SMU: Lecture 4

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

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

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).

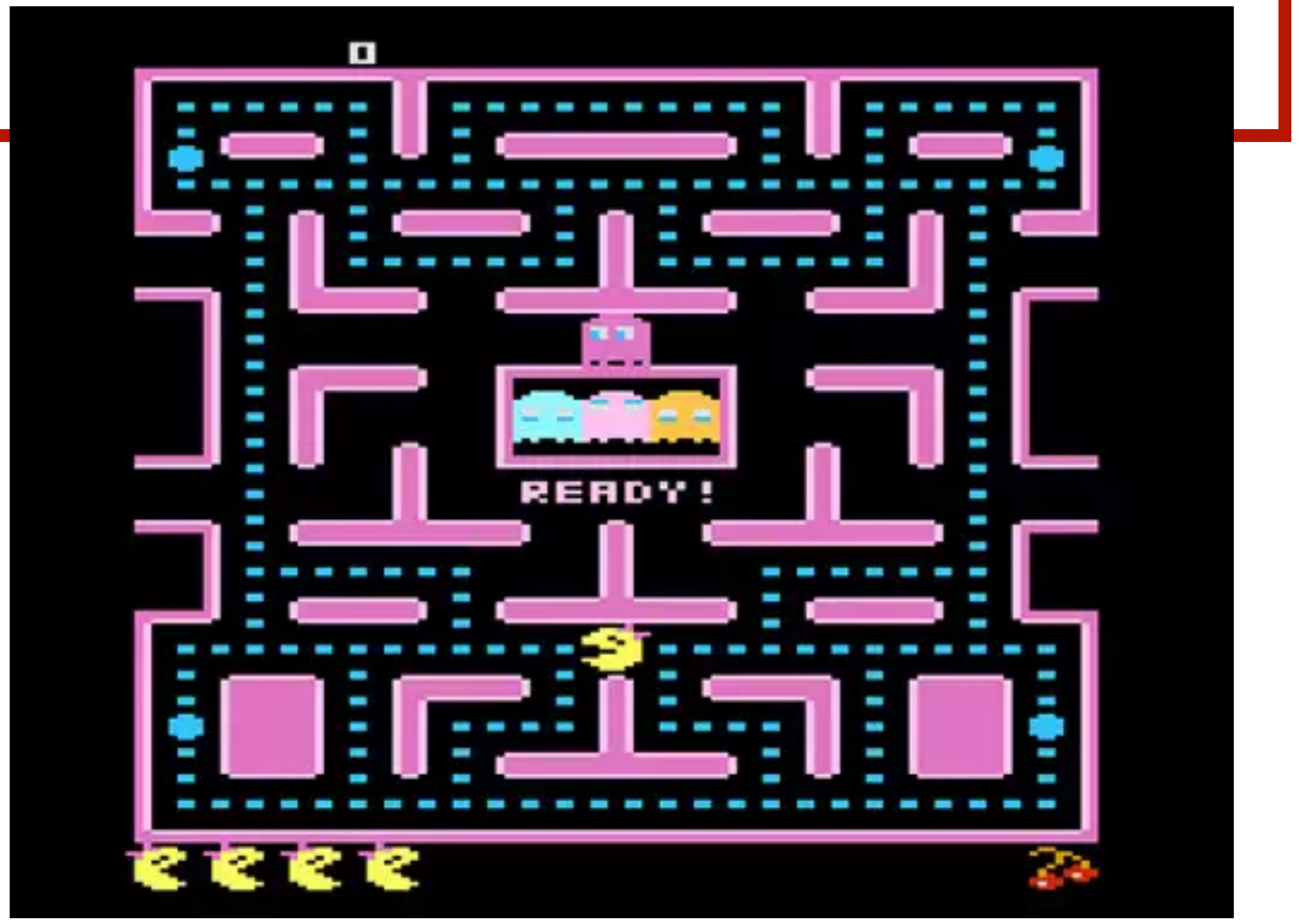
- **What we need is *function approximation.***

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).

