ROB-GY 6323
reinforcement learning and optimal control for robotics

Lecture 10
Deep Q-learning

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			

Policy evaluation with sampling 1
Monte-Carlo methods

Policy evaluation with sampling II
TD-learning

Optimal Value function

$$J = \min_u g(x, u) + \alpha J(f(x, u))$$

Value Function for Policy $\mu(x) : x \to u$

$$J_\mu(x) = g(x, \mu(x)) + \alpha J_\mu(f(x, \mu(x)))$$

Action-value function $Q(x_t, u_t)$

$$Q(x_t, u_t) = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u)$$

$$J_\mu(x_t) \overset{?}{=} \underline{g(x_t, \mu(x_t)) + \alpha J_\mu(x_{t+1})}$$

current guess
of cost for state x_t

actual return
following time t

Temporal difference
error (TD error)

$$\delta_t = g(x_t, \mu(x_t)) + \alpha J_\mu(x_{t+1}) - J_\mu(x_t)$$

Temporal difference learning

[Sutton 1988]
[Samuel 1959]

TD(0) learning for estimating J_μ

Input: policy to be evaluated μ

Choose a step size $\gamma \in [0, 1]$

Initialize J_μ for all states x

For each episode of length N:

Choose an initial state x_0

Loop for each step of the episode:

Do $\mu(x_t)$

Observe x_{t+1} Compute $g(x_t, \mu(x_t))$

Update $J_\mu(x_t) \leftarrow J_\mu(x_t) + \gamma \delta_t$

using $\delta_t = g(x_t, \mu(x_t)) + \alpha J_\mu(x_{t+1}) - J_\mu(x_t)$

How can we improve the policy?

Q-learning: off-policy TD control

[Watkins 1989]

Action-value function $Q(x_t, u_t)$

$$Q(x_t, u_t) = g(x_t, u_t) + \alpha J^*(x_{t+1})$$

$Q(x_t, u_t)$: cost of doing u_t at state x_t and behaving optimally after

$J^*(x_t) = \min_u Q(x_{t+1}, u)$ **Optimal value function**

$\mu^*(x) = \arg\min_u Q(x_t, u)$ **Optimal policy**

Q-learning: off-policy TD control

[Watkins 1989]

Action-value function $Q(x_t, u_t)$

$$Q(x_t, u_t) = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u)$$

$$J^*(x_t) = \min_u Q(x_{t+1}, u)$$ Optimal value function

TD-error $\delta_t = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u) - Q(x_t, u_t)$

Update $Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \gamma \delta_t$

Q-learning

$$\delta_t = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u) - Q(x_t, u_t)$$

$Q(x, u)$ is stored as a table

	$x = 0$	$x = 0.1$	$x = 0.2$		$x = 6.2$
$u = -5$	$Q(x_1, u_1)$			$\cdots$	
$u = -4.9$				$\cdots$	
				$\cdots$	
	$\vdots$	$\vdots$		$\ddots$	
$u = 0$					
	$\vdots$	$\vdots$		$\ddots$	
$u = 5$				$\cdots$	

The exploration/exploitation trade-off

How do we choose the policy in an episode?

Current optimal guess: $u_t = \arg\min_u Q(x_t, u)$

If we always choose the optimal guess, we might miss better actions/states that we would never try

If we only choose random actions, we might be not be able to get a good guess for Q

ϵ -greedy policy

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

Q-learning

[Watkins 1989]

Choose a step size $\gamma \in [0, 1]$ and small ϵ

Initialize $Q(x, u)$ for all states x and actions u

For each episode:

Choose an initial state x_0

Loop for each step of the episode:

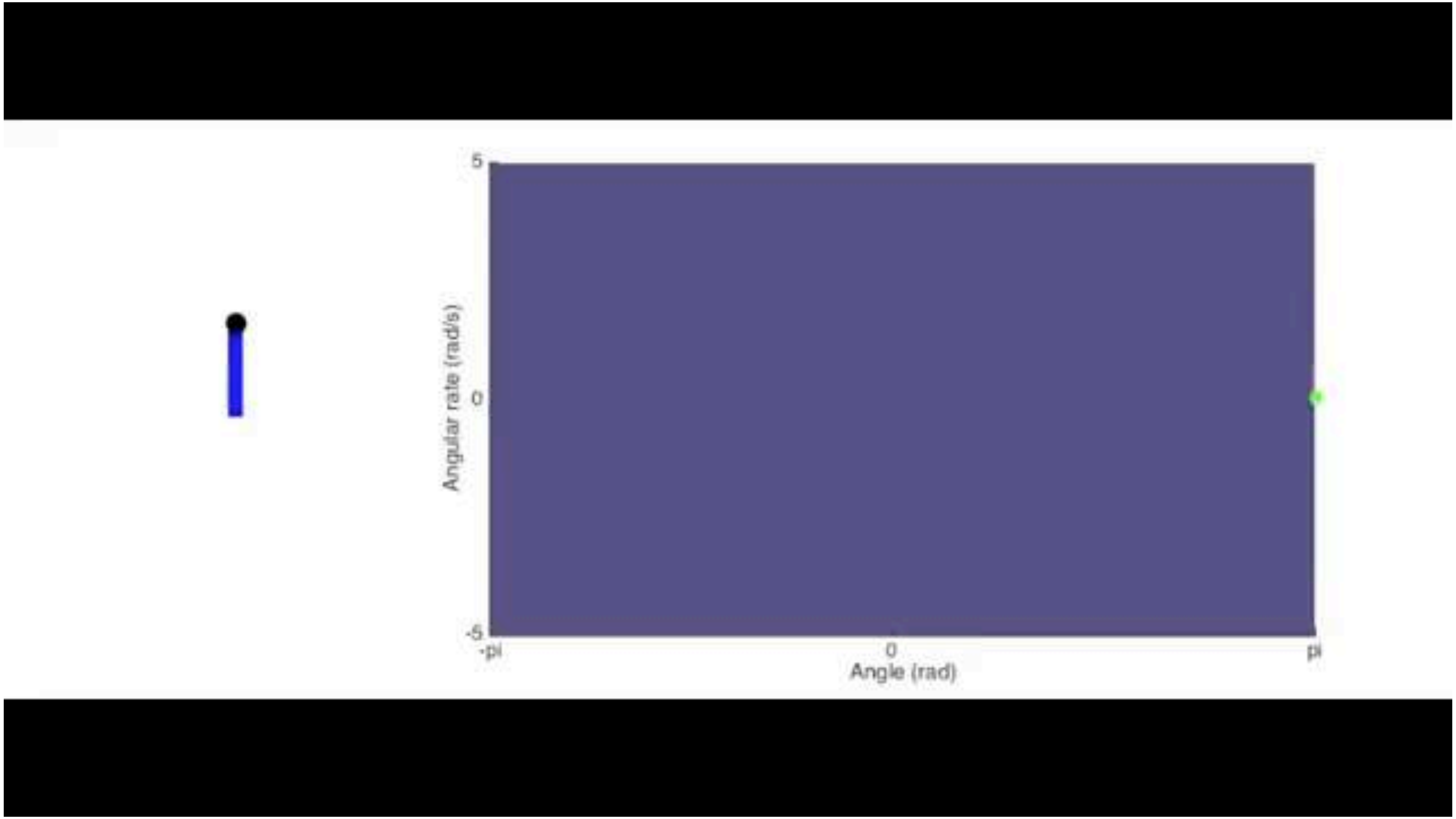
Choose an action using an ϵ-greedy policy from Q

Observe x_{t+1} Compute $g(x_t, \mu(x_t))$

Update $Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \gamma\delta_t$

using $\delta_t = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u) - Q(x_t, u_t)$

Q-learning



[source: https://www.youtube.com/watch?v=YLAWnYAsai8]

Q-learning

Model-free approach to learn optimal policies
(value iteration with a twist)

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \gamma \delta_t$$

$$\delta_t = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u) - Q(x_t, u_t)$$

Guaranteed to converge at infinity BUT can take a long time!

Need to store Q / discrete actions and states

Need to compute the min of Q (expensive!)

Typical RL problems

Robotics RL problems

Set of actions is discrete

State is discrete (countable)

State is continuous

Action space is continuous

Most methods designed for discrete state/action models do not carry over to continuous state/action models

Q-learning

$$\delta_t = g(x_t, u_t) + \alpha \min_u Q(x_{t+1}, u) - Q(x_t, u_t)$$

Can we get rid of the discrete states and actions and the table?

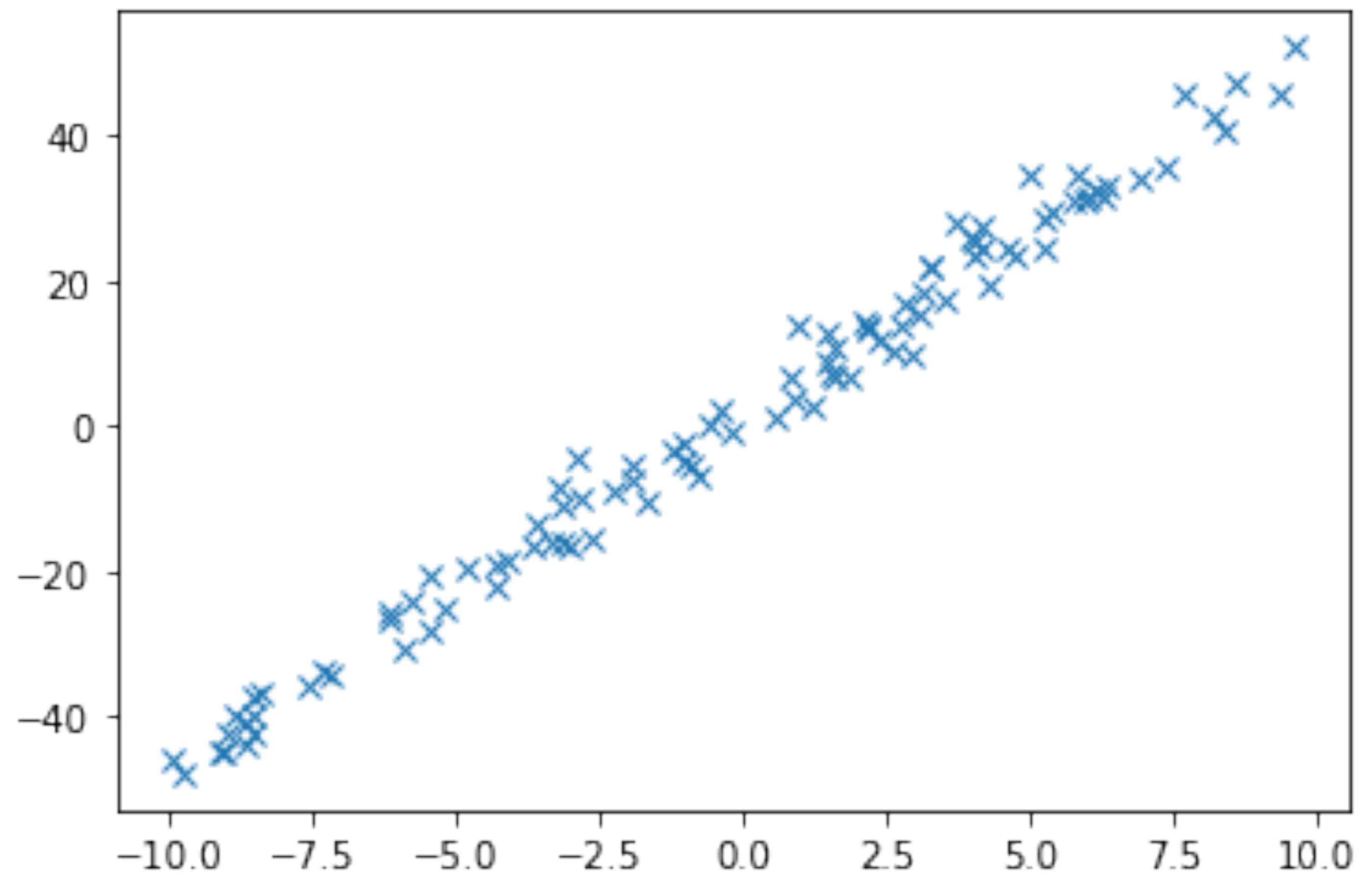
$Q(x, u)$ is a function - can we approximate it?

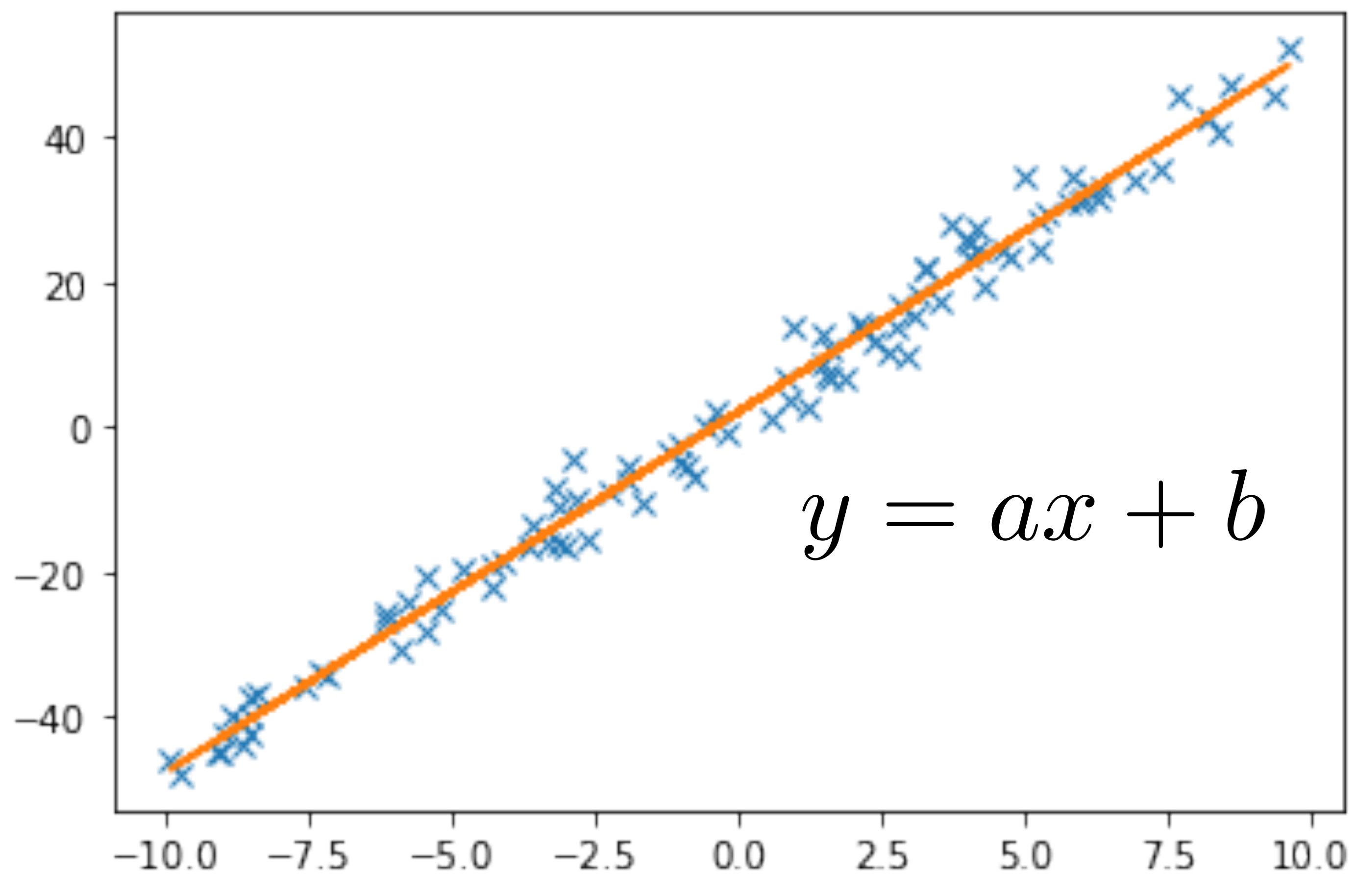
	$x = 0$	$x = 0.1$	$x = 0.2$		$x = 6.2$
$u = -5$	$Q(x_1, u_1)$			$\cdots$	
$u = -4.9$				$\cdots$	
				$\cdots$	
	$\vdots$	$\vdots$		$\ddots$	
$u = 0$					
	$\vdots$	$\vdots$		$\ddots$	
$u = 5$				$\cdots$	

