



SISSA

≡≡≡

Neural Networks: Theory and Practice

Lecture I

Nicoletta Krachmalnicoff

SISSA - Astrophysics and Cosmology PhD school

December, 2019

Who am I?

Office: 514
email: nkrach@sissa.it

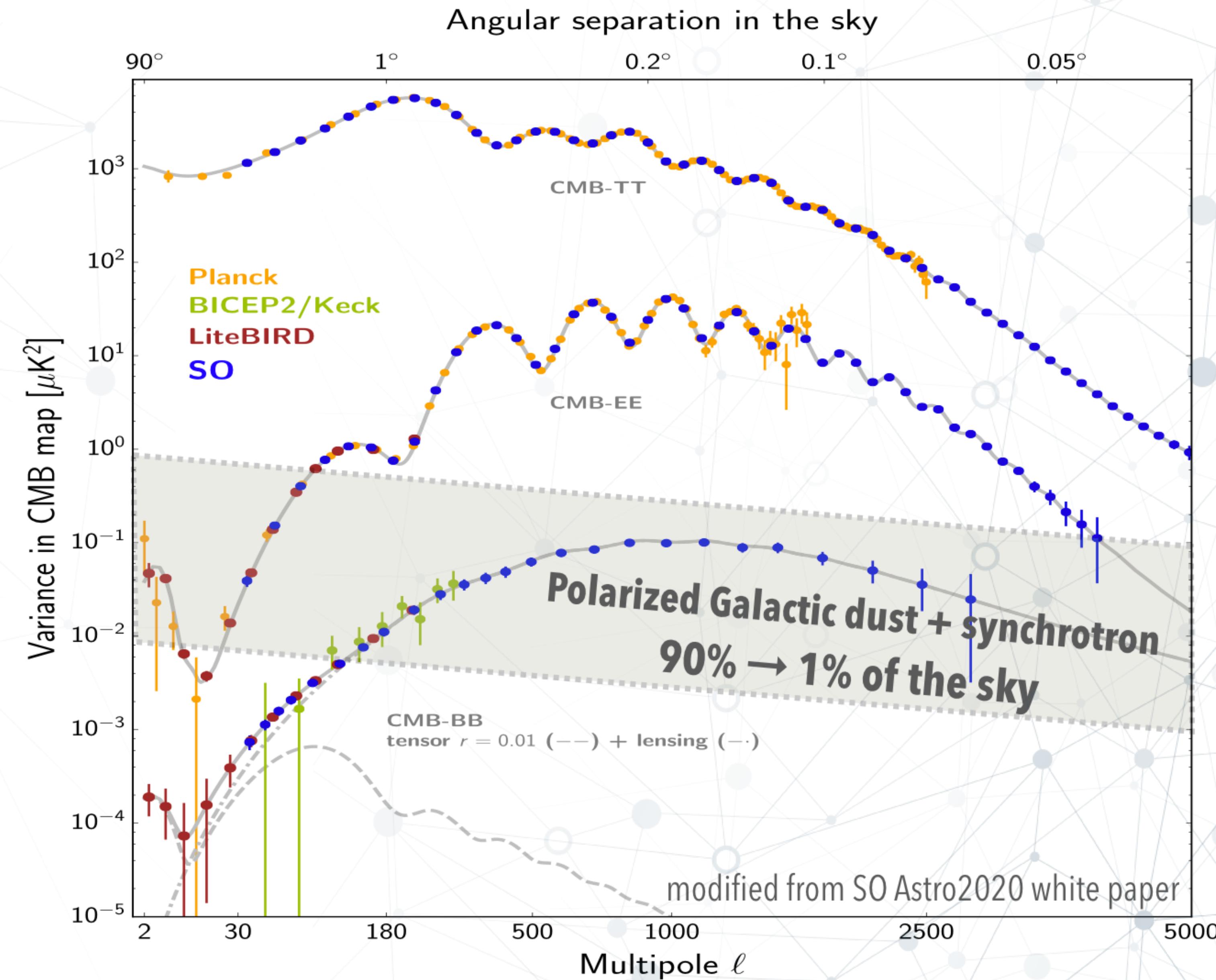
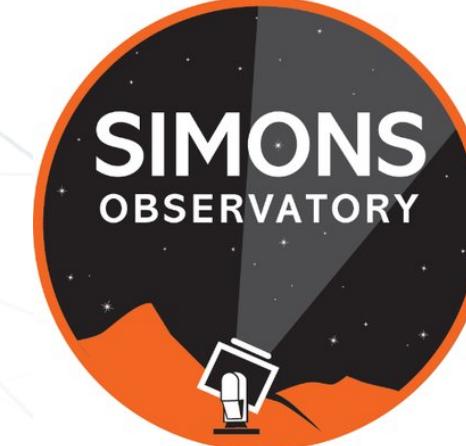
CMB data analysis