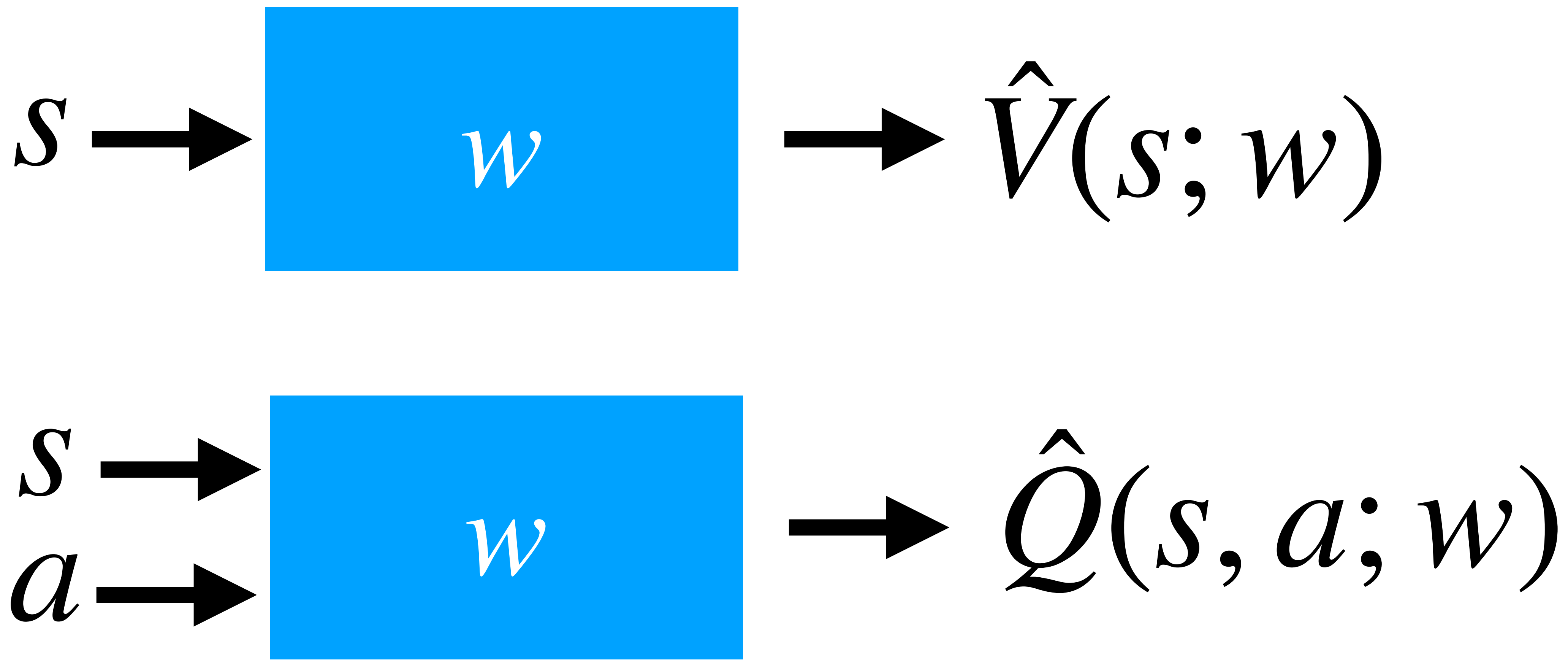
- **What we need is *function approximation.***