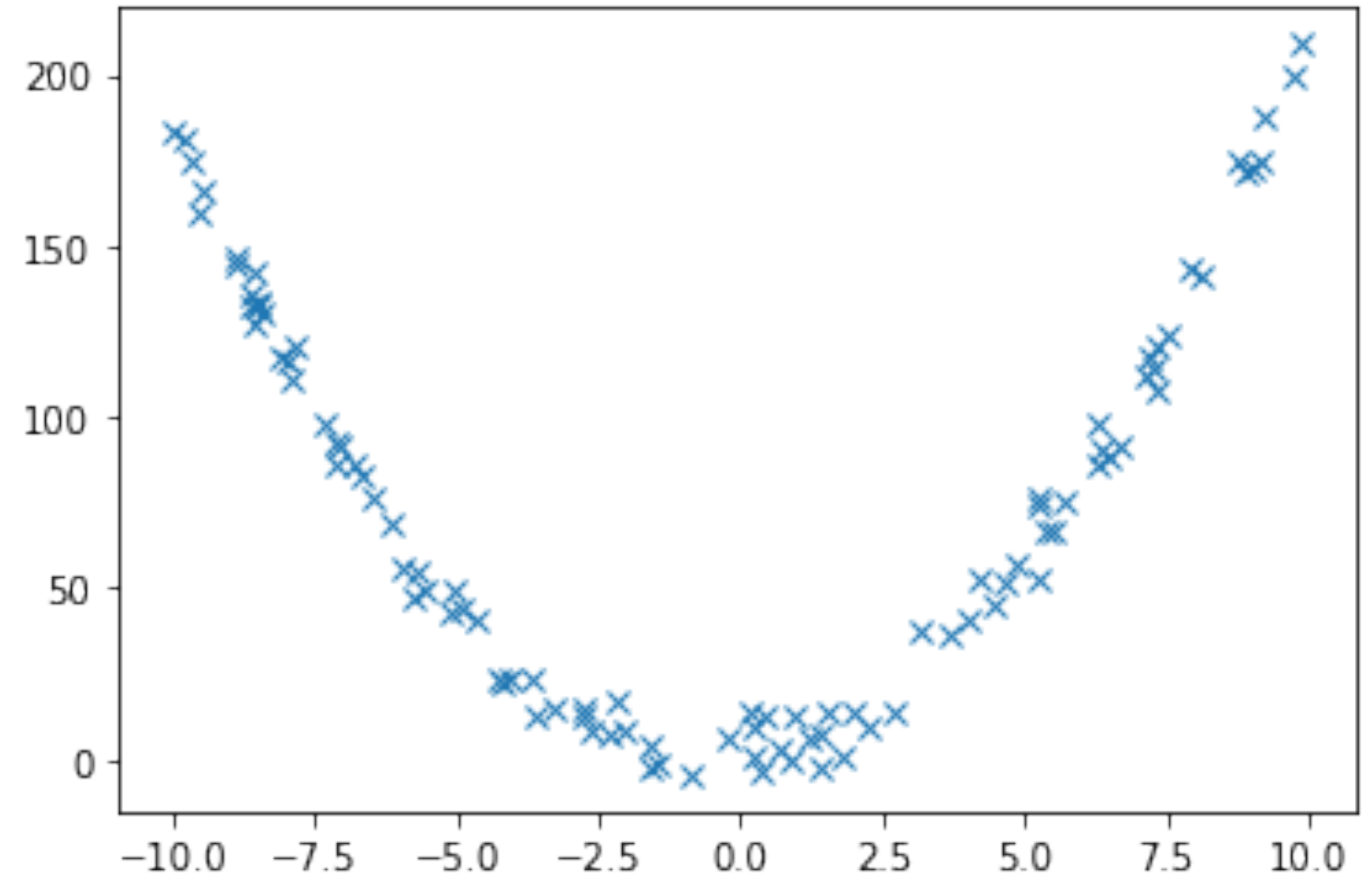
Function approximation

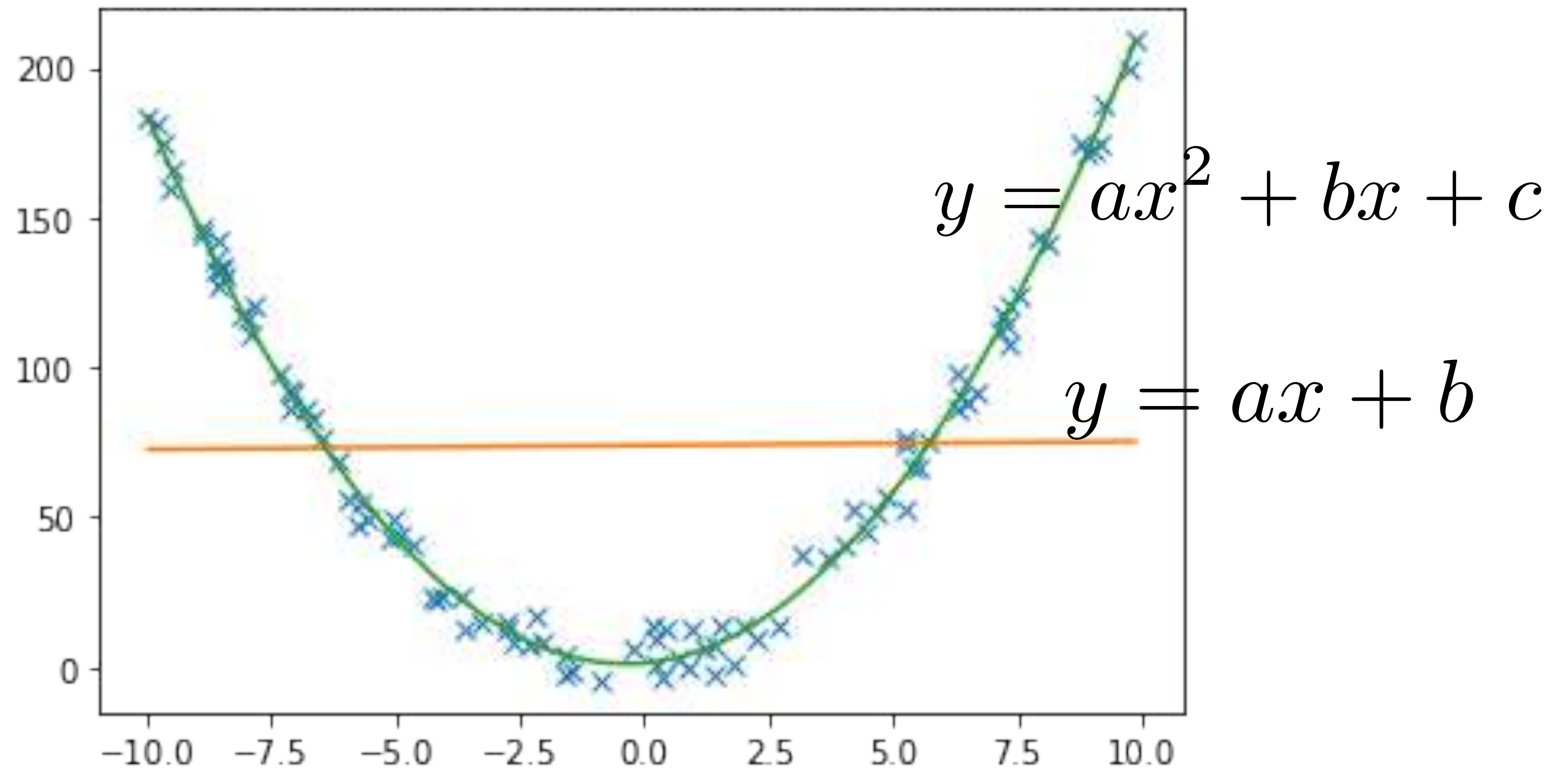
A quick intro to neural networks

Function approximation / Supervised learning









Linear least squares

Given N data point $(x_1, y_1)\ (x_2, y_2) \cdots (x_N, y_N)$

$$y = \sum_{k=0}^{K} a_k x^k$$

Find a function that is a linear combination of polynomials of the input x

Minimize the least square error between the output data and the function

$$\min_{a_0 \cdots a_K} \sum_{i=0}^{N-1} \left(\sum_{k=0}^{K} a_k x_i^k - y_i\right)^2$$

Linear least squares

Minimize the least square error between the output data and the function

$$\min_{a_0 \cdots a_K} \sum_{i=0}^{N-1} (\sum_{k=0}^{K} a_k x_i^k - y_i)^2$$

We can write these relations in matrix form by noticing that

$$\sum_{k=0}^{K} a_k x_i^k = \begin{bmatrix} 1 & x_i & x_i^2 & \cdots & x_i^K \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_K \end{bmatrix}$$

Linear least squares

We can write these relations in matrix form by noticing that

$$\sum_{k=0}^{K} a_k x_i^k = \begin{bmatrix} 1 & x_i & x_i^2 & \cdots & x_i^K \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_K \end{bmatrix}$$

Using all the data points and the knowledge of the degree K and we can then construct the $N \times K$ matrix

$$X = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^K \\ 1 & x_1 & x_1^2 & \cdots & x_1^K \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^K \end{bmatrix}$$

and the vector

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \qquad a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_K \end{bmatrix}$$

where each row i of X and Y is defined by the sample i from the dataset.

Linear least squares

We now have

$$Xa - Y = \begin{bmatrix} \sum_{k=0}^{K} a_k x_0^k - y_0 \\ \sum_{k=0}^{K} a_k x_1^k - y_1 \\ \vdots \\ \sum_{k=0}^{K} a_k x_{N-1}^k - y_{N-1} \end{bmatrix}$$

and the original problem can be written as

$$\min_a (Xa - Y)^T (Xa - Y)$$

which is equal to

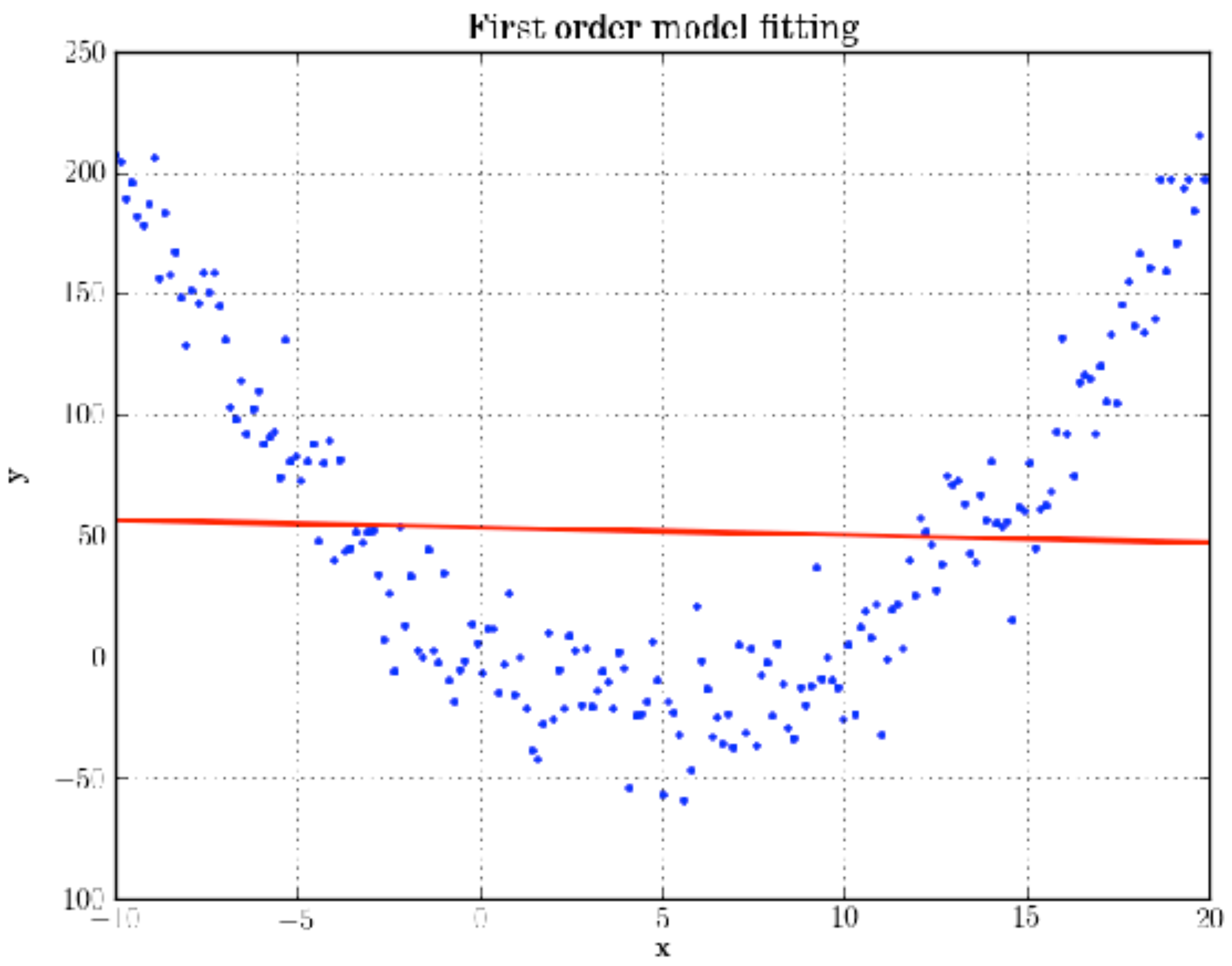
$$\min_a a^T X^T X a - 2Y^T X a + Y^2$$

$$\frac{\partial}{\partial a}\left(a^T X^T X a - 2Y^T X a + Y^2\right) = 2X^T X a - 2X^T Y = 0 \qquad a = (X^T X)^{-1} X^T Y$$

