MLM: Deep RL

Monday, March 7, 2022

(Heavily inspired by the Stanford RL Course of Prof. Emma Brunskill, but all potential errors are mine.)

Plan for Today

- A very short recap of important concepts from last lectures.
- Value function approximation.
- Control with value function approximation.
- Intro to Bandits.

Part 1: Recap (Q-Learning)

State-Action Value Q

- **Definition:**

$$Q^{\pi}(s, a) = R(s, a) + \gamma \cdot \sum_{s' \in S} P(s' \mid s, a) \cdot V^{\pi}(s').$$

- **Intuition:**
  - The value of the return that we obtain if we first take the action a in the state s and then follow the policy π (including when we visit s again).
  - *Think of it as perturbing the policy π — we deviate from following the policy π only in the first step in s.*

ε-Greedy Policy

We assume ties are decided consistently

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|A|} & \text{when } a = \arg\max_{a \in A} Q(s, a) \\ \frac{\varepsilon}{|A|} & \text{when } a \neq \arg\max_{a \in A} Q(s, a) \end{cases}$$

Q-Learning

1. **Initialize:** set π to be some ε-greedy policy, set $t = 0$
2. **Sample** a using the distribution given by π_0 in the state s_0 *(for sampling, we will use the notation $a \sim \pi(s)$)*. **Take** the action a and **observe** r_0, s_1.
3. **While** s_t is not a terminal state:
    1. **Take** action $a \sim \pi(s_t)$ and observe r_t, s_{t+1}.
    2. $Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t) \right)$
    3. $\pi := \varepsilon$-greedy(Q)
    4. Set $t := t + 1$. Update ε, α /* *see next slides* */

Part 2: RL with Function Approximation (Problem Description)

Limitations of What We Saw So Far

- In the previous lectures, we assumed discrete MDPs with number of states that was not too large (i.e. the set S was not too large).

- Now imagine that we want to learn to play Atari games (which is what DeepMind did!) and we want to do it from the pixel inputs. How many states would we need if we wanted to use what we learned in the previous lectures? … Then we would need at least $128^{160 \cdot 192}$ states (128 colors with resolution 160 x 192 pixels).

- **What we need is *function approximation.***

Limitations of What We Saw So Far

- In the previous lectures, we assumed discrete MDPs with number of states that was not too large (i.e. the set S was not too large).

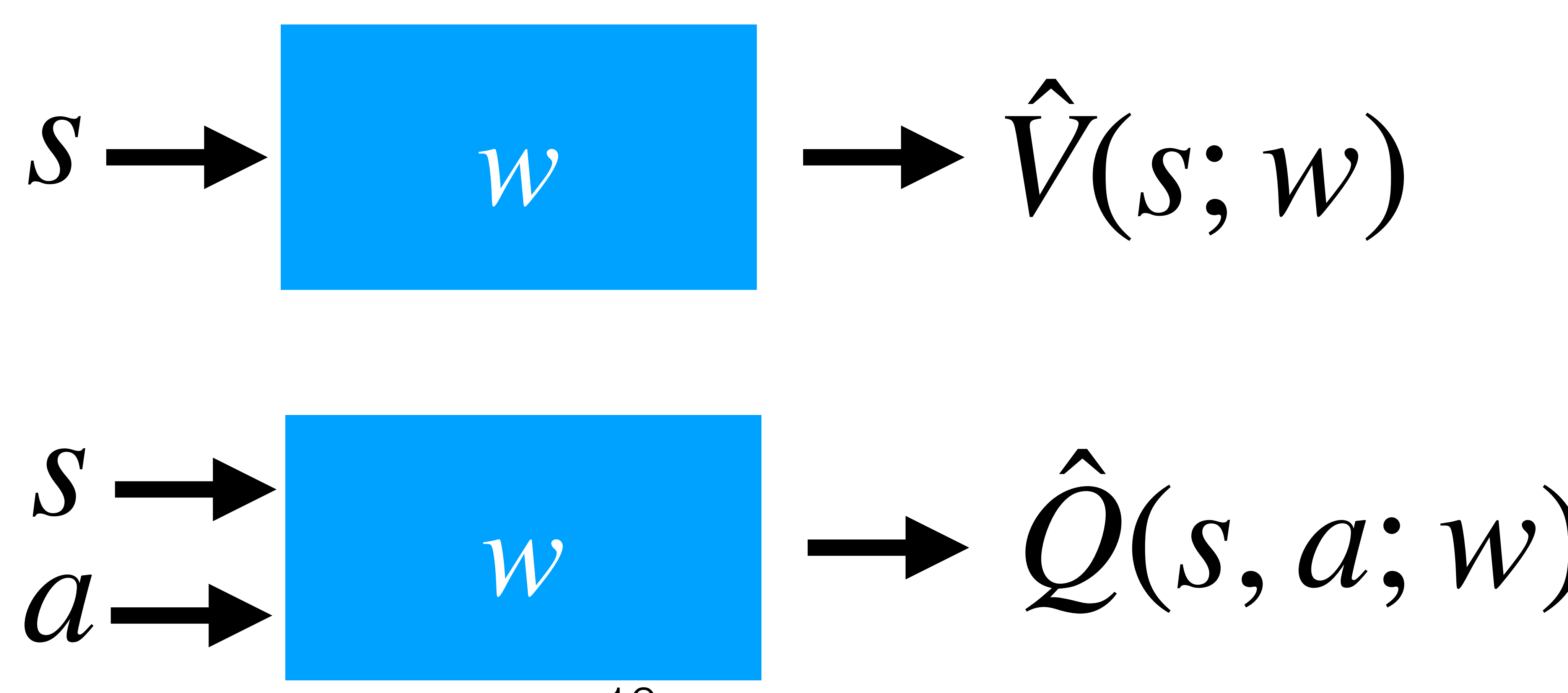
- Now imagine that we want to learn to play Atari games (which is what DeepMind did!) and we want to do it from the pixel inputs. How many states would we need if we wanted to use what we learned in the previous lectures? … Then we would need at least $128^{160 \cdot 192}$ states (128 colors with resolution 160 x 192 pixels).