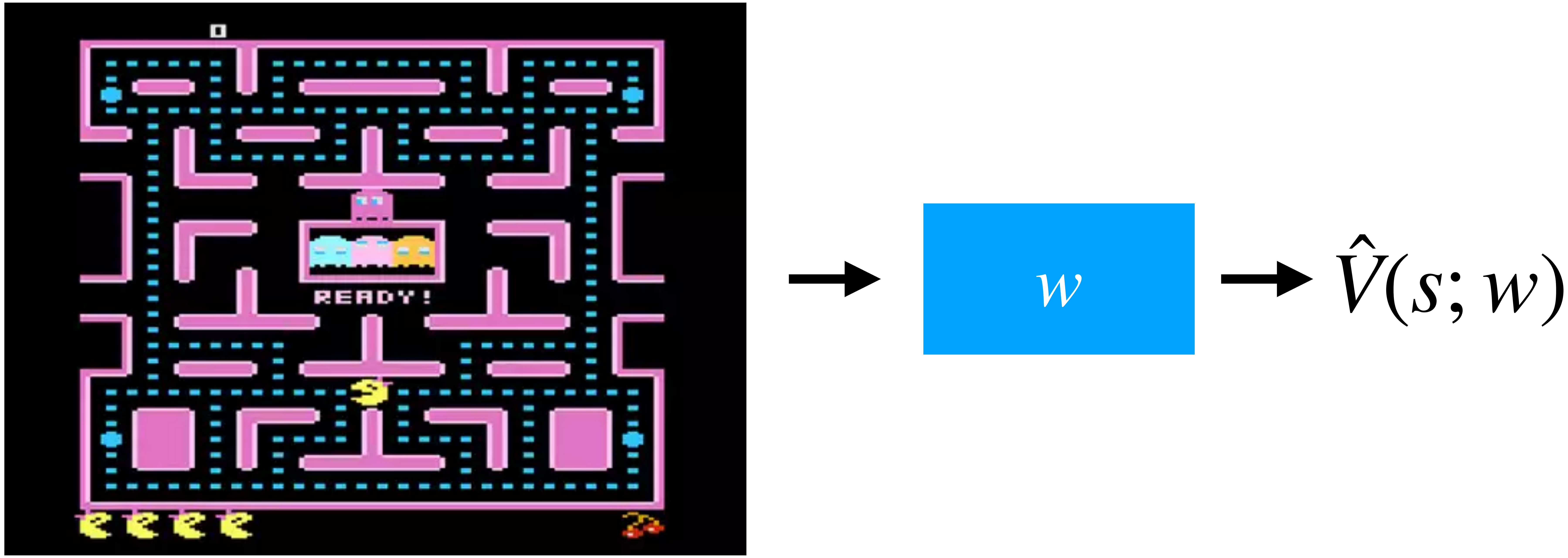


Basic Idea

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

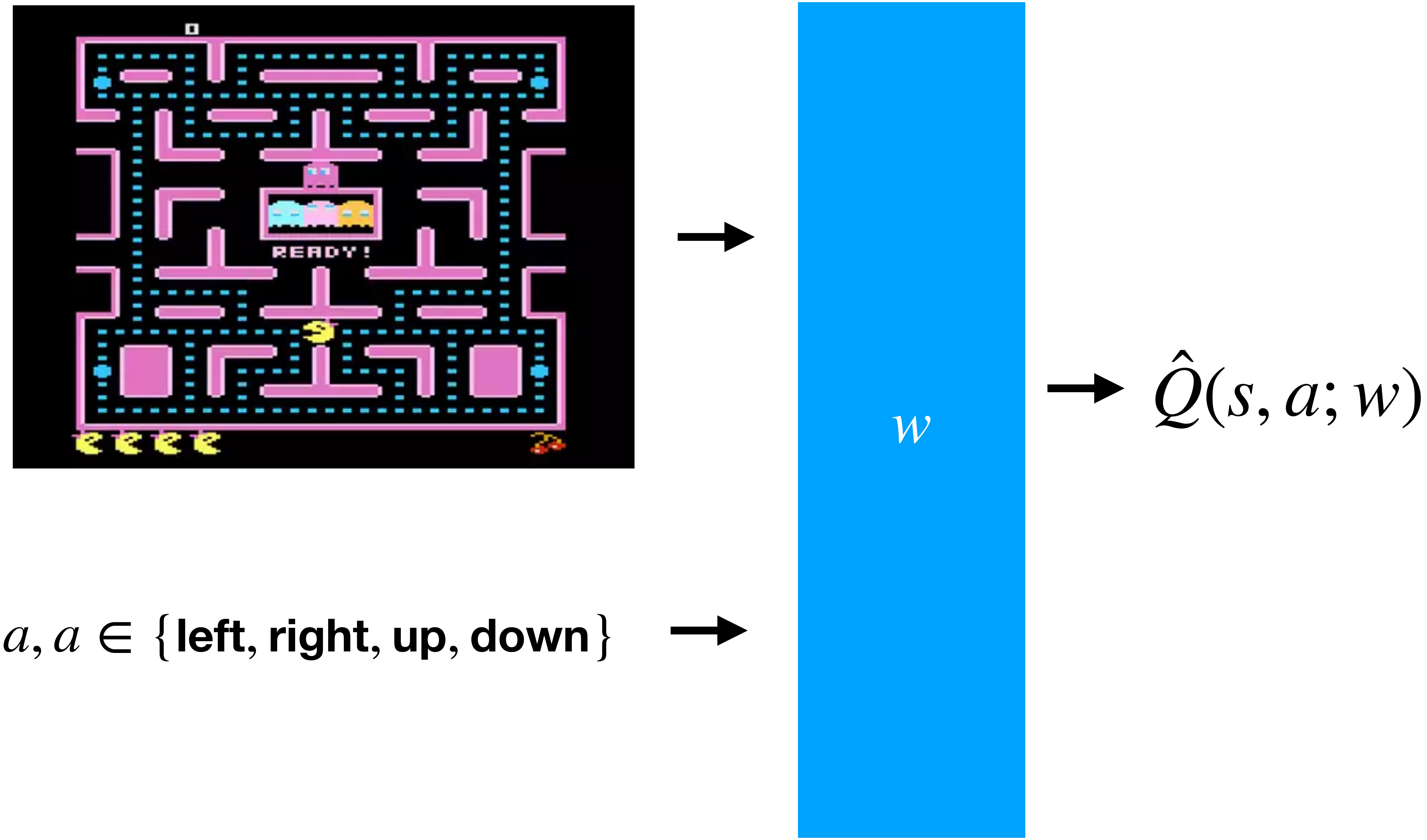
- Represent the state value function $V(s)$ or the state-action value function $Q(s, a)$ approximately using a function from some parametrized family, e.g. as a neural network, linear function, decision tree…





