ROB-GY 6323
reinforcement learning and optimal control for robotics

Lecture 11
Policy gradient and actor-critic methods

Course material
All necessary material will be posted on Brightspace
Code will be posted on the Github site of the class

https://github.com/righetti/optlearningcontrol

Discussions/Forum with Slack

Contact
ludovic.righetti@nyu.edu
Office hours in person
Wednesday 3pm to 4pm
370 Jay street - room 801

Course Assistant

Armand Jordana
aj2988@nyu.edu
Office hours Monday 1pm to 2pm
Rogers Hall 515

any other time by appointment only

Tentative schedule (subject to change)

Week	Lecture		Homework	Project
1	Intro	Lecture 1: introduction		
2	Trajectory optimization	Lecture 2: Basics of optimization	HW 1	
3		Lecture 3: QPs		
4		Lecture 4: Nonlinear optimal control		
5		Lecture 5: Model-predictive control		
6		Lecture 6: Sampling-based optimal control	HW 2	
7	Policy optimization	Lecture 7: Bellman's principle		
8		Lecture 8: Value iteration / policy iteration		Project 1
9		Lecture 9: Q-learning	HW 3	
10		Lecture 10: Deep Q learning		
11		Lecture 11: Actor-critic algorithms		
12		Lecture 12: Learning by demonstration	HW 4	Project 2
13		Lecture 13: Monte-Carlo Tree Search		
14		Lecture 14: Beyond the class		
15	Finals week			

HW3 is due tonight

Project 1 is due Nov 22nd

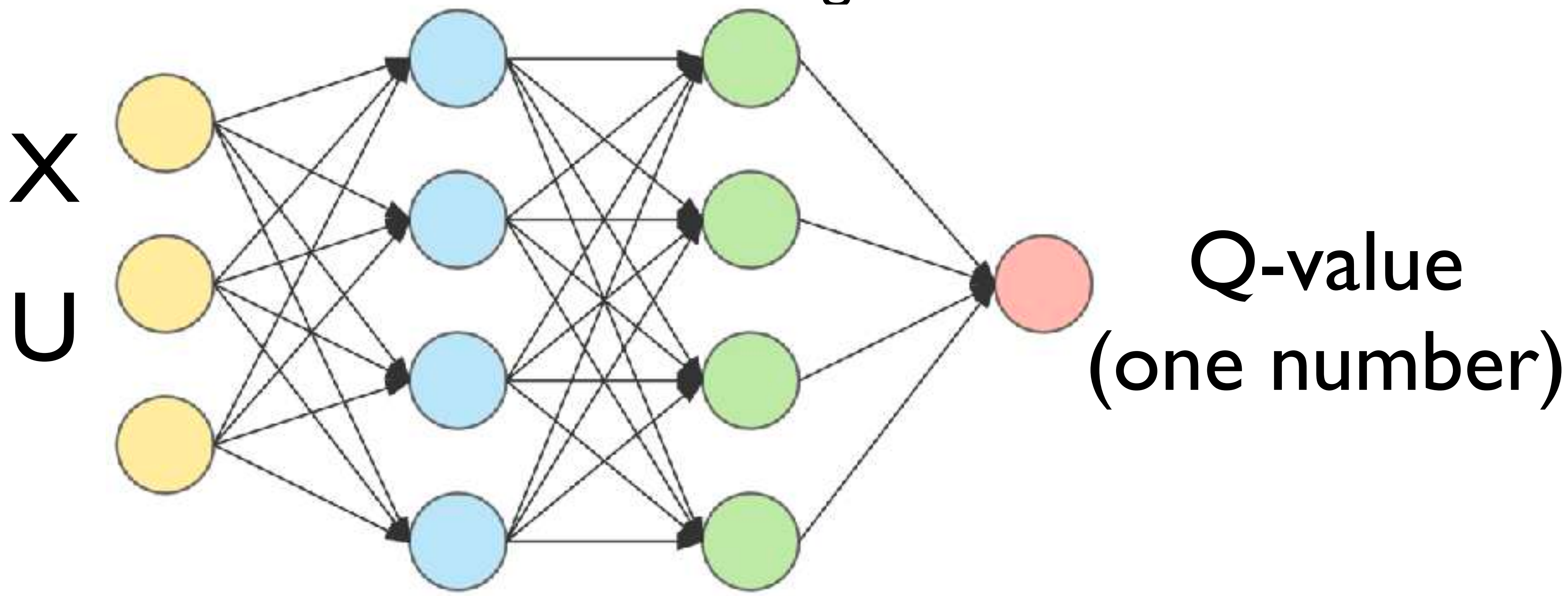
Q-learning with neural networks

Q-learning with a table cannot work for high-dimensional spaces nor for continuous state/action spaces!

<u>Idea</u>: replace the table with a function approximator (e.g. a neural network) - still assume discrete number of actions

$$Q(x,u) \simeq Q(x,u,w)$$

Neural network weights



Q-learning with neural networks

The problem can be written as a least square problem

We can compute the right side of Bellman equation
from data collected during one episode

$$y_t = g(x_t, u_t) + \alpha \min_a Q(x_{t+1}, a, w)$$

and then do one step of gradient descent on the weights
of the neural network to minimize the TD error

$$\min_w ||y_t - Q(x_t, u_t, w)||^2$$

Q-learning with neural networks

Initialize $Q(x, u, w)$ with random weights w

For each episode:

Choose an initial state x_0

Loop for each step of the episode:

Choose an action u_t using an ϵ-greedy policy from Q

Observe the next state x_{t+1}

Compute $y_t = g(x_t, u_t) + \alpha \min_a Q(x_{t+1}, a, w)$

Update the weights of the neural network by doing one iteration of stochastic gradient descent

$$\min_w ||y_t - Q(x_t, u_t, w)||^2$$

Deep Q-network (DQN)

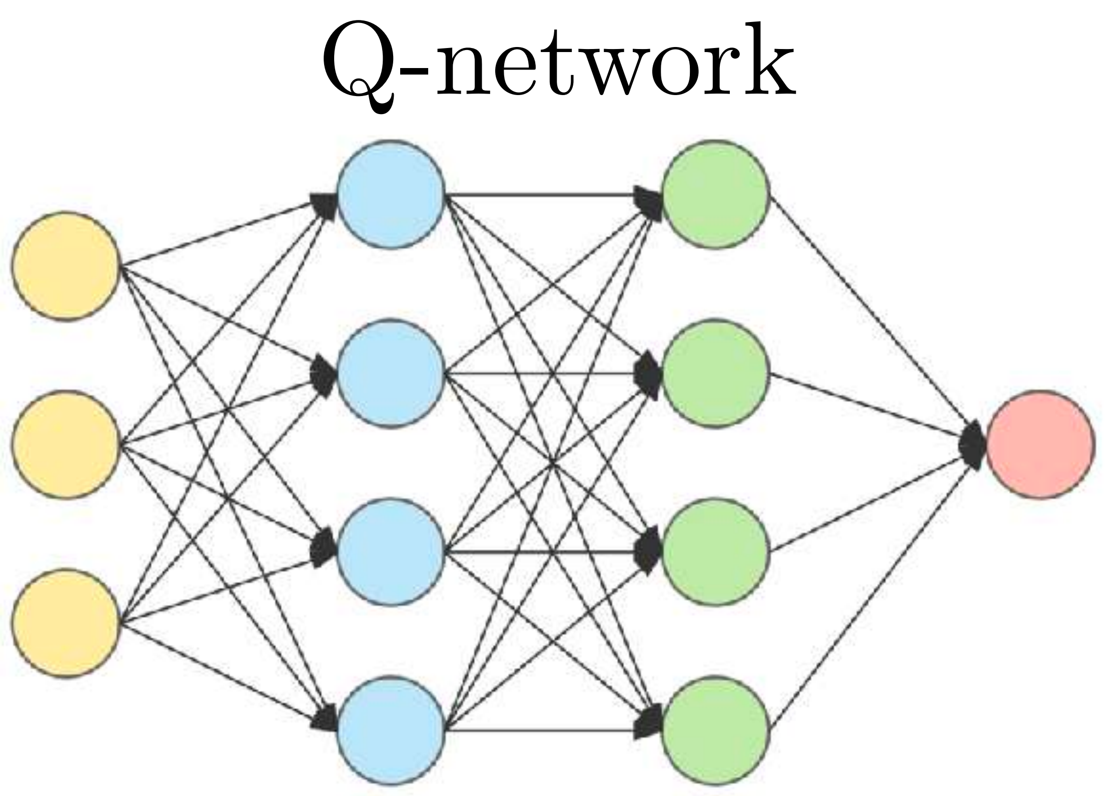
[Mnih et al., Nature, 2015]

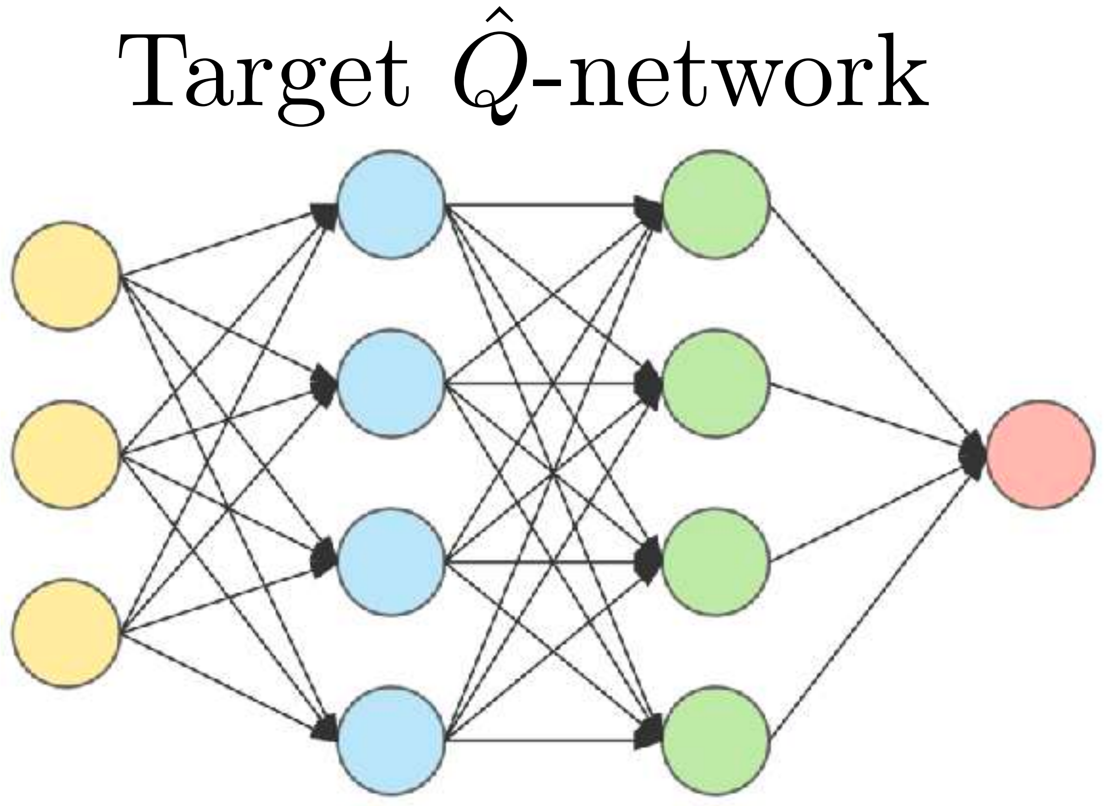
<u>Problem</u>: a direct (naive) approach using solely current episode data tend to be unstable (i.e. it diverges):

- The sequence of observations are correlated
- Small changes in Q can lead to large changes in policy

<u>Solution 1)</u>
Use a "replay" memory of a previous samples from which we randomly sample the next training batch (remove correlations)

<u>Solution 2)</u>
Use 2 Q-networks to avoid correlations due to updates





Deep Q-network (DQN)

[Mnih et al., Nature, 2015]

Initialize replay memory D of size N

Initialize Q-network with random weights θ

Initialize target $\hat{Q}$ function with weights $\theta^- = \theta$

For each episode:

Start from an initial state x_0

Loop for each step t of the episode:

Choose a control action u_t using Q (e.g. ϵ-greedy policy)

Do u_t and observe the next state x_{t+1}

Compute $y_t = g(x_t, u_t) + \alpha \min_a \hat{Q}(x_{t+1}, a, \theta^-)$ ← ! here we use the target network

