S&DS 365 / 665

Intermediate Machine Learning

Reinforcement Learning: Policy Methods

November 4

Yale

Reminders

- Quiz 4 on Wednesday, November 6
  - Graphs and conditional independence
  - Laplacians and GNNs
  - Q-learning
- Assignment 4 is out; due Monday November 18

Outline

- Deep Q-Learning: Recap
- Automatic differentiation
- Minimal DQN example
- Policy iteration
- Policy gradients

Main reference: “Reinforcement Learning: An Introduction” (Second Edition), Richard S. Sutton and Andrew G. Barto

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.

"Reinforcement Learning: An Introduction" (Second Edition), Richard S. Sutton and Andrew G. Barto

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 (if they are making optimal decisions)

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

"Reinforcement Learning: An Introduction" (Second Edition), Richard S. Sutton and Andrew G. Barto

Principle: Bellman equation

Value function optimality

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

Principle: Bellman equation

Q-function optimality

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

Algorithm from this principle

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
```

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:

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

Let y_t be one step of “play”:

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

Adjust the parameters θ to make the squared error small (SGD):

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

Strategy

Adjust the parameters θ to make the squared error small

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

How? Carry out SGD

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

using backpropagation

Replay buffer

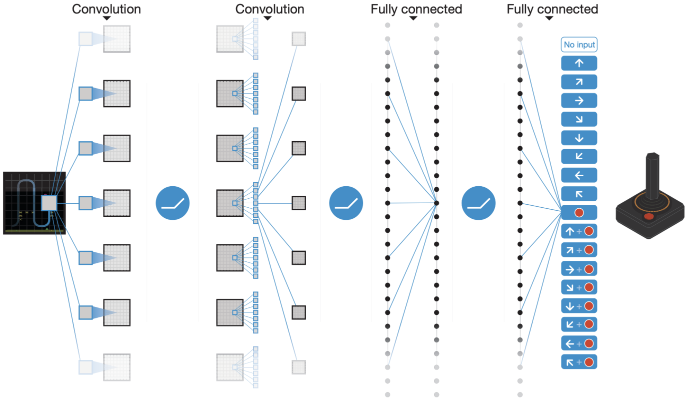
- *Q*-learning carried out using minibatches from a replay buffer of sequences that are “remembered and replayed”

