathcal{B} \sim \mathcal{U}(\mathcal{D})$
- Random sampling "breaks" correlations, allowing for a more stable optimization
- Also reuses experiences during training $\rightarrow$ improved sample efficiency

Note

Replay buffers can only be used for **off-policy** algorithms. We use experiences collected from previous (different) policies, which change as we update θ.

DQN Pseudo Code

Algorithm Deep Q-networks (DQN)

Initialize value network Q with random parameters θ
Initialize target network with parameters $\bar{\theta} = \theta$
Initialize an empty replay buffer $\mathcal{D} = \{\}$
for every episode **do**
  for time step $t = 0, 1, 2, \ldots$ **do**
    Observe current state s^t
    With probability ϵ: choose random action $a^t \in A$
    Otherwise: choose $a^t \in \arg\max_a Q(s^t, a; \theta)$
    Apply action a^t; observe reward r^t and next state s^{t+1}
    Store transition (s^t, a^t, r^t, s^{t+1}) in replay buffer $\mathcal{D}$
    Sample random mini-batch of B transitions (s^k, a^k, r^k, s^{k+1}) from $\mathcal{D}$
    if s^{k+1} is terminal **then**
      Targets $y^k \leftarrow r^k$
    else
      Targets $y^k \leftarrow r^k + \gamma \max_{a'} Q(s^{k+1}, a'; \bar{\theta})$
    Loss $\mathcal{L}(\theta) \leftarrow \frac{1}{B} \sum_{k=1}^{B} \left(y^k - Q(s^k, a^k; \theta)\right)^2$
    Update parameters θ by minimising the loss $\mathcal{L}(\theta)$
    In a set interval, update target network parameters $\bar{\theta}$

Problem

DQN tends to **overestimate** values with 1-step targets

$$y^k \leftarrow r^k + \gamma \max_{a'} Q(s^{k+1}, a'; \bar{\theta})$$

- Using the max operator, select the maximum value estimate for the target
- Since our value estimates do not necessarily reflect the true value function, the max operation will likely select an overestimated action-value estimate
- This can slow down the convergence of the algorithm as the agent spends too much time exploring states with overestimated values

Double Q-Learning

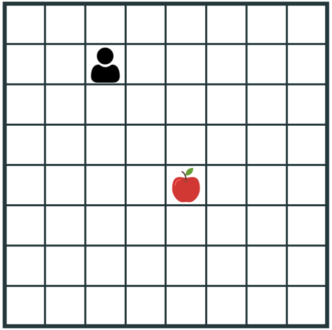
Solution

Double Q-learning reduces this overestimation bias by decoupling the action selection from value estimation using separate function approximations.

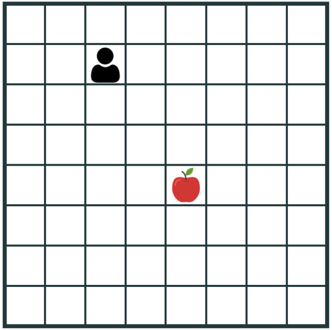
- This can be achieved with minimal changes in DQN
- DDQN uses the primary Q-network (with parameters θ) to select actions while using the target network (with parameters $\bar{\theta}$) to estimate the value of actions
- The target thus becomes:

$$y^t = \begin{cases} r^t & \text{if } s^{t+1} \text{ is terminal} \\ r^t + \gamma Q(s^{t+1}, \operatorname{arg\,max}_{a'} Q(s^{t+1}, a'; \theta); \bar{\theta}) & \text{otherwise} \end{cases}$$

