



S&DS 365 / 665  
Intermediate Machine Learning

# Reinforcement Learning: Policy Methods

April 11

Yale

# Reminders

- Quiz 3: Mean 90% (great work!)
- Assignment 3 due this Wednesday
- Assignment 4 (last!) out on Wednesday

# Outline

- Deep Q-Learning: Recap
- Minimal DQN example
- TF tape and automatic differentiation
- Landscape of RL Methods
- Policy iteration
- Policy gradients

# Important RL concepts

*Policy*: A mapping from states to actions. An algorithm/rule to make decisions at each time step, designed to maximize the long term reward.

# Important RL concepts

**Value function:** A mapping from states to total reward. The total reward the agent can expect to accumulate in the future, starting from that state.

Rewards are short term. Values are predictions of future rewards.

*This week we'll introduce methods that estimate the policy, or estimate the value function and policy together*

# Recall: Q-learning

- Maintains a “quality” variable  $Q(s, a)$  for taking action  $a$  in state  $s$

## Recall: Q-learning

- Maintains a “quality” variable  $Q(s, a)$  for taking action  $a$  in state  $s$
- A measure of the cumulative rewards obtained by the algorithm when it takes action  $a$  in state  $s$

# Recall: Q-learning

- Maintains a “quality” variable  $Q(s, a)$  for taking action  $a$  in state  $s$
- A measure of the cumulative rewards obtained by the algorithm when it takes action  $a$  in state  $s$
- Quality should not be assessed purely based on the reward the action has in the current time step

## Recall: Q-learning

- Maintains a “quality” variable  $Q(s, a)$  for taking action  $a$  in state  $s$
- A measure of the cumulative rewards obtained by the algorithm when it takes action  $a$  in state  $s$
- Quality should not be assessed purely based on the reward the action has in the current time step
- Need to take into account the future rewards

## Recall: Q-learning

- Maintains a “quality” variable  $Q(s, a)$  for taking action  $a$  in state  $s$
- A measure of the cumulative rewards obtained by the algorithm when it takes action  $a$  in state  $s$
- Quality should not be assessed purely based on the reward the action has in the current time step
- Need to take into account the future rewards
- A type of gradient ascent algorithm

## Q-Learning

Parameters: step size  $\alpha$ , exploration probability  $\varepsilon$ , discount factor  $\gamma$   
Initialize  $Q(s, a)$  arbitrarily, except  $Q(\text{terminal}, \cdot) = 0$

**Loop** for each episode:

    Initialize state  $s$

**Loop** for each step of episode:

        Choose action  $a$  using  $Q$  with  $\varepsilon$ -greedy policy

        Take action  $a$ ; observe reward  $r$  and new state  $s'$

$$Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

$$s \leftarrow s'$$

**Until**  $s$  is terminal

## Recall: Bellman equation

The optimality condition for the value function  $v_*$  is

$$\begin{aligned}v_*(s) &= \max_a \sum_{s',r} p(s', r | s, a) \left\{ r + \gamma v_*(s') \right\} \\&= \max_a \mathbb{E} \left[ R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a \right]\end{aligned}$$

# Recall: Bellman equation

## Value function optimality

$$v_*(s) = \max_a \mathbb{E} \left[ R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a \right]$$

# Recall: Bellman equation

The optimality condition for the  $Q$ -function

$$\begin{aligned} Q_*(s, a) &= \sum_{s', r} p(s', r | s, a) \left\{ r + \gamma \max_{a'} Q_*(s', a') \right\} \\ &= \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \end{aligned}$$

# Recall: Bellman equation

## **$Q$ -function optimality**

$$Q_*(s, a) = \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right]$$

# Comment on Q-learning

- Q-learning is an example of *temporal difference (TD) learning*
- It is an “off-policy” approach that is practical if the space of states and actions is small
- Value iteration is analogous approach for learning value function

# Deep reinforcement learning: Motivation

- Direct implementation of Q-learning only possible for small state and action spaces
- For large state spaces we need to map states to “features”
- Deep RL uses a multilayer neural network to learn these features and the  $Q$ -function

# Strategy

Objective:

$$Q(s, a; \theta) = \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta) \mid S_t = s, A_t = a \right]$$

Let  $y_t$  be a sample from this conditional distribution:

$$y_t = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta_{\text{current}})$$