Replay buffer

Role of replay buffer (inspired by neuroscience, “dreaming”)

- Prevent “forgetting” how to play early parts of a game
- Remove correlations between nearby state transitions
- Prevent cycling behavior, due to target changing

Second generation DQN



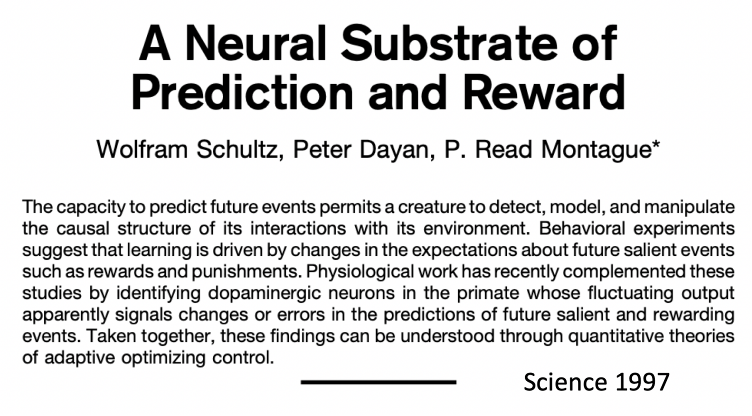
https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf

When does learning take place?

Recall that Bellman equation is a constraint on Q as an expectation.

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

Neuroscience connection



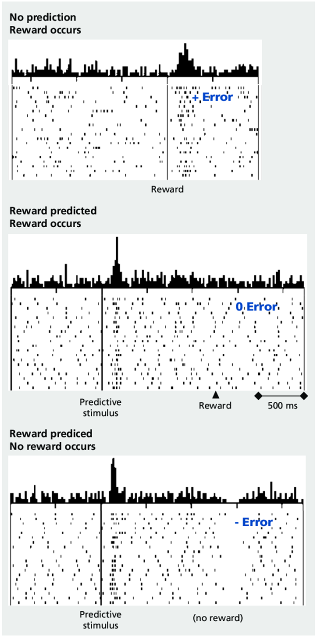
A Neural Substrate of Prediction and Reward

Wolfram Schultz, Peter Dayan, P. Read Montague*

The capacity to predict future events permits a creature to detect, model, and manipulate the causal structure of its interactions with its environment. Behavioral experiments suggest that learning is driven by changes in the expectations about future salient events such as rewards and punishments. Physiological work has recently complemented these studies by identifying dopaminergic neurons in the primate whose fluctuating output apparently signals changes or errors in the predictions of future salient and rewarding events. Taken together, these findings can be understood through quantitative theories of adaptive optimizing control.

Science 1997

Neuroscience connection



Automatic differentiation

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

`https://www.tensorflow.org/guide/autodiff`

`https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html`

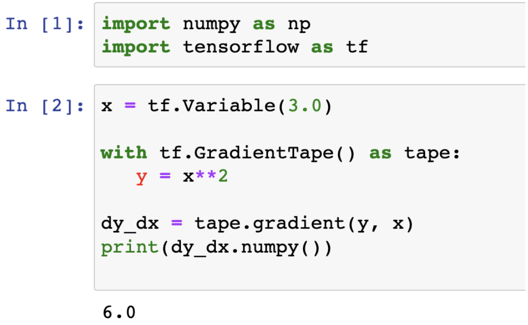
Automatic differentiation: Gradient collection

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”

`https://en.wikipedia.org/wiki/Automatic_differentiation`

Automatic differentiation: Hello world!

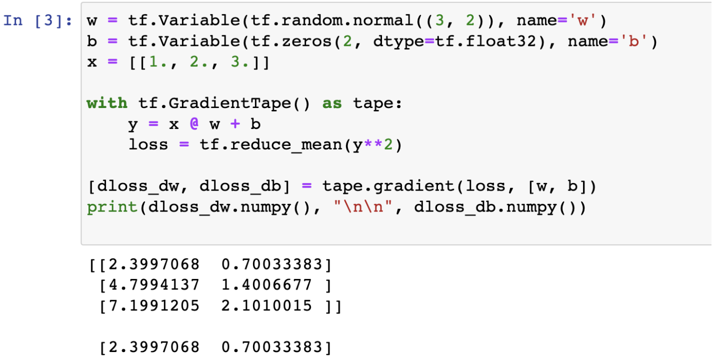