Store (x_t, u_t, y_t, x_{t+1}) in memory D

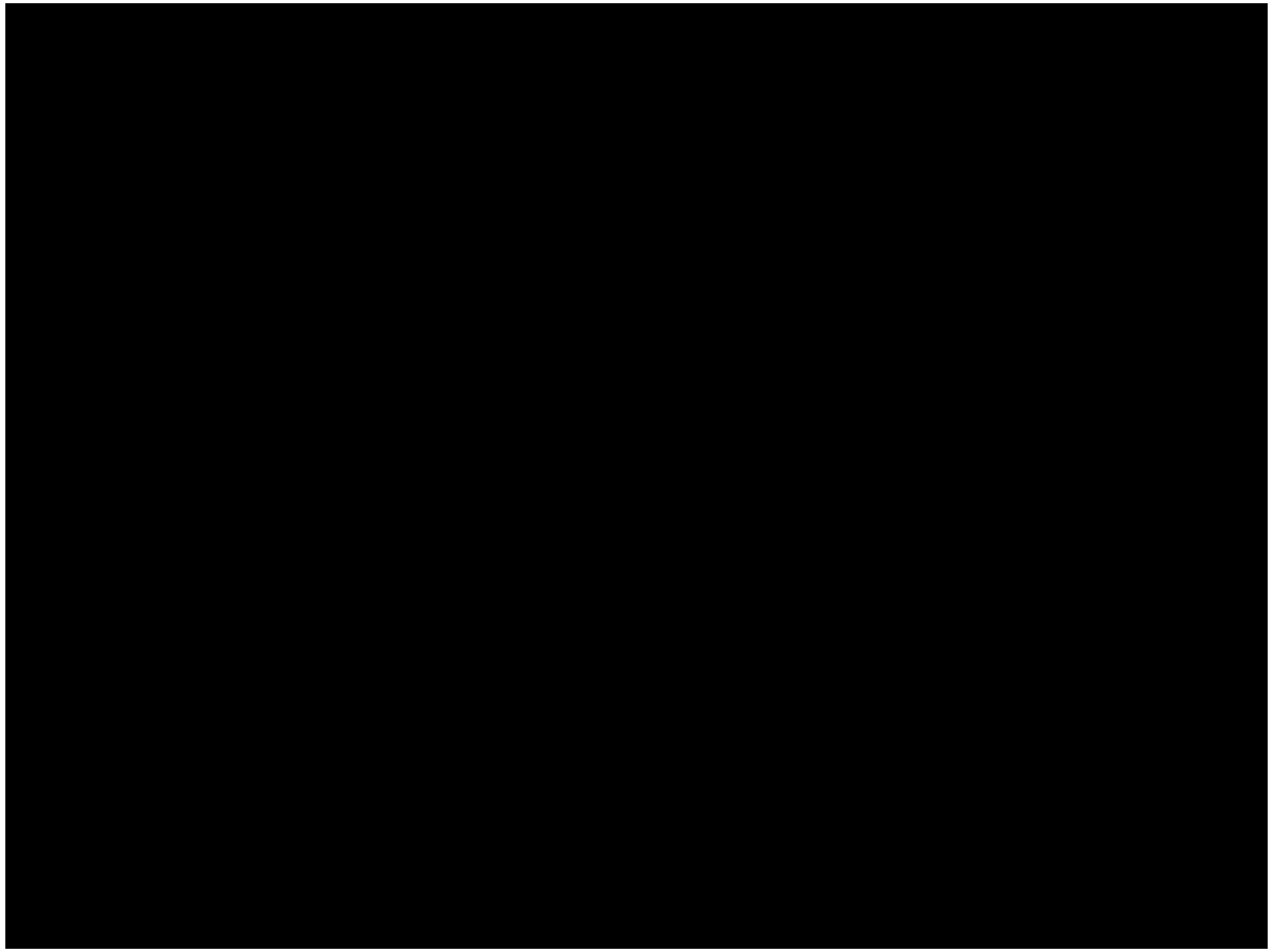
Sample minibatch K of transitions (x_k, u_k, y_k, x_{k+1}) from D

Gradient descent on θ to minimize $\sum_K ||Q(x_k, u_k, \theta) - y_k||^2$

Every C steps reset the target network by setting $\theta^- = \theta$

Deep Q-network (DQN)

[Mnih et al., Nature, 2015]



Our results so far in Reinforcement Learning:

- TD(0)
  => evaluate the value function of a policy
  => a policy update step as in the policy iteration algorithm is necessary to improve policies
  $$\mu_{k+1} = \arg\min_u g(x,u) + \alpha J_{\mu_k}(f(x,u))$$

- Q-learning
  => learn the Q function
  => min over control to find the optimal policy

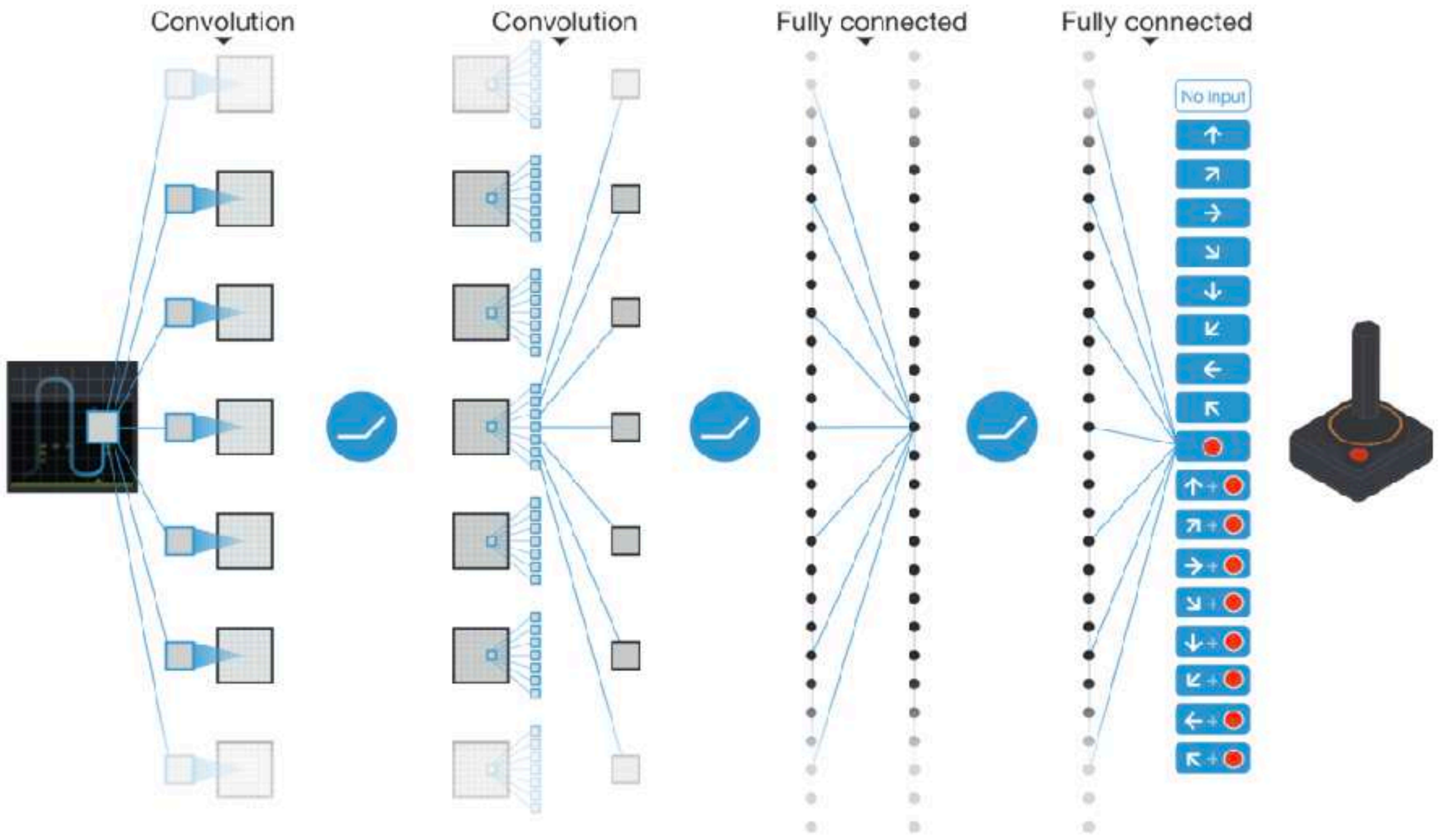
  => replace tables with function approximators (NN)

These methods "learn" value functions <u>then</u> compute a policy - they are called <u>value-based approaches</u>

Can we learn <u>directly</u> the policy?

Now we can do Q-learning using continuous states and high dimensional inputs!

What about a continuous action space?



What about continuous action space?

Problem: we need to evaluate the min to be able to do Q-learning with a function approximator

$$||Q(x_t, u_t, \theta) - g(x_t, u_t) - \alpha \min_a \hat{Q}(x_{t+1}, a, \theta^-)||2$$

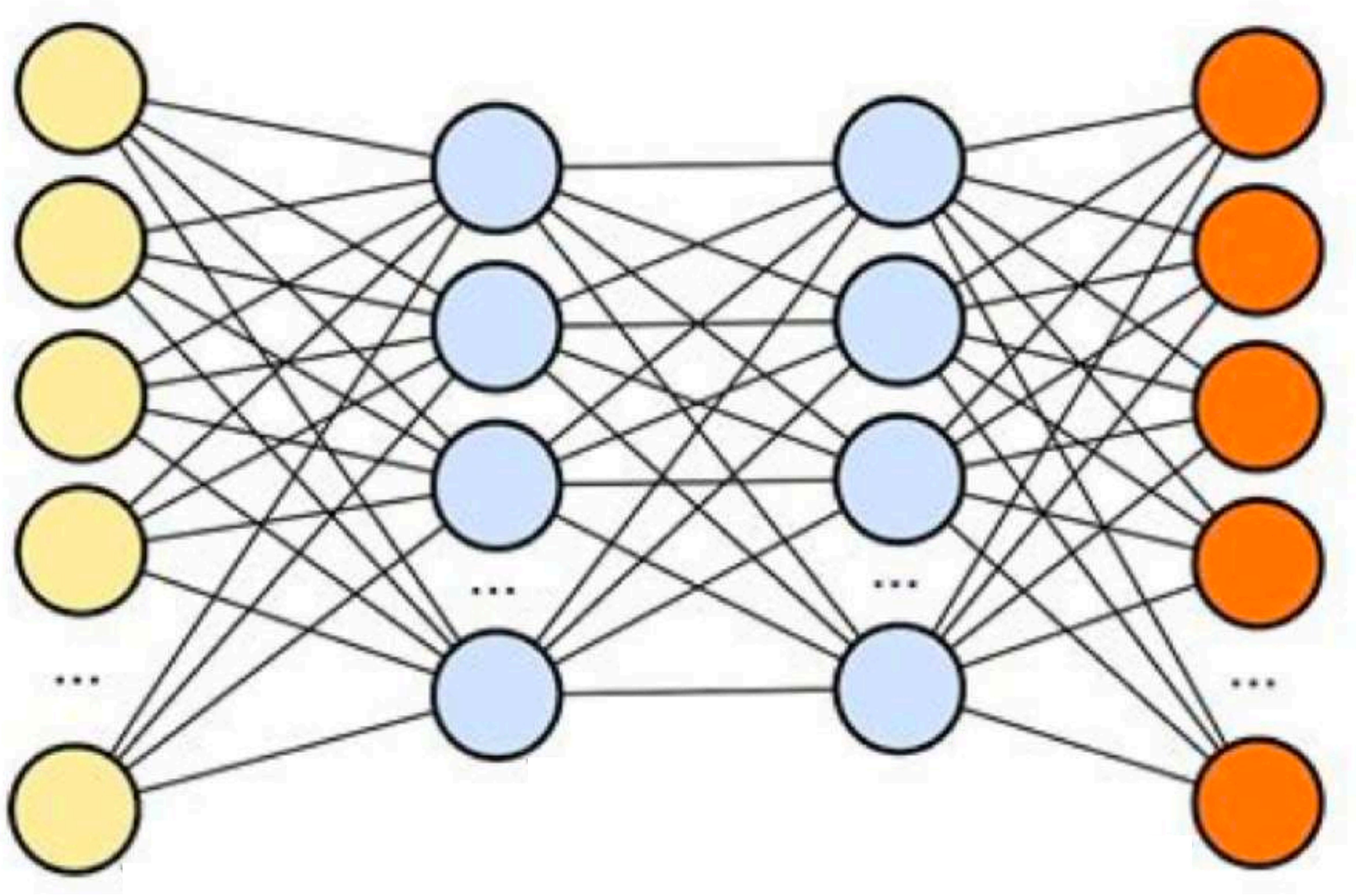
Solution: use another neural network to approximate the min operator (i.e. to approximate the optimal policy)

A primer on actor-critic algorithms
Deep Deterministic Policy Gradient

Back to DQN with "policy gradient"

Let $\pi(S_t, \theta^\pi)$ an approximation of a policy with a NN (weights θ^π)

Input X



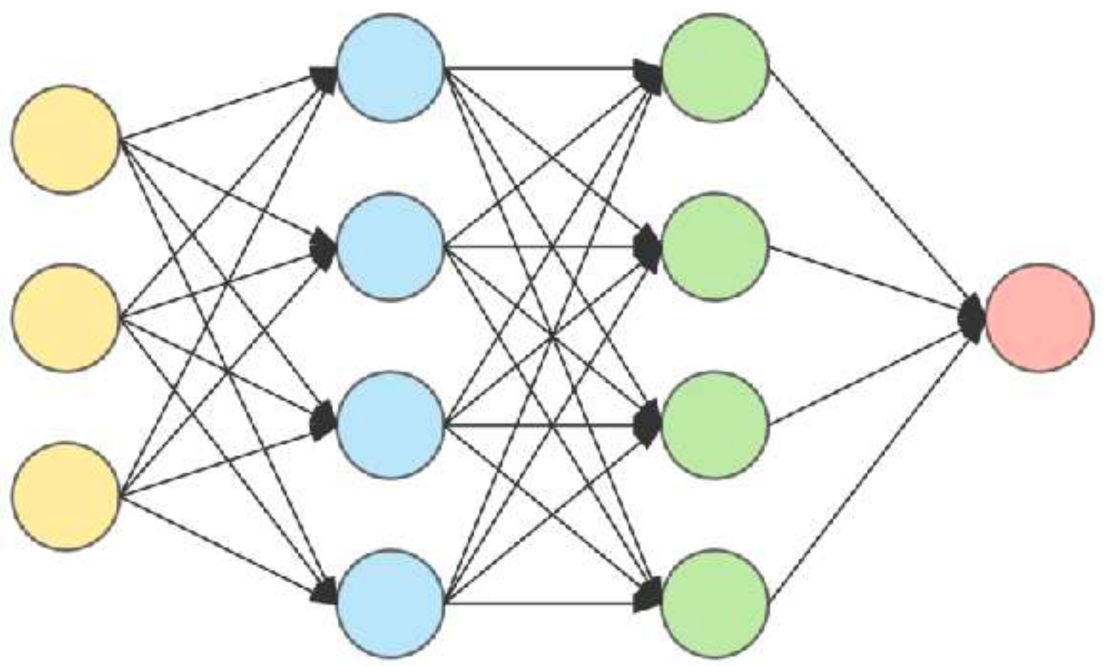
Output U

Deep Deterministic Policy Gradient (DDPG)