Adjust the parameters  $\theta$  to make the squared error small (SGD):

$$(y_t - Q(s, a; \theta))^2$$

# Strategy

Adjust the parameters  $\theta$  to make the squared error small

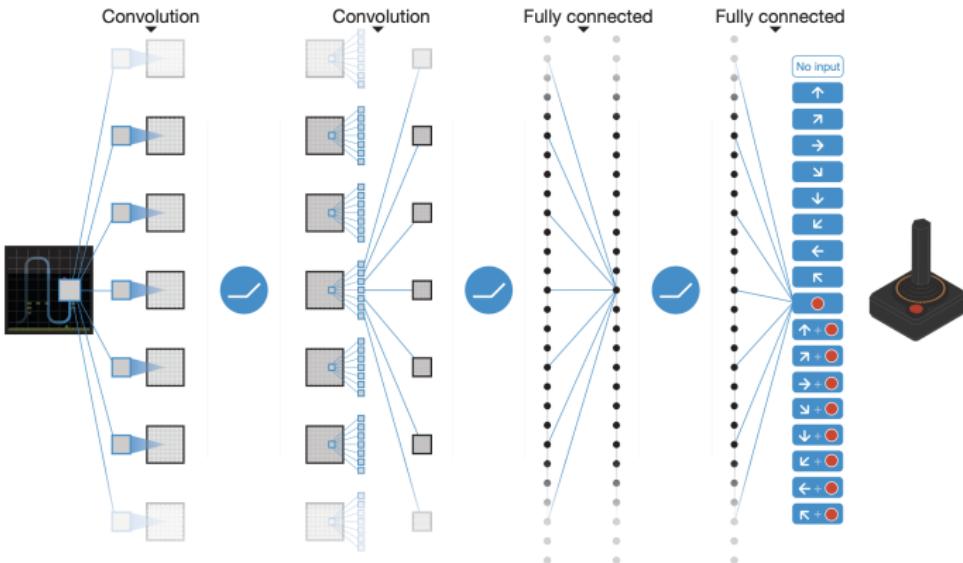
$$(y_t - Q(s, a; \theta))^2$$

How? Carry out SGD

$$\theta \longleftarrow \theta + \eta (y_t - Q(s, a; \theta)) \nabla_{\theta} Q(s, a; \theta)$$

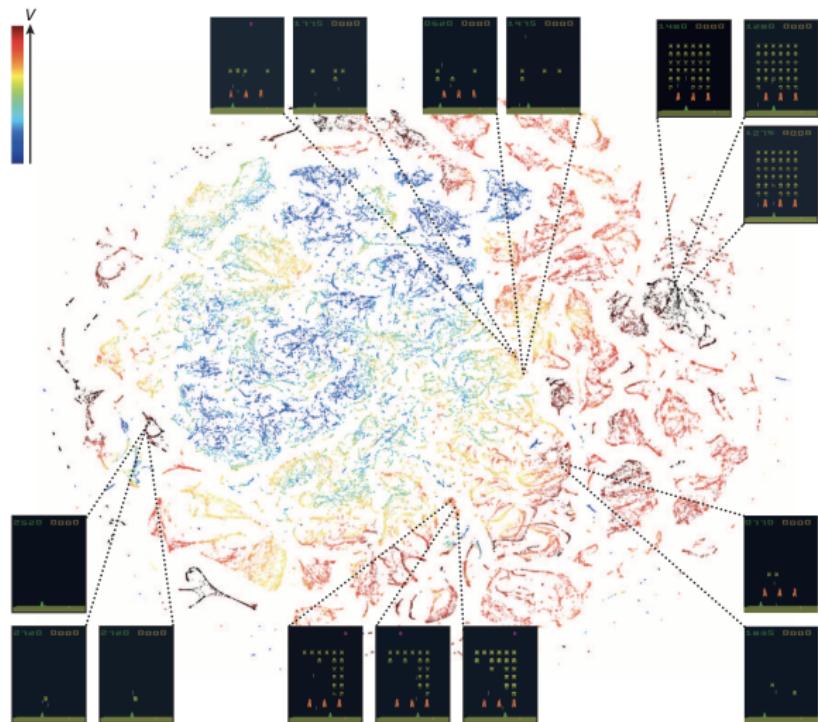
using backpropagation

# Second generation DQN



<https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf>

# Second generation DQN: Interpretation



t-SNE representations of last layer for Space Invaders, color-coded for  $v_*$ .

# When does learning take place?

Recall from Bellman equation that  $y_t$  is an expectation.

