



Навчання з підкріпленням

Лекція 7: Методи апроксимації функції цінності

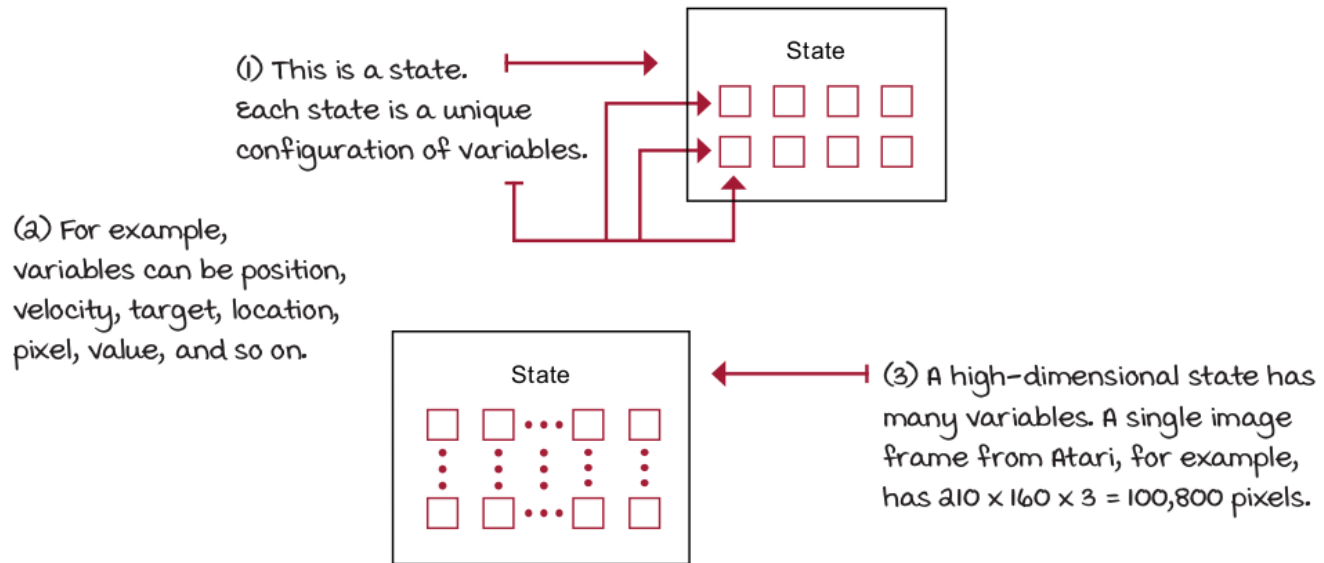
Кочура Юрій Петрович
iuriy.kochura@gmail.com
[@y_kochura](#)

Сьогодні

- Вступ
- Інкрементні методи
- Пакетні методи

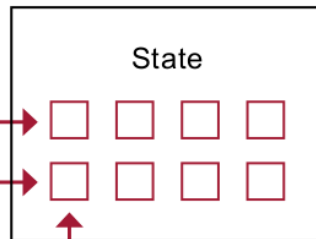
Вступ

High-dimensional state spaces



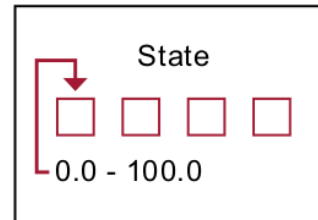
Continuous state spaces

(1) This is a state.
Each state is a
unique configuration
of variables.



(2) variables can be
position, velocity,
target, location, pixel,
value, and so on.

(3) A continuous state-space has at least
one variable that can take on an infinite
number of values. For example, position,
angles, and altitude are variables that
can have infinitesimal accuracy: say, 2.1,
or 2.12, or 2.123, and so on.