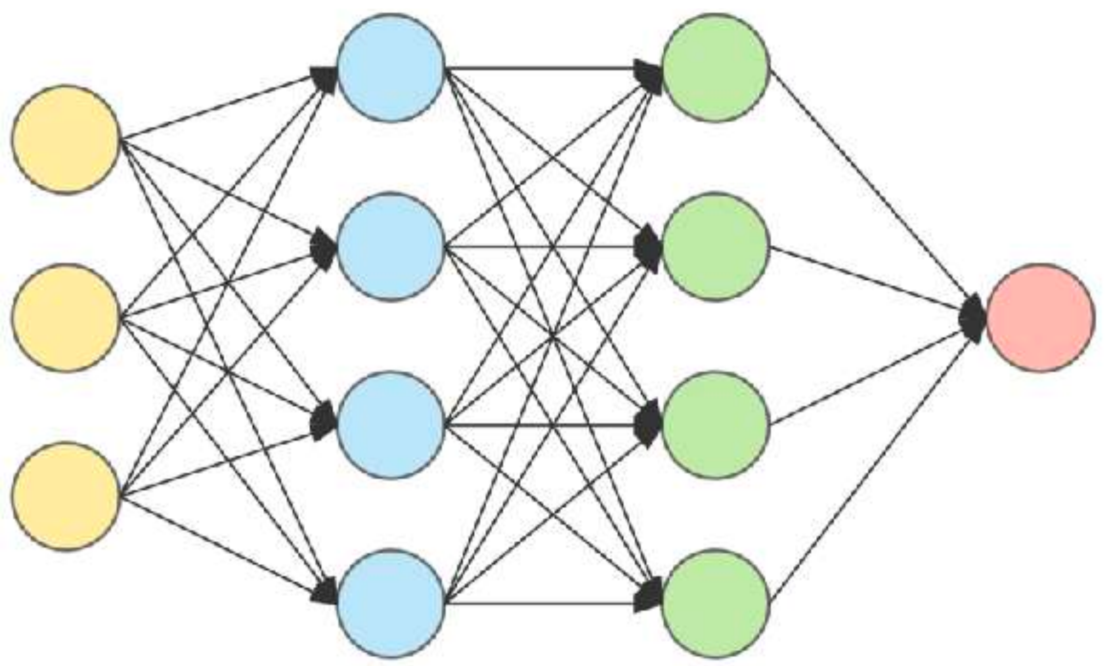
[Lillicrap et al., ICML, 2016]

Policy network (actor) - Q-network (critic)
DDPG => Same as DQN + policy network

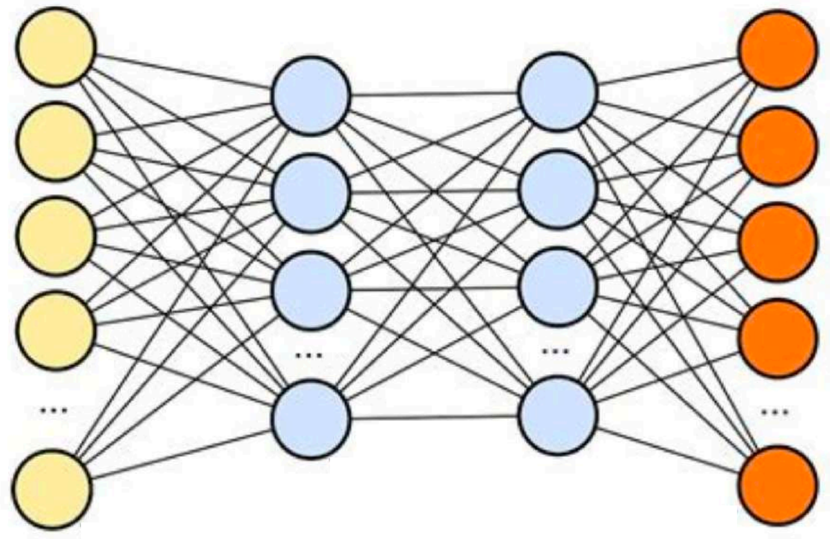
Q-network



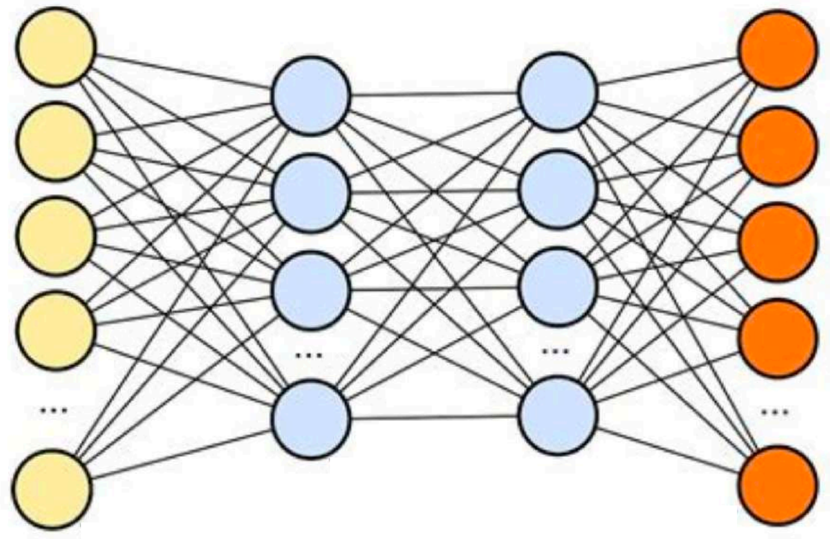
Target $\hat{Q}$-network



Policy network



Target policy network



DDPG

[Lillicrap et al., ICML, 2016]

Initialization

Initialize replay memory D of size N

Initialize the weights of the action-value Q_θ and policy π_ϕ networks

Set the weights $\theta_{target} = \theta$ and $\phi_{target} = \phi$ of the target networks $Q_{\theta_{target}}$ and $\pi_{\phi_{target}}$

For each episode

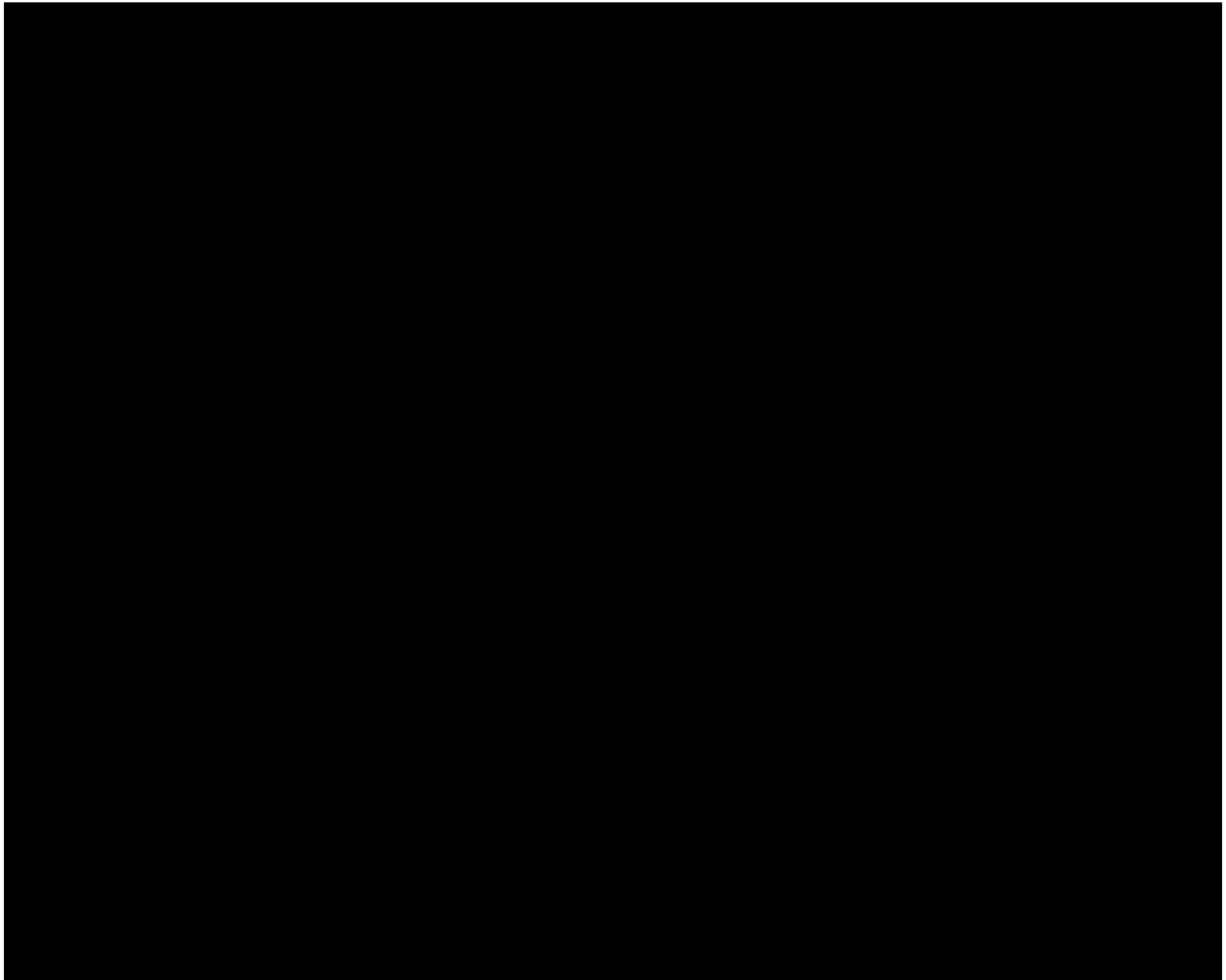
Start from an initial state x_0

Loop for each step t of the episode:

- Choose $u_t = \pi_\phi(x_t) + noise$ (to explore)
- Apply u_t and get the next state x_{t+1}
- Compute the instantaneous cost $c_t = g(x_t, u_t)$
- Store (x_t, u_t, c_t, x_{t+1}) in the replay memory D
- Every few iterations update the networks:
  - Sample minibatch of B elements in replay memory D
  - <u>Improve Q</u>: gradient descent on θ to minimize $\frac{1}{B}\sum_{i=0}^{B}\left(Q_\theta(x_i, u_i) - c_i - \alpha Q_{\theta_{target}}(x_{i+1}, \pi_{\phi_{target}}(x_{i+1}))\right)^2$
  - <u>Improve policy</u>: gradient descent on ϕ to minimize $\frac{1}{B}\sum_{i=0}^{B} Q_\theta(x_i, \pi_\phi(x_i))$
  - Update the target networks: $\begin{array}{l}\theta_{target} \leftarrow \tau\theta + (1-\tau)\theta_{target} \\ \phi_{target} \leftarrow \tau\phi + (1-\tau)\phi_{target}\end{array}$

DDPG

[Lillicrap et al., ICML, 2016]



DDPG is an actor-critic methods

It uses the Q-function to optimize a policy directly

Question Can we directly compute the policy without knowing the Q- or value functions?

Answer Yes! for example using policy gradients

Policy gradient methods

Assume that we have a parametrized policy $u = \pi(x, \theta)$

Can we find a relation between the policy parameters θ
and the associated performance? e.g. find $J(\theta) = V_\pi(x_0)$?

Can we find the gradient $\frac{\partial}{\partial \theta} J(\theta) = \nabla J(\theta)$?