*Learning takes place when expectations are violated. The receipt of the reward itself does not cause changes.*

## Next: Simple DQN example

But first: TensorFlow automatic differentiation

- For supervised problems, loss function is  $L(\hat{Y}, Y)$
- Can program this directly
- Often RL loss functions are built up dynamically
- Automatic differentiation allows us to handle this

<https://www.tensorflow.org/guide/autodiff>

# Automatic differentiation: Hello world!

```
In [1]: import numpy as np  
import tensorflow as tf
```

```
In [2]: x = tf.Variable(3.0)  
  
with tf.GradientTape() as tape:  
    y = x**2  
  
dy_dx = tape.gradient(y, x)  
print(dy_dx.numpy())
```

6.0

# Automatic differentiation: Hello world!

```
In [3]: w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]

with tf.GradientTape() as tape:
    y = x @ w + b
    loss = tf.reduce_mean(y**2)

[dloss_dw, dloss_db] = tape.gradient(loss, [w, b])
print(dloss_dw.numpy(), "\n\n", dloss_db.numpy())
```

```
[[2.3997068  0.70033383]
 [4.7994137  1.4006677 ]
 [7.1991205  2.1010015 ]]
[2.3997068  0.70033383]
```

# Automatic differentiation: Gradient collection

TensorFlow supports automatic differentiation by recording relevant operations executed inside the context of a “tape”

`tf.GradientTape`

It then uses the record to compute the numerical values of gradients using “reverse mode differentiation”

# Automatic differentiation: Parameter updates

Parameters are then updated as shown here:

```
In [4]: opt = keras.optimizers.Adam(learning_rate=0.001)
    _ = opt.apply_gradients(zip([dloss_dw, dloss_db], [w, b]))
```

We'll see examples shortly

# Multi-armed bandits



# Multi-armed bandits

- The rewards are independent and noisy
- Arm  $k$  has expected payoff  $\mu_k$  with variance  $\sigma_k^2$  on each pull
- Each time step, pull an arm and observe the resulting reward
- Played often enough, can estimate mean reward of each arm
- What is the best policy?
- Exploration-exploitation tradeoff

# Multi-armed bandits

We'll treat this as an RL problem and hit it with a big hammer:  
Deep Q-learning

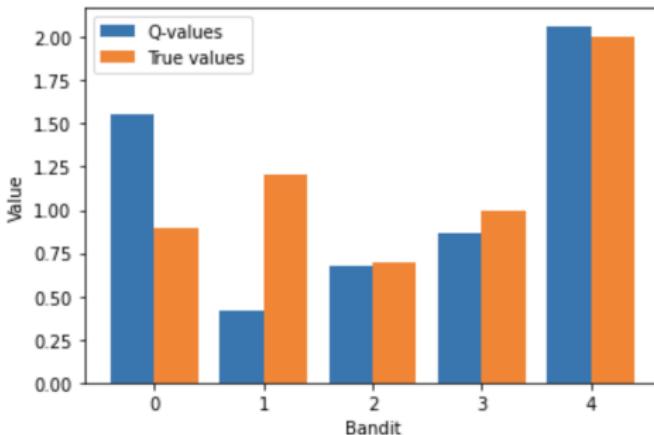
# Multi-armed bandits

```
=====episode 10000 =====
```

```
Q-values ['1.556', '0.412', '0.675', '0.866', '2.065']
```

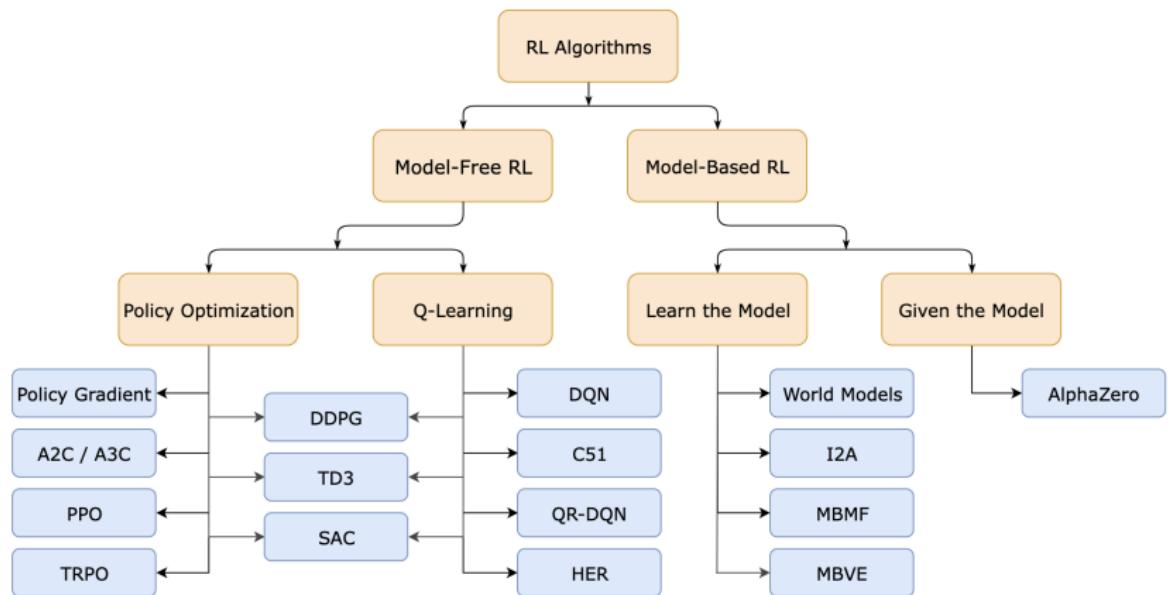
```
Deviation ['72.8%', '-65.7%', '-3.6%', '-13.4%', '3.3%']
```

```
<Figure size 864x504 with 0 Axes>
```