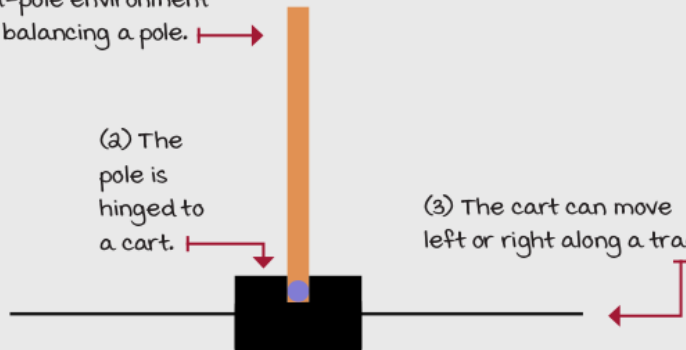


This is the cart-pole environment

(1) The cart-pole environment consists of balancing a pole. →

(2) The pole is hinged to a cart. ↓

(3) The cart can move left or right along a track. ←



Its state space is comprised of four variables:

- The cart position on the track (x-axis) with a range from -2.4 to 2.4
- The cart velocity along the track (x-axis) with a range from $-\infty$ to ∞
- The pole angle with a range of ~ -40 degrees to ~ 40 degrees
- The pole velocity at the tip with a range of $-\infty$ to ∞

There are two available actions in every state:

- Action 0 applies a -1 force to the cart (push it left)
- Action 1 applies a $+1$ force to the cart (push it right)

You reach a terminal state if

- The pole angle is more than 12 degrees away from the vertical position
- The cart center is more than 2.4 units from the center of the track
- The episode count reaches 500 time steps (more on this later)

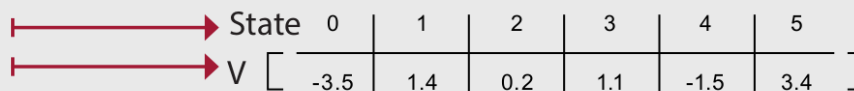
The reward function is

- $+1$ for every time step

Апроксимація функцій має переваги

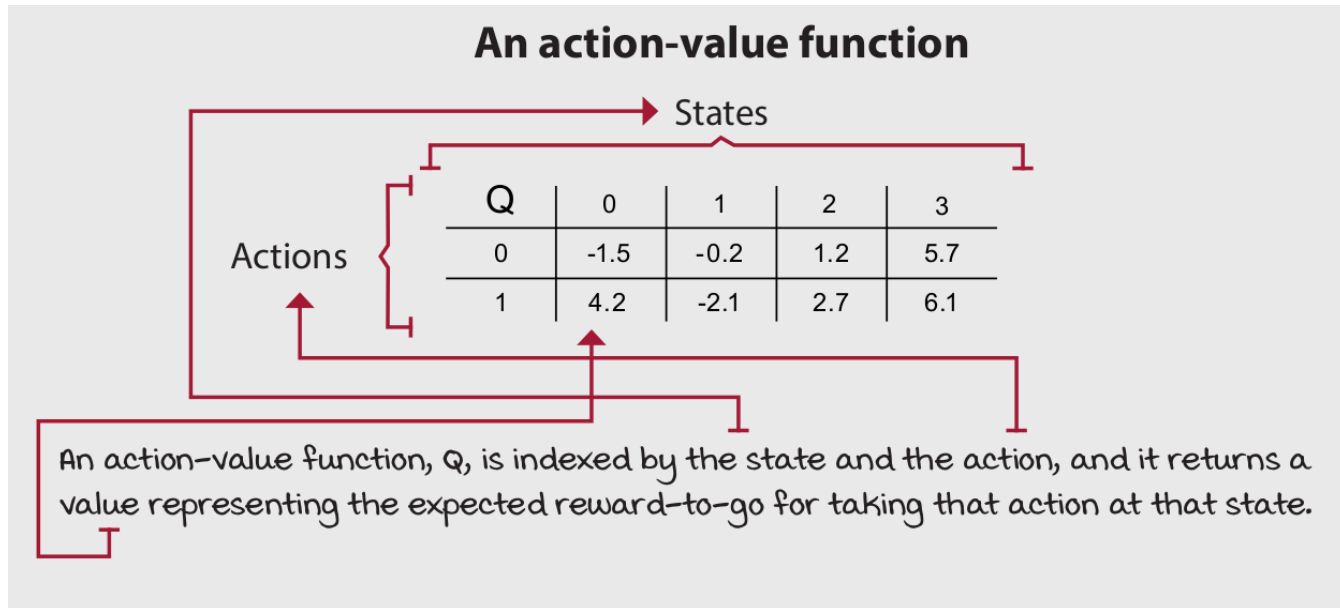
A state-value function

(1) A state-value function is indexed by the state, and it returns a value representing the expected reward-to-go at the given state.