With the gradient, we can improve the policy with gradient descent

$$\theta \leftarrow \theta - \gamma \nabla J(\theta)$$

Policy gradient methods

Historically policy gradient methods have been first derived using stochastic policies

Recently policy gradient algorithms for deterministic policies are also used
(this led to the original DDPG paper)

Stochastic policies

We will derive the policy gradient for stochastic policies

Let's assume a stochastic policy $\pi(u|x,\theta) = \Pr\{u_t = u | x_t = x, \theta\}$

Stochastic policy: example 1

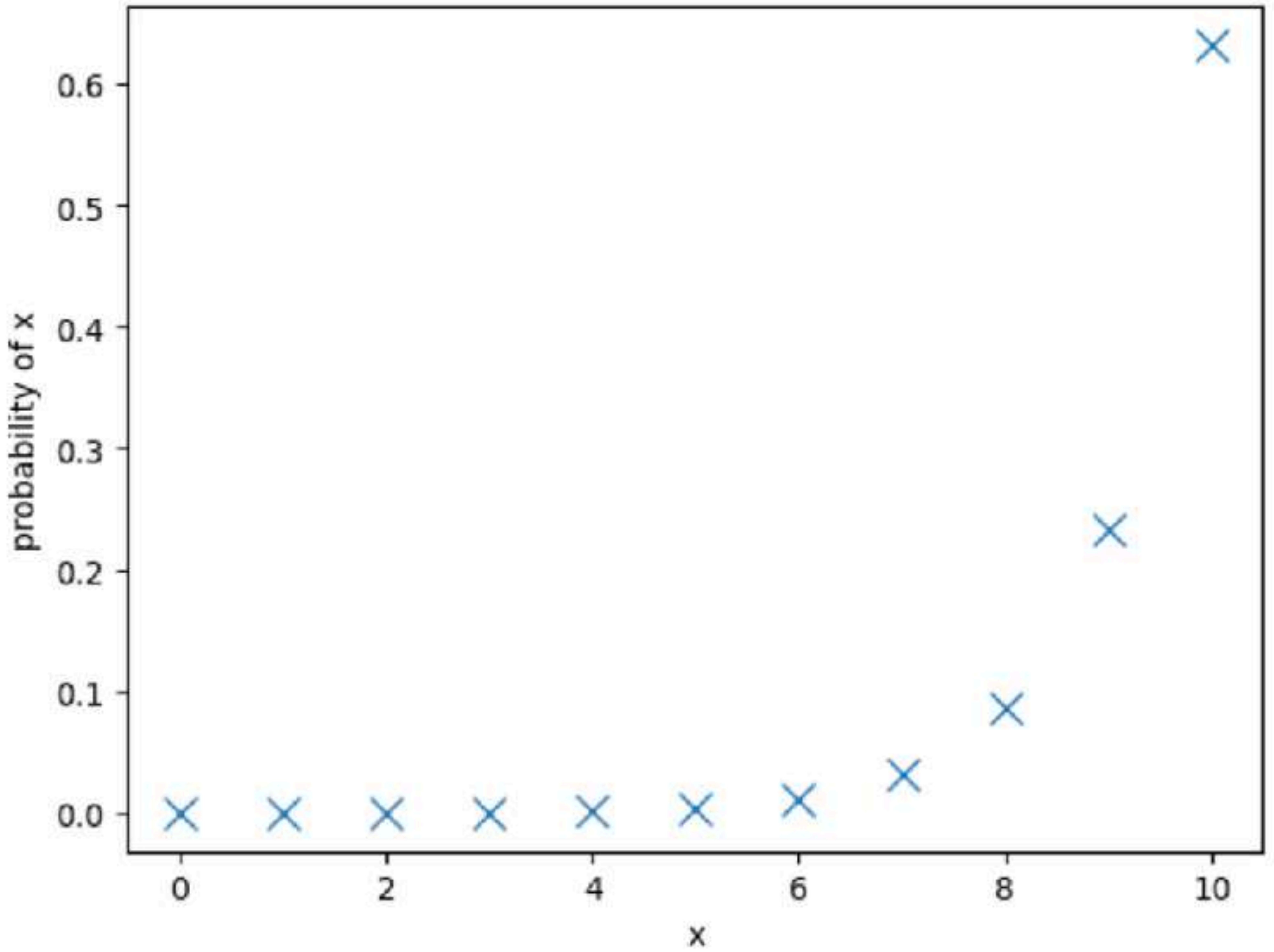
ϵ-greedy policies are stochastic

$$u_t = \begin{cases} \arg\min_u Q(x_t, u) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

The softmax function

Given a vector $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ the softmax function returns a vector of probability distribution where the highest entries in x have the highest probability. Each entry i of the returned vector is $\frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{N} \mathrm{e}^{x_j}}$

```python
import numpy as np

def softmax(x):
    x_exp = np.exp(x)
    return x_exp/np.sum(x_exp)
```



Stochastic policy: example 2

Exponential soft-max distributions: $\pi(u|x,\theta) = \dfrac{e^{h(x,u,\theta)}}{\sum_a e^{h(x,a,\theta)}}$

where $h(x,u,\theta)$ reflects preferences for each state-action pair

Assume we have three control $u = -1,\ 0,\ or\ 1$ and that $x = 0$

If $h(0,-1) = 1$, $h(0,0) = 5$ and $h(0,1) = 0$, we have

Stochastic policy: example 2

Exponential soft-max distributions: $\pi(u|x,\theta) = \dfrac{e^{h(x,u,\theta)}}{\sum_a e^{h(x,a,\theta)}}$

where $h(x,u,\theta)$ reflects preferences for each state-action pair

Assume we have three control $u = -1,\ 0,\ or\ 1$ and that $x = 0$

If $h(0,-1) = 1$, $h(0,0) = 5$ and $h(0,1) = 0$, we have

$$\pi(u=-1|x=0) = \frac{e^1}{e^1 + e^5 + e^0} \simeq 0.018$$

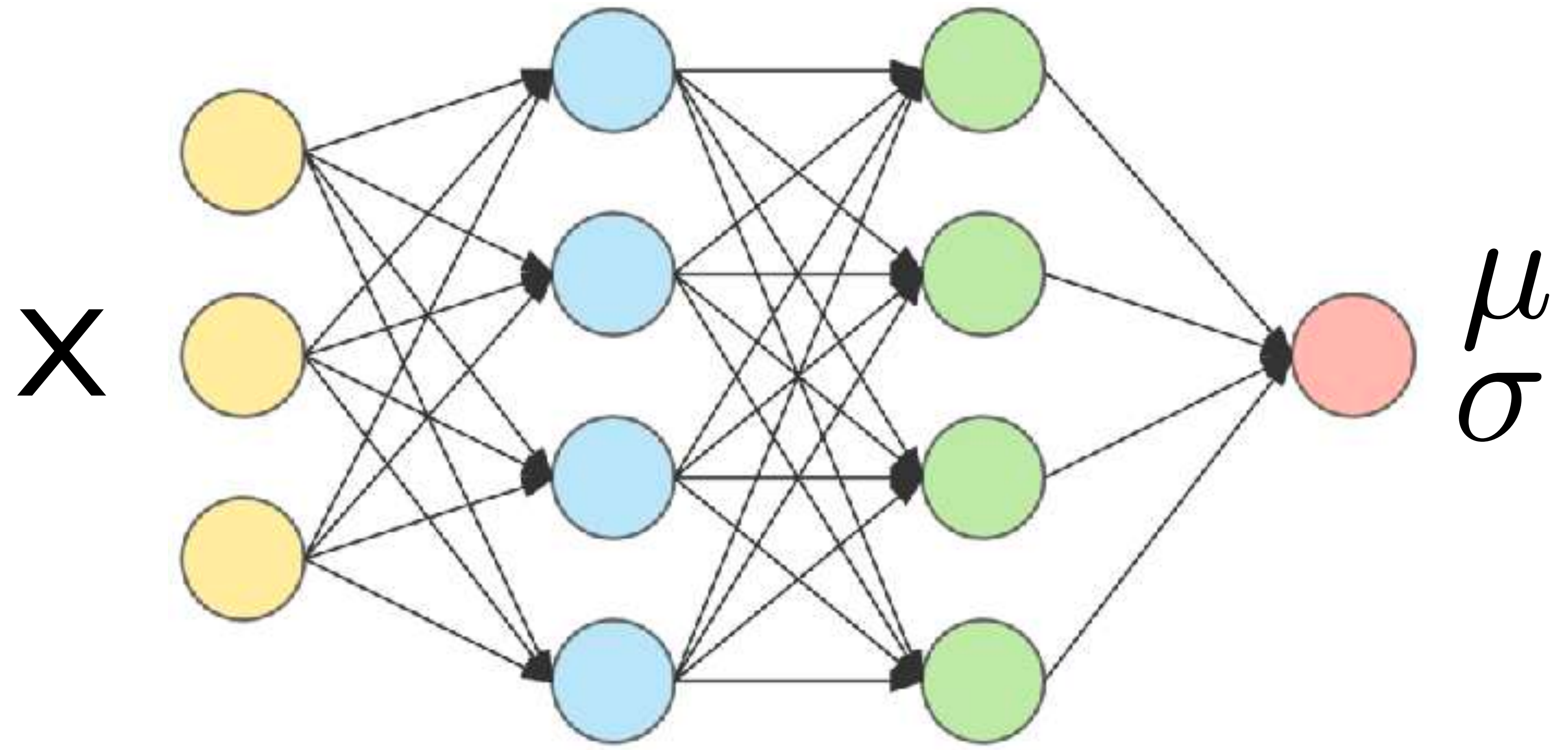
$$\pi(u=0|x=0) = \frac{e^5}{e^1 + e^5 + e^0} \simeq 0.976$$

$$\pi(u=1|x=0) = \frac{e^0}{e^1 + e^5 + e^0} \simeq 0.006$$

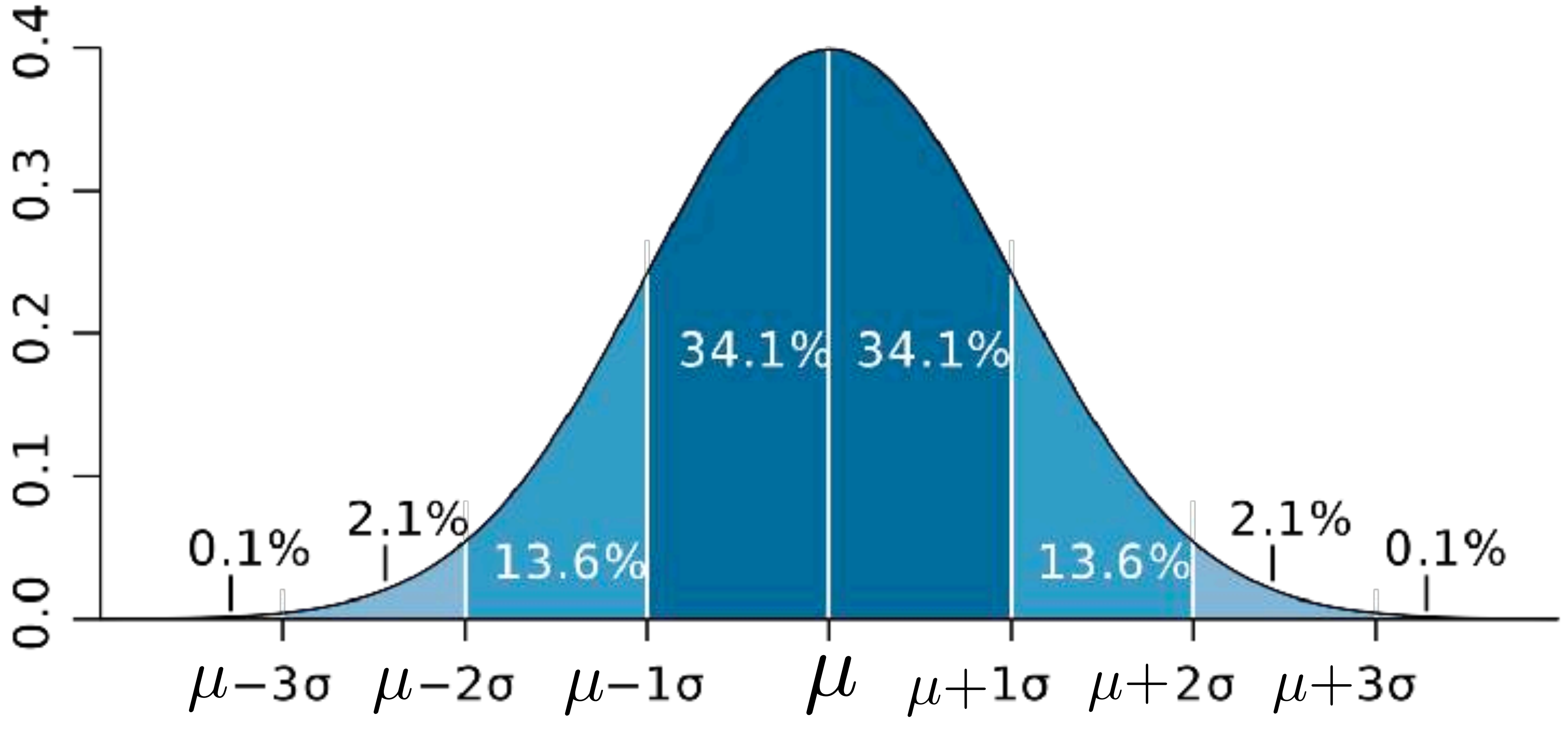
we see that $\pi(-1|0)+\pi(0|0)+\pi(1|0) = 1$, a probability distribution

Stochastic policy: example 3

Gaussian policies parametrized by a neural network



$$u \sim \mathcal{N}(\mu, \sigma^2)$$



Policy gradient theorem

Let's define $J(\theta) = \mathbb{E}_{u_n \sim \pi_\theta}\left[\sum_{n=0}^{N} \alpha^n g(x_n, u_n)\right]$

Evaluating the policy gradient (Monte-Carlo)

REINFORCE

[Williams, 1992]

Initialize the policy parameters θ for an input policy $\pi(u|x,\theta)$

Choose a step size γ (using discount factor α)

Loop forever (for each episode):

Generate an episode $x_0, u_0, x_1, u_1, \cdots, x_N, u_N$ following π

For each step t of the episode

$$G_t = \sum_{k=t}^{T} \alpha^k g(x_k, u_k)$$

$$\theta \leftarrow \theta - \gamma G_t \nabla_\theta \left[\ln \pi(u_t|x_t,\theta)\right]$$

Policy gradients with baseline

Taking the expectation of the cost can lead to very high variance in the gradient
=> makes learning very difficult

It is often a good idea to shift the cost by a state dependent "baseline"

$$\mathbb{E}_{u_n \sim \pi_\theta} \left[\sum_{n=0}^{N} \alpha^n (g(x_n, u_n) - b(x_n)) \right]$$

Since the baseline does not depend on the control, the policy gradient remains the same (swapping g(x,u) by g(x,u)-b(x))

A good baseline is using an estimate of the value function v(x)
In this case we measure the "advantage" of the policy with respect to v(x)

REINFORCE with baseline

[Williams, 1992]

Replace $\theta \leftarrow \theta - \gamma G_t \nabla_\theta \left[\ln \pi(u_t | x_t, \theta) \right]$

with $\theta \leftarrow \theta - \gamma \Big(G_t - b(x) \Big) \cdot \nabla_\theta \left[\ln \pi(u_t | x_t, \theta) \right]$

where for example b(x) is an approximation of the value function
(this can help normalize the gradient step)

REINFORCE with baseline [Williams, 1992]