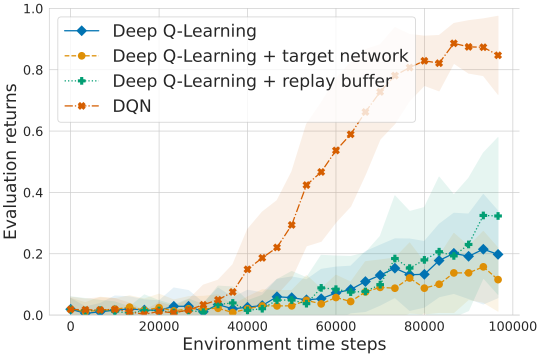
Deep Q-learning in Practice



(a) Single-agent level-based foraging environment

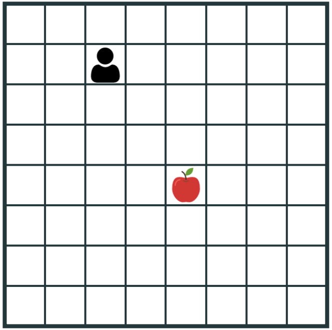


(a) Single-agent level-based foraging environment

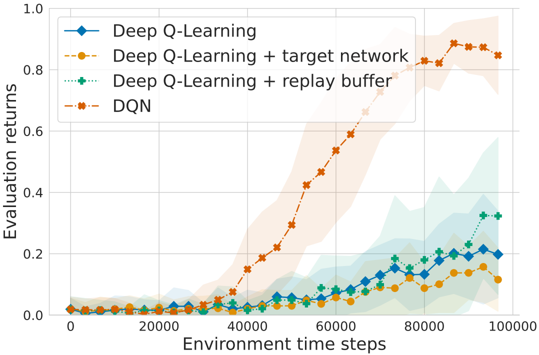


(b) Learning curves

Deep Q-learning in Practice



(a) Single-agent level-based foraging environment



(b) Learning curves

Note that in isolation, neither the addition of **target networks** nor of a **replay buffer** are sufficient to stably train the agent with deep Q-learning in this environment.

Policy Gradient Algorithms

Policy Gradients

So far, we have considered a parameterized value function, but we can also directly parameterize the policy π.

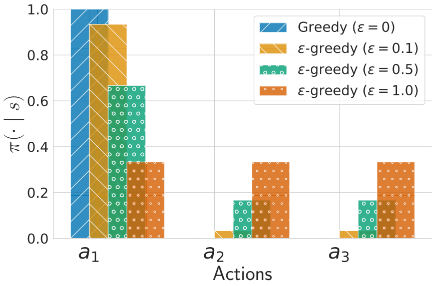
- For instance, we can use a NN for our policy π with parameters ϕ.
- The policy network receives a state s as input and outputs a scalar value for each action
- These scalars $l(s, a)$ represent the preference of the policy to select action a in state s

So far, we have considered a parameterized value function, but we can also directly parameterize the policy π.

- For instance, we can use a NN for our policy π with parameters ϕ.
- The policy network receives a state s as input and outputs a scalar value for each action
- These scalars $l(s, a)$ represent the preference of the policy to select action a in state s
- These preferences are then transformed into a probability distribution across the action space using a **softmax** function:

$$\pi(a \mid s; \phi) = \frac{e^{l(s,a;\phi)}}{\sum_{a' \in A} e^{l(s,a';\phi)}}$$

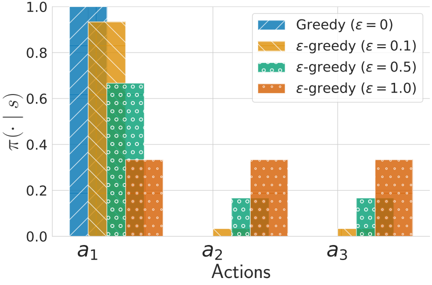
Advantages of Learning a Policy



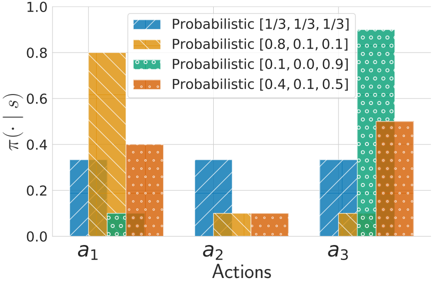
ϵ-greedy policies

- ϵ-greedy policies struggle to represent diverse probabilistic policies

Advantages of Learning a Policy



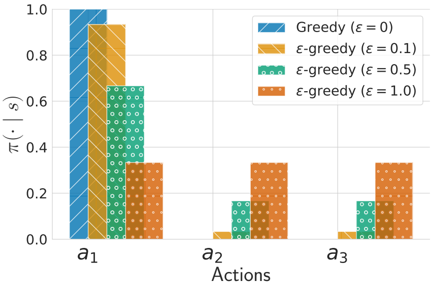
ϵ-greedy policies



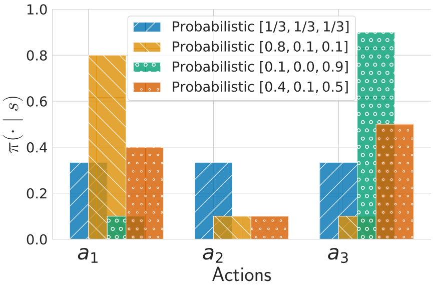
Probabilistic policies

- ϵ-greedy policies struggle to represent diverse probabilistic policies
- Policy gradient algorithms allow us to represent **any** probabilistic policy