State	0	1	2	3	4	5
V	-3.5	1.4	0.2	1.1	-1.5	3.4

Апроксимація функцій має переваги

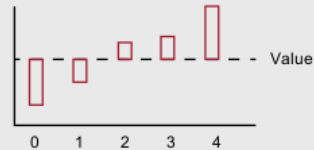


A state-value function with and without function approximation

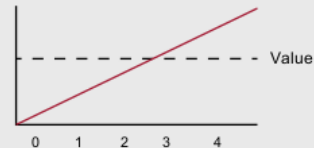
(1) Imagine this state-value function.

$$V = [-2.5, -1.1, 0.7, 3.2, 7.6]$$

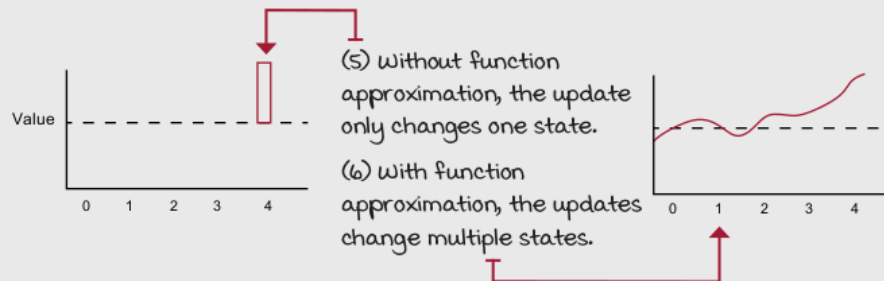
(2) Without function approximation, each value is independent.



(3) With function approximation, the underlying relationship of the states can be learned and exploited.



(4) The benefit of using function approximation is particularly obvious if you imagine these plots after even a single update.



(7) Of course, this is a simplified example, but it helps illustrate what's happening. What would be different in "real" examples?

First, if we approximate an action-value function, Q , we'd have to add another dimension.

Also, with a non-linear function approximator, such as a neural network, more complex relationships can be discovered.

Апроксимація функцій має переваги



BOIL IT DOWN

Reasons for using function approximation

Our motivation for using function approximation isn't only to solve problems that aren't solvable otherwise, but also to solve problems more efficiently.

First decision point: Selecting a value function to approximate

- The state-value function $v(s)$
- The action-value function $q(s, a)$
- The action-advantage function $a(s, a) = q(s, a) - v(s)$

Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

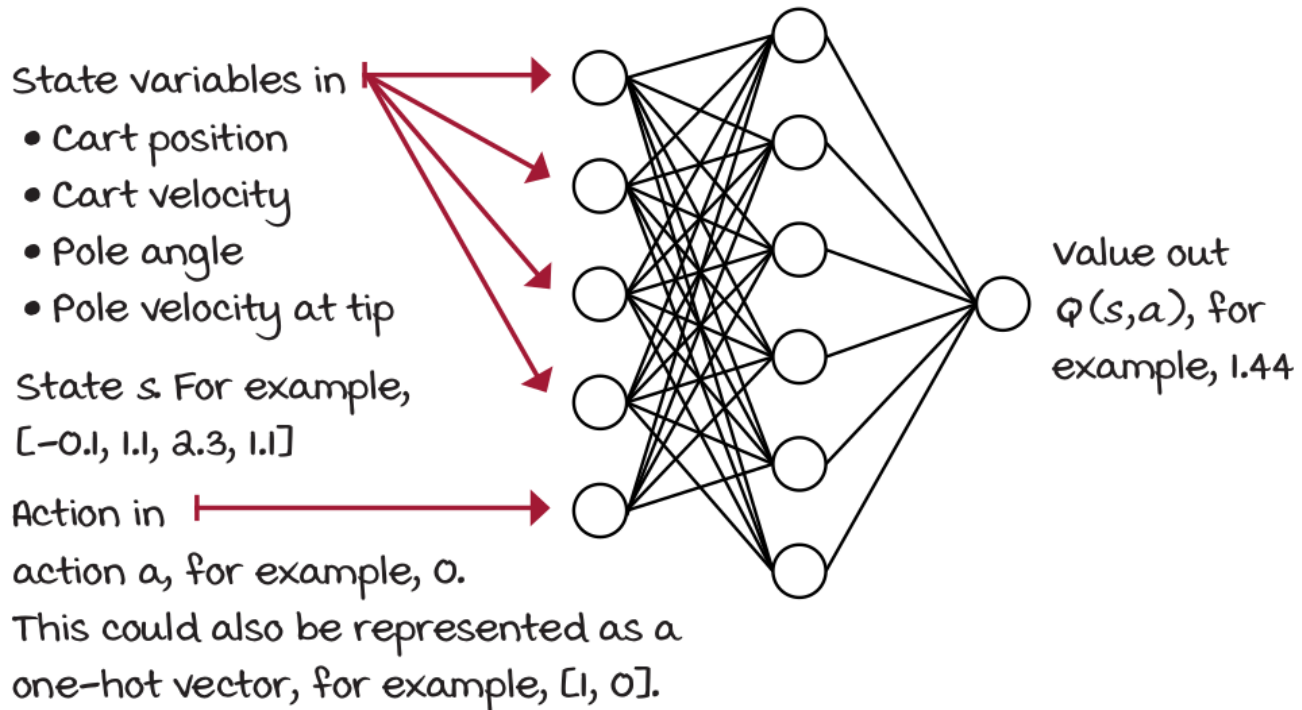
Which Function Approximator?