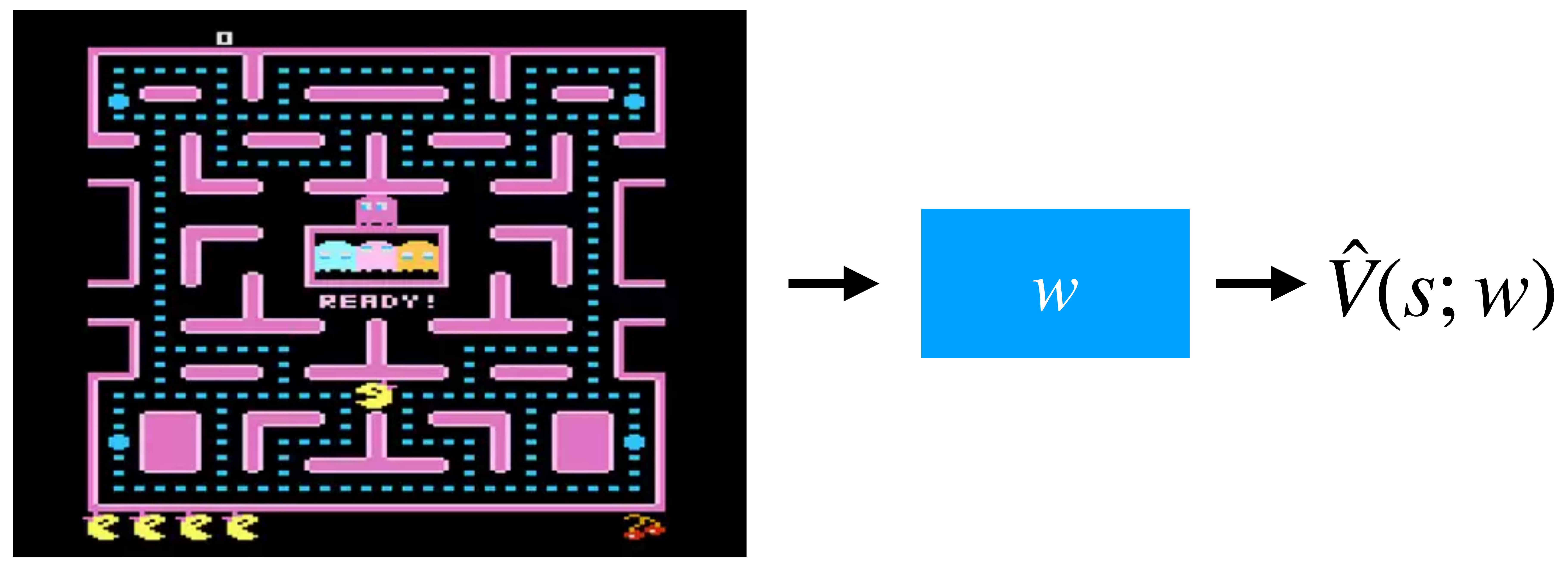
- **What we need is *function approximation.***

Basic Idea

- Do not represent the state value function V or the state-action value function Q explicitly.

- Represent the state value function $V(s)$ or the state-action value function $Q(s, a)$ approximately using a function from some parametrized family, e.g. as a neural network, linear function, decision tree…




0
READY!
w
$\hat{V}(s; w)$

$a, a \in \{$**left, right, up, down**$\}$

w

$\hat{Q}(s, a; w)$

State Representation

- States will be represented by feature vectors.
- The feature vector of a state s will be denoted as $\mathbf{x}(s)$ and we can think of it as a function mapping states to some vector space, e.g. $\mathbb{R}^d$, i.e. $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^T$.
- **Examples:**
    - Atari: the feature vector can, e.g., contain the intensities of the pixels (concatenated).
    - Pole balancing: physical features such as velocities, angles…

Linear Functions

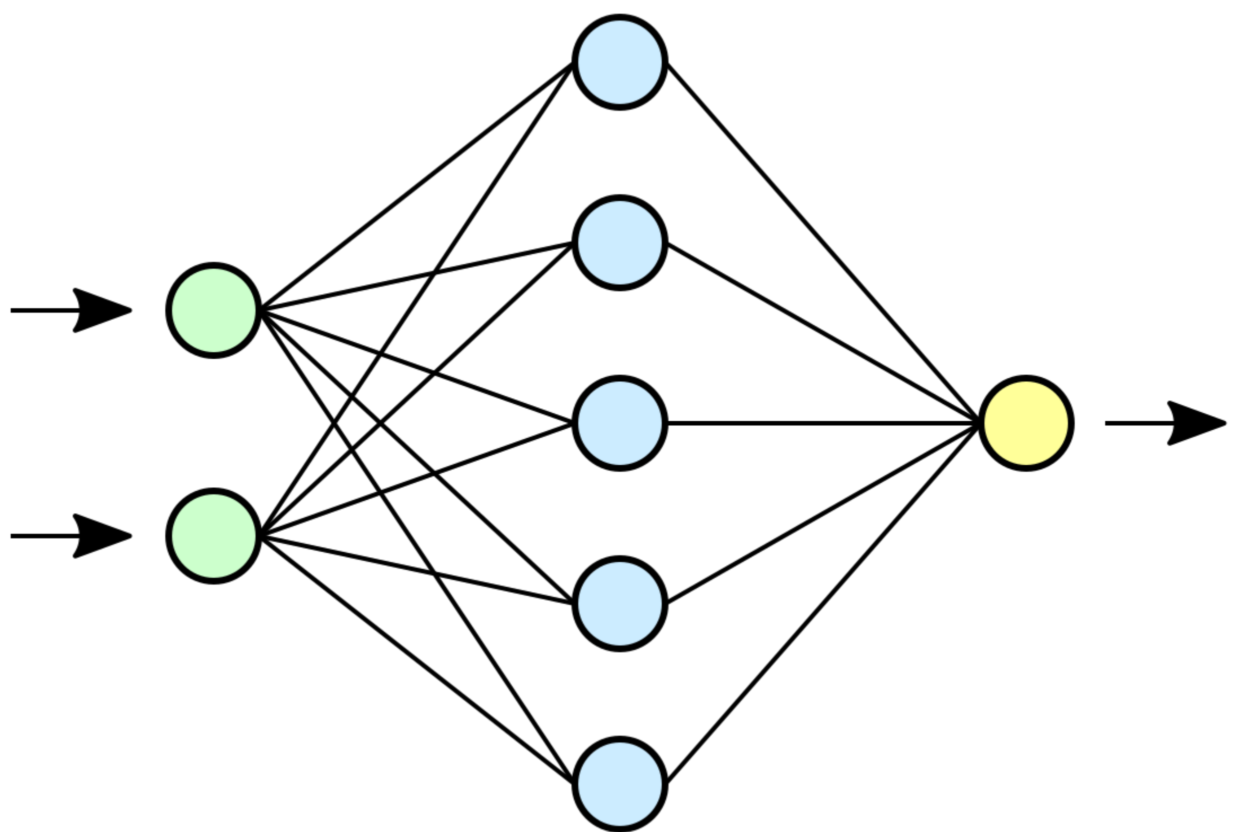
- Scalar product of a weight vector with the feature vector, which represents the state:

$$\hat{V}^{\pi}(s; \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s).$$

- Linear function approximations can work well but need good features (which requires feature engineering).