```
In [1]: import numpy as np
        import tensorflow as tf
```

```
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
```

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

Automatic differentiation: Hello world!



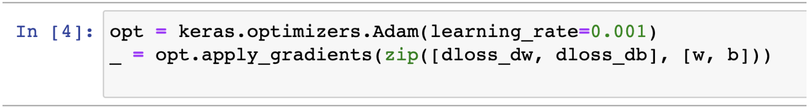
```
In [3]: w = tf.Variable(tf.random.normal((3, 2)), name='w')
        b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
        x = [[1., 2., 3.]]

        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]
```

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

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

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

Multi-armed bandits

Multi-armed bandits

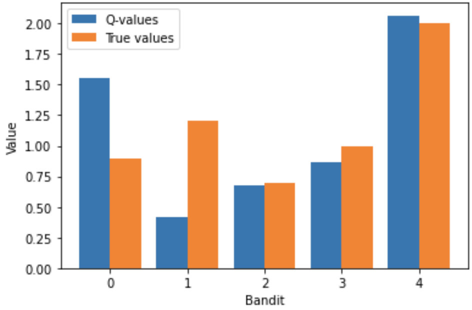
- The rewards are independent and noisy
- Arm k has expected payoff μ_k with variance σ_k^2 on each pull
- Each time step, pull an arm and observe the resulting reward
- Played often enough, can estimate mean reward of each arm
- What is the best policy?
- Exploration-exploitation tradeoff
- “Contextual bandits” add covariate vector — but actions don’t impact the environment

Multi-armed bandits

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

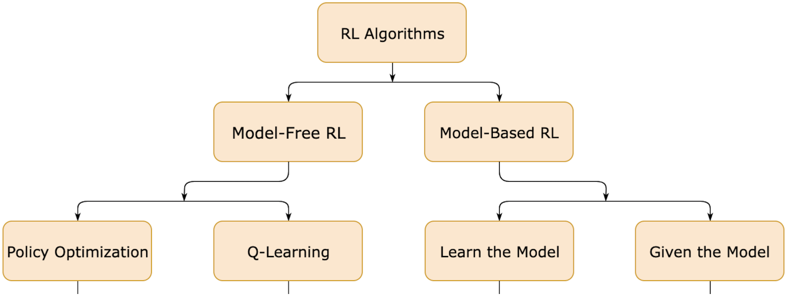
Multi-armed bandits

```
======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



https://tinyurl.com/25vruwtp

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 evaluation

Compute the value function for the current policy

Policy evaluation

```
Loop:
    $\Delta \leftarrow 0$
    Loop for each state $s$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))(r + \gamma V(s'))$
        $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
Until $\Delta < \varepsilon$
```

Policy improvement

Update the policy to match the value function

Policy improvement

```
stable $\leftarrow$ True
For each state $s$:
    $a_{old} \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, \pi(s))(r + \gamma V(s'))$
    If $a_{old} \neq \pi(s)$ then stable $\leftarrow$ False
if stable $=$ False return to policy evaluation
```

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 <u>gradient</u> 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 $\pi_\theta(a \mid s)$ over actions given state s.

The episode unfolds as a random sequence τ

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

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

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

For simplicity, shown without discounting

Calculating the gradient

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

$$\mathbb{E}_\theta(R(\tau)) = \int p(\tau \mid \theta) R(\tau)\, d\tau$$

$$p(\tau \mid \theta) = \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t)\, p(s_{t+1}, r_{t+1} \mid s_t, a_t)$$

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

$$\mathbb{E}_\theta(R(\tau)) = \int p(\tau \mid \theta) R(\tau)\, d\tau$$

$$p(\tau \mid \theta) = \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t)\, p(s_{t+1}, r_{t+1} \mid s_t, a_t)$$

It follows that

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

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

Calculating the gradient

Now we use

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

Need regularity conditions to exchange derivative and integral, such as uniform continuity of the derivatives of the policy.

Approximating the gradient

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

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

Approximating the gradient

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

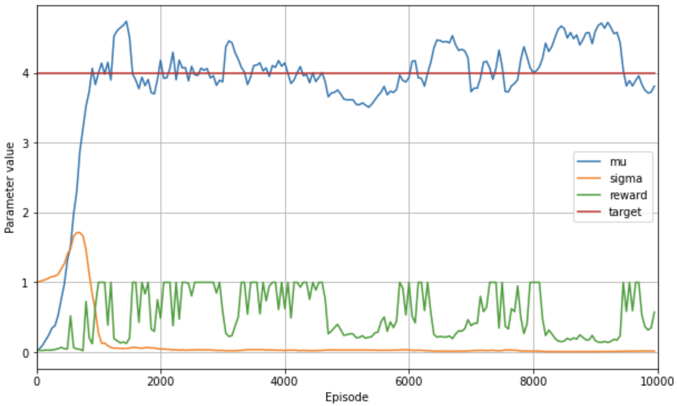
$$\begin{aligned}
\nabla_\theta \mathcal{J}(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} R(\tau^{(i)}) \nabla_\theta \log p(\tau^{(i)} \mid \theta) \\
&= \frac{1}{N} \sum_{i=1}^{N} R(\tau^{(i)}) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \\
&\equiv \widehat{\nabla_\theta \mathcal{J}(\theta)}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

The policy gradient algorithm is then

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

Simple example

Simple example



Let’s go to the notebook

Summary

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