We consider **differentiable** function approximators, e.g.

- **Linear combinations of features**
- **Neural network**
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

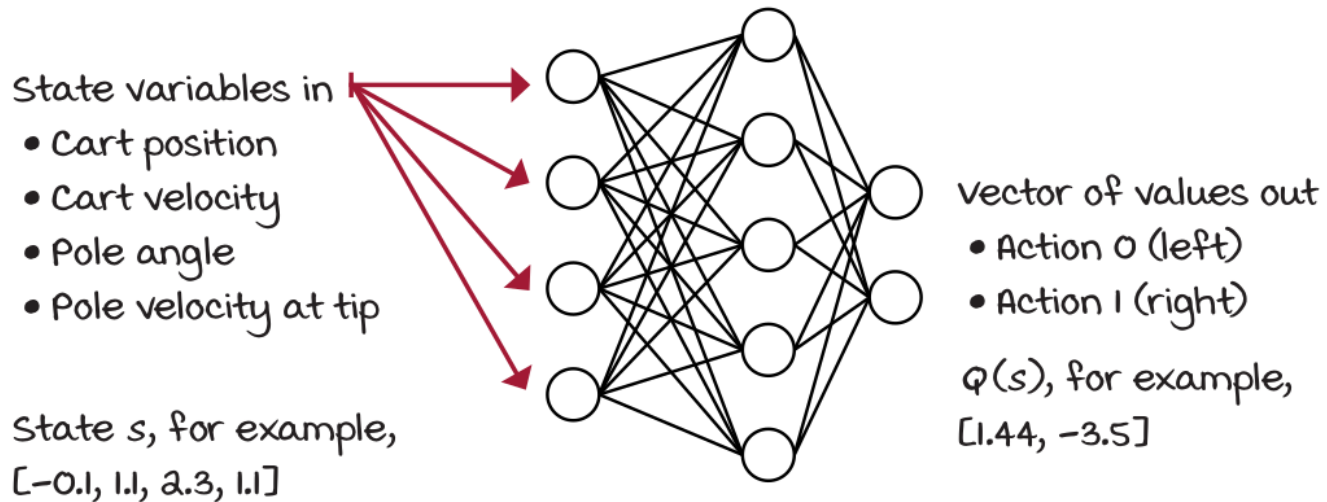
Second decision point: Selecting a neural network architecture

State-action-in-value-out architecture



Second decision point: Selecting a neural network architecture

State-in-values-out architecture



Third decision point: Selecting what to optimize



SHOW ME THE MATH

Ideal objective

(1) An ideal objective in value-based deep reinforcement learning would be to minimize the loss with respect to the optimal action-value function q^* .

(2) We want to have an estimate of q^* , Q , that tracks exactly that optimal function.

$$L_i(\theta_i) = \mathbb{E}_{s,a} \left[\left(q_*(s, a) - Q(s, a; \theta_i) \right)^2 \right]$$

(3) If we had a solid estimate of q^* , we then could use a greedy action with respect to these estimates to get near-optimal behavior—only if we had that q^* .

(4) Obviously, I'm not talking about having access to q^* so that we can use it; otherwise, there's no need for learning. I'm talking about access to sampling the q^* some way: regression-style ML.



REFRESH MY MEMORY

Optimal action-value function

(1) As a reminder, here's the definition of the optimal action-value function.

(2) This is just telling us that the optimal action-value function ...