Neural Networks

- Neural network (*well, you know them*):



- In this lecture we will think of neural networks simply as blackboxes $g(\mathbf{x}; \mathbf{w})$ which we can evaluate and for which we can compute the gradients $\nabla_{\mathbf{w}} g(\mathbf{x}; \mathbf{w})$ efficiently *(we will usually omit the subscript* $\mathbf{w}$ *from* $\nabla_{\mathbf{w}}$ *when it is clear from the context)*.

- In particular, the approximation will have the form $V^{\pi}(s; \mathbf{w}) = g(\mathbf{x}(s); \mathbf{w})$, where g is some neural network…

Part 3: Some Background

Gradient Descent (1/3)

- A method for finding a (local) optimum of a function.
- In our setting, we want to find $\mathbf{w} \in \mathbb{R}^d$ that is a local minimum of a function $J(\mathbf{w})$.
- We do that using *gradient descent.*

Gradient Descent (2/3)

Gradient:

$$\nabla J(\mathbf{w}) = \left(\frac{\partial J}{\partial w_1}(\mathbf{w}), \frac{\partial J}{\partial w_2}(\mathbf{w}), \ldots, \frac{\partial J}{\partial w_d}(\mathbf{w}) \right)$$

Example:

$$J(\mathbf{w}) = w_1 \cdot w_2 + w_1, \mathbf{w} \in \mathbb{R}^2.$$

Then

$$\nabla J(\mathbf{w}) = (w_2 + 1, w_1).$$

Gradient Descent (3/3)

Gradient descent update rule:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \cdot \nabla J(\mathbf{w}_n)$$

(gradient descent algorithm iterates this rule).

Stochastic Gradient Descent

- We want to optimize a function $J(\mathbf{w})$ of the form $J(\mathbf{w}) = \mathbb{E}[g(X; \mathbf{w})]$ where X is a random variable.
- We assume that we can sample from the distribution w.r.t. which the expectation is taken.
- Stochastic gradient descent uses samples to approximate the gradient of $J(\mathbf{w})$ using just one sample (*SGD can also use a mini-batch of multiple samples but we will not consider it now for simplicity*) and estimates the gradient of J as:

$$\nabla J(\mathbf{w}) \approx \nabla g(X; \mathbf{w})$$

(instead of $\nabla \mathbb{E}[g(X; \mathbf{w})]$).

- *Assuming that we can exchange the order of expectation and taking gradients (which we can when g is well-behaved), the expected SGD step is the same as the full gradient of J.*

A Useful Property of Mean Squared Loss

Let $Y_1, Y_2, \ldots, Y_n$ be independent random variables following some distribution with expected value $\mu = \mathbb{E}[Y_i], \forall i$.

What is the value y (~prediction) that minimizes the mean squared error $\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - y\right)^2$?

It is the sample average $y = \frac{1}{n}\sum_{i=1}^{n} Y_i$, which, for $n \to \infty$, converges to the mean μ.

Consequence: Learning a predictor under mean squared loss leads to learning a predictor for conditional expectation (*we will explain later what it means for RL*).

Warm-Up: Learning to "Compress" $V^{\pi}(s)$, (1/3)

- Suppose that we know $V^{\pi}(s)$ and can query it but yet want to learn an approximation of it… using a parametric function $\hat{V}^{\pi}(s; \mathbf{w})$…

- We will use mean-squared error to measure how good the approximation is, i.e.:

$$J(\mathbf{w}) = \mathbb{E}_{\pi}\left[\left(V^{\pi}(X) - \hat{V}^{\pi}(X; \mathbf{w})\right)^{2}\right].$$

- How could we train the approximation using SGD?

Warm-Up: Learning to "Compress" $V^{\pi}(s)$, (2/3)

While (some stopping condition):

Sample a state s and compute the gradient of

$$\hat{J}_s(\mathbf{w}) = (V^{\pi}(s) - V^{\pi}(s; \mathbf{w}))^2,$$

which is:

$$\nabla \hat{J}_s(\mathbf{w}) = -2(V^{\pi}(s) - V^{\pi}(s; \mathbf{w})) \cdot \nabla V^{\pi}(s; \mathbf{w}) = 2(V^{\pi}(s; \mathbf{w}) - V^{\pi}(s)) \cdot \nabla V^{\pi}(s; \mathbf{w})$$

Take the gradient step:

$$\mathbf{w} := \mathbf{w} - \alpha \cdot 2(V^{\pi}(s; \mathbf{w}) - V^{\pi}(s)) \cdot \nabla V^{\pi}(s; \mathbf{w})$$

Warm-Up: Learning to "Compress" $V^{\pi}(s)$, (3/3)

- But in reality **we will not have access** to $V^{\pi}(s)$!
- So we cannot compute the gradient step:
  $\mathbf{w} := \mathbf{w} - \alpha \cdot 2(V^{\pi}(s;\mathbf{w}) - V^{\pi}(s)) \cdot \nabla V^{\pi}(s;\mathbf{w})\ldots$
- We will therefore need to combine SGD with what we saw in the previous lectures…

Part 4: Policy Evaluation with Function Approximation

Monte-Carlo Value Function Approximation

Basic Idea (*not yet complete… wait for the next slide*)**:** We can frame the value function approximation problem as a supervised learning problem under MSE loss:

Sample an episode under policy π: $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T$

Training examples: $[s_1, g_1], [s_2, g_2], \ldots, [s_{T-1}, g_{T-1}]$, where g_i denotes the return from the episode from time i.

First-visit or every-visit? See next slide.

First/Every-Visit Monte-Carlo Value Function Approximation

Initialize: $\mathbf{w} =$ some initialization....

For $i = 1,\ldots,N$**:**

Sample episode $e_i := s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, \ldots, s_{i,T_i}$.