Advantages of Learning a Policy



ϵ-greedy policies



Probabilistic policies

- ϵ-greedy policies struggle to represent diverse probabilistic policies
- Policy gradient algorithms allow us to represent **any** probabilistic policy
- Policy gradients are also effective for representing continuous action spaces

Policy Gradient Theorem

How do we update parameters ϕ of the policy? Using the **policy gradient theorem**, we can express the gradient of the performance of a policy with respect to the parameter ϕ of the policy.

$$\nabla_{\phi} J(\phi) \propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^{\pi}(s, a) \nabla_{\phi} \pi(a \mid s; \phi)$$

- J represents a function measuring the quality policy π
- $Pr(s \mid \pi)$ is the state-visitation distribution for policy π
- $Q^{\pi}(s, a)$ is the value for a given action and state under π
- The J function is similar to a loss function with the key difference that we aim to **maximize** rather than minimize it

Policy Gradient Theorem

How do we update parameters ϕ of the policy? Using the **policy gradient theorem**, we can express the gradient of the performance of a policy with respect to the parameter ϕ of the policy.

$$\nabla_\phi J(\phi) \propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi)$$

Note

The policy gradient theorem assumes that $Pr(s \mid \pi)$ are given under the currently optimized policy $\pi \rightarrow$ needs **on-policy** data

Policy Gradient Theorem – Continued

The assumption that $Pr(s \mid \pi)$ is **on-policy** becomes apparent through a simple derivation.

$$\nabla_\phi J(\phi) \propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi)$$

Policy Gradient Theorem – Continued

The assumption that $Pr(s \mid \pi)$ is **on-policy** becomes apparent through a simple derivation.

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi) \right]
\end{aligned}
$$

Policy Gradient Theorem – Continued

The assumption that $Pr(s \mid \pi)$ is **on-policy** becomes apparent through a simple derivation.

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} Q^\pi(s, a) \nabla_\phi \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} \pi(a \mid s; \phi) Q^\pi(s, a) \frac{\nabla_\phi \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right]
\end{aligned}
$$

Policy Gradient Theorem – Continued

The assumption that $Pr(s \mid \pi)$ is **on-policy** becomes apparent through a simple derivation.

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s,a) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} Q^\pi(s,a) \nabla_\phi \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} \pi(a \mid s; \phi) Q^\pi(s,a) \frac{\nabla_\phi \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[Q^\pi(s,a) \frac{\nabla_\phi \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right]
\end{aligned}
$$

Policy Gradient Theorem – Continued

The assumption that $Pr(s \mid \pi)$ is **on-policy** becomes apparent through a simple derivation.

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} Q^\pi(s,a) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} Q^\pi(s,a) \nabla_\phi \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[\sum_{a \in A} \pi(a \mid s; \phi) Q^\pi(s,a) \frac{\nabla_\phi \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[Q^\pi(s,a) \frac{\nabla_\phi \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right] \\
&= \mathbb{E}_{a \sim \pi(\cdot \mid s; \phi)} \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right]
\end{aligned}
$$

Policy Gradient Theorem – Continued

Intuition

$$\nabla_\phi J(\phi) \propto \mathbb{E}_{a\sim\pi(\cdot|s;\phi)}\left[Q^\pi(s,a)\frac{\nabla_\phi \pi(a \mid s;\phi)}{\pi(a \mid s;\phi)}\right]$$

We can interpret the components of the policy gradient theorem as follows:

- $\nabla_\phi \pi(a \mid s;\phi)$: gradient of ϕ pointing in the direction that most increase the probability of taking action a when in state s

Policy Gradient Theorem – Continued

Intuition

$$\nabla_\phi J(\phi) \propto \mathbb{E}_{a \sim \pi(\cdot|s;\phi)} \left[Q^\pi(s,a) \frac{\nabla_\phi \pi(a \mid s;\phi)}{\pi(a \mid s;\phi)} \right]$$

We can interpret the components of the policy gradient theorem as follows:

- $\nabla_\phi \pi(a \mid s;\phi)$: gradient of ϕ pointing in the direction that most increase the probability of taking action a when in state s
- $Q^\pi(s,a)$: expected returns as a quality measure of taking action a in state s

Policy Gradient Theorem – Continued

Intuition

$$\nabla_\phi J(\phi) \propto \mathbb{E}_{a\sim\pi(\cdot|s;\phi)}\left[Q^\pi(s,a)\frac{\nabla_\phi \pi(a \mid s;\phi)}{\pi(a \mid s;\phi)}\right]$$