w

$\hat{V}(s; w)$



w

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

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

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

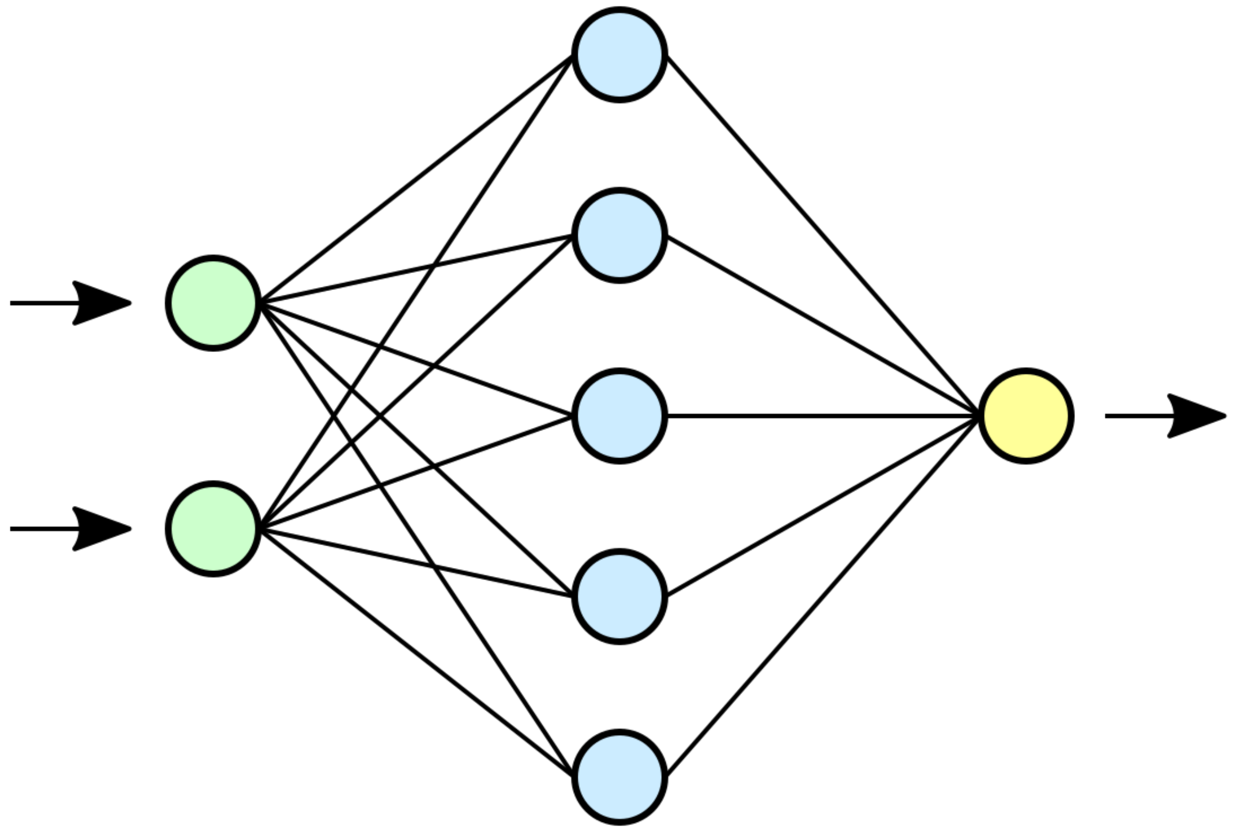
- Scalar product of a weight vector with the feature vector, which represents the state:

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

- 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

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

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:

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

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: $\mathbf{w} := \mathbf{w} - \alpha \cdot 2(V^{\pi}(s; \mathbf{w}) - V^{\pi}(s)) \cdot \nabla V(s; \mathbf{w})\ldots$
- 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 $1 \leq t \leq T_i$:
        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

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

which is the same as

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

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:

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

. 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

```
Input $C$, $\alpha$, $D = \{\}$, Initialize $\boldsymbol{w}$, $\boldsymbol{w}^- = \boldsymbol{w}$, $t = 0$
Get initial state $s_0$
loop
    Sample action $a_t$ given $\epsilon$-greedy policy for current $\hat{Q}(s_t, a; \boldsymbol{w})$
    Observe reward $r_t$ and next state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $D$
    Sample random minibatch of tuples $(s_i, a_i, r_i, s_{i+1})$ from $D$
    for $j$ in minibatch do
        if episode terminated at step $i + 1$ then
            $y_i = r_i$
        else
            $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \boldsymbol{w}^-)$
        end if
        Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}))^2$ for parameters $\boldsymbol{w}$: $\Delta \boldsymbol{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s_i, a_i; \boldsymbol{w})$
    end for
    $t = t + 1$
    if mod(t,C) == 0 then
        $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$
    end if
end loop
```

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

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.