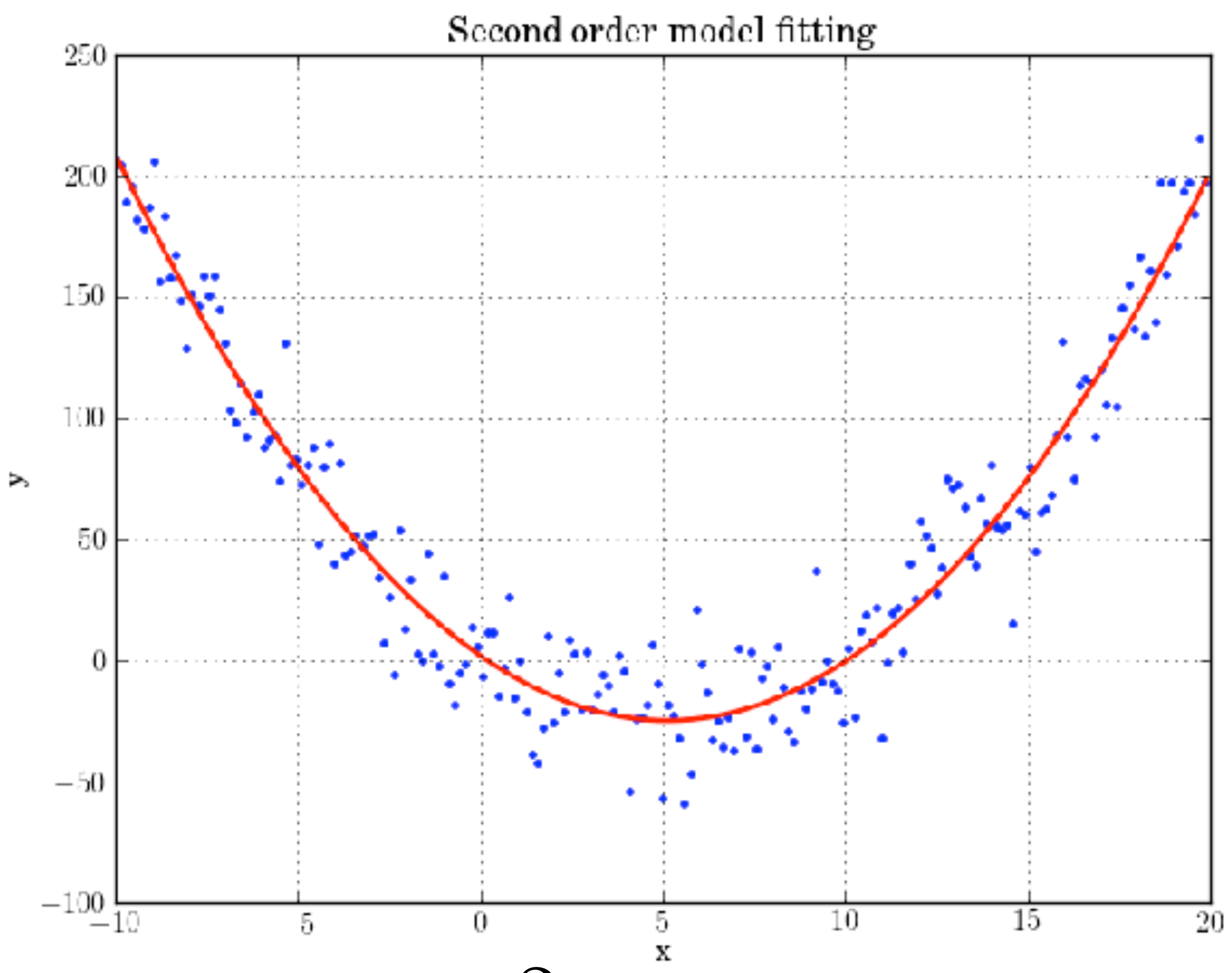
Function approximation

500 noisy samples $\mathcal{D} = \{(x_0, y_0), (x_1, y_1), \cdots, (x_N, y_N)\}$

from a quadratic function



$$y = a_1 x + a_0$$



$$y = a_2 x^2 + a_1 x_1 + a_0$$

Nonlinear least-square

$$y = f(x, w) \qquad\qquad \min \sum_{i=0}^{N} (y_i - f(x_i, \omega))^2$$

we cannot compute w explicitly as before
because the function is not linear in w anymore

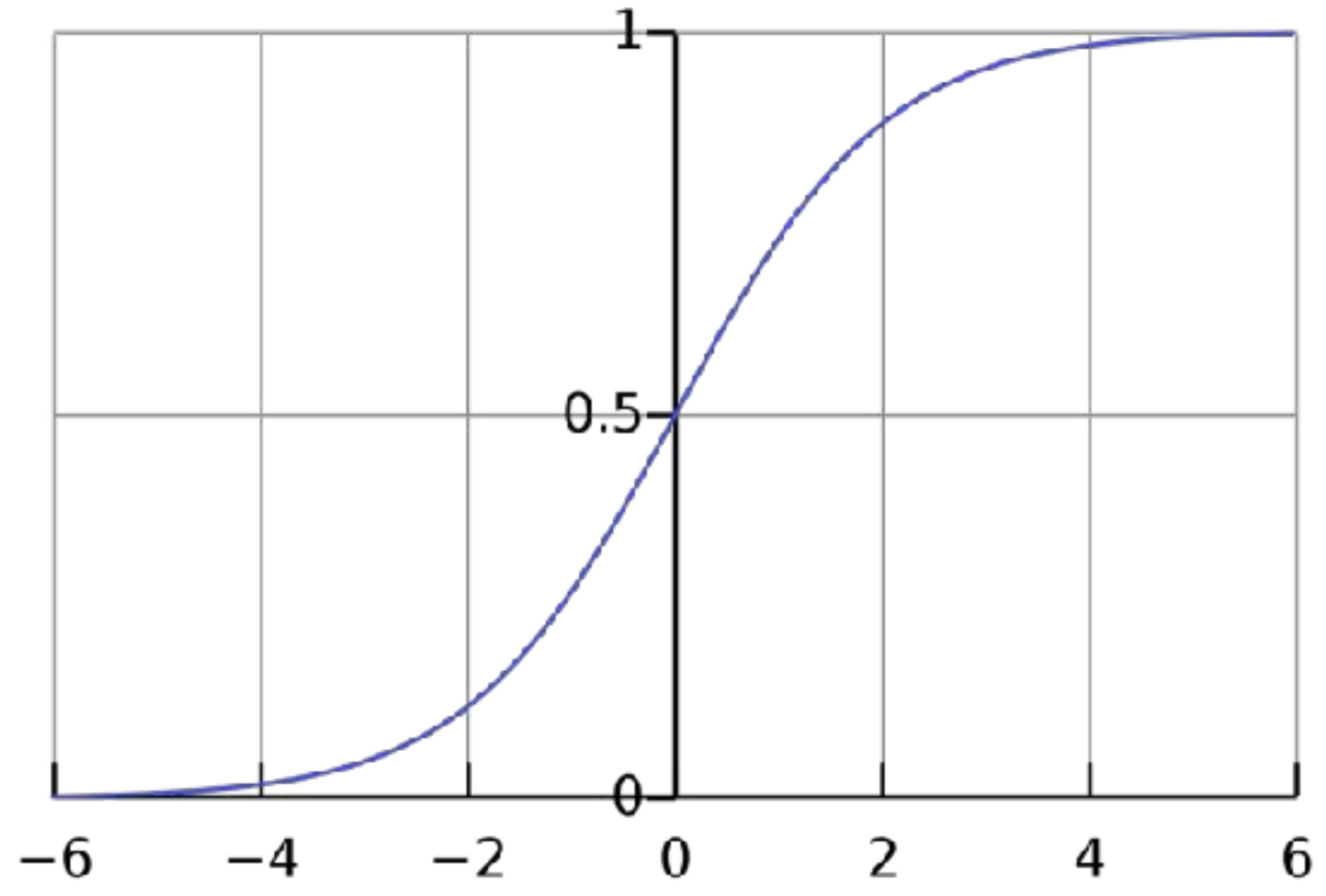
=> do gradient descent to minimize the function

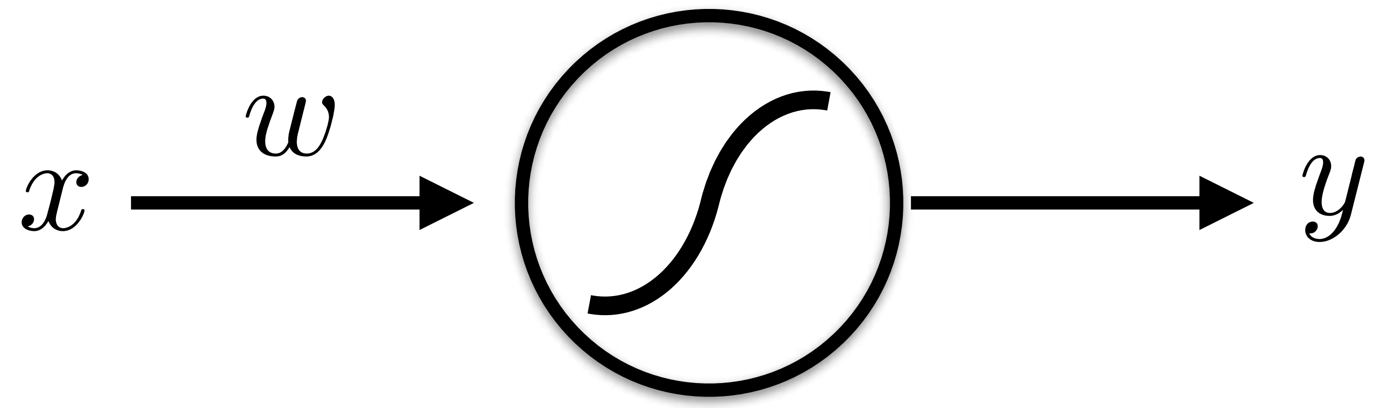
$$\omega \leftarrow \omega - \eta \frac{\partial}{\partial \omega} \left(\sum_{i=0}^{N} (y_i - f(x_i, \omega))^2 \right) = \omega + 2\eta \sum_{i=0}^{N} \left((y_i - f(x_i, \omega)) \frac{\partial f}{\partial \omega} \right)$$

Example: a simple nonlinear function

$$y = \frac{1}{1 + \mathrm{e}^{-wx}}$$

a sigmoid function

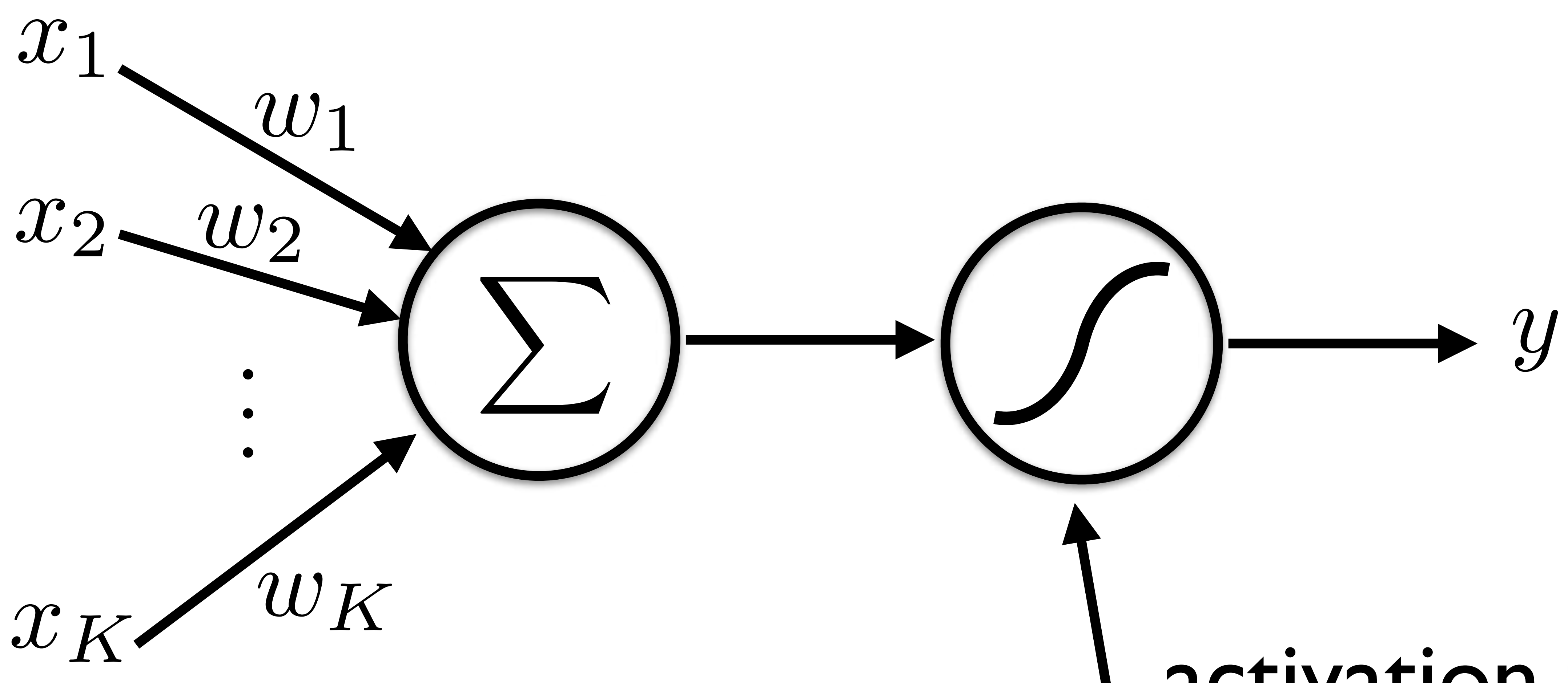




Example: a simple nonlinear function

multiple inputs and one output

$$y = \frac{1}{1 + \mathrm{e}^{-\sum_k w_k x_k}}$$



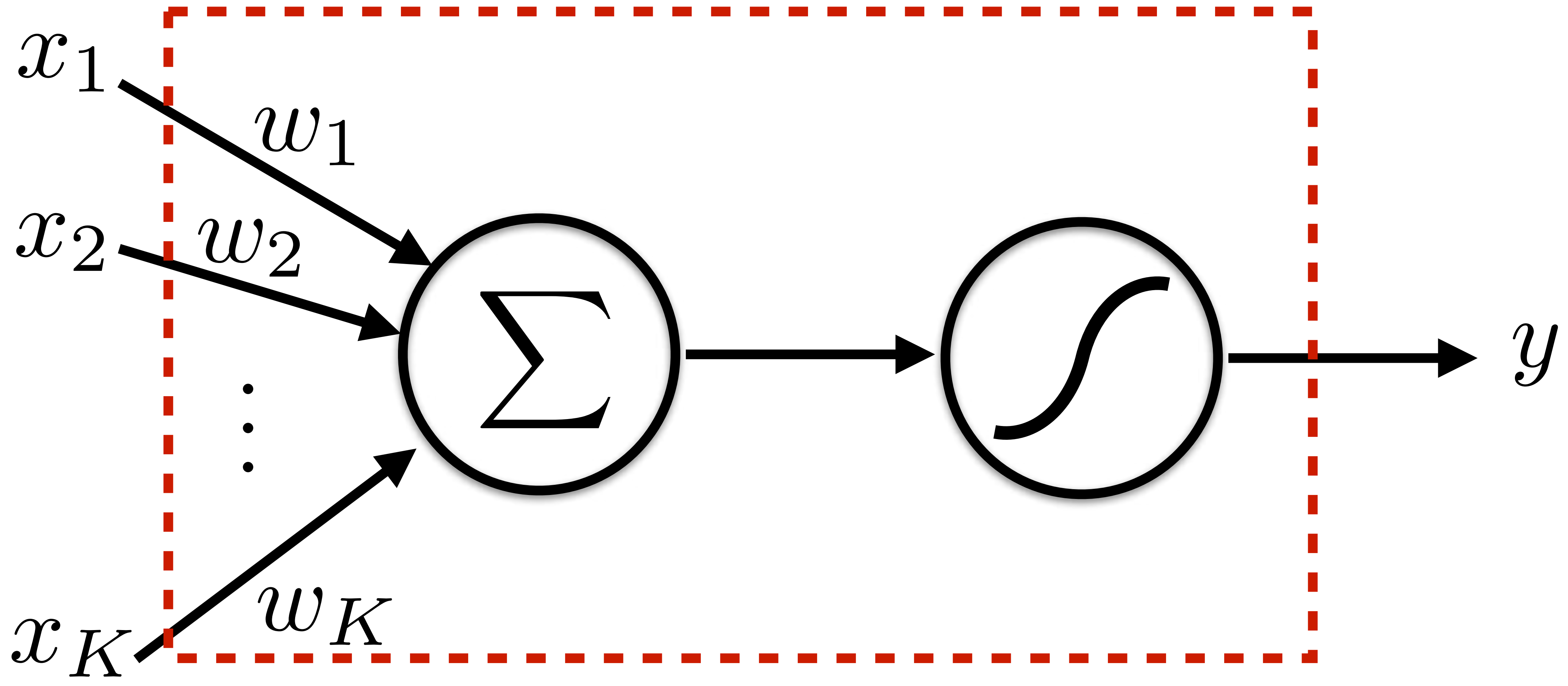
activation function
(nonlinearity)

w_k weights (unknown parameters to find or "learn")