We can interpret the components of the policy gradient theorem as follows:

- $\nabla_\phi \pi(a \mid s;\phi)$: gradient of ϕ pointing in the direction that most increase the probability of taking action a when in state s
- $Q^\pi(s,a)$: expected returns as a quality measure of taking action a in state s
- $\frac{1}{\pi(a|s;\phi)}$: normalization coefficient to account for varying probabilities of different actions under the policy

REINFORCE: Monte Carlo

To apply the policy gradient theorem we must find a way to derive expected returns, one way to do this is to use Monte Carlo (MC) methods.

- REINFORCE is a policy gradient algorithm that uses MC to estimate the expected returns of a policy

REINFORCE: Monte Carlo

To apply the policy gradient theorem we must find a way to derive expected returns, one way to do this is to use Monte Carlo (MC) methods.

- REINFORCE is a policy gradient algorithm that uses MC to estimate the expected returns of a policy
- The algorithms minimize the following loss for an episodic history $h = \{s^0, a^0, r^0, ...s^{T-1}, a^{T-1}, r^{T-1}, s^T\}$:

$$\mathcal{L}(\phi) = -\frac{1}{T}\sum_{t=0}^{T-1}\left(\sum_{\tau=1}^{t-1}\gamma^{\tau-1}\mathcal{R}(s^\tau, a^\tau, s^{\tau+1})\right) log\pi(a^t \mid s^t; \phi)$$

REINFORCE: Monte Carlo

To apply the policy gradient theorem we must find a way to derive expected returns, one way to do this is to use Monte Carlo (MC) methods.

- **REINFORCE** is a policy gradient algorithm that uses MC to estimate the expected returns of a policy
- The algorithms minimize the following loss for an episodic history $h = \{s^0, a^0, r^0, ...s^{T-1}, a^{T-1}, r^{T-1}, s^T\}$:

$$\mathcal{L}(\phi) = -\frac{1}{T}\sum_{t=0}^{T-1}\left(\sum_{\tau=1}^{t-1}\gamma^{\tau-1}\mathcal{R}(s^\tau, a^\tau, s^{\tau+1})\right) log\pi(a^t \mid s^t; \phi)$$

- Note the to-be-minimized loss has a prepended negative sign, as we want to **maximize** expected returns

REINFORCE Pseudocode

Algorithm REINFORCE

Initialize policy network π with random parameters ϕ
for every episode **do**
  for time step $t = 0, 1, 2, \ldots, T-1$ **do**
    Observe current state s^t
    Sample action $a^t \sim \pi(\cdot \mid s^t; \phi)$
    Apply action a^t; observe reward r^t and next state s^{t+1}
  Loss $\mathcal{L}(\phi) \leftarrow -\frac{1}{T} \sum_{t=0}^{T-1} \left(\sum_{\tau=t}^{T-1} \gamma^{\tau-t} r^{\tau} \right) \log \pi(a^t \mid s^t; \phi)$
  Update parameters ϕ by minimizing the loss $\mathcal{L}(\phi)$

Baseline to Reduce Variance

Problem

High variance of MC estimates causes unstable gradients and training.

Baseline to Reduce Variance

Problem

High variance of MC estimates causes unstable gradients and training.

Solution

- One way to reduce variance is subtract a **baseline** from the return estimates
- A common choice of baseline is a state-value function $V(s)$ which can be trained to minimize the loss $\mathcal{L}(\theta) = \frac{1}{T}\sum_{t=1}^{T-1}(u(h^t) - V(s^t;\theta))^2$
- The REINFORCE policy loss would then become

$$\mathcal{L}(\phi) = -\frac{1}{T}\sum_{t=0}^{T-1}\left(u(h^t) - V(s^t;\theta)\right) log\pi(a^t \mid s^t;\phi)$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$\nabla_\phi J(\phi) \propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s, a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi)$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$\begin{aligned}\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s, a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\ &= \mathbb{E}_\pi \left[\left(Q^\pi(s, a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi)\right]\end{aligned}$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[\left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi)\right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right] - \mathbb{E}_\pi \left[b(s) \nabla_\phi \log \pi(a \mid s; \phi)\right]
\end{aligned}
$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[\left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \mathbb{E}_\pi \left[b(s) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi \sum_{a \in A} \pi(a \mid s; \phi)
\end{aligned}
$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[\left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \mathbb{E}_\pi \left[b(s) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi \sum_{a \in A} \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi 1
\end{aligned}
$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s, a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[\left(Q^\pi(s, a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s, a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \mathbb{E}_\pi \left[b(s) \nabla_\phi \log \pi(a \mid s; \phi) \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s, a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi \sum_{a \in A} \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[Q^\pi(s, a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi 1 \\
&= \mathbb{E}_\pi \left[Q^\pi(s, a) \nabla_\phi \log \pi(a \mid s; \phi) \right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) 0
\end{aligned}
$$