For each time step $1 \le t \le T_i$:

If t is the first occurrence of state s in the episode e_i /* This is for first-visit MC */

s is the state visited at time t in the episode e_i

$g_{i,t} := r_{i,t} + \gamma \cdot r_{i,t+1} + \gamma^2 \cdot r_{i,t+2} + \ldots + \gamma^{T_i - t} \cdot r_{i,T_i}$

/* SGD step */

$\mathbf{w} := \mathbf{w} - \alpha \cdot (V^{\pi}(\mathbf{x}(s_t); \mathbf{w}) - g_t) \cdot \nabla V(\mathbf{x}(s_t); \mathbf{w})$

Intuition About Why It Works

- Recall that what we want to estimate is $V^{\pi}(s) = \mathbb{E}[G_t | X_t = s]$.
- When using first-visit MC, each of the training examples $[s_t, g_t]$ is an unbiased (but very noisy!) estimate of $V^{\pi}(s)$. But when we use these examples and try to find a best mean-squared-error fit then we are estimating their expectation which equals $V^{\pi}(s)$. And that is why it works…

Convergence of MC VFA (1/3)

- **Definition (On-Policy Distribution):** Given an MDP and a policy π, we define on-policy distribution P^{π}_{onp} as follows.
  - **In non-episodic settings:** P^{π}_{onp} is the stationary distribution of the MRP that is given by the MDP and the policy (*recall MDP + policy = MRP*).
  - **In episodic settings:** P^{π}_{onp} depends also on the distribution of the initial states P_{init} *(see Sutton's book for details)*.
- In what follows, we denote the on-policy distribution by P^{π}_{onp}.

Convergence of MC VFA (2/3)

- **Definition:** Mean squared error of value function approximation is defined as

$$MSVE_{\pi}(\mathbf{w}) = \sum_{s \in S} P_{onp}^{\pi}(s) \cdot \left(V^{\pi}(s) - \hat{V}^{\pi}(s; \mathbf{w}) \right)^2,$$

which is the same as

$$MSVE_{\pi}(\mathbf{w}) = \mathbb{E}_{X \sim P_{onp}^{\pi}} \left[\left(V^{\pi}(s) - \hat{V}^{\pi}(s; \mathbf{w}) \right)^2 \right].$$

Convergence of MC VFA (3/3)

- **Theorem:** Assume that $\hat{V}^{\pi}(s; \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ (i.e. **we are assuming linear function approximation**). Then MC VFA converges to weights that are optimal in the sense that they minimize $MSVE_{\pi}(\mathbf{w})$.

- **Caution:** This theorem holds for **linear** function approximation, not for general functions! We do not have such guarantees for, e.g., arbitrary neural networks.

Temporal Difference VFA (1/5)

- For temporal difference learning in the tabular setting, we had the following update rule:

$$V^{\pi}(s_t) := V^{\pi}(s_t) + \alpha \cdot \left(\underbrace{r_t + \gamma \cdot V^{\pi}(s_{t+1})}_{\text{TD-target}} - V^{\pi}(s_t) \right).$$

- Now, we will want to have a similar update rule but for the case where $V^{\pi}(s)$ is only approximated by $V^{\pi}(s; \mathbf{w})$.

Temporal Difference VFA (2/5)

Recall the Bellman equation (*for simplicity, we are showing it for deterministic policy*):

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \cdot \sum_{s' \in S} P(s' \,|\, s, \pi(s)) \cdot V(s')$$

which is the same as:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \cdot \mathbb{E}\left[V^{\pi}(X_{t+1}) \,|\, X_t = s\right].$$

We can turn the system of equations above into the following minimization problem:

$$\min_{\mathbf{V}^{\pi}} \sum_{s \in S} P_{onp}(s) \cdot \mathbb{E}\left[\left(R(s, \pi(s)) + \gamma \cdot V^{\pi}(X_{t+1}) - V^{\pi}(s)\right)^2 \middle| \, X_t = s\right].$$

Temporal Difference VFA (3/5)

Next we replace $V^\pi(s)$ by its approximation $\hat{V}^\pi(s;\mathbf{w})$, yielding:

$$\min_{\mathbf{V}^\pi} \sum_{s\in S} P_{onp}(s) \cdot \mathbb{E}\left[\left(R(s,\pi(s)) + \gamma \cdot \hat{V}^\pi(X_{t+1};\mathbf{w}) - \hat{V}^\pi(s;\mathbf{w})\right)^2 \middle| X_t = s\right].$$

Now, instead of the on policy distribution, we will just take the states as they come in an episode and instead of the expectation we will use the tuple (s_t, a_t, r_t, s_{t+1}) which we get in the current episode *(as is common in TD-learning)*. That will lead us to the minimization problem:

$$\min_{\mathbf{w}} \left(R(s_t, r_t) + \gamma \cdot \hat{V}^\pi(s_{t+1};\mathbf{w}) - \hat{V}^\pi(s_t;\mathbf{w})\right)^2$$

Temporal Difference VFA (4/5)

We need to solve:

$$\min_{\mathbf{w}} \left(R(s_t, r_t) + \gamma \cdot \hat{V}^{\pi}(s_{t+1}; \mathbf{w}) - \hat{V}^{\pi}(s_t; \mathbf{w}) \right)^2$$. Denoting

$$J(\mathbf{w}) = \left(R(s_t, r_t) + \gamma \cdot \hat{V}^{\pi}(s_{t+1}; \mathbf{w}) - \hat{V}^{\pi}(s_t; \mathbf{w}) \right)^2$$

we have

$$\nabla J(\mathbf{w}) = 2 \left(R(s_t, r_t) + \gamma \cdot \hat{V}^{\pi}(s_{t+1}; \mathbf{w}) - \hat{V}^{\pi}(s_t; \mathbf{w}) \right) \cdot (\gamma \cdot \nabla \hat{V}^{\pi}(s_{t+1}; \mathbf{w}) - \nabla \hat{V}^{\pi}(s_t; \mathbf{w}))$$

But this is not what TD with function approximation does! TD VFA is a so-called semigradient method. It does not consider the contribution of $\nabla \hat{V}^{\pi}(s_{t+1}; \mathbf{w})$ and considers it fixed.

Temporal Difference VFA (4/5)

We need to solve:

$$\min_{\mathbf{w}} \left(R(s_t, r_t) + \gamma \cdot \hat{V}^\pi(s_{t+1}; \mathbf{w}) - \hat{V}^\pi(s_t; \mathbf{w}) \right)^2. \text{ Denoting}$$

$$J(\mathbf{w}) = \left(R(s_t, r_t) + \gamma \cdot \hat{V}^\pi(s_{t+1}; \mathbf{w}) - \hat{V}^\pi(s_t; \mathbf{w}) \right)^2$$

we have

$$\nabla J(\mathbf{w}) = 2 \left(R(s_t, r_t) + \gamma \cdot \hat{V}^\pi(s_{t+1}; \mathbf{w}) - \hat{V}^\pi(s_t; \mathbf{w}) \right) \cdot (\gamma \cdot \nabla \hat{V}^\pi(s_{t+1}; \mathbf{w}) - \nabla \hat{V}^\pi(s_t; \mathbf{w}))$$

But this is not what TD with function approximation does! TD VFA is a so-called semigradient method. It does not consider the contribution of $\nabla \hat{V}^\pi(s_{t+1}; \mathbf{w})$ and considers it fixed.