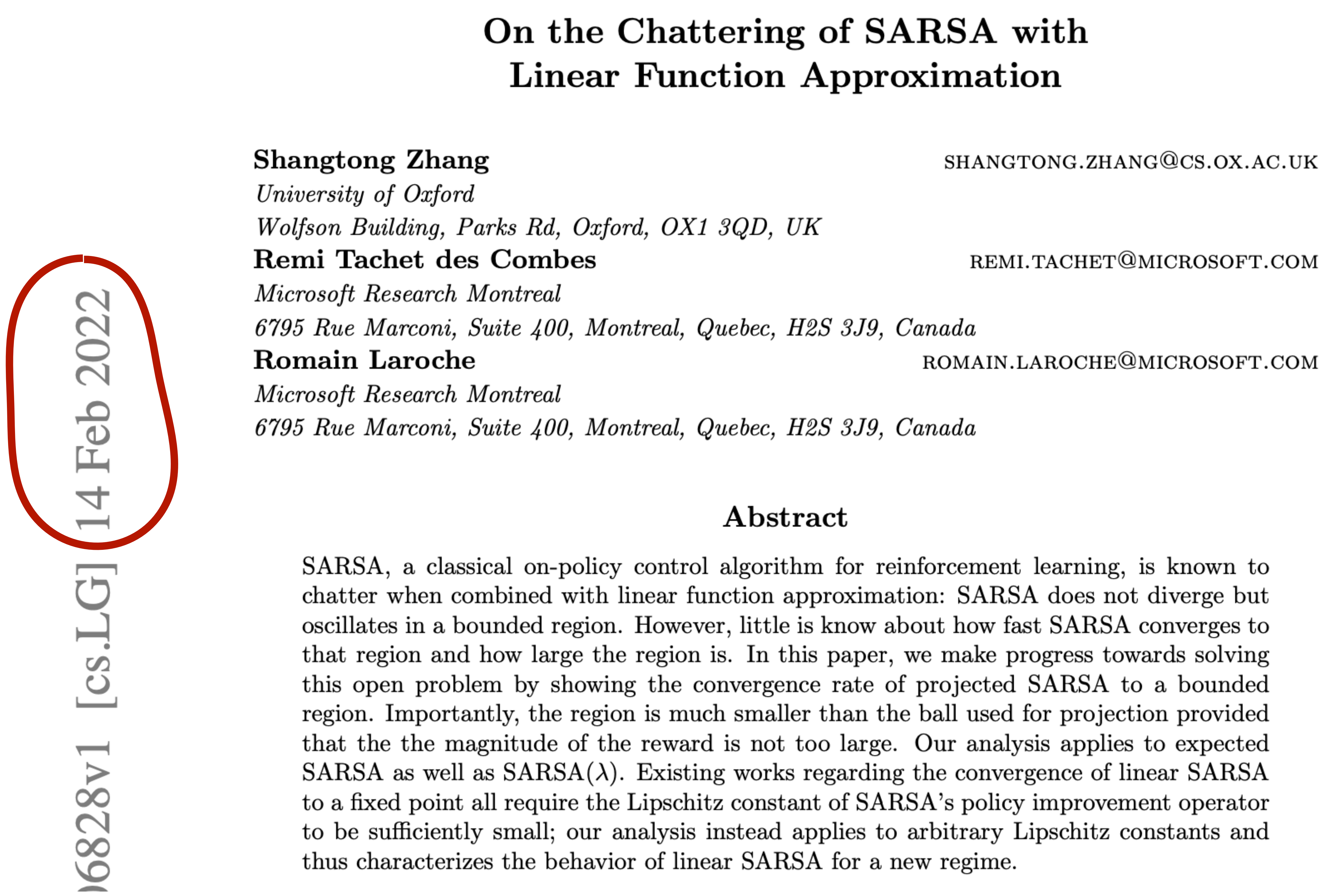
Partial remedies: experience replay and fixed Q-targets.

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

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: Bandits (Introduction)

Efficient Learning

So far we only cared about whether our RL algorithms converge, not that much how fast

We assumed that failed experiments (episodes) do not cost us anything (except, maybe, time). That is the case, e.g., when learning some strategy with a simulator or when playing computer games, but not, e.g., when optimizing an advertisement campaign…

We can generally study efficient learning for MDPs but in this course we will only look at efficient learning for multi-armed bandits (which are simpler but still interesting and used in practice).

Multi-Armed Bandits

1

2

3

4

$$P[R = r \,|\, A = i]$$

We can choose actions $\{1,2,3,4\}$ *and each of them leads to a different distribution of rewards.*

Setting

Multi-armed bandit is essentially a degenerate MDP that contains a single state.

Definition: A multi-armed bandit is given by:

A set A containing m actions $a_1, a_2, \ldots, a_m$ (each can be thought of as "pulling an arm").

Reward distributions $P[R_t = r \,|\, A_t = a]$, that is the distribution of rewards at time t given the action at time t.

At each step, the agent takes an action and receives a reward sampled from the above distribution.

The *informal* goal is to maximize the reward $\sum_{t=1}^{T} R_t$…. of course, this is a random variable.

Example

Your PR team created m different advertisements. You are now supposed to show these advertisements to people and maximize the number of times they click on them.

This can be modelled using multi-armed bandits:

The action a_i corresponds to displaying the i-th advertisement from our collection.

We get reward 1 when the person clicks on the advertisement and 0 otherwise.

Clearly, the probabilities $P[R_t = 1 \,|\, A_t = 1]$, $P[R_t = 1 \,|\, A_t = 2]$, … will be different (different advertisements will have different quality).

Regret (1/3)

Action-value: $Q(a) = \mathbb{E}[R_t | A_t = a]$.

Similar to MDPs where we had $Q^\pi(s, a)$. However, we do not need s because we now have only one state. So we could rewrite it as $Q^\pi(a)$. But then, since the action only affects the immediate reward and not to which state we get, the whole notion of policy is not very important for Q in this setting, so we drop that as well and end up with $Q(a) = \mathbb{E}[R_t | A_t = a]$.

Regret (2/3)

Optimal value: $V^* = \max_{a \in A} Q(a) = \max_{a \in A} \mathbb{E}[R_t \,|\, A_t = a].$

Optimal action: $a^* = \arg\max_{a \in A} Q(a)$**.**

Regret: $L_t = V^* - Q(A_t)$**.**

That is, regret is the "opportunity loss" at time t. Note that we use expected value in the definition of regret (recall how we defined $Q(a)$). That means we are not measuring regret directly in terms of what we observe. Since the parameters of bandits will generally be unknown, it also means we will not be able to compute regret directly.

Regret (3/3)

Total regret: $L_T^{tot} = \sum_{t=1}^{T} L_t = \sum_{t=1}^{T} (V^* - Q(A_t))$.