Baselines Continued

The inclusion of the baseline does not affect the gradients in expectation:

$$
\begin{aligned}
\nabla_\phi J(\phi) &\propto \sum_{s \in S} \Pr(s \mid \pi) \sum_{a \in A} \left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[\left(Q^\pi(s,a) - b(s)\right) \nabla_\phi \log \pi(a \mid s; \phi)\right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right] - \mathbb{E}_\pi \left[b(s) \nabla_\phi \log \pi(a \mid s; \phi)\right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi \sum_{a \in A} \pi(a \mid s; \phi) \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) \nabla_\phi 1 \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right] - \sum_{s \in S} \Pr(s \mid \pi) b(s) 0 \\
&= \mathbb{E}_\pi \left[Q^\pi(s,a) \nabla_\phi \log \pi(a \mid s; \phi)\right]
\end{aligned}
$$

Actor-Critic Algorithms

The policy gradient theorem allows us to optimize our parameterized policy; we still need to approximate expected returns.

- MC methods have high variance and require an entire episode to compute estimates
- **Actor-critic** algorithms aim to reduce variance and update more often by using **bootstrapped return estimates**

$$\begin{aligned}\mathbb{E}_\pi\left[u(h^t) \mid s^t\right] &= \mathbb{E}_\pi\left[R(s^t, a^t, s^{t+1}) + \gamma u(h^{t+1}) \mid s^t, a^t \sim \pi(\cdot \mid s^t)\right] \\ &= \mathbb{E}_\pi\left[R(s^t, a^t, s^{t+1}) + \gamma V(s^{t+1}) \mid s^t, a^t \sim \pi(\cdot \mid s^t)\right]\end{aligned}$$

- We now train a **critic** (value function approximator) alongside the **actor** (parameterized policy) which acts as a baseline

Balancing Bias and Variance

Problem

MC estimates have high **variance** and bootstrapping introduces **bias** since the value function might not yet approximate the true expected returns.

Balancing Bias and Variance

Problem

MC estimates have high **variance** and bootstrapping introduces **bias** since the value function might not yet approximate the true expected returns.

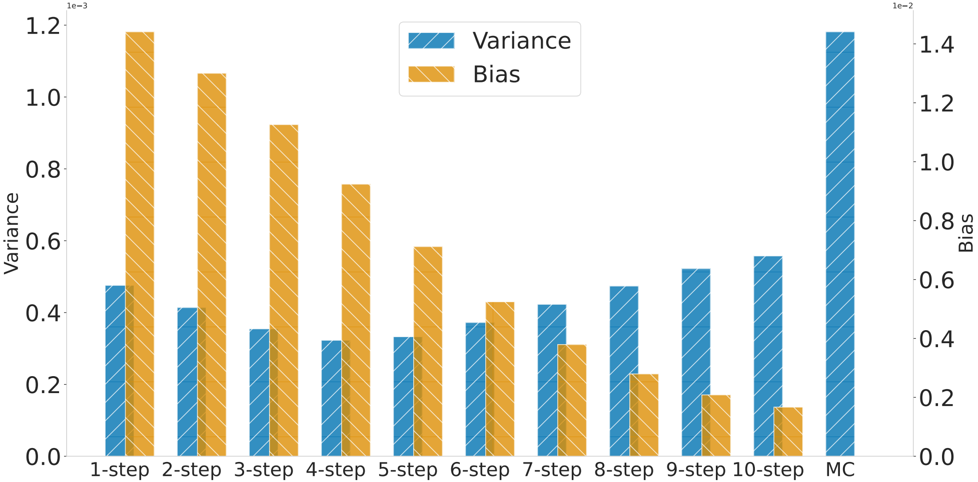
Solution

N-step returns allow us to balance between bias and variance:

$$\mathbb{E}_\pi\left[u(h^t) \mid s^t\right] = \mathbb{E}_\pi\left[\left(\sum_{\tau=0}^{N-1} \gamma^\tau \mathcal{R}(s^{t+\tau}, a^{t+\tau}, s^{t+\tau+1})\right) + \gamma^N V(s^{t+N}) \middle| s^t, a^\tau \sim \pi(\cdot \mid s^\tau)\right]$$

$N = 1 \rightarrow$ one-step bootstrapped returns $\ldots$ $N = T \rightarrow$ MC returns

Balancing Bias and Variance – Continued



Advantage Actor Critic (A2C)

Advantage actor-critic is a foundation actor-critic algorithm which uses the **advantage** of a policy to guide the policy gradients.