Example: a simple nonlinear function

multiple inputs and one output

$$y = \frac{1}{1 + \mathrm{e}^{-\sum_k w_k x_k}}$$



one "neuron"

$$\min_w \sum_{n=0}^{N} \left(y_i - \frac{1}{1 + \mathrm{e}^{-\sum_k w_k x_{k,i}}} \right)$$

Example: a simple nonlinear function

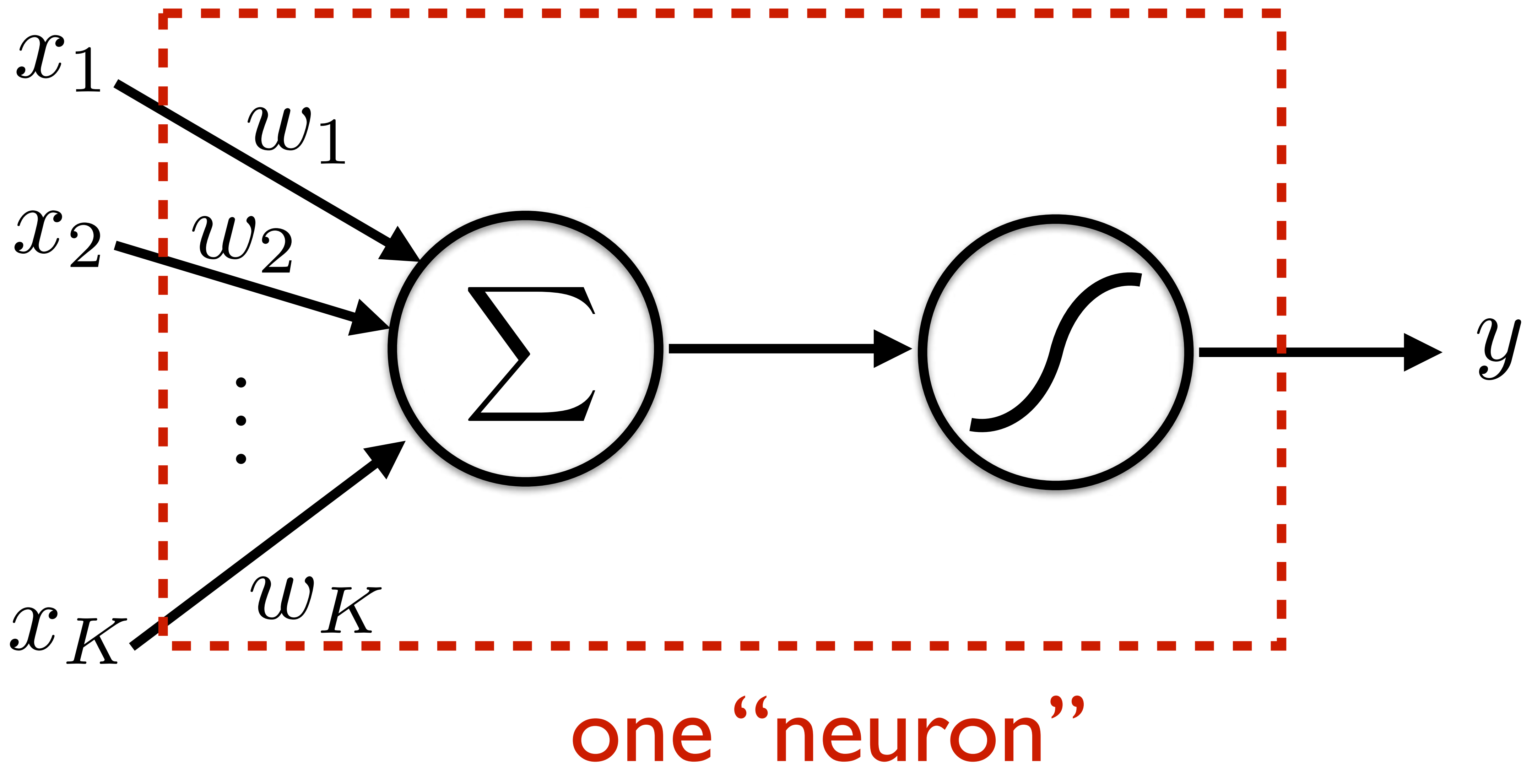
$$\min_{w} \sum_{n=0}^{N} \left(y_i - \frac{1}{1 + \mathrm{e}^{-\sum_k w_k x_{k,i}}} \right)$$

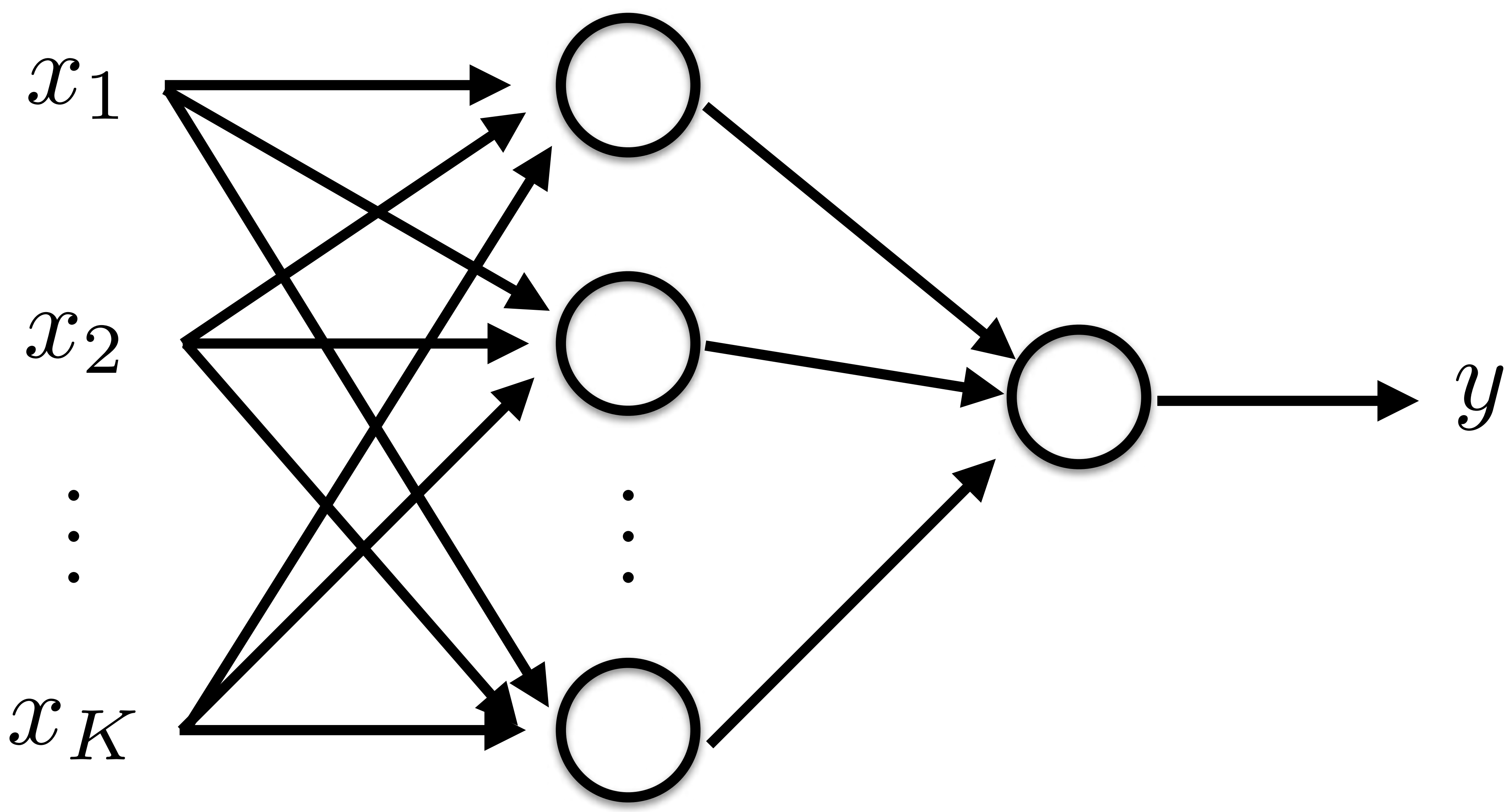
Dataset has multidimensional input

$$\mathcal{D} = \{(x_{0,0}, x_{1,0}, \cdots, x_{M,0}, y_0), (x_{0,1}, x_{1,1}, \cdots, x_{M,1}, y_1), \cdots, (x_{0,N}, x_{1,N}, \cdots, x_{M,N}, y_N)\}$$

Gradient descent enables the optimization of the unknown parameters (or neuron weights) w

One layer neural network

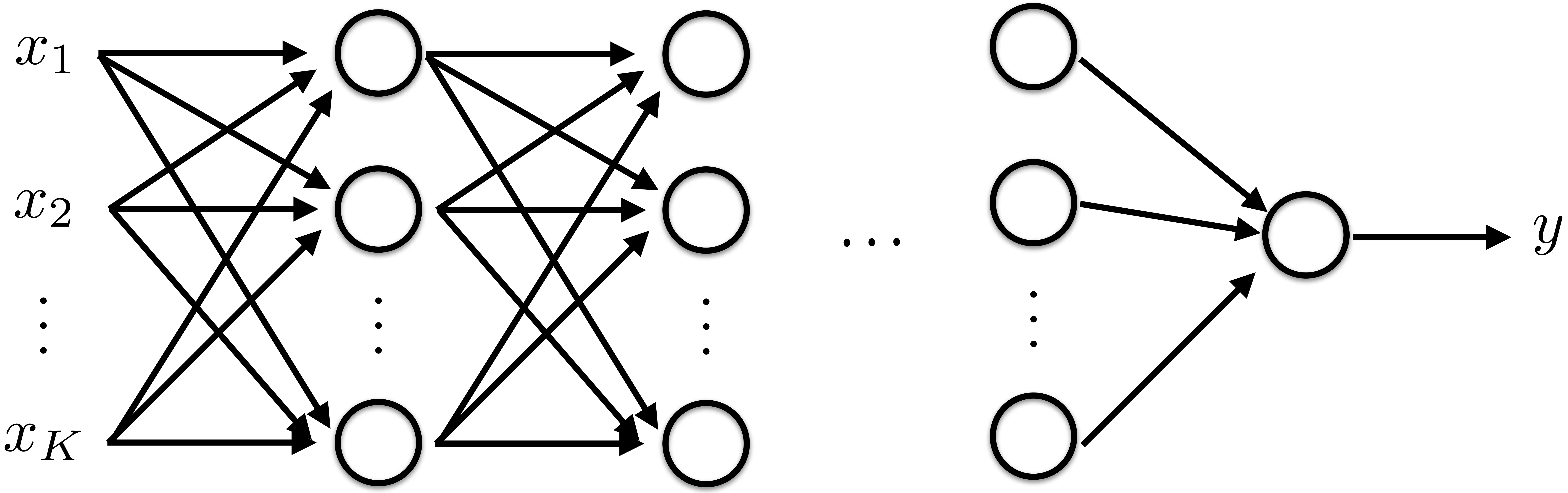




Simple neural network: a combination of neurons to create a (complex) nonlinear function