Initialize parameters θ_V for value function $V(x, \theta_V)$

Initialize parameters θ_π for policy function $\pi(u|x, \theta_\pi)$

Choose step sizes $\gamma_\pi > 0$ and $\gamma_V > 0$

Loop forever (for each episode):

Generate an episode $x_0, u_0, x_1, u_1, \cdots, x_N, u_N$ following π

For each step t of the episode

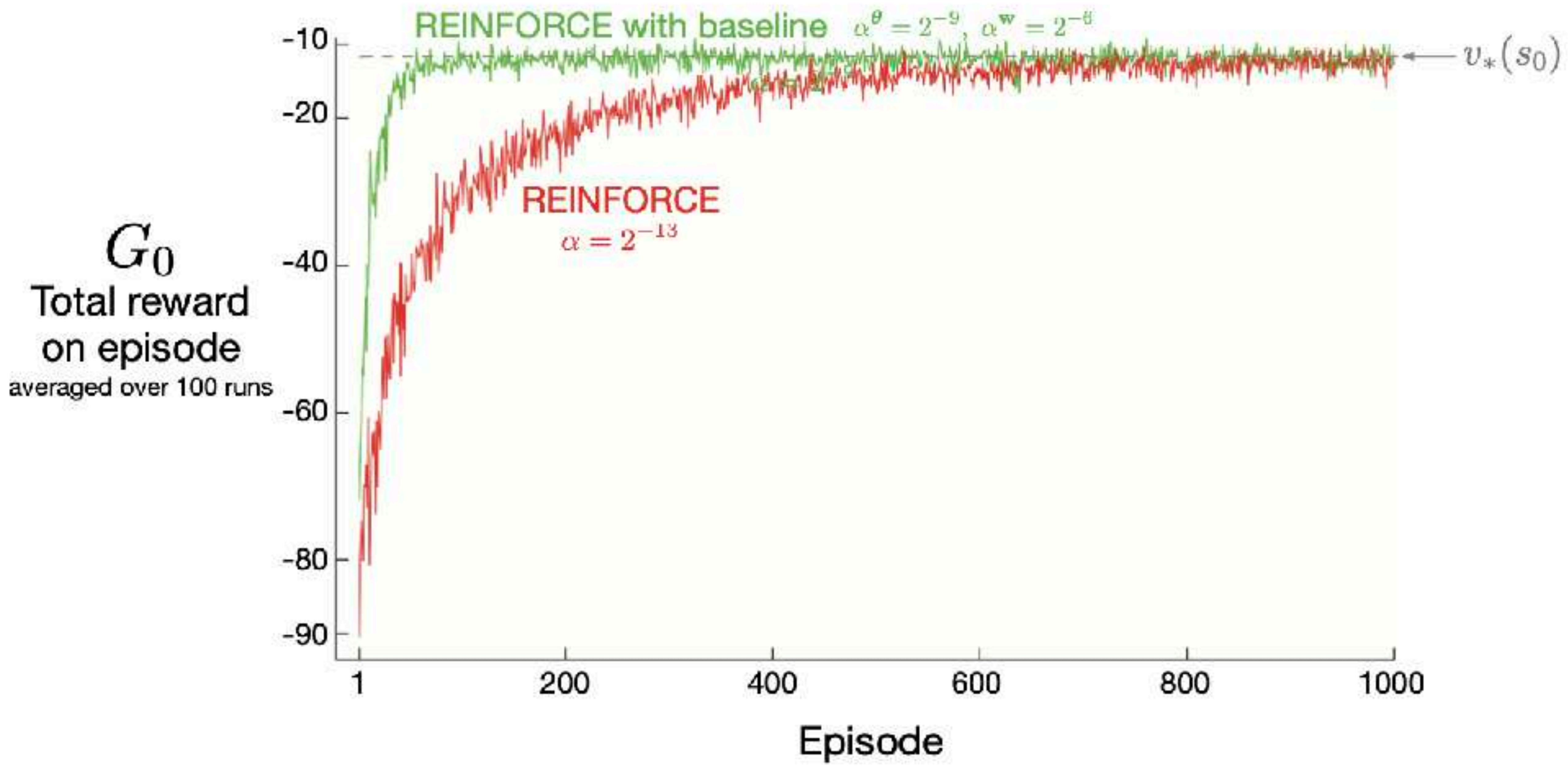
$$G_t = \sum_{k=t}^{T} \alpha^k g(x_k, u_k)$$

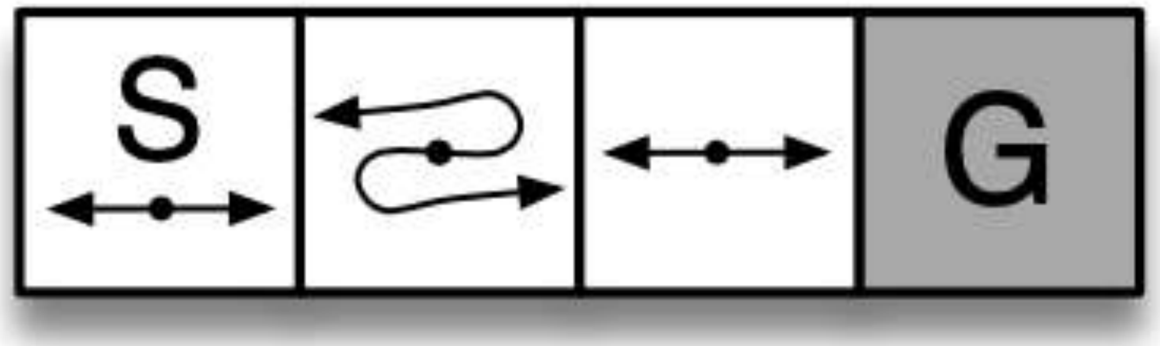
$$\theta_V \leftarrow \theta_V - \gamma_V \Big(V(x_t) - G_t \Big) \cdot \nabla_{\theta_V} V(x_t, \theta_V)$$

$$\theta_\pi \leftarrow \theta_\pi - \gamma_\pi \Big(G_t - V(x_t) \Big) \cdot \nabla_{\theta_\pi} \left[\ln \pi(u_t|x_t, \theta_\pi) \right]$$

REINFORCE with baseline

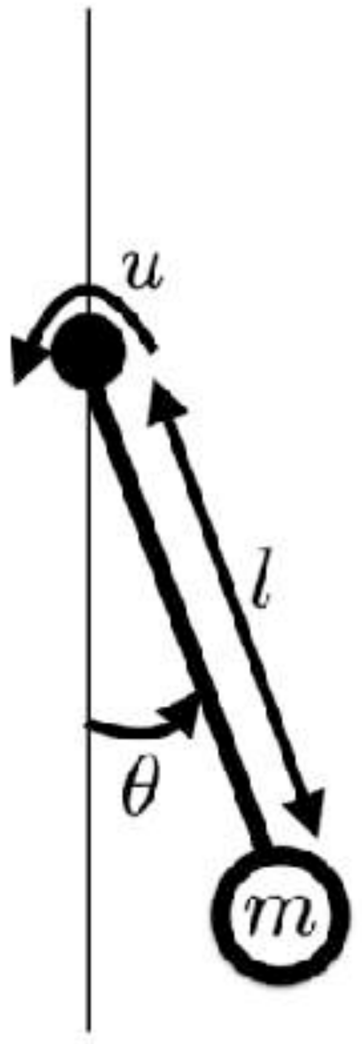
[Williams, 1992]





REINFORCE

$$\min \sum_{i=0}^{N} \alpha^i g(\theta_i, \omega_i, u_i)$$



$$g(x, v, u) = (x - \pi)^2 + 0.01v^2 + 0.00001u^2$$

$$u = [-5,\ 0,\ 5]$$

Softmax stochastic policy

$$\pi(u|x, \theta) = \frac{e^{h(x,u,\theta_\pi)}}{\sum_a e^{h(x,a,\theta)}}$$

$$h(x, u, \theta_\pi) = \theta_\pi^T \Psi(x, u)$$

$$\psi_{k,l,c,0}(\theta, \omega, u) = \frac{e^{-\frac{(u-u_c)^2}{0.002}}}{\sqrt{2\pi 0.001}} \cos(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

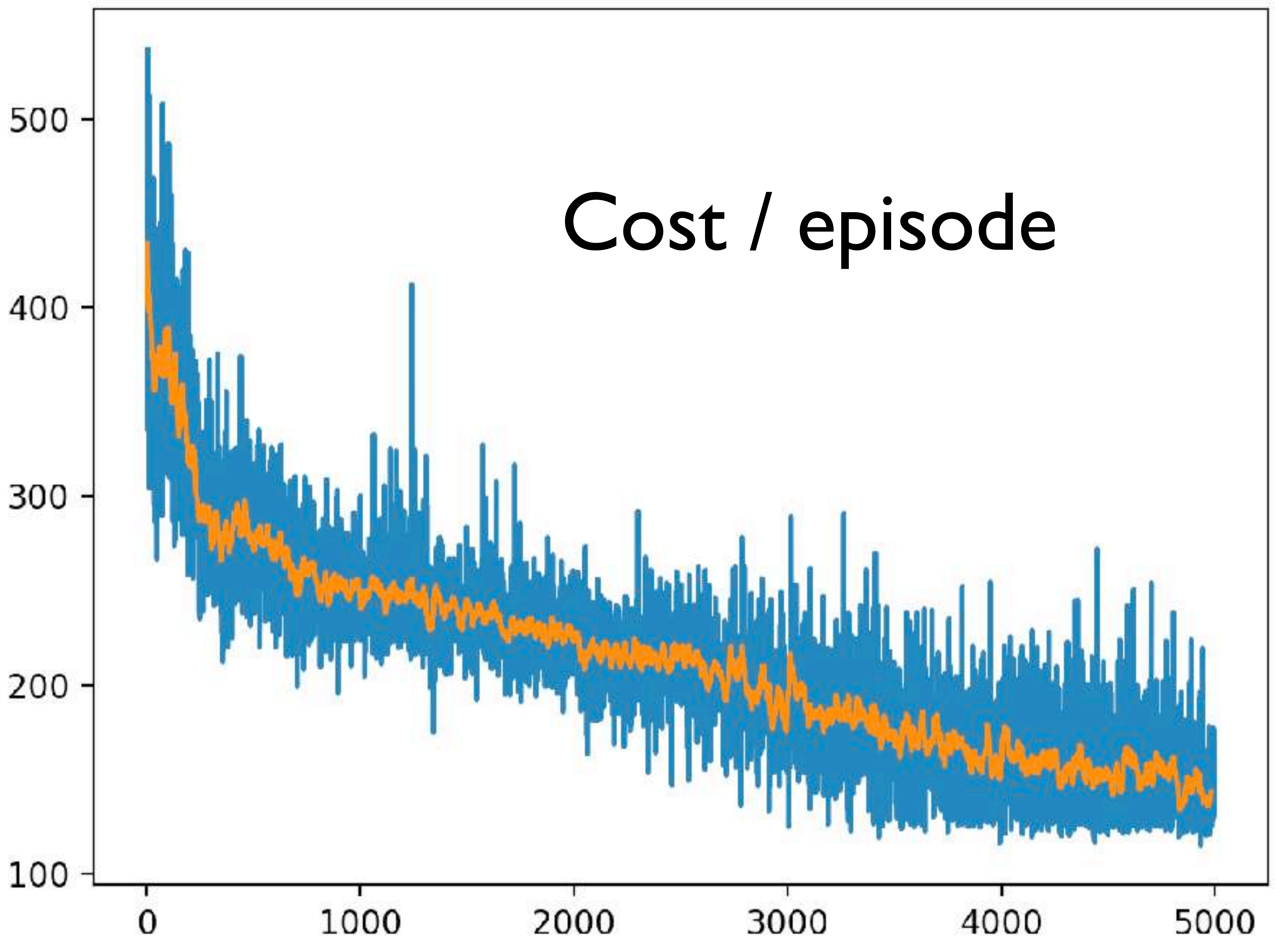
$$\psi_{k,l,c,1}(\theta, \omega, u) = \frac{e^{-\frac{(u-u_c)^2}{0.002}}}{\sqrt{2\pi 0.001}} \sin(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

```python
class StochasticPolicyPeriodicFeatures:
    """
    This class implements a stochastic policy with linear sum of nonlinear features
    the features are periodic functions multiplied by a radial basis function of u
    """
    def __init__(self, controls, order = 2):
        """
        class constructor - controls is the array of control inputs, order is the order of the periodic basis
        """
        self.controls =  controls.copy()
        self.num_controls = len(self.controls)
        self.exp_basis = np.zeros([self.num_controls])
        self.order = order

        # the vector of basis functions
        self.basis_vector = np.zeros([2*self.num_controls*(self.order+1)**2])

        # the linear parameters to learn
        self.theta = np.zeros_like(self.basis_vector)

    def basis(self, x, u):
        """
        Returns the vector of basis functions evaluated at x,u
        """
        dx = x[0]
        dy = x[1]/6. * np.pi
        count = 0
        for c in self.controls:
            du = 1/(np.sqrt(2*np.pi*0.001)) * np.exp(-(u-c)**2/0.002)
            for j,k in itertools.product(range(self.order+1), range(self.order+1)):
                self.basis_vector[count] = du * np.cos(j*dx + k*dy)
                self.basis_vector[count+1] = du * np.sin(j*dx + k*dy)
                count += 2
        return self.basis_vector

    def get_distribution(self, x):
        """
        Computes pi(u|x) for all u
        returns an array of pi and an array of basis functions (row is the control index and column is the )
        """
        dist = np.zeros_like(self.controls)
        basis_fun = np.zeros([len(self.theta), len(self.controls)])
        for i,u in enumerate(self.controls):
            # this gives the basis function evaluated as (x,u)
            basis_fun[:,i] = self.basis(x,u)
            # dist gives exp(theta * basis_function)
            dist[i] = np.exp(self.theta.dot(basis_fun[:,i]))

        # we sum the exponentials
        sm = np.sum(dist)
        # dist is rescaled by the sum of exponentials (we now have a probability distribution)
        dist = dist/sm
        return dist, basis_fun

    def sample(self, x):
        """
        sample from the stochastic policy given x
        it returns the index of the control and its value
        """
        probs, basis = self.get_distribution(x)
        index = np.random.choice(len(self.controls), p=probs)
        return index, self.controls[index]
```