Minimizing total regret is the same as maximizing the expected sum of rewards (i.e. return).

Example

Consider again the example with advertisements, say we have 2 different advertisements that we can use, so $A = \{a_1, a_2\}$.

Suppose that:

$P[\text{Person t clicks on ad} \,|\, A_t = a_1] = 0.8$, $P[\text{Person t clicks on ad} \,|\, A_t = a_2] = 0.5$

So $\mathbb{E}[R_t | A_t = a_1] = 0.8$, $\mathbb{E}[R_t | A_t = a_2] = 0.5$.

Let us have the following deterministic sequence of actions:

$a_1, 1, a_1, 0, a_2, 1, a_1, 1, a_2, 0, a_1, 1, a_1, 0, a_1, 1, a_1, 1, a_1, 1$

What is the total regret of this episode?

We have $V^* = 0.8$, $V^* - Q(a_1) = 0$, $V^* - Q(a_2) = 0.8 - 0.5 = 0.3$.

So the total regret is:

$$0 + 0 + 0.3 + 0 + 0.3 + 0 + 0 + \ldots + 0 = 0.6.$$

What We Want… (1/2)

We want to find algorithms where the regret will grow slowly with the number of time steps taken.

Note that:

When regret does not grow at all after some time, that means that we are already taking the optimal action.

Regret is the difference between best possible return and the return under our strategy. So when the regret grows slowly, it means we are already doing quite well.

What We Want… (2/2)

If we knew the expectations $\mathbb{E}[R_t | A_t = a]$ *then the problem would be trivial, but it would not be reinforcement learning.*

We could try to first estimate $\mathbb{E}[R_t | A_t = a]$ *by taking actions completely randomly. However, then in this first part we would incur high regret and it is also not clear how long we should be estimating (because that actually depends on the values of* $\mathbb{E}[R_t | A_t = a]$*)… So we will need something smarter.*

Greedy Methods (Why They Would Not Work)

Greedy Algorithm

```
Initialization: Do several passes over all actions and compute estimates $\hat{Q}(a)$ for all $a \in A$.
Maintain counter $N(a)$ with the number of times an action was used.
While (some stopping condition):
    Select the action $a_t \in A$ which maximizes $\hat{Q}(a)$.
    Use the selected action and observe $r_t$.
    Set $N(a_t) := N(a_t) + 1$.
    Set $\hat{Q}(a_t) := \hat{Q}(a_t) + \frac{1}{N(a_t)}(r_t - Q(a_t))$.**
```

$$** \left(\begin{aligned} \underbrace{Q(a_t)}_{=\frac{1}{N(a_t)-1}(r_{i_1}+\ldots+r_{i_{t-1}})} + \frac{1}{N(a_t)} r_{i_t} - \frac{1}{N(a_t)} Q(a_t) &= \frac{N(a_t)(r_{i_1} + \ldots + r_{i_{t-1}}) + (N(a_t) - t) r_{i_t} - (N(a_t) - 1)\frac{1}{N(a_t)-1}(r_{i_1} + \ldots + r_{i_{t-1}})}{(N(a_t) - 1)N(a_t)} \\ &= \frac{(N(a_t) - 1)(r_{i_1} + \ldots + r_{i_{t-1}}) + (N(a_t) - t) r_{i_t}}{(N(a_t) - 1)N(a_t)} = \frac{1}{N(a_t)}(r_{i_1} + r_{i_2} + \ldots + r_{i_t}) \end{aligned} \right)$$

Why Greedy Will Not Work Well

This will be similar to why purely greedy methods do not work well for RL (as we saw before, where we solved the problem by using ε-greedy methods.

Example (Continue with our previous example):

$$\mathbb{E}[R_t | A_t = a_1] = 0.8, \mathbb{E}[R_t | A_t = a_2] = 0.5.$$

For greedy methods, we need some initialization (e.g. passing over all the actions a couple of times).

Suppose that our initial estimates for Q are $\hat{Q}(a_1) = 0$ and $\hat{Q}(a_2) = 0.5$ (which can happen if we are unlucky in the initialization).

Then we will never select a_1 even though it is the optimal action. So **regret will grow linearly with time** in this case.

ε-Greedy Methods (*Also not that great…*)

ε-Greedy (Basic Idea)

Similarly to what we did in the previous lectures…

```
**Initialization:** Do several passes over all actions and compute estimates $\hat{Q}(a)$ for all $a \in A$. Maintain counter $N(a)$ with the number of times an action was used.
**While (some stopping condition):**
    With probability $1 - \varepsilon$:
        Select the action $a_t \in A$ which maximizes $\hat{Q}(a)$.
    Else:
        Select an action $a_t \in A$ uniformly at random.
    Use the selected action and observe $r_t$.
    Set $N(a_t) := N(a_t) + 1$.
    Set $\hat{Q}(a_t) := \hat{Q}(a_t) + \frac{1}{N(a_t)}(r_t - Q(a_t))$.
```

Regret of ε-Greedy Methods

If we keep ε constant during the run of the ε-greedy algorithm then we will incur regret growing linearly with the number of time steps—in every step we have probability $\varepsilon - \dfrac{\varepsilon}{|A|}$ of picking a suboptimal action (assuming no ties) which will incur a regret of at least $V^* - \max_{a \neq a^*} Q(a)$

So also not great…

We might try to set ε to be a function of t (as we did before) but it turns out to be tricky and need to know a lot about $Q(a)$'s in advance.