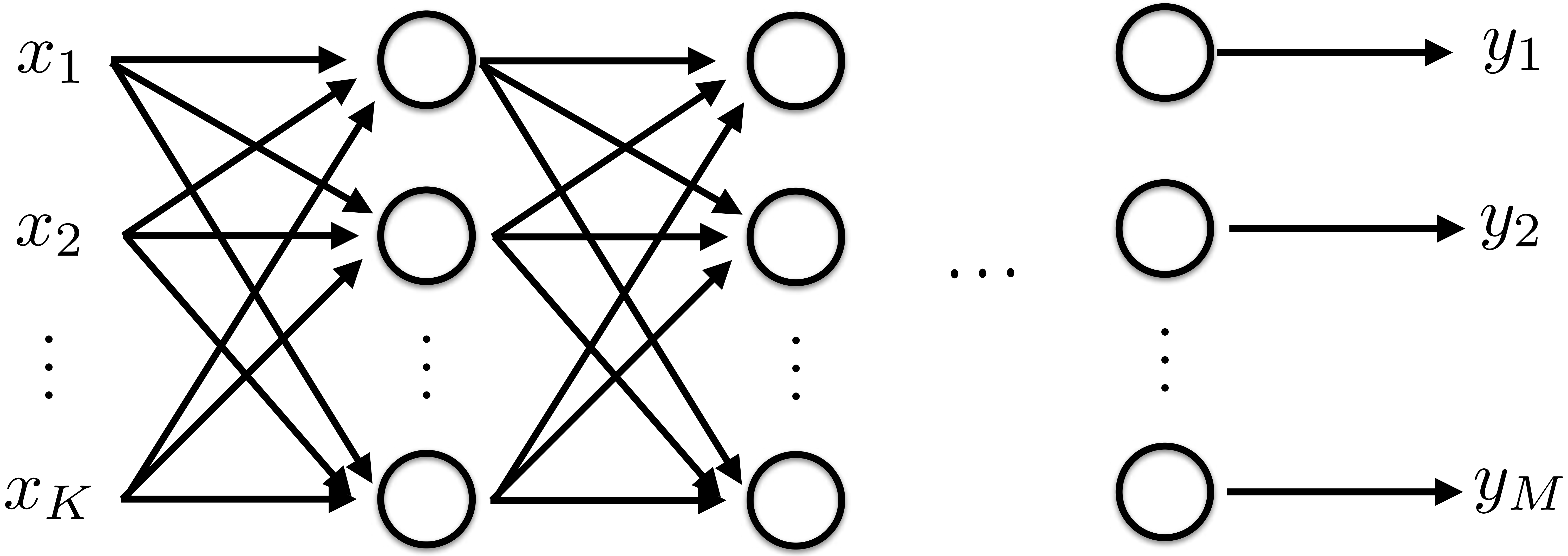
Deep neural network

A neural network with many layers

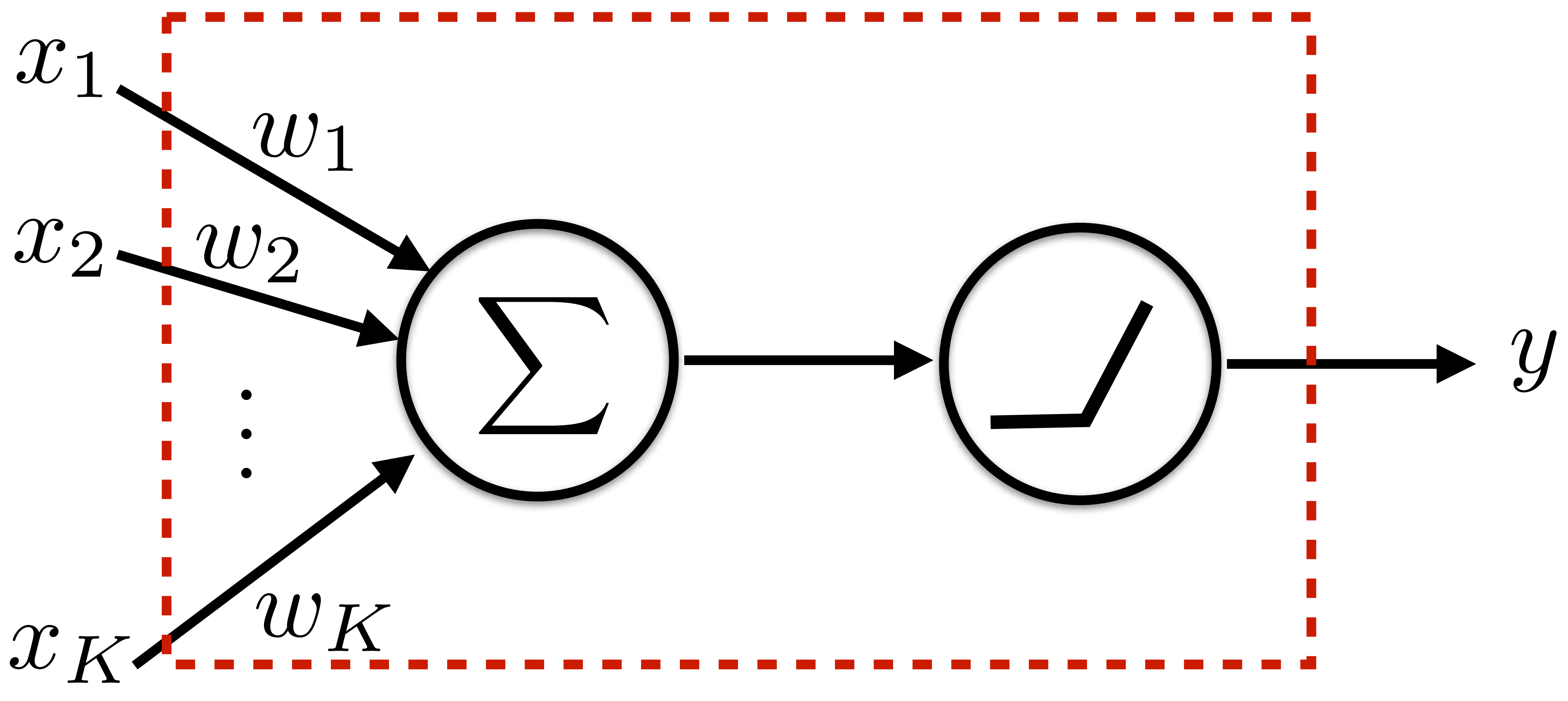


Example of a fully connected deep neural network

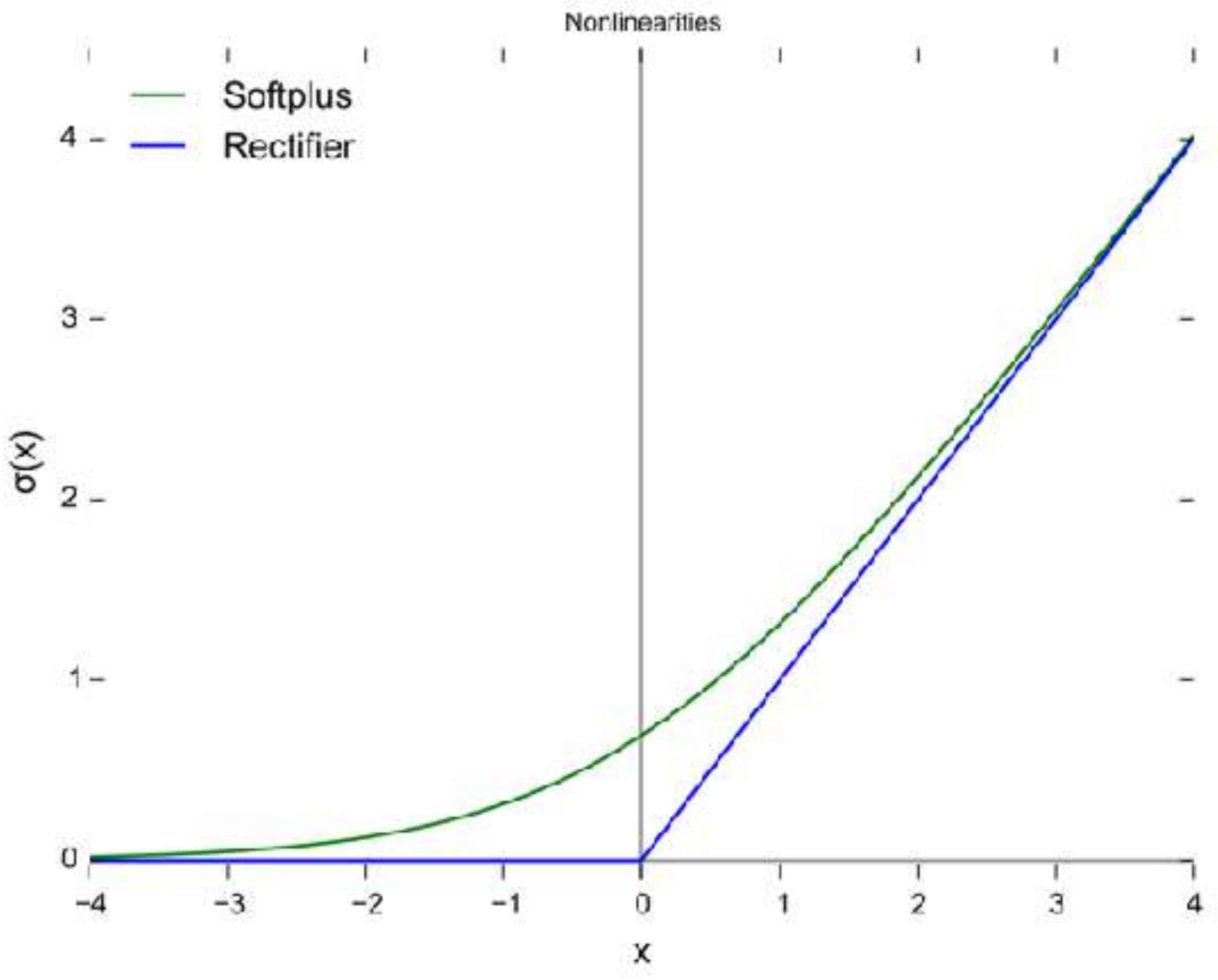
Multi-dimensional output



Different activation functions



one "neuron"



we can replace the sigmoid by other nonlinear functions, popular ones are

- Rectified Linear Units (ReLU) $y = \max\{0, \sum_k w_k x_k\}$
- Softplus $y = \log(1 + e^{\sum_k w_k x_k})$

Different architectures

We can decide:

- How many layers
- How many neurons per layer
- Which activation functions are used
- How layers connect to each other
- etc

Stochastic gradient descent

Usually we cannot do gradient descent using all the data point available in our dataset!

Stochastic gradient descent: randomly select a small number of data points (a mini-batch) and do gradient descent using these points

Libraries to “train” neural networks

PyTorch

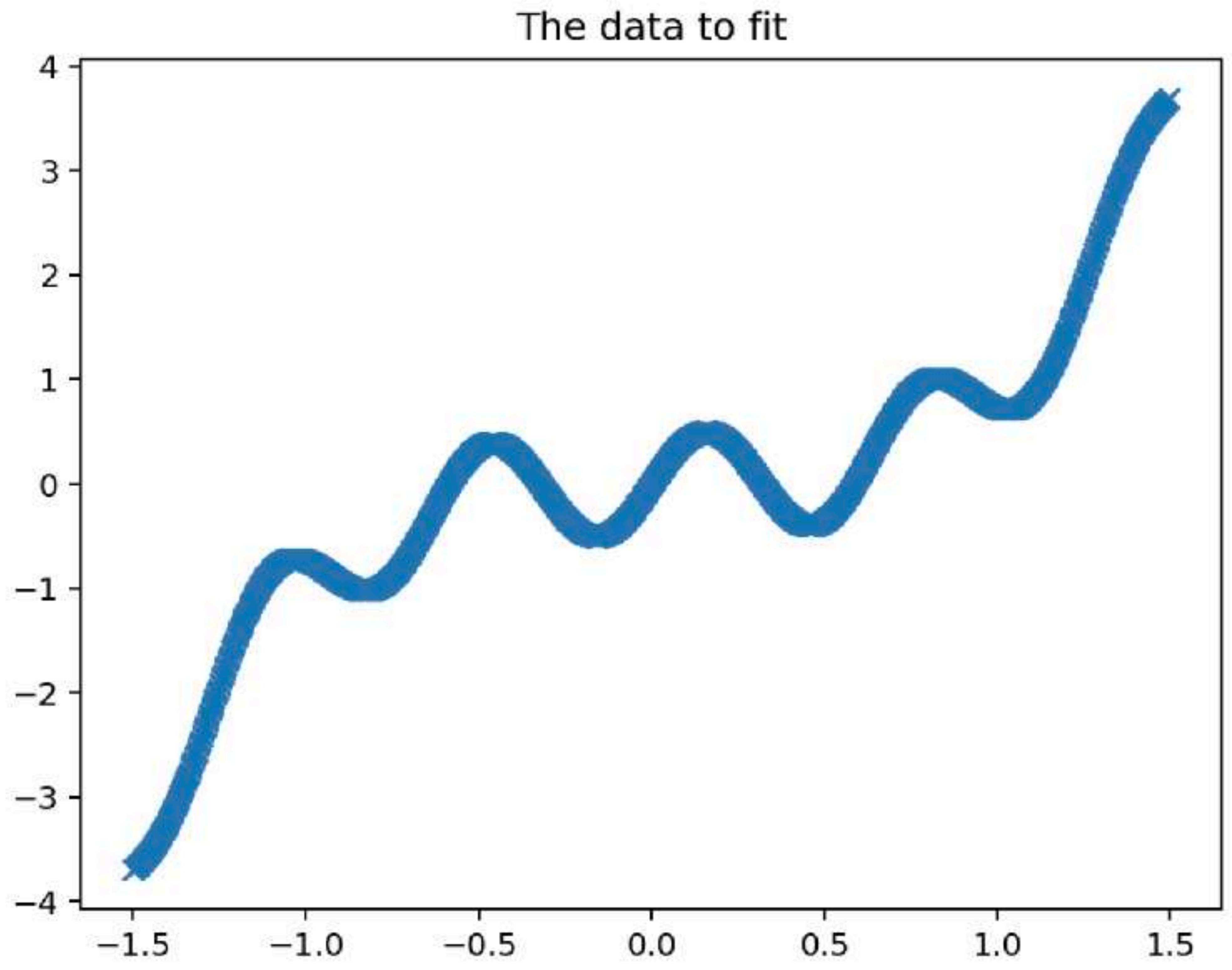
Example

In [1]:

```
%matplotlib notebook

import torch

import numpy as np
import matplotlib as mp
import matplotlib.pylab as plt
```

In [2]:

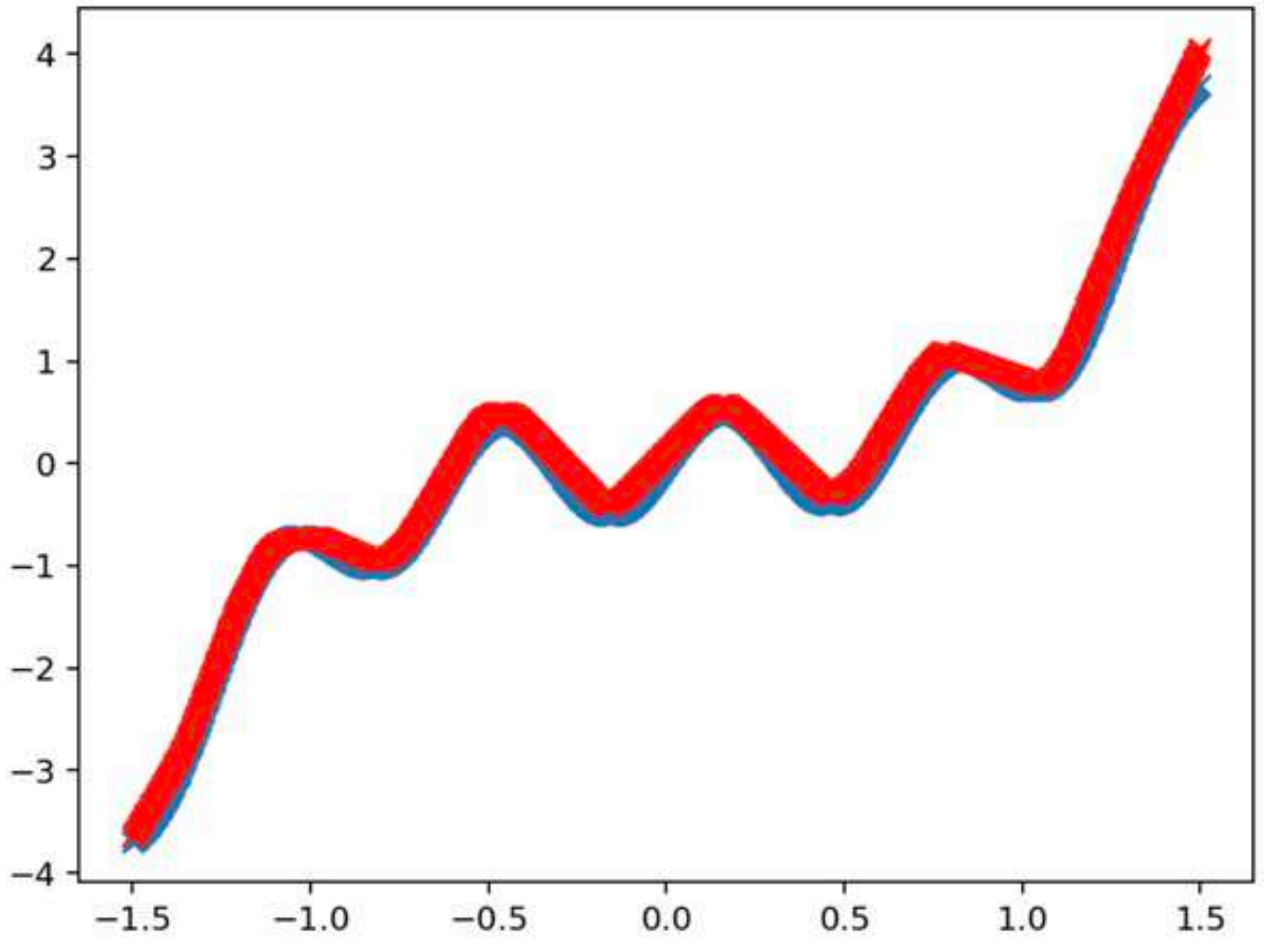
```
## we create the data  to learn from
N = 1000 # number of data points
x = torch.linspace(-1.5,1.5,steps=N).reshape(N, 1)
y = x**3 + 0.5*torch.sin(10*x)

plt.figure()
plt.plot(x.numpy(),y.numpy(), 'x')
plt.title('The data to fit')
```



In [6]:

```
# # we create another model with 3 hidden layers
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# we define the learning rate and select an optimizer
learning_rate = 1e-3
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# we learn doing 5000 iterations
for t in range(10000):
    # sample a mini batch
    sample_index = torch.tensor(np.random.choice(N, batch_size))

    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x[sample_index])

    # Compute the least-square loss.
    loss = loss_fn(y_pred, y[sample_index])

    # use the optimizer object to zero all of the gradients for the variables it will update,
    # i.e. the weights of the model. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # compute gradient of the loss with respect to model parameters (backward autodiff)
    loss.backward()

    # call the step function of the optimizer to make one update of the parameters
    optimizer.step()

y_pred = model(x)
plt.figure()
plt.plot(x.numpy(),y.numpy(), 'x')
plt.plot(x.numpy(),y_pred.detach().numpy(), 'rx')
```

```
# D_in is input dimension;
# H is dimension of the hidden layers; D_out is output dimension.
batch_size = 32
D_in, H, D_out = 1, 64, 1
```