(3) ... is the policy that gives ...

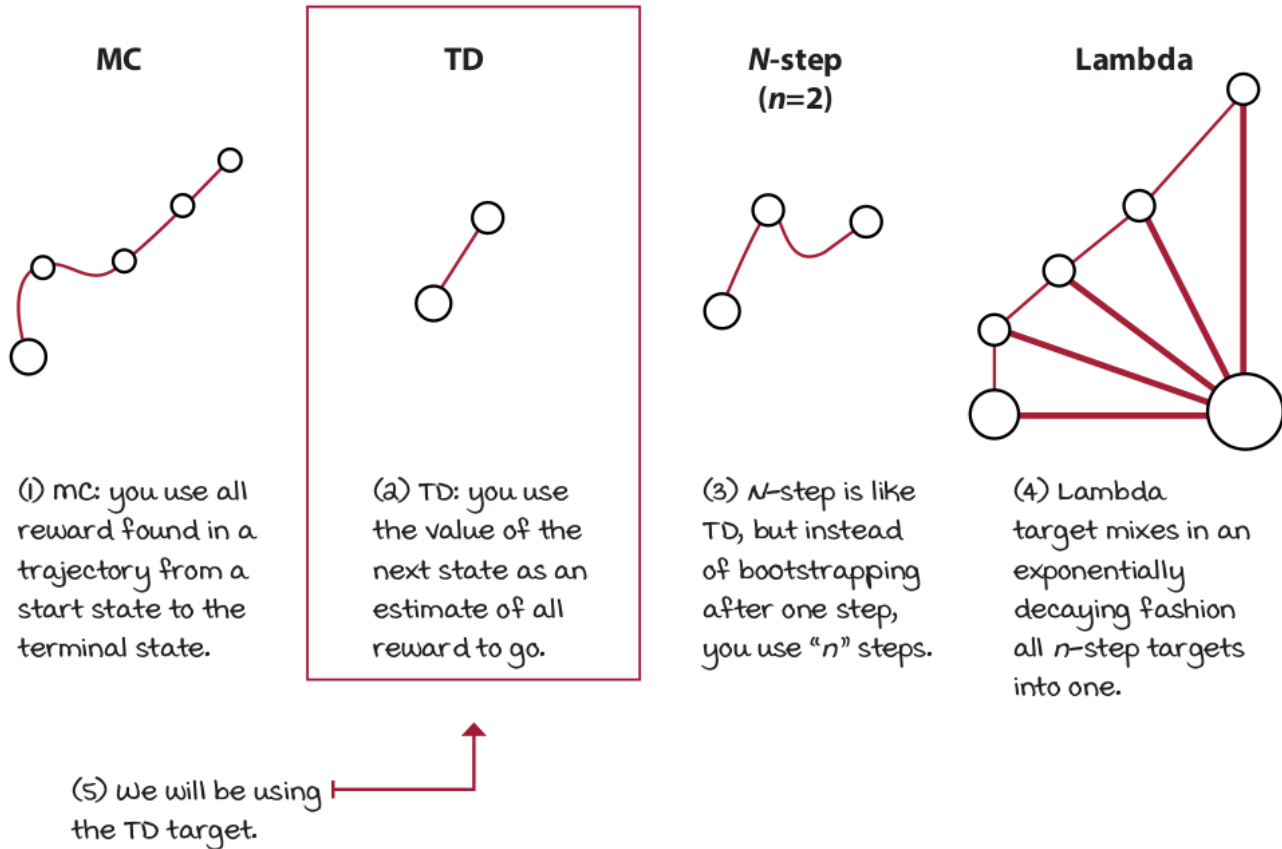
$$q_*(s, a) = \max_{\pi} \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a], \forall s \in S, \forall a \in A(s)$$

(4) ... the maximum expected return ...

(5) ... from each and every action in each and every state.

Fourth decision point: Selecting the targets for policy evaluation

MC, TD, n -step, and lambda targets



Fifth decision point: Selecting an exploration strategy

Another thing we need to decide is which policy improvement step to use for our generalized policy iteration needs.