Temporal Difference VFA (5/5)

The TD update rule for value function approximation is:

$$\mathbf{w} := \mathbf{w} + \alpha \left(r_t + \gamma \cdot \hat{V}^{\pi}(s_{t+1}; \mathbf{w}) - \hat{V}^{\pi}(s_t; \mathbf{w}) \right) \cdot \nabla \hat{V}^{\pi}(s_t; \mathbf{w})$$

Convergence of TD VFA with Linear Functions

- As for MC VFA, we will use the on-policy distribution P^{π}_{onp} and define the mean squared error w.r.t. it, that is…

$$MSVE_{\pi}(\mathbf{w}) = \sum_{s \in S} P^{\pi}_{onp}(s) \cdot \left(V^{\pi}(s) - \hat{V}^{\pi}(s; \mathbf{w}) \right)^2$$

- **Theorem:** Let $\mathbf{w}_{TD}$ be the weight vector to which TD VFA converges. Then it holds:

$$MSVE_{\pi}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \cdot \min_{\mathbf{w}} MSVE_{\pi}(\mathbf{w}).$$

- Recall that for MC VFA with linear functions we had convergence of mean squared error to $\min_{\mathbf{w}} MSVE_{\pi}(\mathbf{w})$.

Part 5: Control with Function Approximation

Basic Idea

- *Same ideas, just plugging them into what we were doing in the last lecture, but there are caveats…*
- Instead of approximating V^{π}, we need to approximate $Q^{\pi}(s, a)$.
- The algorithms are similar to those we saw last week (MC, SARSA, Q-Learning). **Important: the idea of using ε-greedy policies.** The motivation is the same but we use $Q^{\pi}(s, a; \mathbf{w})$.

Basic Idea

- Recall the structure of RL algorithms from the last lecture:
  - Maintain an estimate of Q-function.
  - Compute an ε-greedy π policy w.r.t. the Q-function estimate.
  - Use the policy π, either for an episode (MC methods) or for a step (SARSA and Q-learning).
  - Update the Q-function estimate (here we rely on the ideas from value function approximation).

Representing State-Action Pairs

- For control RL problems, we need to encode both states and actions together.
- The feature vector of a state-action pair (s, a) will be denoted as $\mathbf{x}(s, a)$ and we can think of it as a function mapping state-action pairs to some vector space, e.g. $\mathbb{R}^d$, i.e. $\mathbf{x}(s, a) = (x_1(s, a), x_2(s, a), \ldots, x_d(s, a))^T$.

Approximation of Q-Function

- **Linear function approximation:** Scalar product of a weight vector with the feature vector, which represents the state-action pair:

$$\hat{Q}^{\pi}(s, a; \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s, a).$$

- **Neural network function approximation:**

$$\hat{Q}^{\pi}(s, a; \mathbf{w}) = g(\mathbf{x}(s, a); \mathbf{w})$$

where g is a function represented as a neural network.

Weight Updates

- MC:

$$\mathbf{w} := \mathbf{w} + \alpha \cdot \left(g_t - \hat{Q}(s_t, a_t; \mathbf{w})\right) \cdot \nabla \hat{Q}(s_t, a_t; \mathbf{w})$$

- SARSA:

$$\mathbf{w} := \mathbf{w} + \alpha \cdot \left(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})\right) \cdot \nabla \hat{Q}(s_t, a_t; \mathbf{w})$$

- Q-Learning:

$$\mathbf{w} := \mathbf{w} + \alpha \cdot \left(r + \gamma \max_{a \in A} \hat{Q}(s_{t+1}, a; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})\right) \cdot \nabla \hat{Q}(s_t, a_t; \mathbf{w})$$

Deep Q-Learning

1: Input C, α, $D = \{\}$, Initialize $\mathbf{w}$, $\mathbf{w}^- = \mathbf{w}$, $t = 0$
2: Get initial state s_0
3: **loop**
4: Sample action a_t given ϵ-greedy policy for current $\hat{Q}(s_t, a; \mathbf{w})$
5: Observe reward r_t and next state s_{t+1}
6: Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
7: Sample random minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from D
8: **for** j in minibatch **do**
9: **if** episode terminated at step $i + 1$ **then**
10: $y_i = r_i$
11: **else**
12: $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$
13: **end if**
14: Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$ for parameters $\mathbf{w}$: $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))\nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$
15: **end for**
16: $t = t + 1$
17: **if** mod(t,C) == 0 **then**
18: $\mathbf{w}^- \leftarrow \mathbf{w}$
19: **end if**
20: **end loop**

With Neural Networks…

Convergence is not guaranteed.

Two of the reasons why Q-learning with VFA may diverge: correlations between samples and non-stationary targets.

Partial remedies: experience replay and fixed Q-targets.

There are many variations proposed in the literature with many tricks to improve deep Q-learning and many are still appearing…

Convergence of MC, SARSA and Q-Learning

	Tabular	Linear	NN
MC	✅	Chattering (may oscilate at the end but not diverge)	❌
SARSA	✅	Chattering (may oscilate at the end but not diverge)	❌
Q-Learning	✅	❌	❌

Convergence

On the Chattering of SARSA with Linear Function Approximation

Shangtong Zhang SHANGTONG.ZHANG@CS.OX.AC.UK
University of Oxford
Wolfson Building, Parks Rd, Oxford, OX1 3QD, UK
Remi Tachet des Combes REMI.TACHET@MICROSOFT.COM
Microsoft Research Montreal
6795 Rue Marconi, Suite 400, Montreal, Quebec, H2S 3J9, Canada
Romain Laroche ROMAIN.LAROCHE@MICROSOFT.COM
Microsoft Research Montreal
6795 Rue Marconi, Suite 400, Montreal, Quebec, H2S 3J9, Canada

06828v1 [cs.LG] 14 Feb 2022

Abstract

SARSA, a classical on-policy control algorithm for reinforcement learning, is known to chatter when combined with linear function approximation: SARSA does not diverge but oscillates in a bounded region. However, little is know about how fast SARSA converges to that region and how large the region is. In this paper, we make progress towards solving this open problem by showing the convergence rate of projected SARSA to a bounded region. Importantly, the region is much smaller than the ball used for projection provided that the the magnitude of the reward is not too large. Our analysis applies to expected SARSA as well as SARSA(λ). Existing works regarding the convergence of linear SARSA to a fixed point all require the Lipschitz constant of SARSA's policy improvement operator to be sufficiently small; our analysis instead applies to arbitrary Lipschitz constants and thus characterizes the behavior of linear SARSA for a new regime.