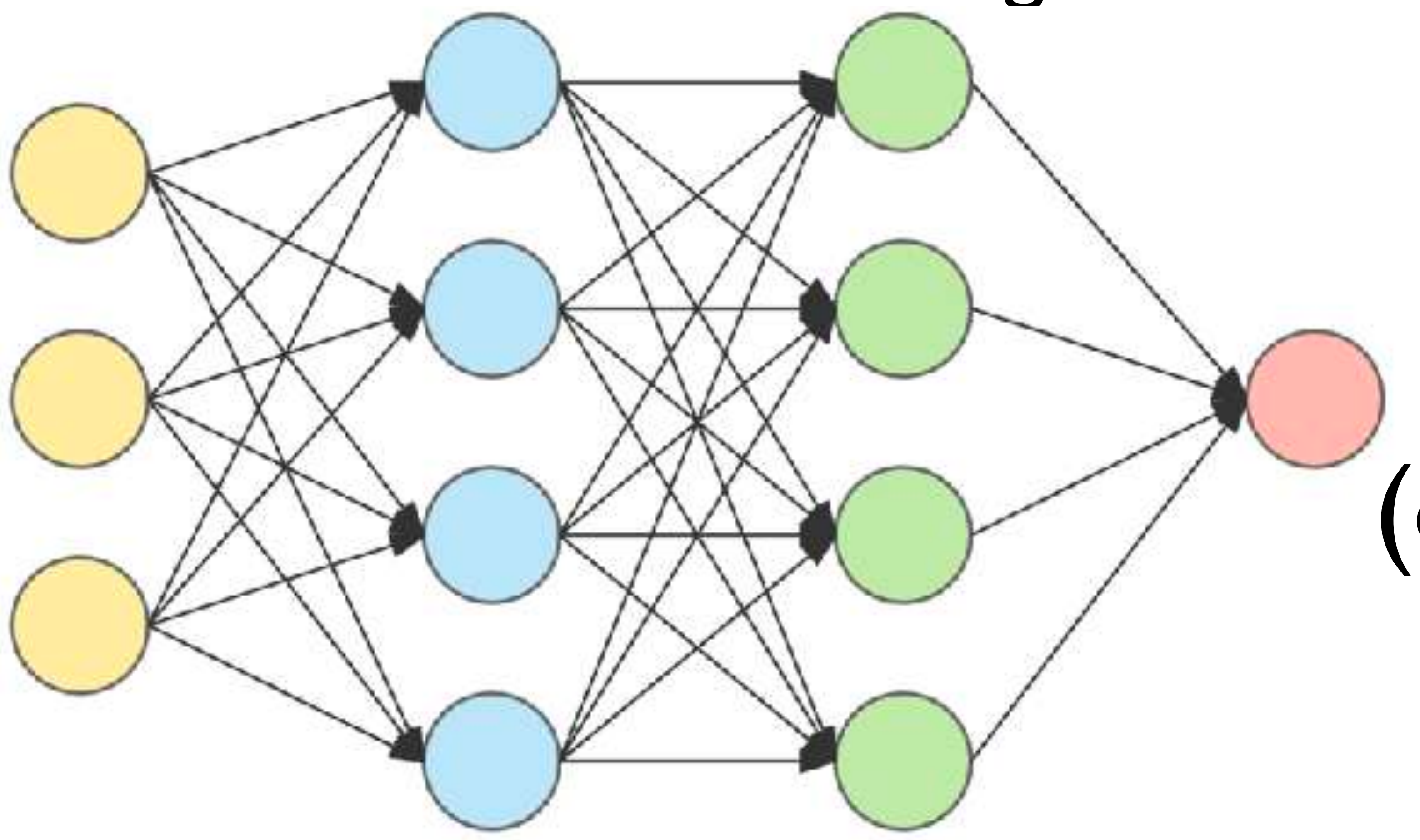
Back to Q-learning

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



Back to Q-learning

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 a neural network

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

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

Back to Q-learning

<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

Back to Q-learning

[Mnih et al., Nature, 2015]

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

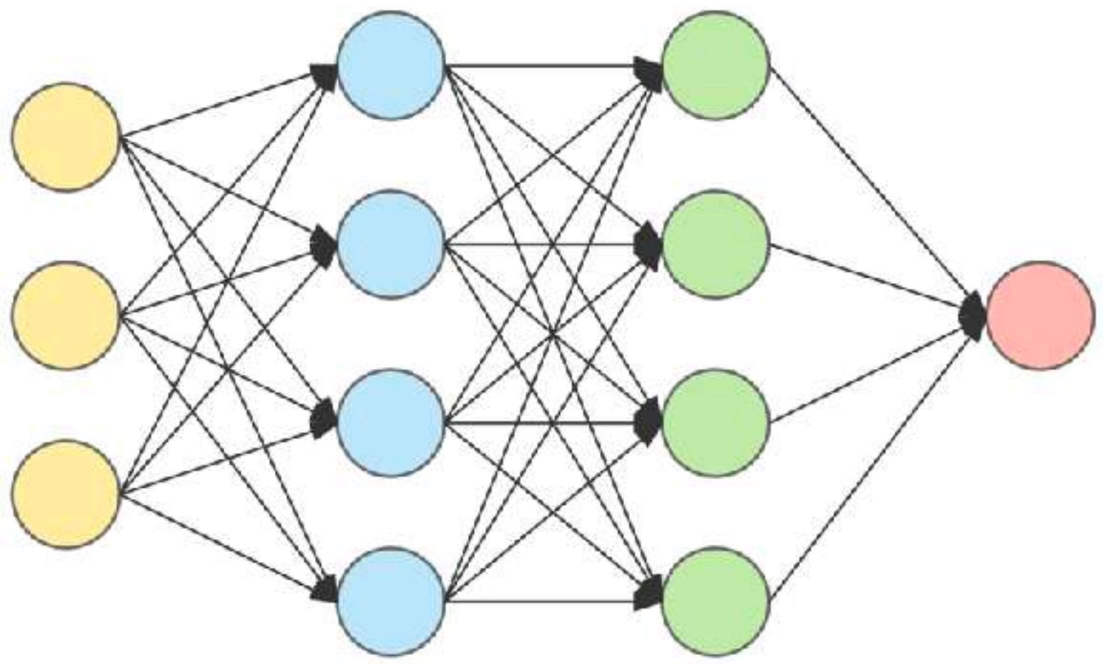
<u>Solution 1)</u>

Use a “replay” memory of a previous samples from which we randomly sample the next training batch (remove correlations)

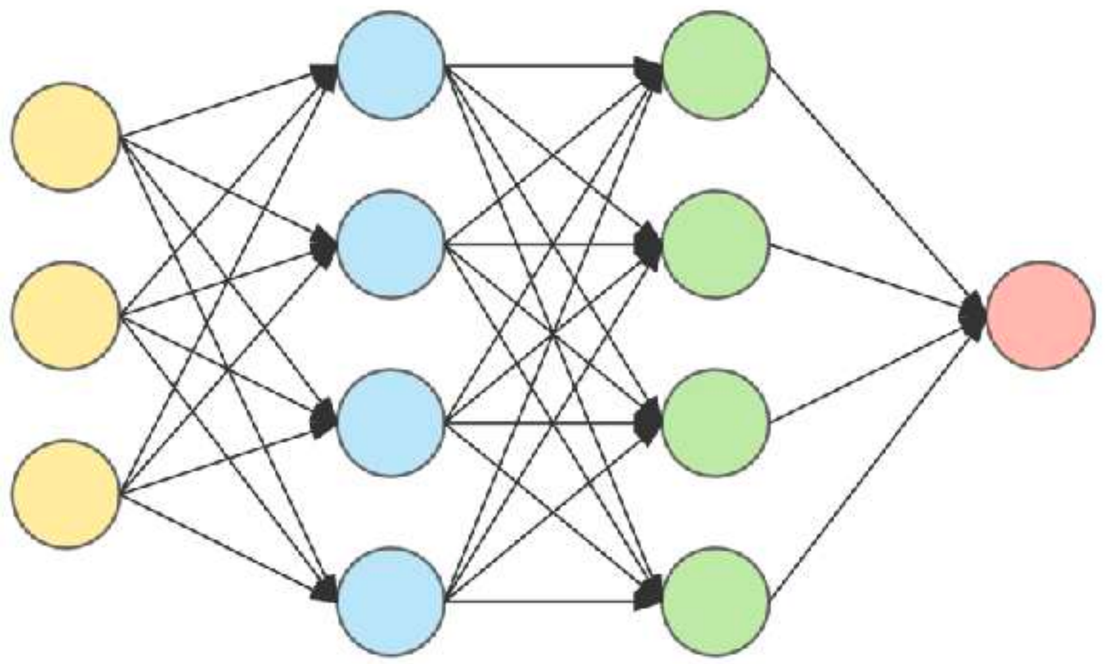
<u>Solution 2)</u>

Use 2 Q-networks to avoid correlations due to updates

Q-network



Target $\hat{Q}$-network



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

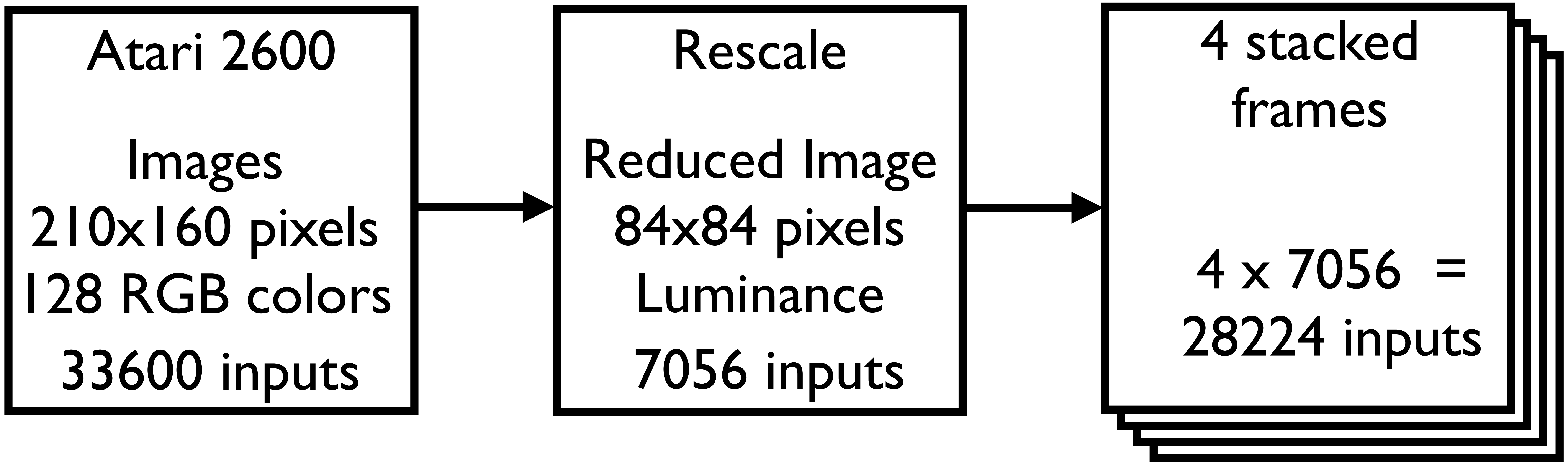
Gradient descent on θ to minimize $\sum_K ||Q(x_k, u_k, \theta) - y_k||^2$

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

Deep Q-network (DQN)

[Mnih et al., Nature, 2015]

Pre-processing states of the system



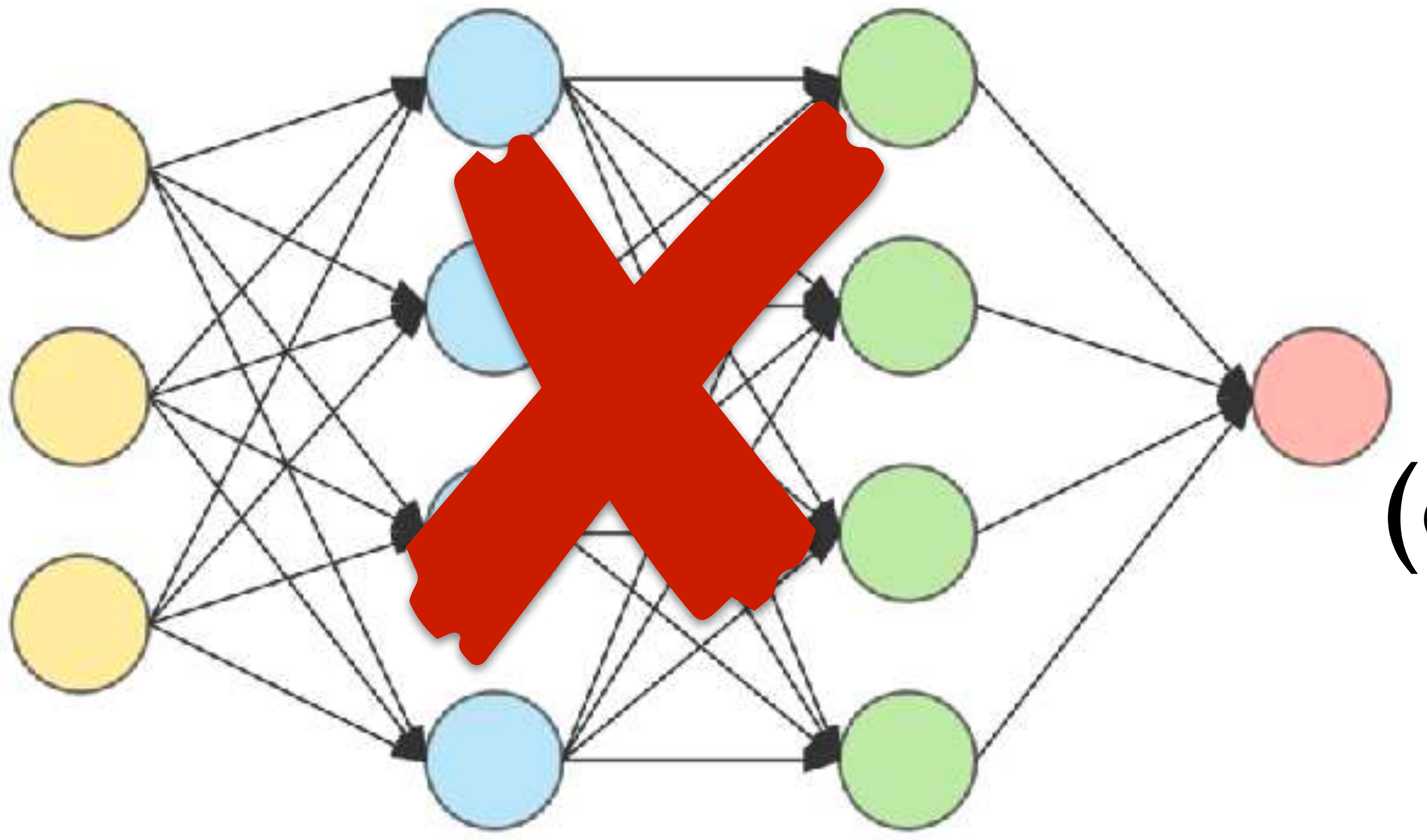
Deep Q-network (DQN)

[Mnih et al., Nature, 2015]