Optimism Under Uncertainty

UCB Algorithm: Basic Idea

Upper-Confidence Bound (UCB) Algorithm

For every action $a \in A$, maintain an upper bound $U_t(a)$ *(the upper bound will change with time, that is why it is indexed by t).*

In every time step t, take the action that has the maximum upper bound, i.e. take the action $\arg\max_{a \in A} U_t(a)$.

After observing the reward, update the estimates.

UCB Algorithm

Initialization:

Take every action $a \in A$ once and record the rewards in $\hat{Q}(a)$.

$t := 1$

Loop:

Compute upper confidence bounds for all actions $a_i \in A$:

$$U_t(a_i) = \hat{Q}(a_i) + \sqrt{\frac{1}{2N(a_i)} \log \frac{2t^2}{\delta}}$$

Use the action $a_t = \arg\max_{a \in A} U_t(a)$ and observe the reward r_t.

Update $N(a_t) := N(a_1) + 1$

Update $\hat{Q}(a_t) := \hat{Q}(a_t) + \frac{1}{N(a_t)}(r_t - Q(a_t))$.

$t := t + 1$

UCB Theorem

With probability at least $1 - 2\delta m$, we have for the regret of the UCB algorithm:

$$L_T^{tot} \leq 2\sqrt{\frac{Tm}{2}\log\frac{T^2}{\delta}}.$$

Sublinear regret!!!!

Conclusions

- There is a lot more about bandits than we could cover here… and about sample-efficient reinforcement learning in general.

If you want to know more…

Lattimore, Tor, and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

Available online: https://tor-lattimore.com/downloads/book/book.pdf

EXTRA

Proof (1/12)

Claim: If all upper bounds $U_t(a_1), U_t(a_2), \ldots, U_t(a_m)$ satisfy $U_t(a_i) \geq Q(a_i)$, i.e. if none of them underestimates the true value, then for the action a_t selected at time t, it must hold

$$U_t(a_t) \geq U(a*) \geq Q(a*) = V*.$$

Easy to see why…

Proof (2/12)

First, we will state an auxiliary statement (*which you probably know from other courses).*

Theorem (Hoeffding's Inequality): Let $X_1, X_2, \ldots, X_N$ be independent random variables bounded on the interval $[a; b]$. Let $\overline{X}_N = \frac{1}{N}\sum_{i=1}^{N} X_i$. Then it holds

$$P\left[\overline{X}_N - \mathbb{E}[\overline{X}_N] \geq \xi\right] \leq \exp\left(-\frac{2N\xi^2}{(b-a)^2}\right),$$

$$P\left[\mathbb{E}[\overline{X}_N] - \overline{X}_N \geq \xi\right] \leq \exp\left(-\frac{2N\xi^2}{(b-a)^2}\right),$$

$$P\left[|\overline{X}_N - \mathbb{E}[\overline{X}_N]| \geq \xi\right] \leq 2\exp\left(-\frac{2N\xi^2}{(b-a)^2}\right).$$

Proof (3/12)

Our $\overline{X}_N$ will be $\hat{Q}_t(a_i)$, i.e. the estimate of $\hat{Q}(a_i)$, and our N will therefore be $N_t(a_i)$, i.e. number of times a_i was used.

We have $\mathbb{E}[\hat{Q}_t(a_i)] = Q(a_i)$.

We will want to find ξ_t (one value for each t) such that

$$P\left[\,|\,Q(a_i) - \hat{Q}(a_i)\,| \geq \xi_t\right] \leq 2\exp\left(-\frac{2N_t(a_i)\xi_t^2}{(b-a)^2}\right) = \frac{\delta}{t^2},$$

where t is the current number of time steps.

Proof (4/12)

We have

$$P\left[\,|Q(a_i) - \hat{Q}(a_i)| \geq \xi_t\right] \leq 2\exp\left(-\frac{2N_t(a_i)\xi_t^2}{(b-a)^2}\right) = \frac{\delta}{t^2},$$

$$-\frac{2N(a_i)\xi_t^2}{(b-a)^2} = \log\frac{\delta}{2t^2},$$

$$\xi_t = (b-a)\sqrt{\frac{1}{2N_t(a_i)}\log\frac{2t^2}{\delta}}$$

For simplicity we will now assume that $a = 0,\, b = 1$.