Let's go to the notebook

# Landscape of RL algorithms



## Policy iteration: Idea

0. Initialize policy arbitrarily
1. Compute values for current policy (policy evaluation)
2. Update policy to match values (policy improvement)
3. Go to 1.

This will compute an optimal policy—it will satisfy Bellman's equations. Step 2. can only increase the value of the policy.

# Policy iteration: Algorithm

Policy Iteration (using iterative policy evaluation) for estimating  $\pi \approx \pi_*$

1. Initialization

$V(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$

2. Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

Loop for each  $s \in \mathcal{S}$ :

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$  (a small positive number determining the accuracy of estimation)

3. Policy Improvement

*policy-stable*  $\leftarrow$  true

For each  $s \in \mathcal{S}$ :

$$\text{old-action} \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

If  $\text{old-action} \neq \pi(s)$ , then *policy-stable*  $\leftarrow$  false

If *policy-stable*, then stop and return  $V \approx v_*$  and  $\pi \approx \pi_*$ ; else go to 2

# Policy iteration

- As for vanilla Q-learning, this only works for small state spaces
- A “tabular” method, computes all values  $V(s)$  and actions  $\pi(s)$

## Policy gradient methods

- Parameterize the policy— $\pi_\theta(s)$ —and use features of states
- Perform gradient ascent over those parameters
- Well-suited to deep learning approaches
- Why use an on-policy method? May be possible to estimate a good policy without accurately estimating the value function

# Policy gradient methods: Loss function

We start with the loss function: Expected reward  $\mathcal{J}(\theta) = \mathbb{E}(R)$

- Parameterize the policy— $\pi(s; \theta)$ —and use features of states
- Perform gradient ascent of  $\mathcal{J}(\theta)$
- Well-suited to deep learning approaches

# Policy gradient methods: Loss function

Policy is probability distribution  $\pi_\theta(a|s)$  over actions given state  $s$ .

The episode unfolds as a random sequence  $\tau$

$$\tau : (s_0, a_0) \rightarrow (s_1, r_1, a_1) \rightarrow (s_2, r_2, a_2) \rightarrow \dots \rightarrow (s_T, r_T, a_T) \rightarrow s_{T+1}$$

where  $s_{T+1}$  is a terminal state. Receive reward  $R(\tau)$ , for example

$$R(\tau) = \sum_{t=1}^T r_t$$

Objective function  $\mathcal{J}$  is expected reward

$$\mathcal{J}(\theta) = \mathbb{E}_\theta(R(\tau))$$

# Calculating the gradient

Using Markov property, calculate  $\mathbb{E}_\theta(R(\tau))$  as

$$\mathbb{E}_\theta(R(\tau)) = \int p(\tau | \theta) R(\tau) d\tau$$
$$p(\tau | \theta) = \prod_{t=0}^T \pi_\theta(a_t | s_t) p(s_{t+1}, r_{t+1} | s_t, a_t)$$

---

If states or rewards are finite the integral becomes a sum, or a mix of sums and integrals.

# Calculating the gradient

Using Markov property, calculate  $\mathbb{E}_\theta(R(\tau))$  as

$$\mathbb{E}_\theta(R(\tau)) = \int p(\tau | \theta) R(\tau) d\tau$$
$$p(\tau | \theta) = \prod_{t=0}^T \pi_\theta(a_t | s_t) p(s_{t+1}, r_{t+1} | s_t, a_t)$$

It follows that

$$\nabla_\theta \log p(\tau | \theta) = \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) = \sum_{t=0}^T \frac{\nabla_\theta \pi_\theta(a_t | s_t)}{\pi_\theta(a_t | s_t)}$$

---

If states or rewards are finite the integral becomes a sum, or a mix of sums and integrals.