Network Architecture

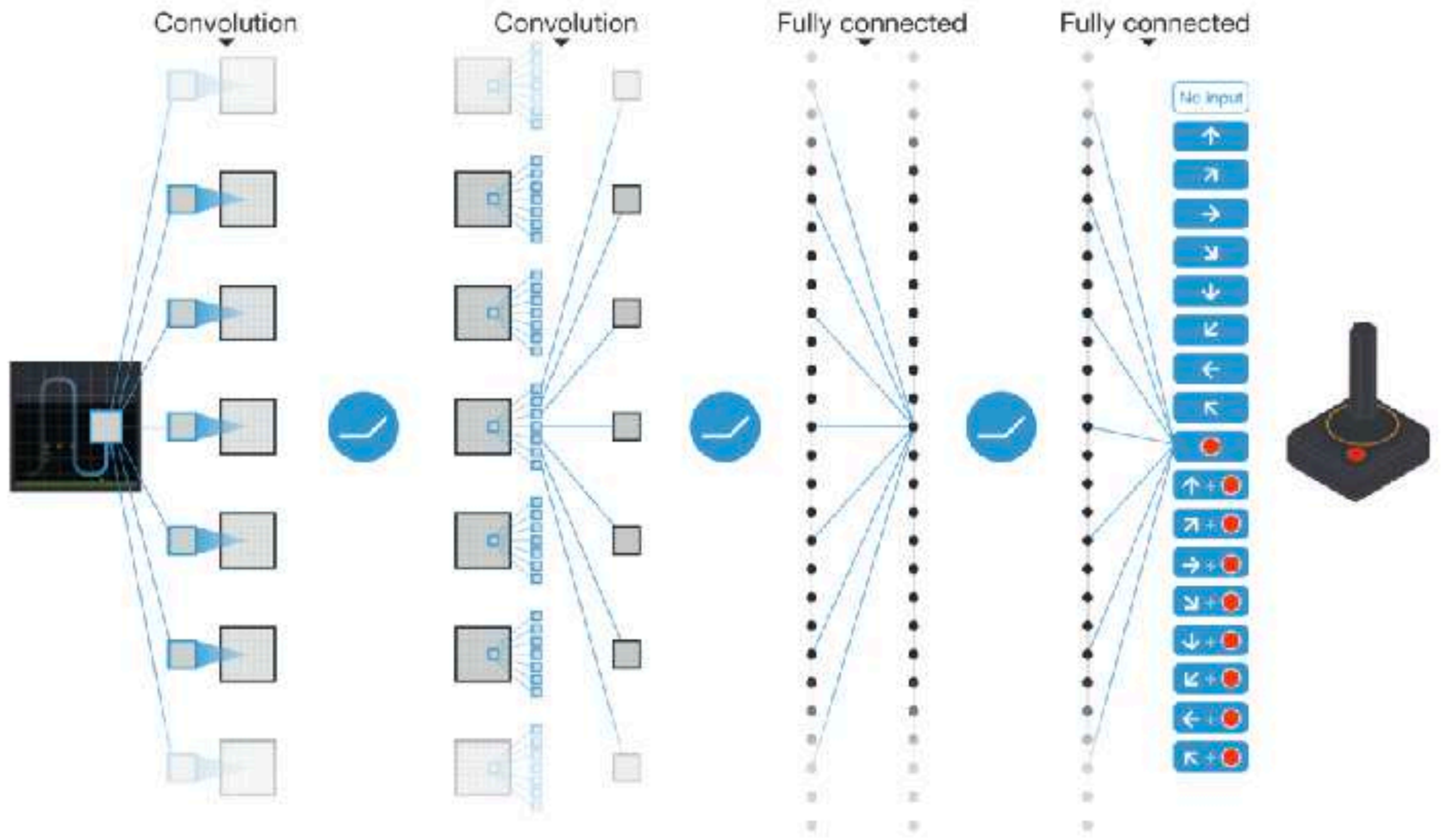
X

U



Q-value
(one number)

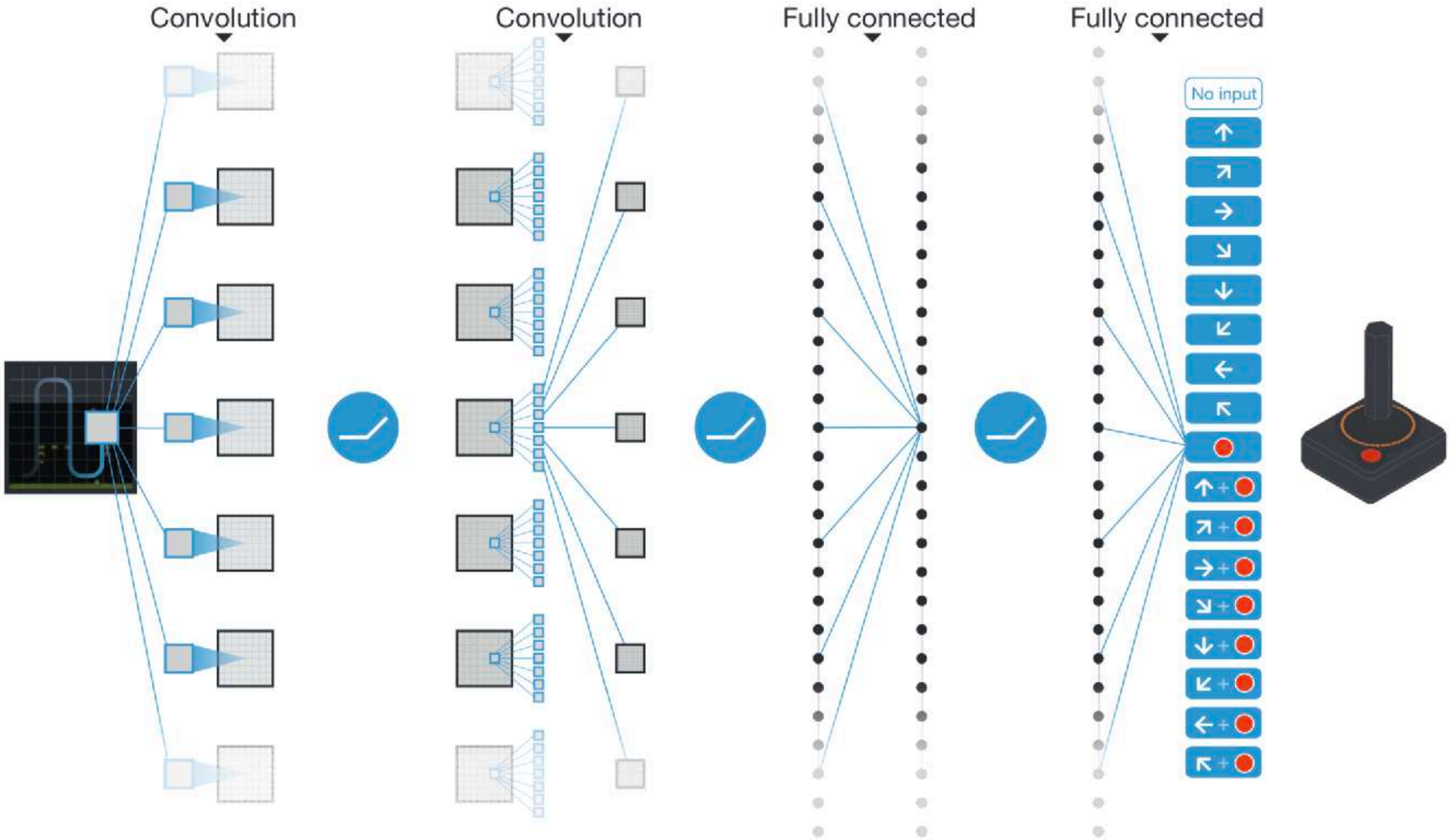
X



One Q-value per action
(multiple outputs)

Deep Q-network (DQN)

[Mnih et al., Nature, 2015]



Q-network architecture for Atari game playing - Each output corresponds to one action entry of the Q function

Deep Q-network (DQN)

[Mnih et al., Nature, 2015]

Training

49 games:

- a different Q network is used for each game
- same parameters for learning each game

mini-batches of size 32

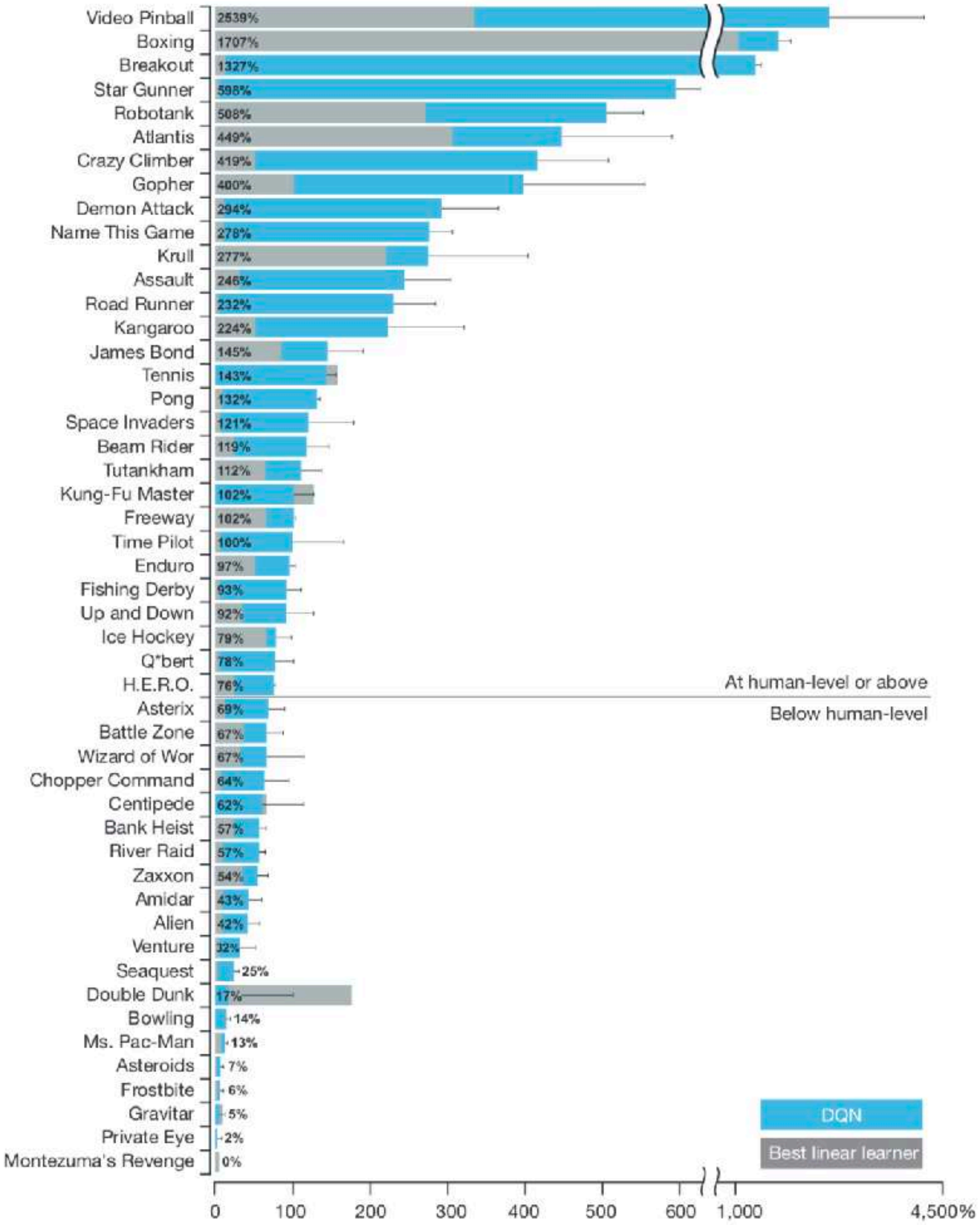
ϵ-greedy with $\epsilon = 1$ at the beginning of learning and linearly decreases until $\epsilon = 0.1$ after first 1 million frames

trained on 50 million frames
=> 32 days of game experience in total!
(the human player was allowed only 2h of training)

replay memory size: 1 million samples (FIFO)

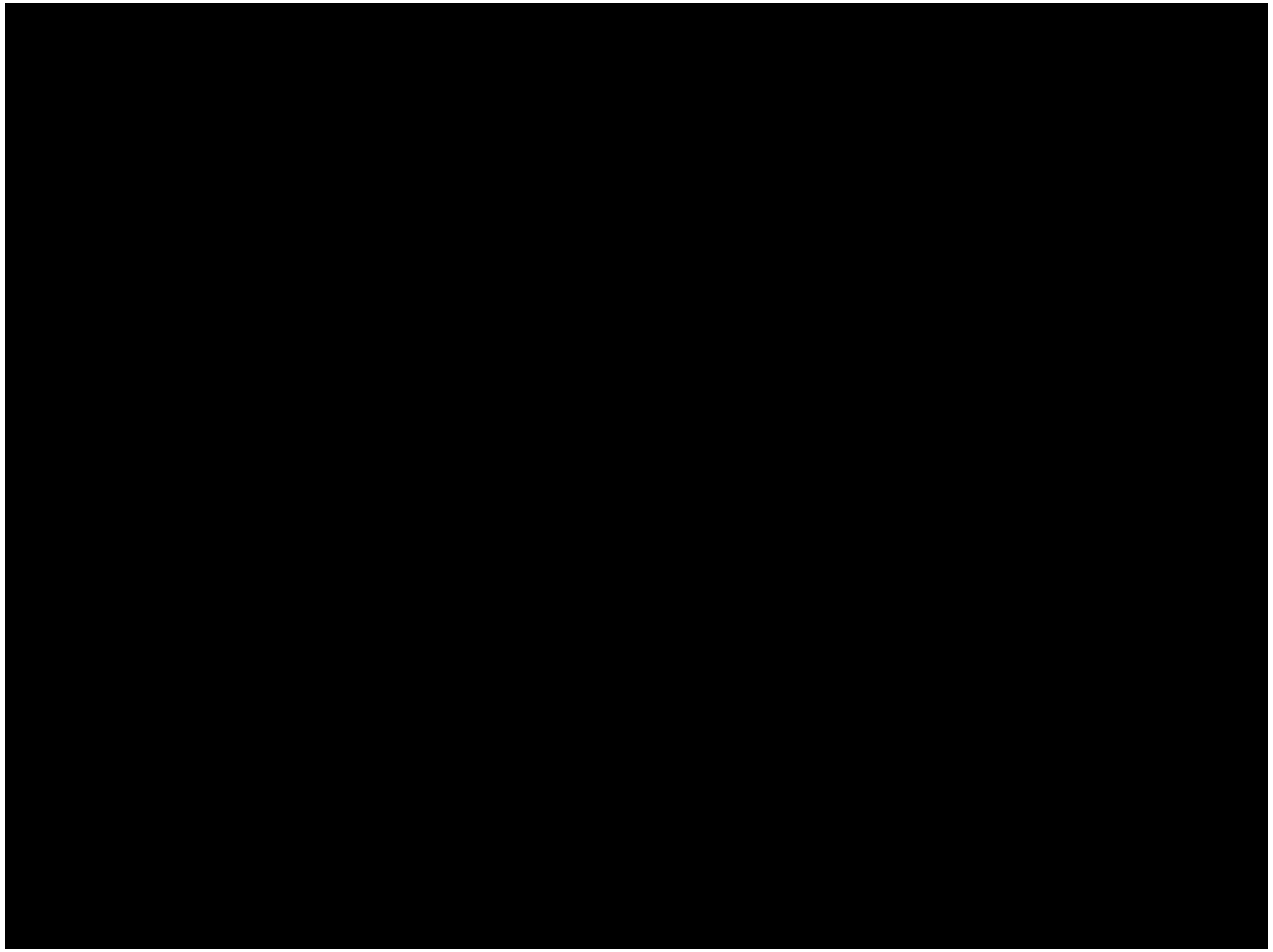
Deep Q-network (DQN)

[Mnih et al., Nature, 2015]