Proof (5/12)

That is, the upper bounds $U_t(a_i)$ will be:

$$U_t(a_i) = \hat{Q}(a_i) + \sqrt{\frac{1}{2N_t(a_i)} \log \frac{2t^2}{\delta}}.$$

And we will also have lower bounds $L_t(a_i)$:

$$L_t(a_i) = \hat{Q}(a_i) - \sqrt{\frac{1}{2N_t(a_i)} \log \frac{2t^2}{\delta}}.$$

Proof (6/12)

Let A_t be the **action selected at time t.**

We will now bound the probability that at least some of the bounds are incorrect *(we will see in a moment why we want this)*.

$$P\left[\bigvee_{t=1}^{T}\bigvee_{i=1}^{m} Q(a_i) \notin [L_t(a_i); U_t(a_i)]\right] \leq$$

$$\leq \sum_{t=1}^{T}\sum_{i=1}^{m} P[\,|\,Q(a_i) - \hat{Q}_t(a_i)\,| > \xi_t] \leq \sum_{t=1}^{T}\sum_{i=1}^{m}\frac{\delta}{t^2} = m\delta\sum_{t=1}^{T}\frac{1}{t^2}.$$

Proof (7/12)

We can now use the famous identity $\sum_{t=1}^{\infty} \frac{1}{t^2} = \frac{\pi^2}{6}$ *(which is smaller than 2)*.**

So we can bound:

$$P\left[\bigvee_{t=1}^{T} \bigvee_{i=1}^{m} U_t(a_i) \notin [L_t(a_i); U_t(a_i)]\right] \leq 2m\delta.$$

That means that the probability that all lower and upper bounds are valid at all time steps is at least $1 - 2m\delta$.

We will use this in a moment.

**** We actually do not need this fancy result to get the constant 2 (see the additional slide)**

Proof (8/12)

Let A_t be the **action selected at time t.**

We will now bound the probability that at least one of the upper bounds $U_1(A_1), U_2(A_2), \dots$ is lower than $U(a^*)$.

We can notice that the event that at least one action has wrong confidence bounds over the course of T time steps, formally written as

$$\bigvee_{t=1}^{T} \bigvee_{i=1}^{m} U_t(a_i) \notin [L_t(a_i); U_t(a_i)]$$

is a necessary condition for at least one of the upper bounds $U_1(A_1), U_2(A_2), \dots$ to be lower than $U(a^*)$.

Therefore we can bound this probability also by $1 - 2\delta m$.

Proof (9/12)

Let us now compute the regret of this algorithm:

$$\text{Regret}(T) = \sum_{t=1}^{T} \big(Q(a*) - Q(A_t)\big) = \sum_{t=1}^{T} \big(U_t(A_t) - Q(A_t) + Q(a*) - U_t(A_t)\big)$$

We have that $Q(a*) < U_t(A_t)$ with probability at least $1 - 2m\delta$ (*from the previous slide!*) Hence we can bound the above as:

$$\text{Regret}(T) \leq \sum_{t=1}^{T} \big(U_t(A_t) - Q(A_t)\big).$$

Proof (10/12)

Now we will play with

$$\text{Regret}(T) \leq \sum_{t=1}^{T} \big(U_t(A_t) - Q(A_t)\big).$$

Recall that we defined $U_t(a_i) = \hat{Q}(a_i) + \sqrt{\frac{1}{2N_t(a_i)} \log \frac{2t^2}{\delta}}$ for all $a_i \in A$.

Hence we get

$$\text{Regret}(T) \leq \sum_{t=1}^{T} \left(\hat{Q}(A_t) + \sqrt{\frac{1}{2N_t(A_t)} \log \frac{2t^2}{\delta}} - Q(A_t) \right).$$

Proof (11/12)

Now we need to do something with

$$\text{Regret}(T) \leq \sum_{t=1}^{T} \left(\hat{Q}(A_t) + \sqrt{\frac{1}{2N_t(A_t)} \log \frac{t^2}{\delta}} - Q(A_t) \right).$$

Since we have that, with probability at least $1 - 2\delta m$, we have for all $a_t \in A$

$$\left| \hat{Q}(a_t) - Q(a_t) \right| \leq \sqrt{\frac{1}{2N_t(A_t)} \log \frac{t^2}{\delta}}.$$

We can bound the regret, with probability at least $1 - 2\delta m$, as

$$\text{Regret}(T) \leq \sum_{t=1}^{T} 2\sqrt{\frac{1}{2N_t(A_t)} \log \frac{t^2}{\delta}} = \sum_{t=1}^{T} \sqrt{\frac{2}{N_t(A_t)} \log \frac{t^2}{\delta}}.$$

Proof (12/12)

Finally we have, with probability at least $1 - 2\delta m$,

$$\text{Regret}(T) \leq \sum_{t=1}^{T} \sqrt{\frac{2}{N_t(A_t)} \log \frac{t^2}{\delta}} = \sqrt{\log \frac{t^2}{\delta}} \sum_{t=1}^{T} \sqrt{\frac{2}{N_t(A_t)}} =$$

$$= \sqrt{2 \log \frac{t^2}{\delta}} \sum_{i=1}^{m} \sum_{j=1}^{N_T(a_i)} \sqrt{\frac{1}{j}} \leq 2\sqrt{\frac{Tm}{2} \log \frac{T^2}{\delta}}.$$

Sublinear regret!!!!

Additional Slide

(Why $\sum_{t=1}^{\infty} \frac{1}{t^2} = \frac{\pi^2}{6}$ is not needed)

Bounding

$$\sum_{t=1}^{\infty} \frac{1}{t^2} \leq 1 + \int_1^{\infty} \frac{1}{t^2} dt = 1 + \left[-\frac{1}{t} \right]_1^{\infty} = 2.$$

END OF SLIDES