# Calculating the gradient

Now we use

$$\begin{aligned}\nabla_{\theta} \mathcal{J}(\theta) &= \nabla_{\theta} \mathbb{E}_{\theta} R(\tau) \\&= \nabla_{\theta} \int R(\tau) p(\tau | \theta) d\tau \\&= \int R(\tau) \nabla_{\theta} p(\tau | \theta) d\tau \\&= \int R(\tau) \frac{\nabla_{\theta} p(\tau | \theta)}{p(\tau | \theta)} p(\tau | \theta) d\tau \\&= \mathbb{E}_{\theta} \left( R(\tau) \nabla_{\theta} \log p(\tau | \theta) \right)\end{aligned}$$

# Approximating the gradient

Since it's an expectation, can approximate by sampling:

$$\begin{aligned}\nabla_{\theta} \mathcal{J}(\theta) &\approx \frac{1}{N} \sum_{i=1}^N R(\tau^{(i)}) \nabla_{\theta} \log p(\tau^{(i)} | \theta) \\ &= \frac{1}{N} \sum_{i=1}^N R(\tau^{(i)}) \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t^{(i)} | s_t^{(i)}) \\ &\equiv \widehat{\nabla_{\theta} \mathcal{J}(\theta)}\end{aligned}$$

# Approximating the gradient

Since it's an expectation, can approximate by sampling:

$$\begin{aligned}\nabla_{\theta} \mathcal{J}(\theta) &\approx \frac{1}{N} \sum_{i=1}^N R(\tau^{(i)}) \nabla_{\theta} \log p(\tau^{(i)} | \theta) \\ &= \frac{1}{N} \sum_{i=1}^N R(\tau^{(i)}) \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t^{(i)} | s_t^{(i)}) \\ &\equiv \widehat{\nabla_{\theta} \mathcal{J}(\theta)}\end{aligned}$$

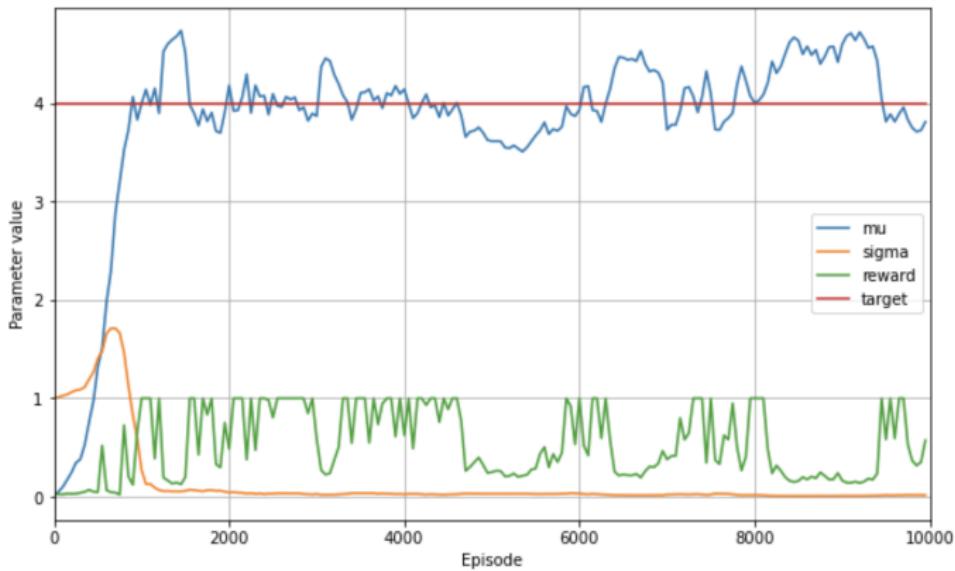
The policy gradient algorithm is then

$$\theta \leftarrow \theta + \eta \widehat{\nabla_{\theta} \mathcal{J}(\theta)}$$

# Simple example



# Simple example



Let's go to the notebook

# Summary

- Policy methods estimate  $\pi(a | s)$
- Policy iteration can be used for small state/action spaces
- Otherwise, parameterize  $\pi_\theta(a | s)$  and use gradient ascent
  - ▶ Change in expected reward calculated with “grad-log trick”
- Automatic differentiation is used for deep learning models