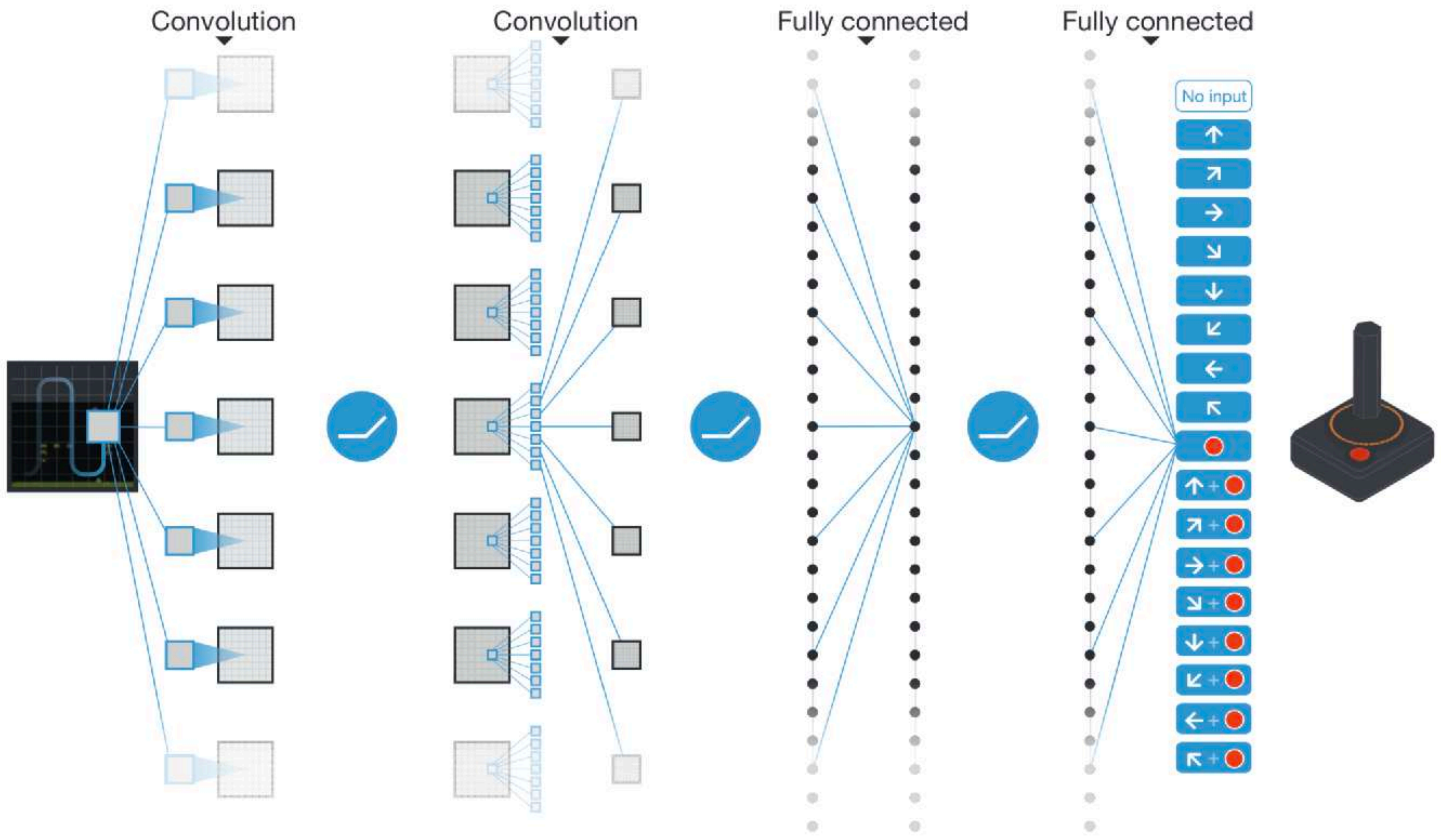
Deep Q-network (DQN)

[Mnih et al., Nature, 2015]



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

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

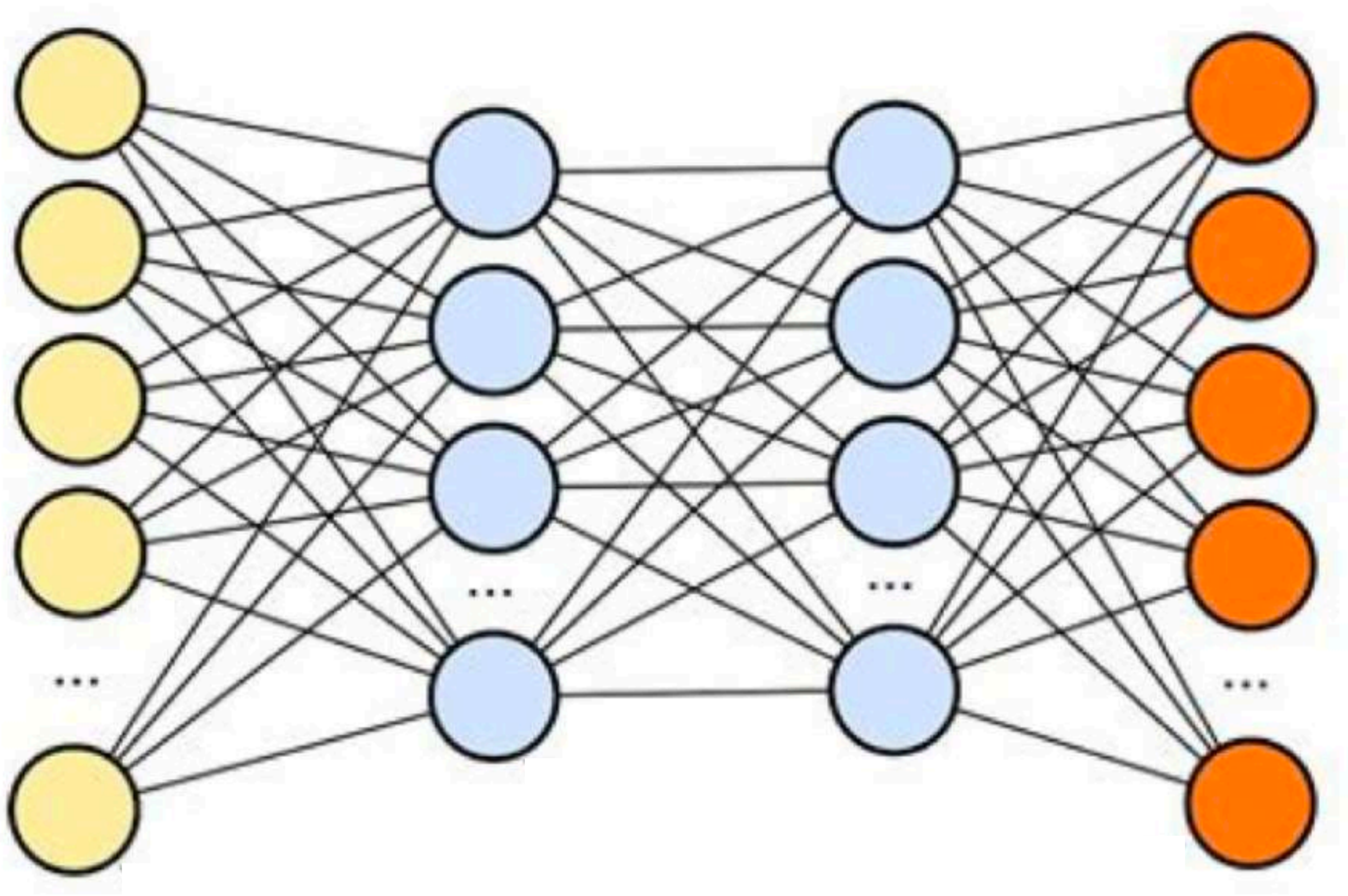
Solution: use another neural network to approximate the min operator (i.e. to approximate the optimal policy)

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



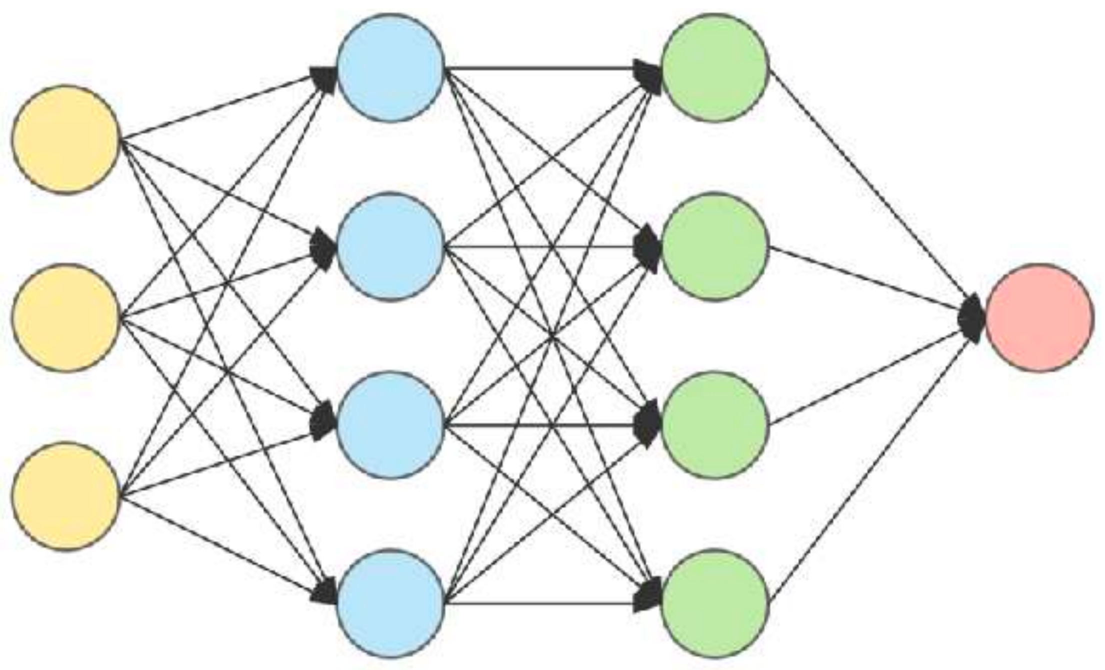
Output U

Deep Deterministic Policy Gradient (DDPG)

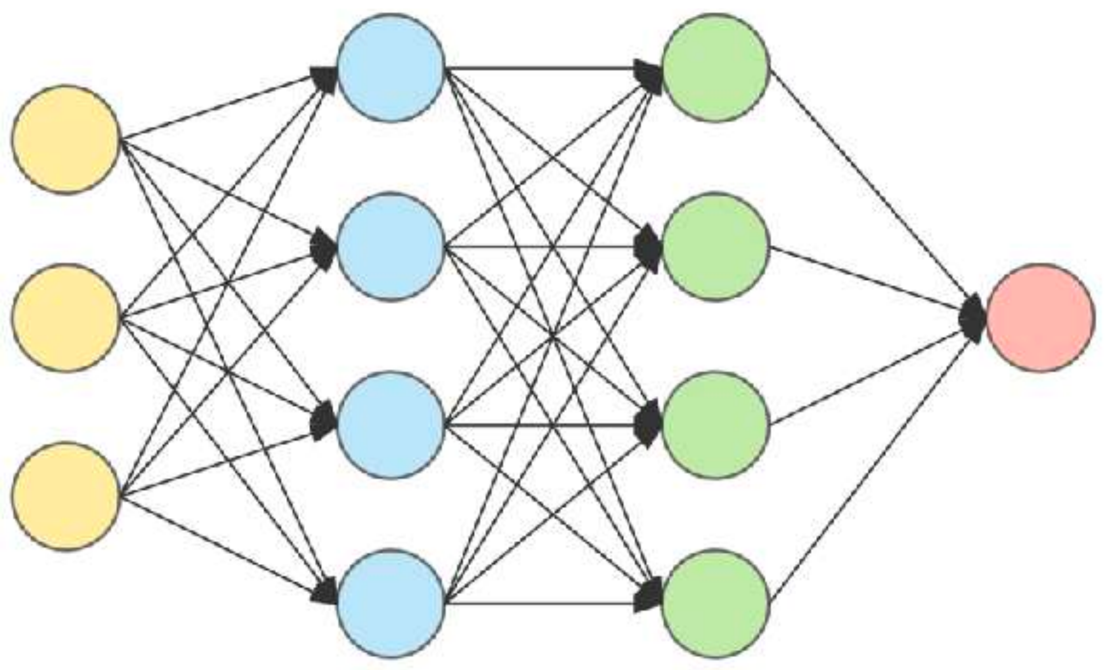
[Lillicrap et al., ICML, 2016]

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

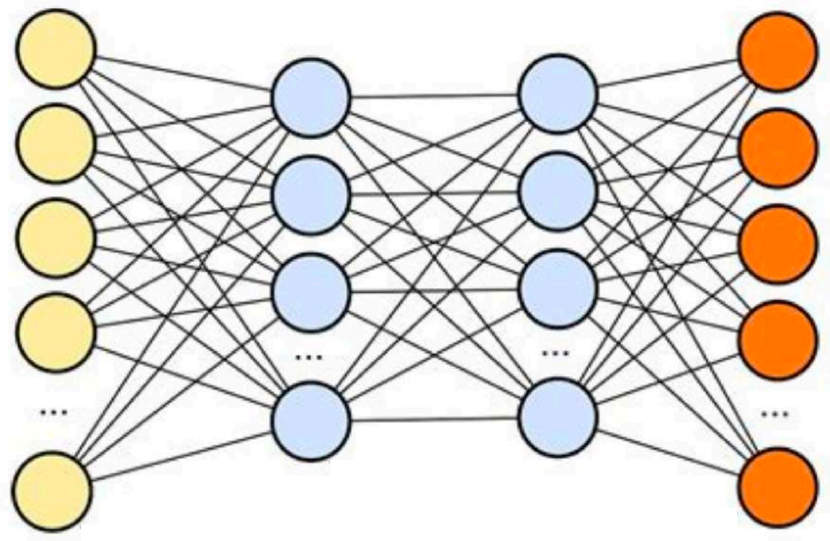
Q-network



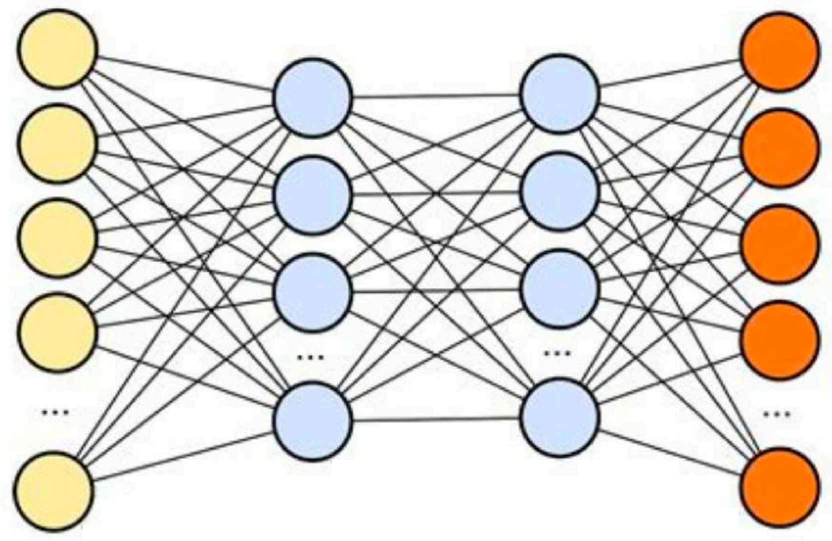
Target $\hat{Q}$-network



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
  - Improve Q: gradient descent on θ to minimize $\frac{1}{B}\sum_{i=0}^{B}\left(Q_\theta(x_i, u_i) - c_i - \alpha Q_{\theta_{target}}(x_{i+1}, \pi_{\phi_{target}}(x_{i+1}))\right)^2$
  - Improve policy: 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]