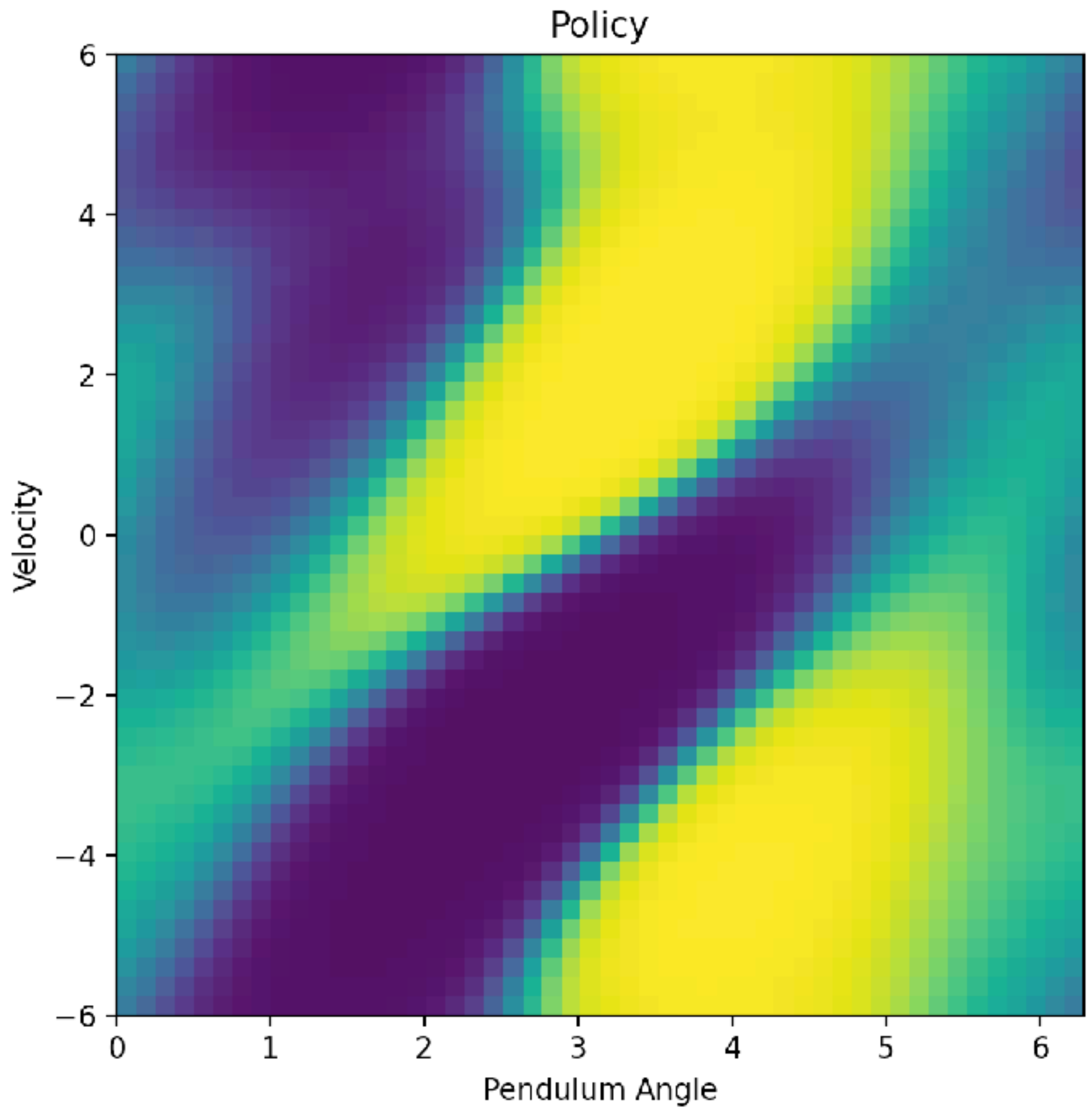
```python
class Reinforce:
    """
    An implementation of the reinforce algorithm (without baseline)
    """

    def __init__(self, model, cost, policy, discount_factor=0.99,
                 episode_length=100, policy_learning_rate = 0.000001, value_learning_rate = 0.01):

        self.model = model
        self.cost = cost

        self.policy = policy

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

    def iterate(self, num_iter=1):
        learning_progress = []

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj = np.zeros([self.episode_length+1, self.model.num_states])
            u_traj = np.zeros([self.episode_length, 1])
            u_index = np.zeros([self.episode_length], dtype=np.int)
            cost_traj = np.zeros([self.episode_length])

            for j in range(self.episode_length):
                u_index[j], u_traj[j,:] = self.policy.sample(x_traj[j,:])
                cost_traj[j] = self.cost(x_traj[j,:], u_traj[j,0])
                x_traj[j+1,:] = self.model.step(x_traj[j,:], u_traj[j,:])[:,0]

            # now we learn computing backwards
            G = 0.
            for j in range(self.episode_length-1, -1, -1):
                G = cost_traj[j] + self.discount_factor * G
                dist, basis = self.policy.get_distribution(x_traj[j,:])
                grad = basis[:,u_index[j]] - basis.dot(dist)
                self.policy.theta -= self.policy_learning_rate * (self.discount_factor**j) * G * grad

            learning_progress.append(G)

        return learning_progress
```

```
pendulum = Pendulum()
policy = StochasticPolicyPeriodicFeatures(controls = pendulum.controls, order = 2)
reinforce_nob = Reinforce(pendulum, cost, policy, value, episode_length=100, discount_factor=0.99,
                          policy_learning_rate = 0.0000001)
```

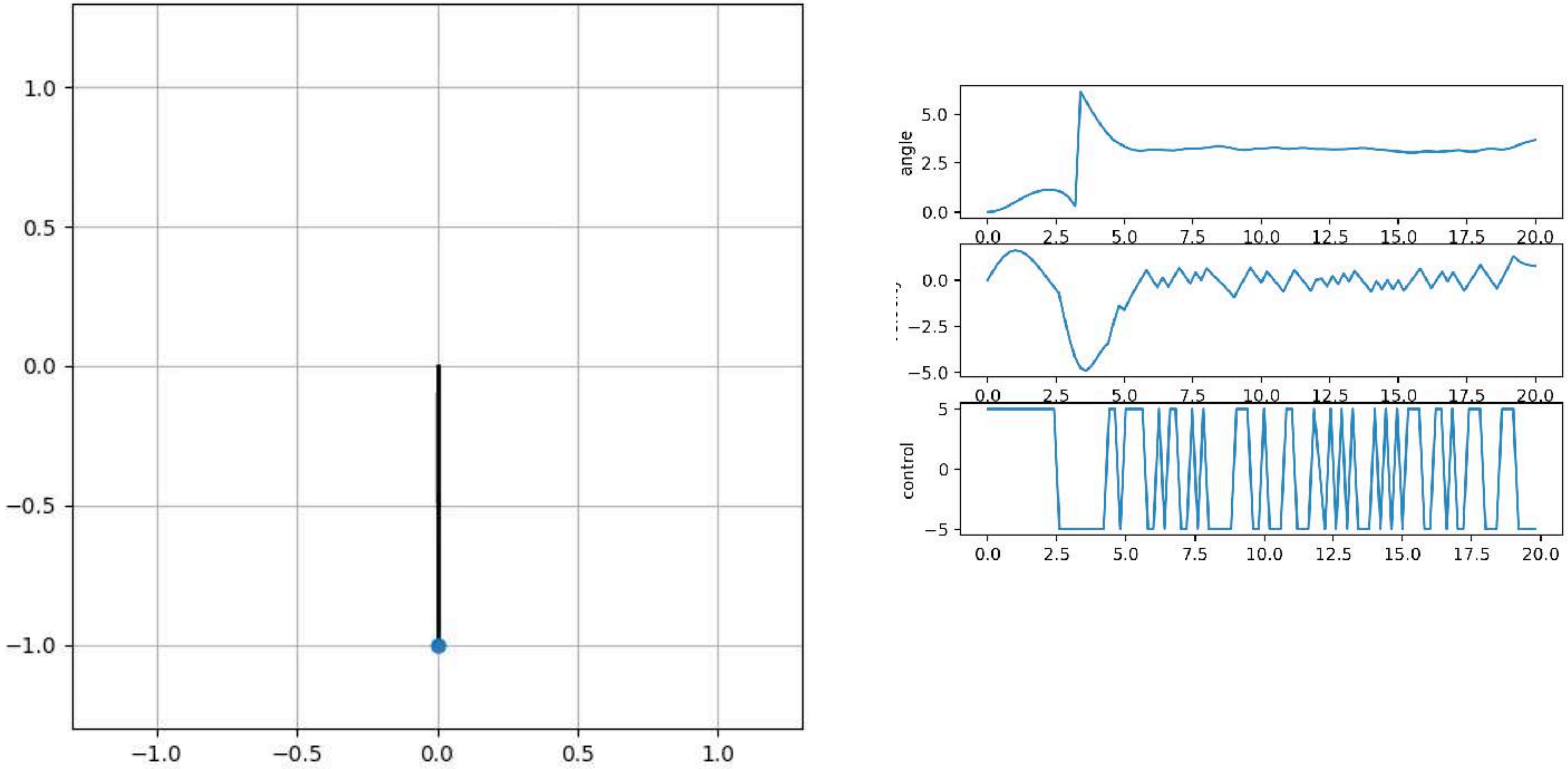
Learning rate 10e-7



Cost / episode
500
400
300
200
100
0
1000
2000
3000
4000
5000



Policy
Velocity
Pendulum Angle
6
4
2
0
−2
−4
−6
0
1
2
3
4
5
6



angle
control

REINFORCE with baseline

$$V(x, \theta_V) = \theta_V^T B(x)$$

$$b_{k,l,0}(\theta, \omega) = \cos(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

$$b_{k,l,1}(\theta, \omega) = \sin(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

```python
class ValueFunctionPeriodicFeatures:
    """
    This class implements a function approximator with linear sum of nonlinear features
    the features are periodic functions
    We use this to approximate the value function
    """

    def __init__(self, order = 2):
        """
        the class constructor - order is the order of the periodic basis
        """

        self.order = order
        self.basis_vector = np.zeros([2*(self.order+1)**2])

        # the parameters to learn
        self.theta = np.zeros_like(self.basis_vector)

    def basis(self, x):
        """
        Returns the vector of basis functions evaluated at x
        """
        dx = x[0]
        dy = x[1]/6. * np.pi
        count = 0
        for j,k in itertools.product(range(self.order+1), range(self.order+1)):
            self.basis_vector[count] = np.cos(j*dx + k*dy)
            self.basis_vector[count+1] = np.sin(j*dx + k*dy)
            count += 2
        return self.basis_vector

    def getValue(self, x):
        """
        returns the value at x and the basis functions evaluated at x
        """
        bs = self.basis(x)
        return bs.dot(self.theta), bs
```