- The advantage is defined as

$$Adv^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) = r +$$

Advantage Actor Critic (A2C)

Advantage actor-critic is a foundation actor-critic algorithm which uses the **advantage** of a policy to guide the policy gradients.

- The advantage is defined as

$$Adv^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) = r+$$

- We can compute the advantage using only a state-value function:

$$Adv(s^t, a^t) = Q(s^t, a^t) - V(s^t) = \begin{cases} r^t - V(s^t) & \text{if } s^{t+1} \text{ is terminal} \\ r^t + \gamma V(s^{t+1}) - V(s^t) & \text{otherwise} \end{cases}$$

Advantage Actor Critic (A2C)

Advantage actor-critic is a foundation actor-critic algorithm which uses the **advantage** of a policy to guide the policy gradients.

- The advantage is defined as

$$Adv^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) = r+$$

- We can compute the advantage using only a state-value function:

$$Adv(s^t, a^t) = Q(s^t, a^t) - V(s^t) = \begin{cases} r^t - V(s^t) & \text{if } s^{t+1} \text{ is terminal} \\ r^t + \gamma V(s^{t+1}) - V(s^t) & \text{otherwise} \end{cases}$$

- We can also still use N-step returns to reduce bias during the advantage computation

A2C Pseudocode

Algorithm A2C

Initialize actor network π with random parameters ϕ
Initialize critic network V with random parameters θ
for every episode **do**
    for time step $t = 0, 1, 2, \ldots$ **do**
        Observe current state s^t
        Sample action $a^t \sim \pi(\cdot \mid s^t; \phi)$
        Apply action a^t; observe reward r^t and next state s^{t+1}
        if s^{t+1} is terminal **then**
            Advantage $Adv(s^t, a^t) \leftarrow r^t - V(s^t; \theta)$
            Critic target $y^t \leftarrow r^t$
        else
            Advantage $Adv(s^t, a^t) \leftarrow r^t + \gamma V(s^{t+1}; \theta) - V(s^t; \theta)$
            Critic target $y^t \leftarrow r^t + \gamma V(s^{t+1}; \theta)$
        Actor loss $\mathcal{L}(\phi) \leftarrow -Adv(s^t, a^t) \log \pi(a^t \mid s^t; \phi)$
        Critic loss $\mathcal{L}(\theta) \leftarrow (y^t - V(s^t; \theta))^2$
        Update parameters ϕ by minimizing the actor loss $\mathcal{L}(\phi)$
        Update parameters θ by minimizing the critic loss $\mathcal{L}(\theta)$

Proximal Policy Optimization (PPO)

Problem

Policy gradient methods can cause significant shifts in the policy with a single update which can worsen the policy!

Solution

Limit the change of the policy in a single update $\rightarrow$ **trust region** of a policy

Proximal policy optimization (PPO) computes an efficient surrogate objective to limit the change in the policy when executing multiple updates:

$$\mathcal{L}(\phi) = -\min \begin{pmatrix} \rho(s^t, a^t) Adv(s^t, a^t), \\ \text{clip}\left(\rho(s^t, a^t), 1-\epsilon, 1+\epsilon\right) Adv(s^t, a^t) \end{pmatrix}$$

PPO Surrogate Objective

$$\mathcal{L}(\phi) = -\min \left(\begin{array}{l} \rho(s^t, a^t)Adv(s^t, a^t), \\ \text{clip}\left(\rho(s^t, a^t), 1-\epsilon, 1+\epsilon\right) Adv(s^t, a^t) \end{array} \right)$$

- ρ represents the importance sampling ratio $\rho(s, a) = \frac{\pi(a|s;\phi)}{\pi_\beta(a|s)}$
- π_β represents the behavior policy followed to select action a^t in state s^t
- ϵ is a hyperparameter that determines the allowed change of the policy