Part 6: Double Q-Learning

Double Q-Learning: Motivation

- The following step causes a maximization bias:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

because, in general:

$\mathbb{E}[\max\{X_1, X_2, \ldots, X_k\}] \neq \max\{\mathbb{E}[X_1], \mathbb{E}[X_2], \ldots, \mathbb{E}[X_k]\}$, and in fact:
$\mathbb{E}[\max\{X_1, X_2, \ldots, X_k\}] \geq \max\{\mathbb{E}[X_1], \mathbb{E}[X_2], \ldots, \mathbb{E}[X_k]\}$.

- So even if the estimates of $Q(s, a)$ were unbiased, $\max_{a \in A} Q(s_{t+1}, a)$ would not have to be unbiased.

Double Q-Learning: Key Idea

- Maintain two different estimates of the state-action value function Q.
- One will be used to select "argmax" element, the other to give the value of the argmax element. This will reduce the maximization bias because…

  …normally we would use $\max_{a \in A} Q(s_{t+1}, a)$… and that could lead to big overestimation… but if we use $Q^B(s_{t+1}, \arg\max_a Q^A(s_{t+1}, a))$ we will reduce this,

Double Q-Learning (Tabular, Not the DL Version)

Double Q-Learning

1: Initialize $Q_1(s,a)$ and $Q_2(s,a)$,$\forall s \in S, a \in A$ $t=0$, initial state $s_t = s_0$
2: **loop**
3: Select a_t using ϵ-greedy $\pi(s) = \arg\max_a Q_1(s_t, a) + Q_2(s_t, a)$
4: Observe (r_t, s_{t+1})
5: **if** (with 0.5 probability) **then**
6: $Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \gamma \underline{Q_2}(s_{t+1}, \arg\max_a \underline{Q_1}(s_{t+1}, a)) - Q_1(s_t, a_t))$
7: **else**
8: $Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + \gamma \underline{Q_1}(s_{t+1}, \arg\max_a \underline{Q_2}(s_{t+1}, a)) - Q_2(s_t, a_t))$
9: **end if**
10: $t = t + 1$
11: **end loop**

Compared to Q-learning, how does this change the: memory requirements, computation requirements per step, amount of data required?

The Deep-Learning Version

- Double DQN (Deep Reinforcement Learning with Double Q-Learning, Van Hasselt et al, AAAI 2016)
- Extend double Q learning to DQN
- Current Q-network $\boldsymbol{w}$ is used to select actions
- Older Q-network $\boldsymbol{w}^-$ is used to evaluate actions

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \overbrace{\hat{Q}(\underbrace{\arg\max_{a'} \hat{Q}(s', a'; \boldsymbol{w})}_{\text{Action selection: } \boldsymbol{w}}; \boldsymbol{w}^-)}^{\text{Action evaluation: } \boldsymbol{w}^-} - \hat{Q}(s, a; \boldsymbol{w}))$$

- How is this different from fixed target network update used in DQN?

Compare It With Deep Q-Learning

1: Input C, α, $D = \{\}$, Initialize $\mathbf{w}$, $\mathbf{w}^- = \mathbf{w}$, $t = 0$
2: Get initial state s_0
3: **loop**
4: Sample action a_t given ϵ-greedy policy for current $\hat{Q}(s_t, a; \mathbf{w})$
5: Observe reward r_t and next state s_{t+1}
6: Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
7: Sample random minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from D
8: **for** j in minibatch **do**
9: **if** episode terminated at step $i + 1$ **then**
10: $y_i = r_i$
11: **else**
12: $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$
13: **end if**
14: Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$ for parameters $\mathbf{w}$: $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))\nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$
15: **end for**
16: $t = t + 1$
17: **if** mod(t,C) == 0 **then**
18: $\mathbf{w}^- \leftarrow \mathbf{w}$
19: **end if**
20: **end loop**

Slide taken from the Reinforcement Learning course by Prof. Emma Brunskill

Policy Gradient

A Simplifying Assumption

In this part, we are going to assume that the dscount factor $\gamma = 1$ (the methods work also for general discount factors, but this will make everything bit less complex).

The First Idea

Before:

We were approximating

$$Q^{\pi}(s, a) \approx Q(s, a; w)$$

Next:

We will directly paramaetrize the policy using a neural network

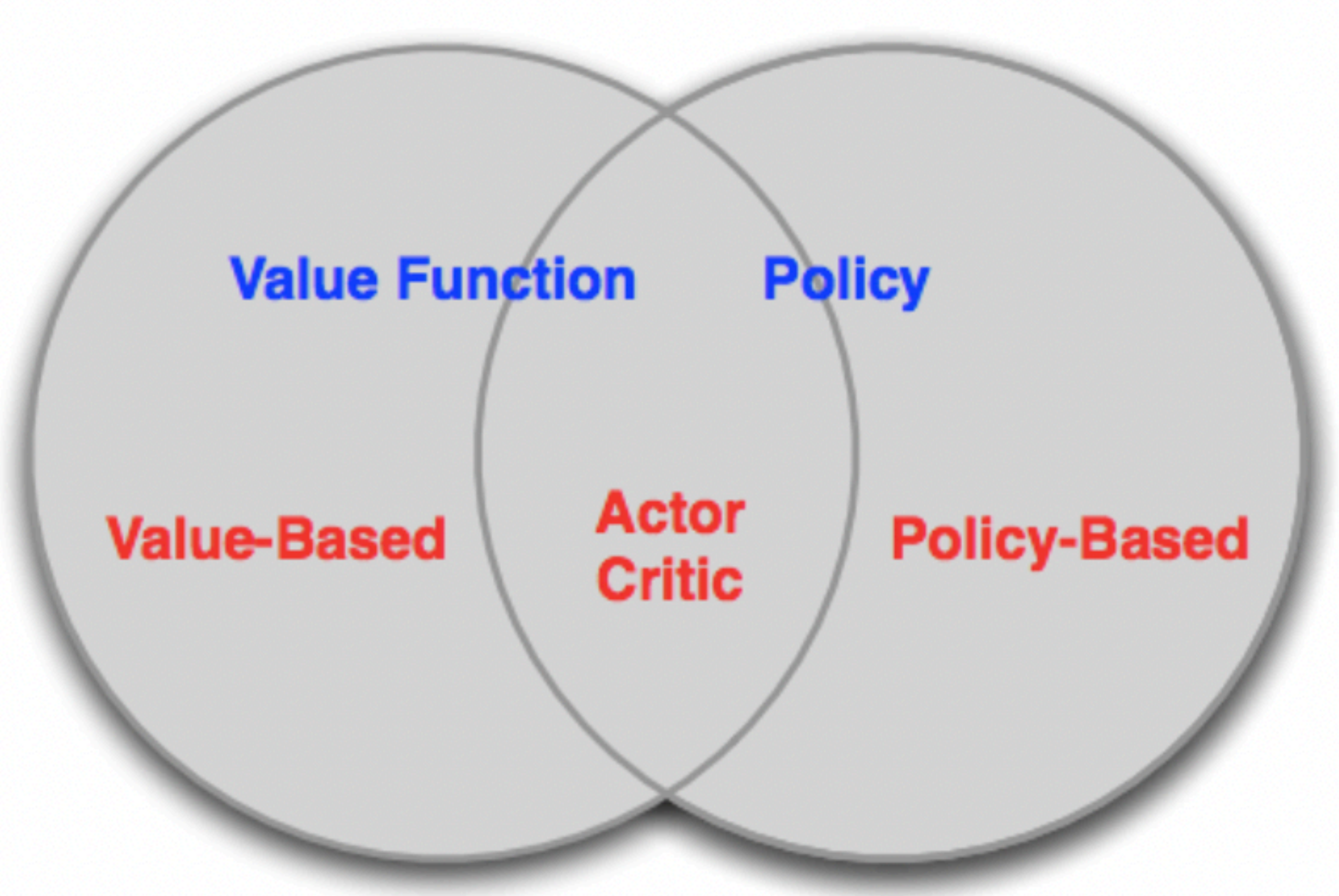
$$\pi(a \mid s; w) = P(a \mid s; w)$$

And then there are also combinations...

Value Based
learned Value Function and implicit policy (e.g. ε-greedy)

Policy Based
No Value Function
Learned Policy

Actor-Critic
Learned Value Function Learned Policy



Note: Slide courtesy professor Emma Brunskill

Gradient-Based Policy Optimization

Goal: Find a policy

$$\pi(a \mid s; w)$$

(represented implicitly through the neural net's weights) that **maximizes**

$$V(s_0; w)$$

for all s_0 or directly one that maximizes the expectation of it over the selection of the initial states, i.e. we are looking for:

$$w = \arg\max_w \mathbb{E}[V(X; w)] = \arg\max_w \mathbb{E}[G_t; w],$$

(but even for that it would be enough to optimize separately for each s_0).

Gradient of the Value Function

The policy-gradient methods will follow the standard gradient ascent idea… since we want to optimize $V(s_0; w)$…

$$\Delta w = \alpha \nabla_w V(s_0; w)$$

Here,

$$\nabla_w V(s_0; w) = \begin{pmatrix} \frac{\partial V(s_0; w)}{\partial w_1} \\ \cdots \\ \frac{\partial V(s_0; w)}{\partial w_m} \end{pmatrix}$$

is the **policy gradient**.

Assumptions

1. We are going to assume that $\pi(a \mid s; w)$ is differentiable in the parameters w whenever the policy $\pi(a \mid s; w)$ is non-zero.

2. We are going to assume that we can compute the gradient $\nabla_w \pi(a \mid s; w)$ (i.e., it is represented in a convenient form).

Let's re-express the value function

Let us use the following notation for trajectories $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, a_{T_1}, s_T)$

$$\mathbb{E}[V(X_0; w)] = \mathbb{E}\left[\sum_{t=0}^{T} R(s_t, a_t); w\right] = \sum_{\tau} P(\tau; w) R(\tau)$$

where τ denotes a trajectory and the sum $\sum_{\tau}$ ranges over all trajectories and

where we defined

$$R(\tau) = \sum_{(s_t, r_t) \in \tau} r_t$$

Our goal is still to maximize…

…the value function $\mathbb{E}[V(X_0; w)]$ which means we are looking for the weights

$$w = \arg\max_{w} \mathbb{E}[V(X_0; w)] = \sum_{\tau} P(\tau; w) R(\tau)$$

Our goal is still to maximize…

…the value function $V(s_0; w)$ which means we are looking for the weights

$$w = \arg\max_w \mathbb{E}[V(X_0; w)] = \sum_\tau P(\tau; w)R(\tau)$$

So we compute the gradient

$$\nabla_w \mathbb{E}[V(X_0; w)] = \nabla_w \sum_\tau P(\tau; w)R(\tau)$$

Computing the gradient…

$$\nabla_w \mathbb{E}[V(X_0; w)] = \nabla_w \sum_\tau P(\tau; w) R(\tau) = \sum_\tau \nabla_w P(\tau; w) R(\tau)$$

Computing the gradient…

$$\nabla_w \mathbb{E}[V(X_0; w)] = \nabla_w \sum_{\tau} P(\tau; w) R(\tau) = \sum_{\tau} \nabla_w P(\tau; w) R(\tau)$$

$$= \sum_{\tau} \frac{P(\tau; w)}{P(\tau; w)} \nabla_w P(\tau; w) R(\tau) = \sum_{\tau} P(\tau; w) R(\tau) \frac{\nabla_w P(\tau; w)}{P(\tau; w)}$$

Computing the gradient…

$$\nabla_w \mathbb{E}[V(X_0; w)] = \nabla_w \sum_\tau P(\tau; w) R(\tau) = \sum_\tau \nabla_w P(\tau; w) R(\tau)$$

$$= \sum_\tau \frac{P(\tau; w)}{P(\tau; w)} \nabla_w P(\tau; w) R(\tau) = \sum_\tau P(\tau; w) R(\tau) \frac{\nabla_w P(\tau; w)}{P(\tau; w)}$$

$$= \sum_\tau P(\tau; w) R(\tau) \nabla_w \log P(\tau; w)$$

We can try to approximate it…

… as in other RL approaches, we approximate it from multiple episodes.

$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \nabla_w \log P(\tau_i; w)$$

We can try to approximate it…

… as in other RL approaches, we approximate it from multiple episodes.

$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \nabla_w \log P(\tau_i; w)$$

But but… we do not have $P(\tau_i; w)$!

Computing $\nabla_w \log P(\tau_i; w)$

$$\nabla_w \log P(\tau^{(i)}; w) = \nabla_w \log \left[\underbrace{P(s_0)}_{\text{Initial state distrib.}} \prod_{t=0}^{T-1} \underbrace{\pi(a_t \mid s_t; w)}_{\text{policy}} \underbrace{P(s_{t+1} \mid s_t, a_t)}_{\text{MDP model}} \right]$$

$$= \nabla_w \left[\log P(s_0) + \sum_{t=0}^{T-1} \log \pi(a_t \mid s_t; w) + \log P(s_{t+1} \mid s_t, a_t) \right]$$

$$= \sum_{t=0}^{T-1} \underbrace{\nabla_w \log \pi(a_t \mid s_t; w)}_{\text{no MDP transitions required!}}$$

Terminology

$\nabla_w \log \pi(a_t \mid s_t; w)$ is called score function

Example: Softmax Policy

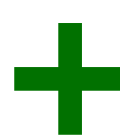
Suppose we represent the policy $\pi(a \mid s; w)$ as a softmax function (think of it as a neural network with softmax output).