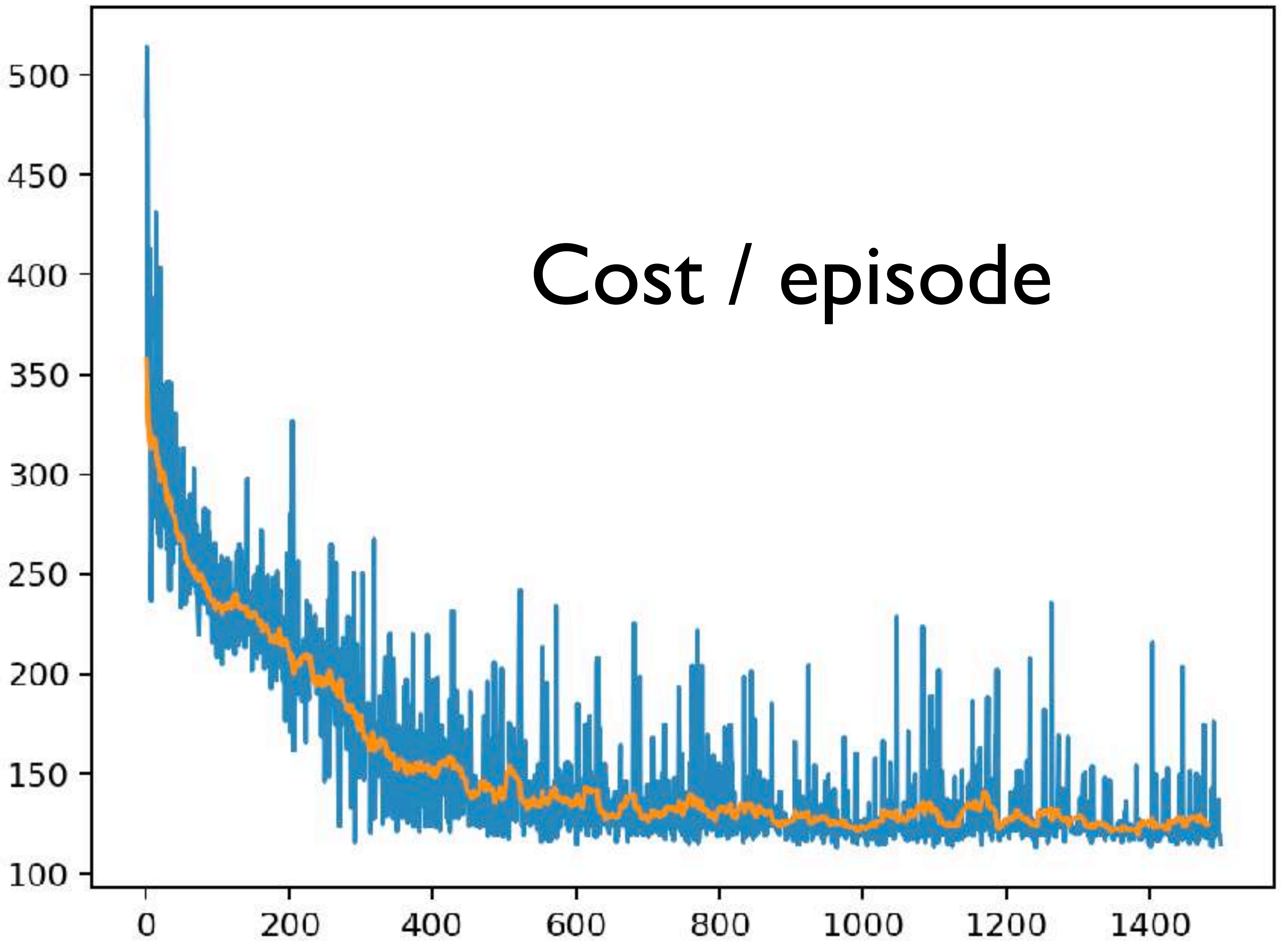
```python
class ReinforceBaseline:
    """
    An implementation of the reinforce algorithm (with or without baseline)
    """

    def __init__(self, model, cost, policy, valuefunction, discount_factor=0.99,
                 episode_length=100, policy_learning_rate = 0.000001, value_learning_rate = 0.01):

        self.model = model
        self.cost = cost

        self.policy = policy
        self.value = valuefunction

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

    def iterate(self, num_iter=1):
        learning_progress = []

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj = np.zeros([self.episode_length+1, self.model.num_states])
            u_traj = np.zeros([self.episode_length, 1])
            u_index = np.zeros([self.episode_length], dtype=np.int)
            cost_traj = np.zeros([self.episode_length])

            for j in range(self.episode_length):
                u_index[j], u_traj[j,:] = self.policy.sample(x_traj[j,:])
                cost_traj[j] = self.cost(x_traj[j,:], u_traj[j,0])
                x_traj[j+1,:] = self.model.step(x_traj[j,:], u_traj[j,:])[:,0]

            # now we learn computing backwards
            G = 0.
            for j in range(self.episode_length-1, -1, -1):
                G = cost_traj[j] + self.discount_factor * G
                dist, basis = self.policy.get_distribution(x_traj[j,:])
                grad = basis[:,u_index[j]] - basis.dot(dist)
                value, grad_value = self.value.getValue(x_traj[j,:])
                delta = (self.discount_factor**j) * (G - value)
                self.value.theta += self.value_learning_rate * delta * grad_value
                self.policy.theta -= self.policy_learning_rate * delta * grad

            learning_progress.append(G)

        return learning_progress
```

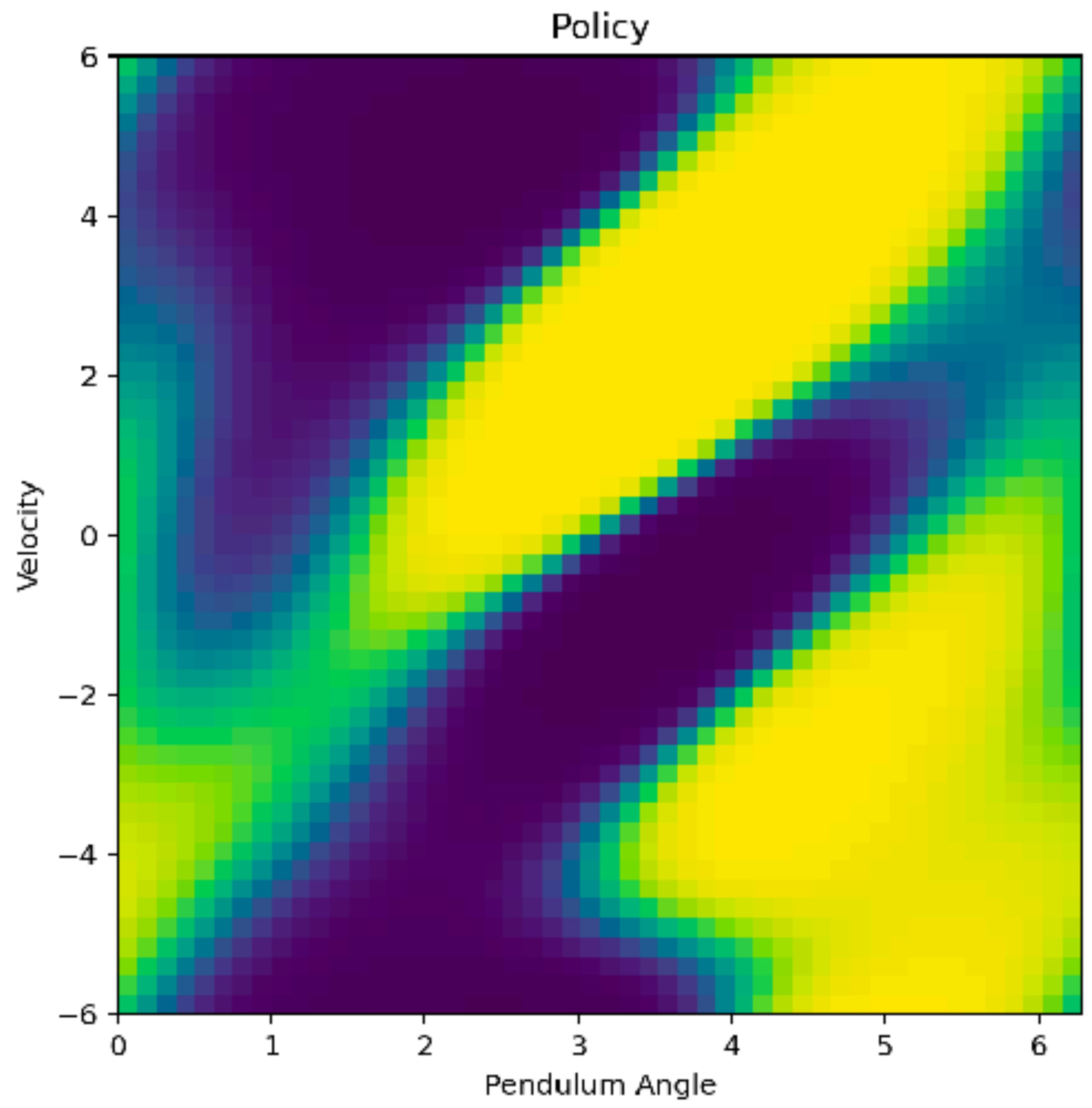
REINFORCE with baseline

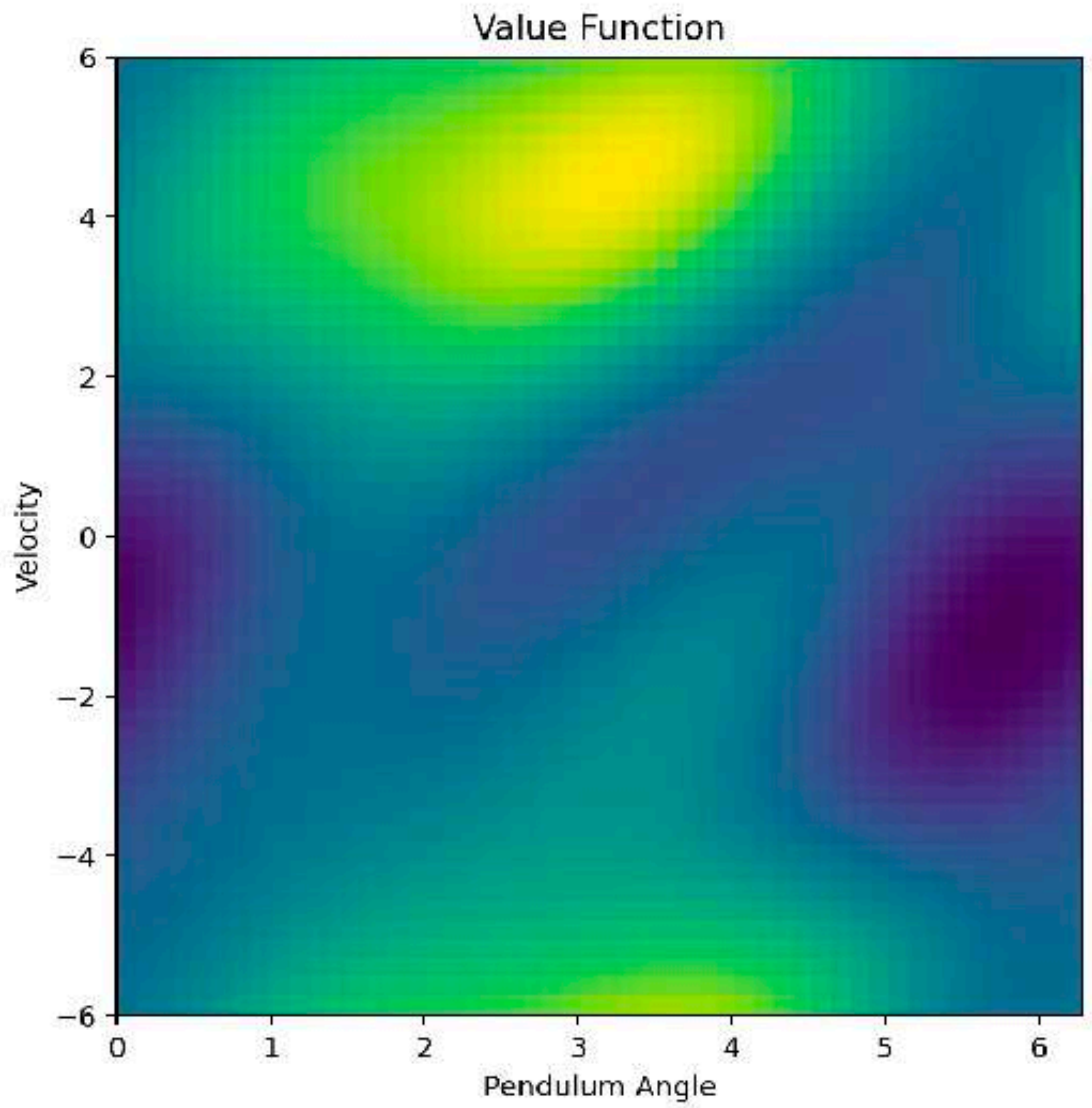
```
pendulum = Pendulum()
policy = StochasticPolicyPeriodicFeatures(controls = pendulum.controls, order = 2)
value = ValueFunctionPeriodicFeatures(order = 2)
reinforce_withb = Reinforce(pendulum, cost, policy, value, episode_length=100, discount_factor=0.99,
                      policy_learning_rate = 0.000001, value_learning_rate = 0.01)
```

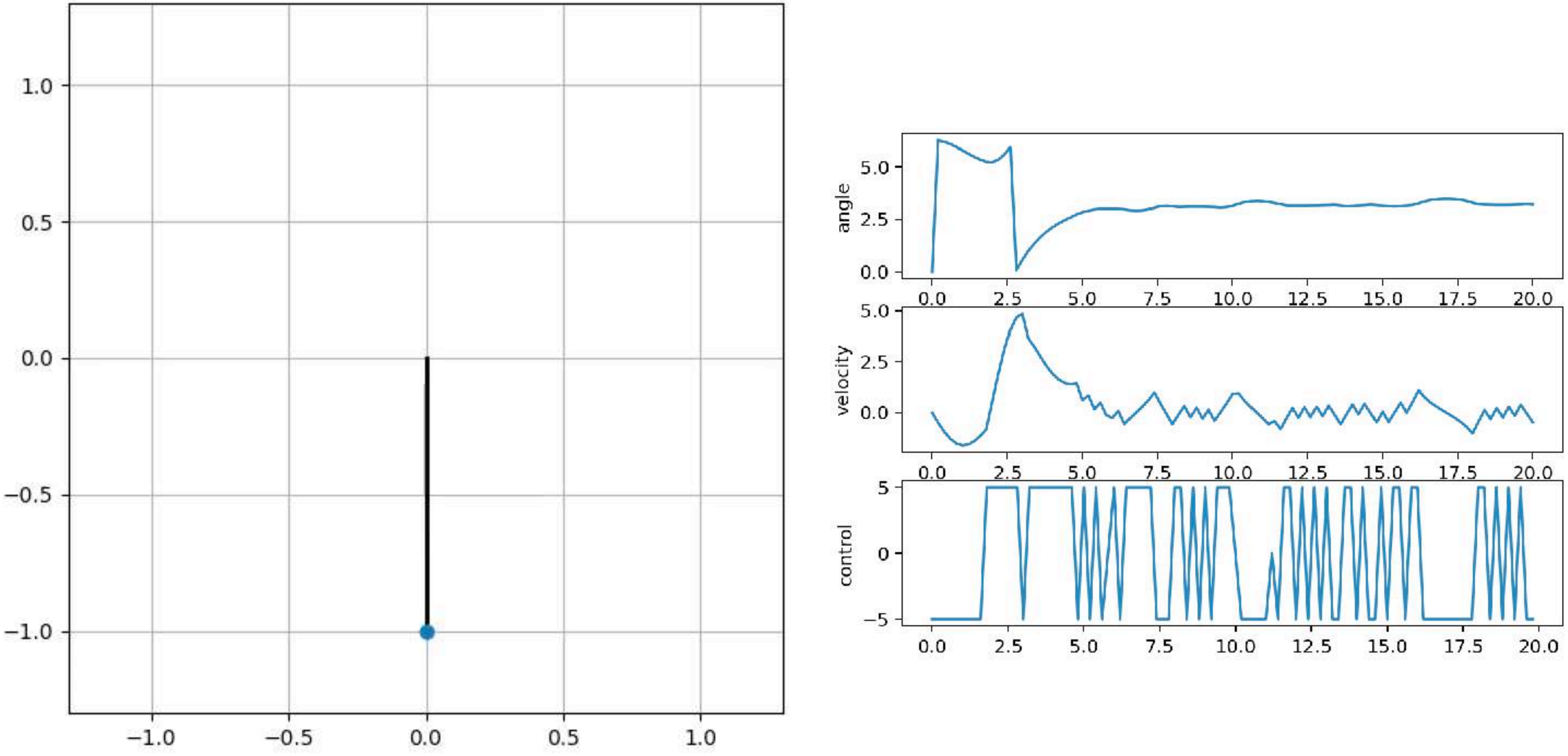
Learning rate 10e-6

REINFORCE with baseline








1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
angle
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
velocity
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
control
5
0
−5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Actor-critic methods