PPO Surrogate Objective

$$\mathcal{L}(\phi) = -\min \left(\begin{array}{l} \rho(s^t, a^t) Adv(s^t, a^t), \\ \text{clip}\left(\rho(s^t, a^t), 1-\epsilon, 1+\epsilon\right) Adv(s^t, a^t) \end{array} \right)$$

- ρ represents the importance sampling ratio $\rho(s, a) = \frac{\pi(a|s;\phi)}{\pi_\beta(a|s)}$
- π_β represents the behavior policy followed to select action a^t in state s^t
- ϵ is a hyperparameter that determines the allowed change of the policy

Importance sampling ratios serves multiple purposes:

- Correct for differences in data distributions of π_β and π
- Represent measure of divergence between π_β and π $\rightarrow$ can be clipped to limit divergence

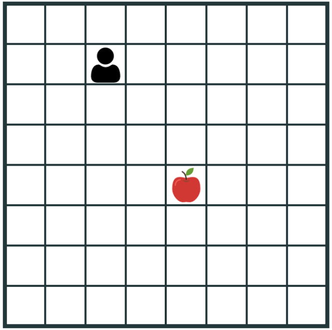
PPO Pseudocode

Algorithm Simplified proximal policy optimization (PPO)

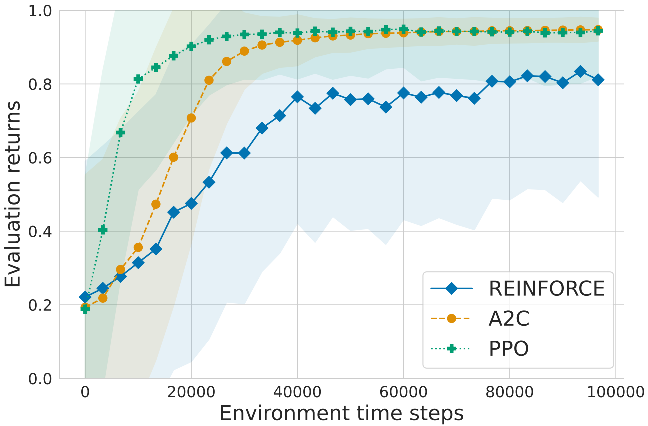
Initialize actor network π with random parameters ϕ
Initialize critic network V with random parameters θ
for every episode **do**
  for time step $t = 0, 1, 2, \ldots$ **do**
    Observe current state s^t
    Sample action $a^t \sim \pi(\cdot \mid s^t; \phi)$
    Apply action a^t; observe reward r^t and next state s^{t+1}
    $\pi_\beta(a^t \mid s^t) \leftarrow \pi(a^t \mid s^t; \phi)$
    for epoch $e = 1, \ldots, N_e$ **do**
      $\rho(s^t, a^t) \leftarrow \pi(a^t \mid s^t; \phi) \div \pi_\beta(a^t \mid s^t)$
      if s^{t+1} is terminal **then**
        Advantage $Adv(s^t, a^t) \leftarrow r^t - V(s^t; \theta)$
        Critic target $y^t \leftarrow r^t$
      else
        Advantage $Adv(s^t, a^t) \leftarrow r^t + \gamma V(s^{t+1}; \theta) - V(s^t; \theta)$
        Critic target $y^t \leftarrow r^t + \gamma V(s^{t+1}; \theta)$
      Actor loss $\mathcal{L}(\phi) \leftarrow -\min \begin{pmatrix} \rho(s^t, a^t) Adv(s^t, a^t), \\ \text{clip}\left(\rho(s^t, a^t), 1 - \epsilon, 1 + \epsilon\right) Adv(s^t, a^t) \end{pmatrix}$
      Critic loss $\mathcal{L}(\theta) \leftarrow \left(y^t - V(s^t; \theta)\right)^2$
      Update parameters ϕ by minimising the actor loss $\mathcal{L}(\phi)$
      Update parameters θ by minimising the critic loss $\mathcal{L}(\theta)$

- PPO executes multiple epochs of updates!
- For first epoch, $\pi = \pi_\beta$ $\rightarrow \rho = 1$
- After first epoch, $\pi \neq \pi_\beta$ $\rightarrow$ needs ρ to correct for off-policy data