I SPEAK PYTHON

Epsilon-greedy exploration strategy

```
class EGreedyStrategy():  
    <...>  
    def select_action(self, model, state):  
        with torch.no_grad():  
            q_values = model(state).cpu().detach()  
            q_values = q_values.data.numpy().squeeze()  
            (a) I make the values "NumPy friendly" and remove an extra dimension.  
            if np.random.rand() > self.epsilon:  
                action = np.argmax(q_values) (3) Then, get a random number and, if greater than epsilon, act greedily.  
            else:  
                action = np.random.randint(len(q_values)) (4) Otherwise, act randomly in the number of actions.  
        <...>  
        return action (5) NOTE: I always query the model to calculate stats. But, you shouldn't do that if your goal is performance!
```

Sixth decision point: Selecting a loss function

- L1
- L2, MSE
- ...

Seventh decision point: Selecting an optimization method

- Batch gradient descent
- Mini-batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent vs. momentum

Batch gradient descent

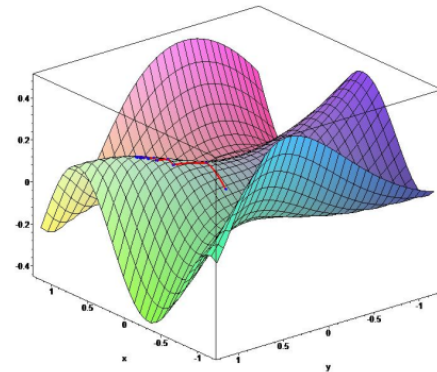
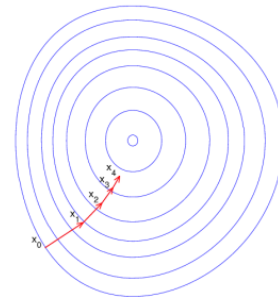
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector \mathbf{w}
- Define the *gradient* of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

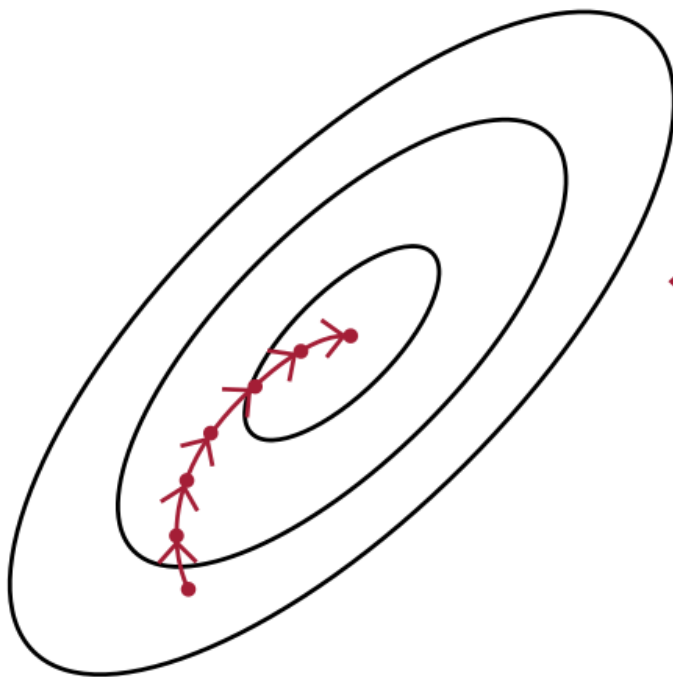
- To find a local minimum of $J(\mathbf{w})$
- Adjust \mathbf{w} in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter

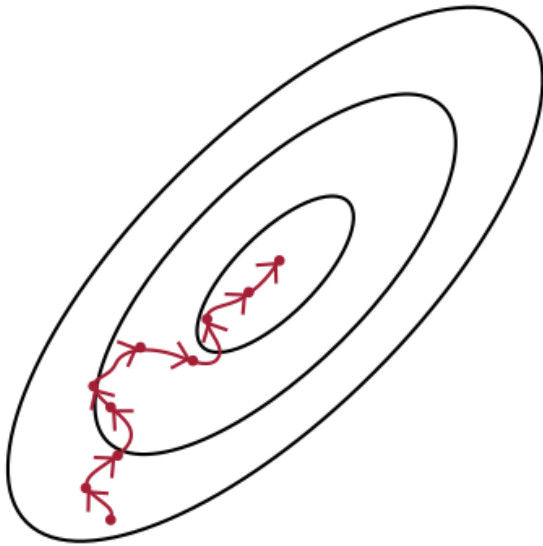


Batch gradient descent



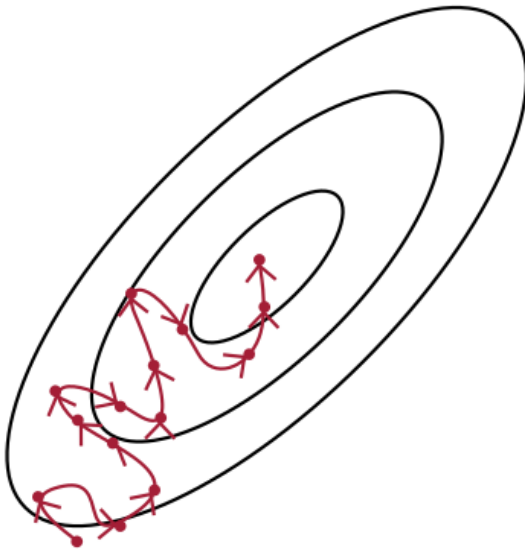
(i) Batch gradient descent goes smoothly toward the target because it uses the entire dataset at once, so lower variance is expected.

Mini-batch gradient descent



- ← (i) In mini-batch gradient descent we use a uniformly sampled mini-batch. This results in noisier updates, but also faster processing of the data.

Stochastic gradient descent

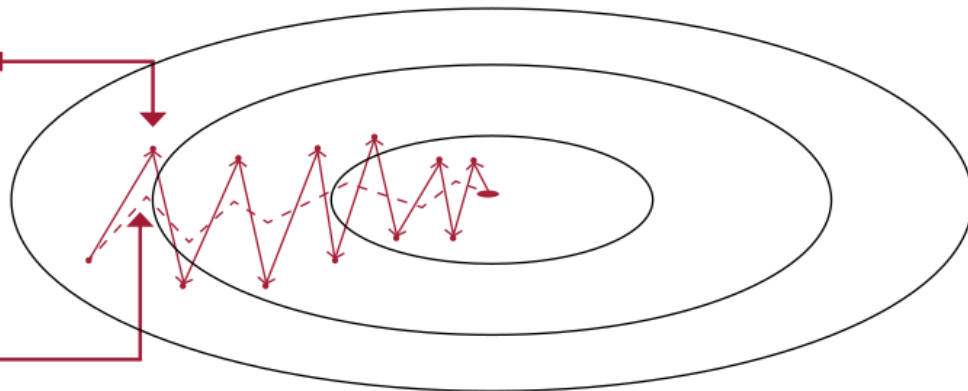


← (i) with stochastic gradient descent, in every iteration we step through only one sample. This makes it a noisy algorithm. It wouldn't be surprising to see several steps taking us further away from the target, and later back toward the target.

Mini-batch gradient descent vs. momentum

(i) mini-batch
gradient descent
from the last image

(a) This would
be momentum.





IT'S IN THE DETAILS

The full neural fitted Q-iteration (NFQ) algorithm

Currently, we've made the following selections:

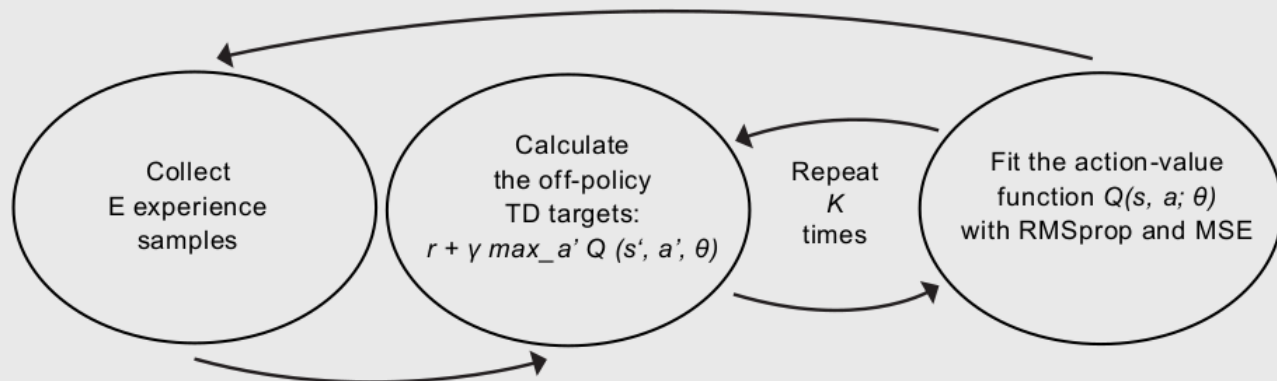
- Approximate the action-value function $Q(s,a; \theta)$.
- Use a state-in-values-out architecture (nodes: 4, 512, 128, 2).
- Optimize the action-value function to approximate the optimal action-value function $q^*(s,a)$.
- Use off-policy TD targets $(r + \gamma \max_{a'} Q(s',a'; \theta))$ to evaluate policies.
- Use an epsilon-greedy strategy (epsilon set to 0.5) to improve policies.
- Use mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0005.