We can use the TD error directly instead of computing the return on the full episode

Actor-critic algorithm with advantage function

Initialize parameters θ_V for value function $V(x, \theta_V)$

Initialize parameters θ_π for policy function $\pi(u|x, \theta_\pi)$

Choose step sizes $\gamma_\pi > 0$ and $\gamma_V > 0$

Loop forever (for each episode):

Initialize the initial state x_0

Loop for the duration of the episode

Get a $u \sim \pi(.|x, \theta)$

Apply action u and get x_{t+1}

Compute advantage $A_t \leftarrow g(x_t, u) + \alpha V(x_{t+1}, \theta_V) - V(x_t, \theta_V)$

$\theta_V \leftarrow \theta_V + \gamma_V \alpha^t A_t \nabla V(x, \theta_V)$

$\theta_\pi \leftarrow \theta_\pi - \gamma_\pi \alpha^t A_t \nabla \ln \pi(u|x, \theta_\pi)$

$I \leftarrow \alpha I$

Policy gradient methods

REINFORCE $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} G_n \nabla_\theta \log \pi(u_n|x_n,\theta)\right] \quad G_n = \sum_{k=n}^{N} \alpha^k g(x_k,u_k)$$

REINFORCE with baseline $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (G_n - V(x_n)) \nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Actor-critic

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (g(x_n,u_n) + \alpha V(x_{n+1}) - V(x_n)) \nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Policy gradient methods

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \nabla_\theta \log \pi(u_n|x_n, \theta)\right]$$

$$\Psi_n = \sum_{k=0}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = g(x_n, u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = Q_\pi(x_n, u_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k) - b(x_n)$$

$$\Psi_n = A_n = Q(x_n, u_n) - V(x_n)$$

Proximal policy optimization (PPO)

Explicit gradient descent

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Equivalent to

$$\min_\theta \mathbb{E}\left[\Psi_n \log \pi(u_n|x_n,\theta)\right]$$

Use the gradient of log to re-arrange the formula

$$\min_\theta \mathbb{E}\left[A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}\right]$$

Proximal policy optimization (PPO)

"Clip" the total scaling

$$\min_{\theta} \mathbb{E}\left[\min\left(A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, clip\left(\frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, 1-\epsilon, 1+\epsilon\right) A_n\right)\right]$$

Run a lot of episodes in <u>parallel</u> (in simulation) to improve the estimation of the gradient and expectation

Proximal policy optimization (PPO)

Evaluating the advantage An

$$\delta_n = g(x_n, u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$A_n = \sum_{k=n}^{N} (\alpha\lambda)^{k-n} \delta_k$$

Proximal policy optimization (PPO)

While not converged

For actors 1, … , P do

Run the policy in the simulator for N time steps

Collect state/action transition

Compute advantage estimates $A_n = \sum_{k=n}^{N} (\alpha\lambda)^{k-n} \delta_k$

End for

Do gradient descent on the cost

$$\min_{\theta} \mathbb{E}\left[\min\left(A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, clip\left(\frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, 1-\epsilon, 1+\epsilon\right) A_n\right)\right]$$

Update the value function estimates (e.g. TD-learning)

Proximal policy optimization (PPO)

Lots of heuristics but it works rather well in practice

Parallelization and clipping help a lot to get good gradient steps

PPO is considered "state of the art" for deep RL in robotics

BUT it is rarely used as is - a lot of engineering around is necessary

Getting started with RL… CleanRL

CleanRL   Search   vwxyzjn/cleanrl v1.0.0 5.7k 642

CleanRL - Overview

license MIT | tests passing | docs success | discord 44 online | Views 13k | code style black | imports isort | Models Huggingface | Open in Colab

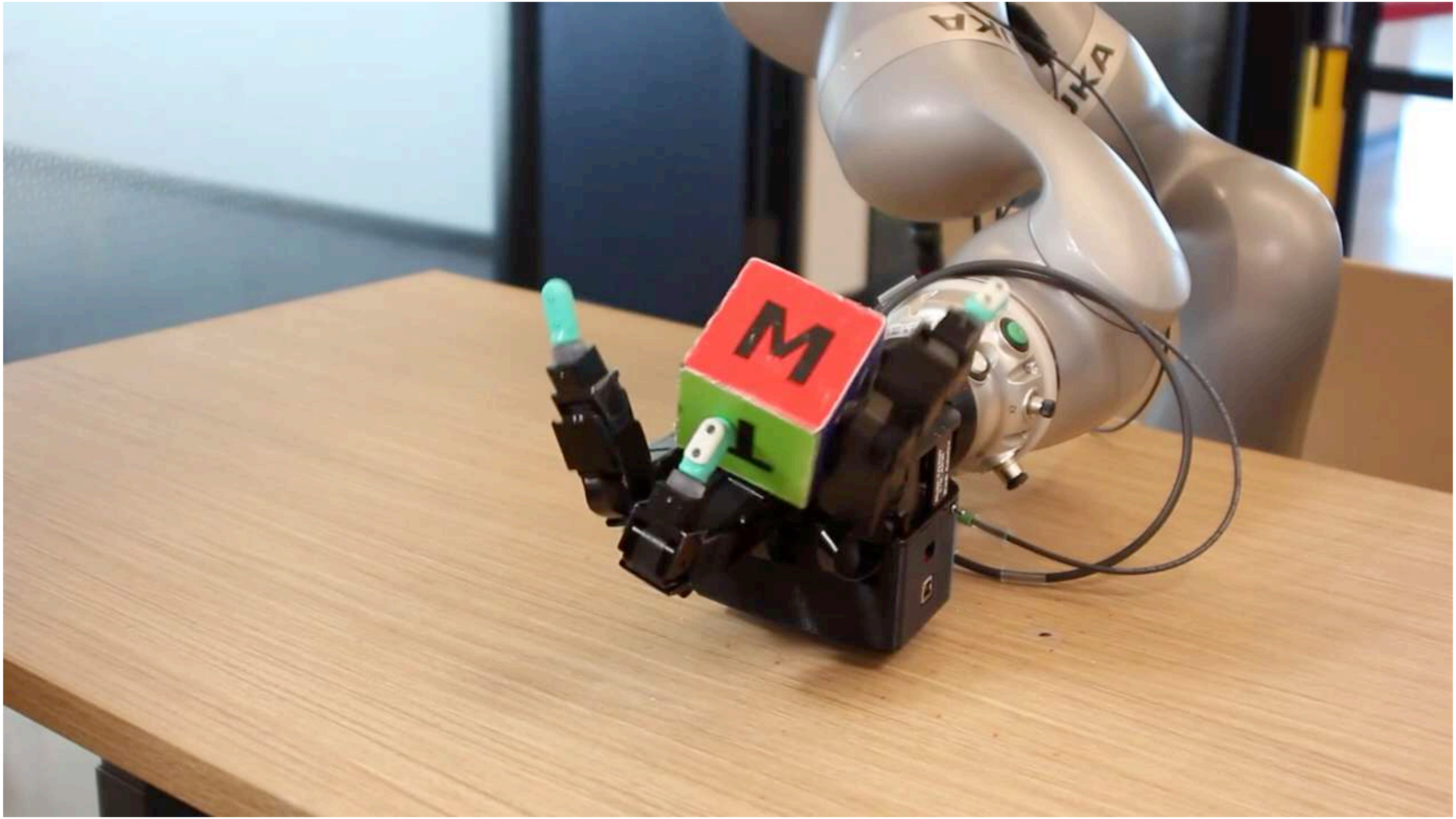
CleanRL is a Deep Reinforcement Learning library that provides high-quality single-file implementation with research-friendly features. The implementation is clean and simple, yet we can scale it to run thousands of experiments using AWS Batch. The highlight features of CleanRL are:

- 📜 Single-file implementation
- *Every detail about an algorithm variant is put into a single standalone file.*
- For example, our `ppo_atari.py` only has 340 lines of code but contains all implementation details on how PPO works with Atari games, **so it is a great reference implementation to read for folks who do not wish to read an entire modular library**.
- 📊 Benchmarked Implementation (7+ algorithms and 34+ games at https://benchmark.cleanrl.dev)
- 📈 Tensorboard Logging
- 🪛 Local Reproducibility via Seeding
- 🎮 Videos of Gameplay Capturing
- 🧫 Experiment Management with Weights and Biases
- 💸 Cloud Integration with docker and AWS

You can read more about CleanRL in our technical paper and documentation.

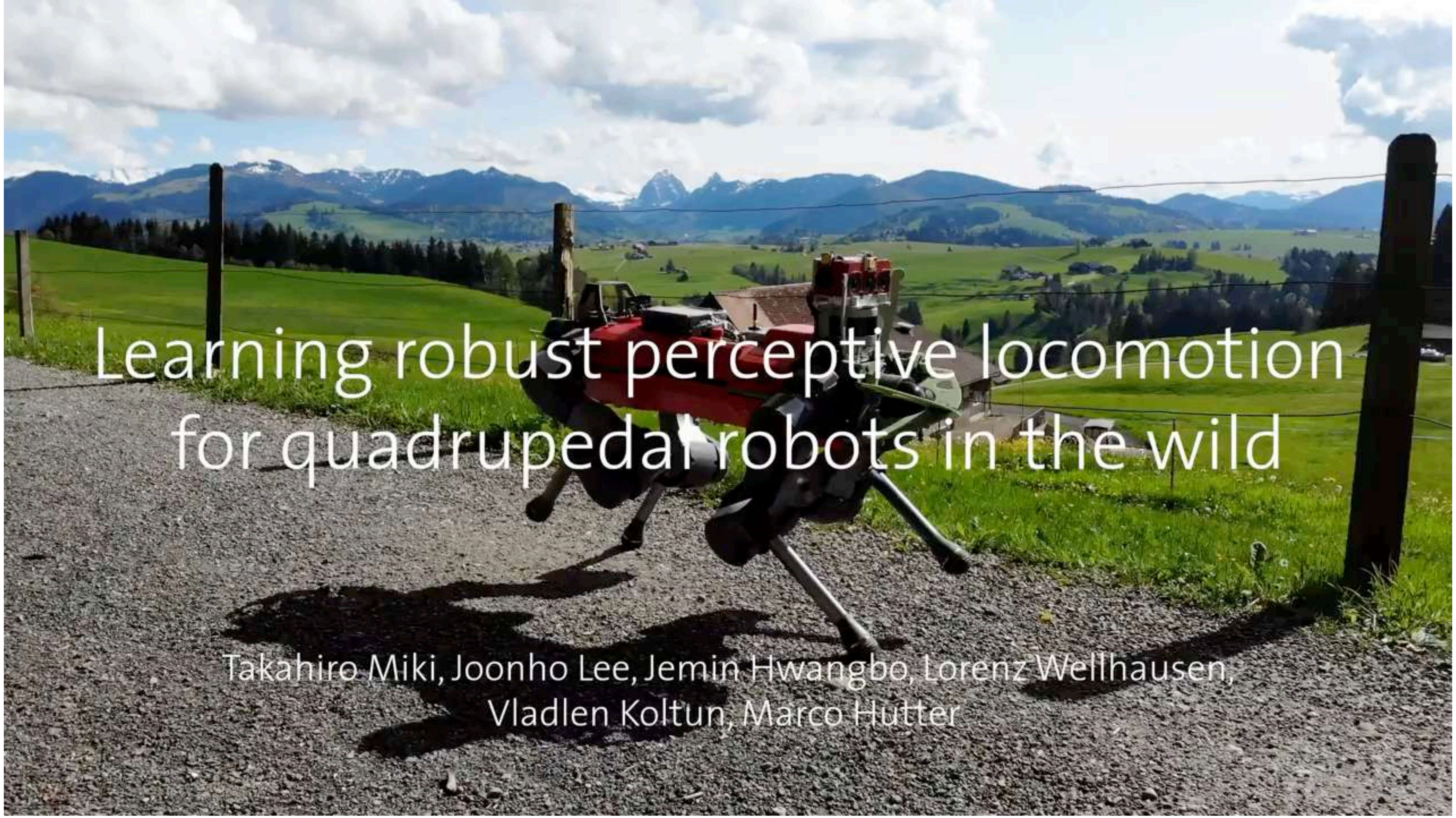
CleanRL only contains implementations of **online** deep reinforcement learning algorithms. If you are looking for **offline** algorithms, please check out corl-team/CORL, which shares a similar design philosophy as CleanRL.

[Chane-Sane et al IROS 2024]



[Handa et al. 2022]

Learning various behaviors



[Miki et al. Science 2022]