That is:

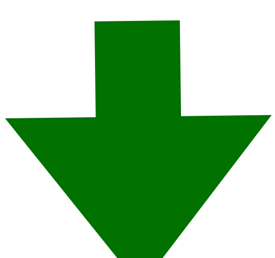
$$\pi(a \mid s; w) = \frac{\exp(\phi(s, a; w))}{\sum_{a' \in A} \exp(\phi(a', s; w))}$$

Combining the pieces

$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \nabla_w \log P(\tau_i; w)$$



$$\nabla_w \log P(\tau^{(i)}; w) = \sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w)$$



$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w)$$

Unfortunately… not the best estimator

Unfortunately, the gradient estimator

$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w)$$

is usually too noisy (i.e., has high variance).

Next, we will take a look at ways to make it less noise and practical.

A Better Estimator: Using Temporal Structure

Idea: Express the gradient for a single reward at time t' instead of expressing it for the value function (which aggregates all the rewards).

This leads to:

$$\nabla_w \mathbb{E}[R_{t'}] = \mathbb{E}\left[R_{t'} \sum_{t=0}^{t'} \nabla_w \log \pi(a_t \mid s_t; w)\right]$$

and after summing over t (and doing simple algebraic manipulations):

$$\nabla_w \mathbb{E}[V(X_0; w)] = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) \sum_{t'=t}^{T-1} R_{t'}\right]$$

One more rewrite…

Recall that (when $\gamma = 1$):

$$G_t = \sum_{t'=t}^{T-1} R_t$$

Therefore we can also write the gradients from the previous slide as:

$$\nabla_w \mathbb{E}[V(X_0; w)] = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) G_t\right]$$

And now the approximation…

$$\nabla_w \mathbb{E}[V(X_0; w)] \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t \mid s_t; w) G_t^{(i)}$$

REINFORCE Algorithm

Initialize policy parameters w

repeat

  Generate an episode using the policy $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T$

  for t = 0, 1, ..., T – 1 do

    Compute return from time t: $G_t = r_t + r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{T-1}$

    Update policy parameters: $w \leftarrow w + \alpha G_t \nabla_w \log \pi(a_t \mid s_t; w)$

  end for

until convergence

return w

Better Policy-Gradient Algorithms

REINFORCE With Baseline

We can reduce the variance of the gradient estimator by using a "baseline", which is just some function of the state variable:

$$\nabla_w \mathbb{E}_\tau[R] = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t \mid s_t; w) \left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t) \right) \right]$$

Why we can do it…

Let's compute the gradient:

$$\nabla_w \mathbb{E}_\tau[R] = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w)\big(G_t - b(s_t)\big)\right], \qquad G_t = \sum_{t'=t}^{T-1} r_{t'}.$$

$$= \underbrace{\mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) G_t\right]}_{\text{standard policy-gradient term}} - \underbrace{\mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) b(s_t)\right]}_{\text{baseline term}}.$$

Why we can do it…

Let's compute the gradient:

$$= \underbrace{\mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) G_t \right]}_{\text{standard policy-gradient term}} - \underbrace{\mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_w \log \pi(a_t \mid s_t; w) b(s_t) \right]}_{\text{baseline term}}.$$

$$\mathbb{E}_w \left[\nabla_w \log \pi(a_t \mid s_t; w)\, b(s_t) \right] = b(s_t) \sum_a \pi(a \mid s_t; w) \nabla_w \log \pi(a \mid s_t; w)$$

$$= b(s_t) \sum_a \nabla_w \pi(a \mid s_t; w) = b(s_t) \nabla_w \sum_a \pi(a \mid s_t; w) = b(s_t) \nabla_w 1 = 0.$$

Why it helps…

…so expectation is the same but variance can be different (ideally smaller)…

$$\mathrm{Var}\left[\nabla_w \log \pi(a_t \mid s_t; w)\big(G_t - b(s_t)\big)\right] \neq \mathrm{Var}\left[\nabla_w \log \pi(a_t \mid s_t; w) G_t\right].$$

Intuition: The baseline should try to remove the part of the return that is due to the state, leaving only how much better or worse the action was than expected.

So where this leads us…

***Intuition**: The baseline should try to remove the part of the return that is due to the state, leaving only how much better or worse the action was than expected.*

A natural choice is then the value function

$$b(s_t) = V^{\pi}(s_t) = \mathbb{E}[G_t \mid s_t] \, .$$

So…

Policy gradient without baseline asks: "Was the return high?"

Policy gradient with baseline asks: "Was the action better than expected?"

$$\underbrace{\nabla_w \log \pi(a_t \mid s_t; w)\, G_t}_{\text{rewards actions from good states}} \quad \Longrightarrow \quad \underbrace{\nabla_w \log \pi(a_t \mid s_t; w)\big(G_t - V^{\pi}(s_t)\big)}_{\text{rewards actions better than the state average}}$$

REINFORCE with Learned Baseline

Repeat:

1. Sample a trajectory using current policy: $a_t \ \pi(\cdot \mid s_t; w)$

2. Compute Monte Carlo returns: $G_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$

3. Use value function as baseline: $b_t = V_\phi(s_t)$

4. Compute centered return: $A_t = G_t - V_\phi(s_t)$

5. Policy update: $w \leftarrow w + \alpha \sum_t \nabla_w \log \pi(a_t \mid s_t; w) A_t$

6. Value-function update: $\phi \leftarrow \phi - \beta \nabla_\phi \sum_t \left(V_\phi(s_t) - G_t \right)^2$

Actor-Critic

Let's just plug the temporal-difference idea back in…

Actor-Critic

Repeat:

1. Actor samples action: $a_t \sim \pi(\,\cdot \mid s_t; w)$

2. Environment returns reward and next state: r_t, s_{t+1}

3. Critic computes TD error: $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$

4. Actor update: $w \leftarrow w + \alpha \nabla_w \log \pi(a_t \mid s_t; w)\delta_t$

5. Critic update: $\phi \leftarrow \phi - \beta \nabla_\phi \left(V_\phi(s_t) - \left(r_t + \gamma V_\phi(s_{t+1}) \right) \right)^2$

6. Continue from next state: $s_t \leftarrow s_{t+1}$