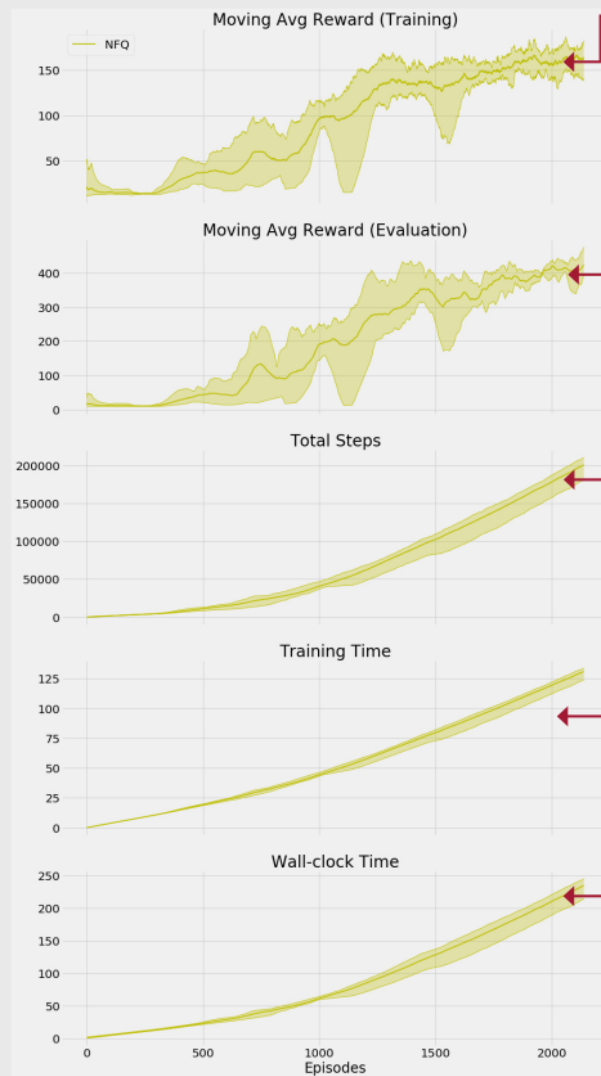
NFQ has three main steps:

1. Collect E experiences: (s, a, r, s', d) tuples. We use 1024 samples.
2. Calculate the off-policy TD targets: $r + \gamma \max_{a'} Q(s', a'; \theta)$.
3. Fit the action-value function $Q(s, a; \theta)$ using MSE and RMSprop.

This algorithm repeats steps 2 and 3 K number of times before going back to step 1. That's what makes it fitted: the nested loop. We'll use 40 fitting steps K .

NFQ





(1) One interesting point is that you can see the training reward never reaches the max of 500 reward per episode. The reason is we're using an epsilon of 0.5. Having such a high exploration rate helps with finding more accurate value functions, but it shows worse performance during training.

(2) On the second figure we plot the mean reward during evaluation steps. The evaluation steps are the best performance we can obtain from the agent.

(3) The main issue with NQF is that it takes too many steps to get decent performance. In other words, in terms of sample efficiency, NQF does poorly. It needs many samples before it gets decent results. It doesn't get the most out of each sample.

(4) The next two plots are related to time. You can see how NQF takes approximately 80 seconds on average to pass the environment. "Training time" is the time excluding evaluation steps, statistics, and so on.

(5) Wall-clock time is how long it takes to run from beginning to end.

Демо

Література

- David Silver, Lecture 6: Value Function Approximation. [[video](#)], [[slides](#)]
- [Reinforcement Learning: An Introduction](#) - Chapter 9: On-policy Prediction with Approximation
- [Reinforcement Learning: An Introduction](#) - Chapter 10: On-policy Control with Approximation
- [Tutorial: Introduction to Reinforcement Learning with Function Approximation](#)
- [Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method](#)

Кінець