Policy Gradient Algorithms in Practice



(a) Single-agent level-based foraging environment



(b) Learning curves

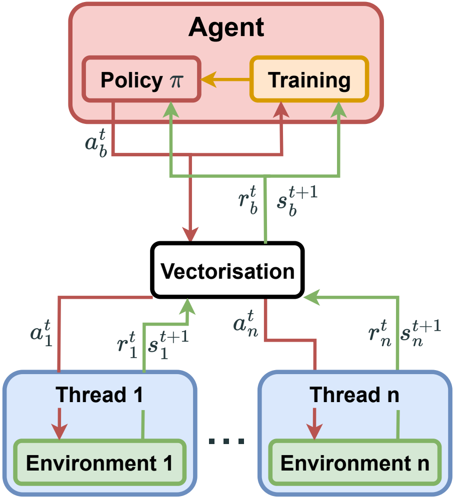
Concurrent Training

Concurrent Training of Policies

Policy gradient algorithms rely on on-policy data, which raises the question of how to deal with correlation in the collected data and how to increase sample efficiency.

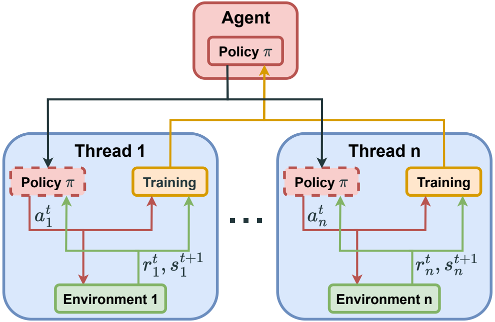
- Concurrent training of policies speeds up training by getting more samples (in parallel), leading to better gradients and breaking of correlation of data
- There are many ways to achieve this, but there are two simple methods commonly used, **synchronous training** and **asynchronous training**

Synchronous Training



- Initiates separate instances of the environment in separate threads
- At each timestep, the agent receives a batch of states and rewards from each thread
- The agent then independently chooses an action for each thread/environment
- Aggregate gradients across batch of experiences $\rightarrow$ more stable and efficient optimization

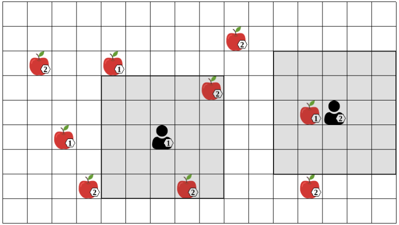
Asynchronous Training



- Asynchronous training parallelizes the optimization of the agent.
- Each thread separately computes the loss and gradients and optimizes the parameters of the agent's network
- Once gradients are computed, the central agent's network is updated
- Asynchronous training is particularly effective if multiple accelerators (e.g. GPUs) are available

Observation, States, and Histories in Practice

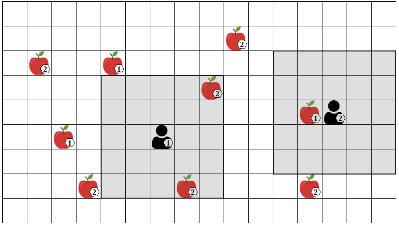
We have thus far only considered algorithms that condition on the entire states of the environment, in practice we however often have **partial observability**.



We want to condition value functions and policies on observation history $h^t = (o^0, ..., o^t)$

Observation, States, and Histories in Practice

We have thus far only considered algorithms that condition on the entire states of the environment, in practice we however often have **partial observability**.

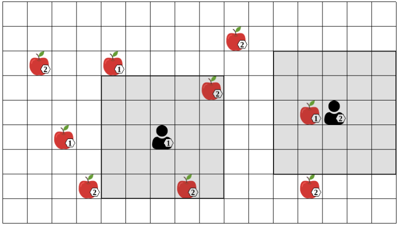


- Feedforward NN assume constant input size, this would require zero-padding vectors to the maximum episode length
- Zero-padding requires knowledge about maximum episode length and results in high-dimensional and sparse inputs

We want to condition value functions and policies on observation history $h^t = (o^0, ..., o^t)$

Observation, States, and Histories in Practice

We have thus far only considered algorithms that condition on the entire states of the environment, in practice we however often have **partial observability**.



We want to condition value functions and policies on observation history $h^t = (o^0, ..., o^t)$

- Feedforward NN assume constant input size, this would require zero-padding vectors to the maximum episode length
- Zero-padding requires knowledge about maximum episode length and results in high-dimensional and sparse inputs
- To avoid this we can use RNNs that process sequences of observations with one observation at a time while maintaining the previous history in the hidden state

Summary

We covered:

- Deep Q-learning
- The moving target problem and correlations of consecutive experiences
- Policy gradient algorithms
- Concurrent training of policies
- Observation, states, and histories under partial observability

Next we'll cover:

- Deep **multi agent** reinforcement learning