planck



POLARBEAR

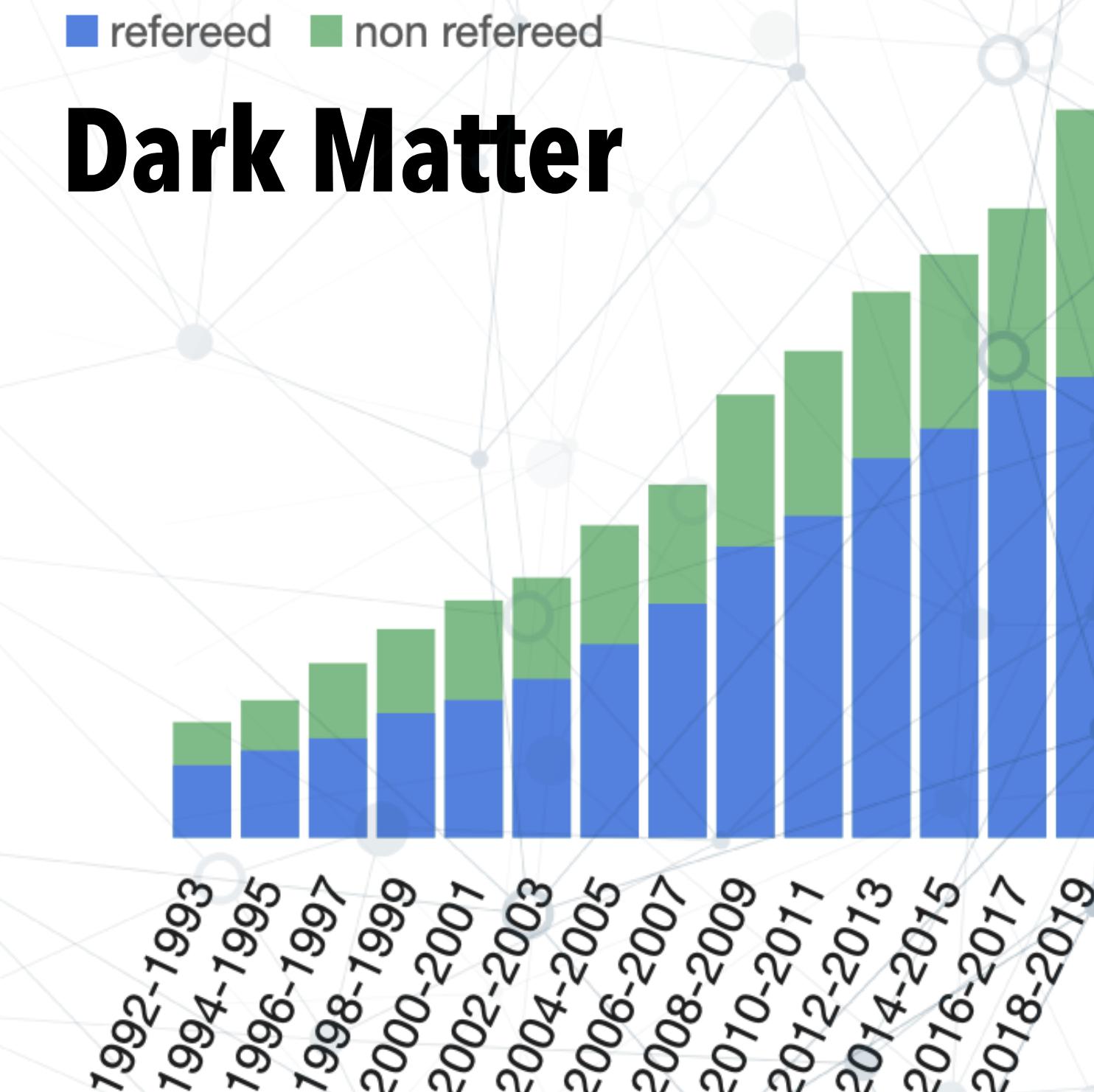
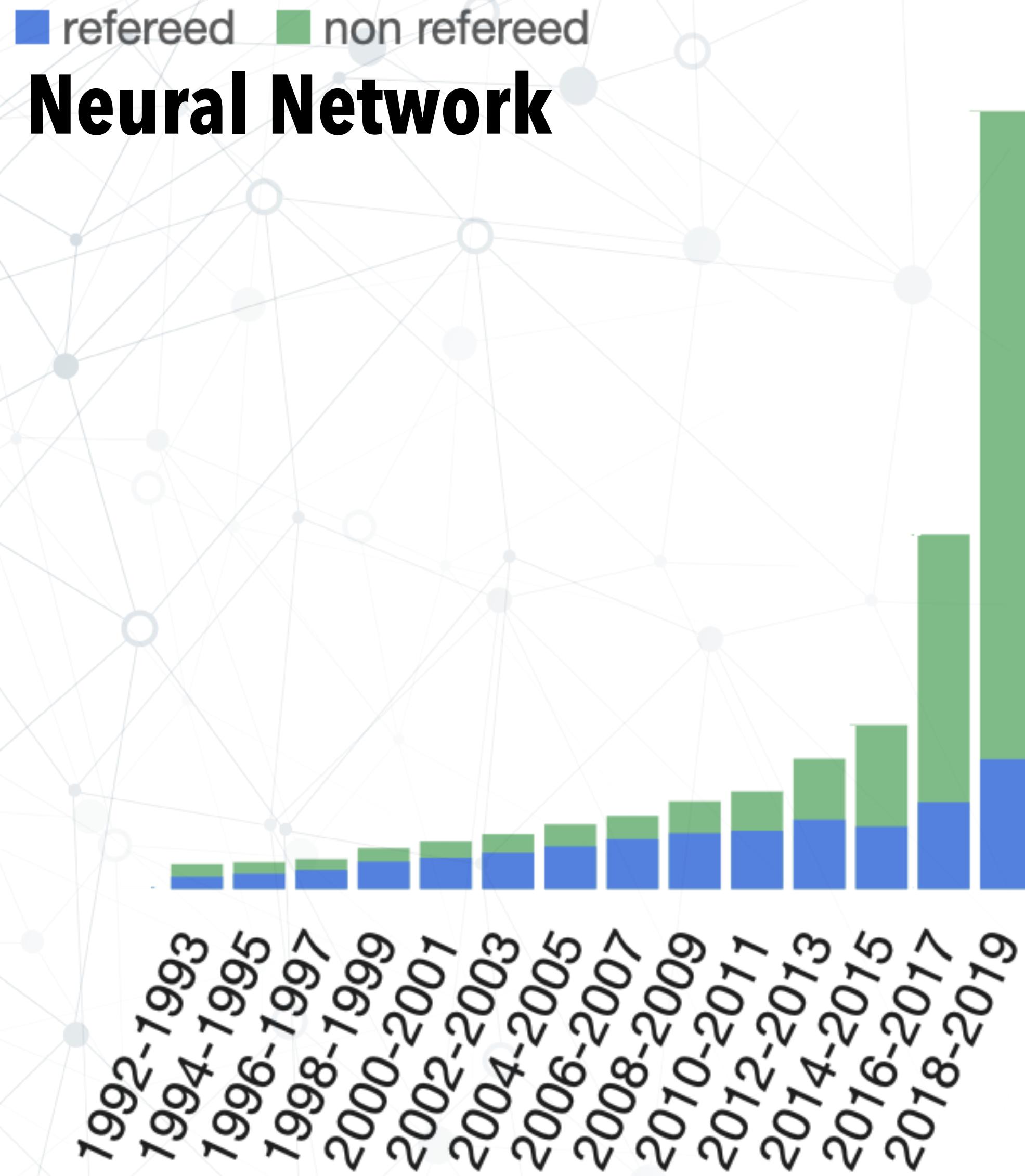


Outline lecture 1

- Introduction: what is a NN?
- Fully connected NNs:
 - Feedforward
 - Loss and cost functions
 - Backpropagation
- **Tutorial 1: coding a simple NN from scratch**
- Libraries for NN development
- Training a NN:
 - Regularization
 - Dropout
 - Batch normalization
 - Mini-batches
- Autoencoders

NN publication metrics

Papers on ADS with "Neural Network " in the text, compared with "Dark Matter"



What is a Neural Network?

$$y = f(\mathbf{x})$$

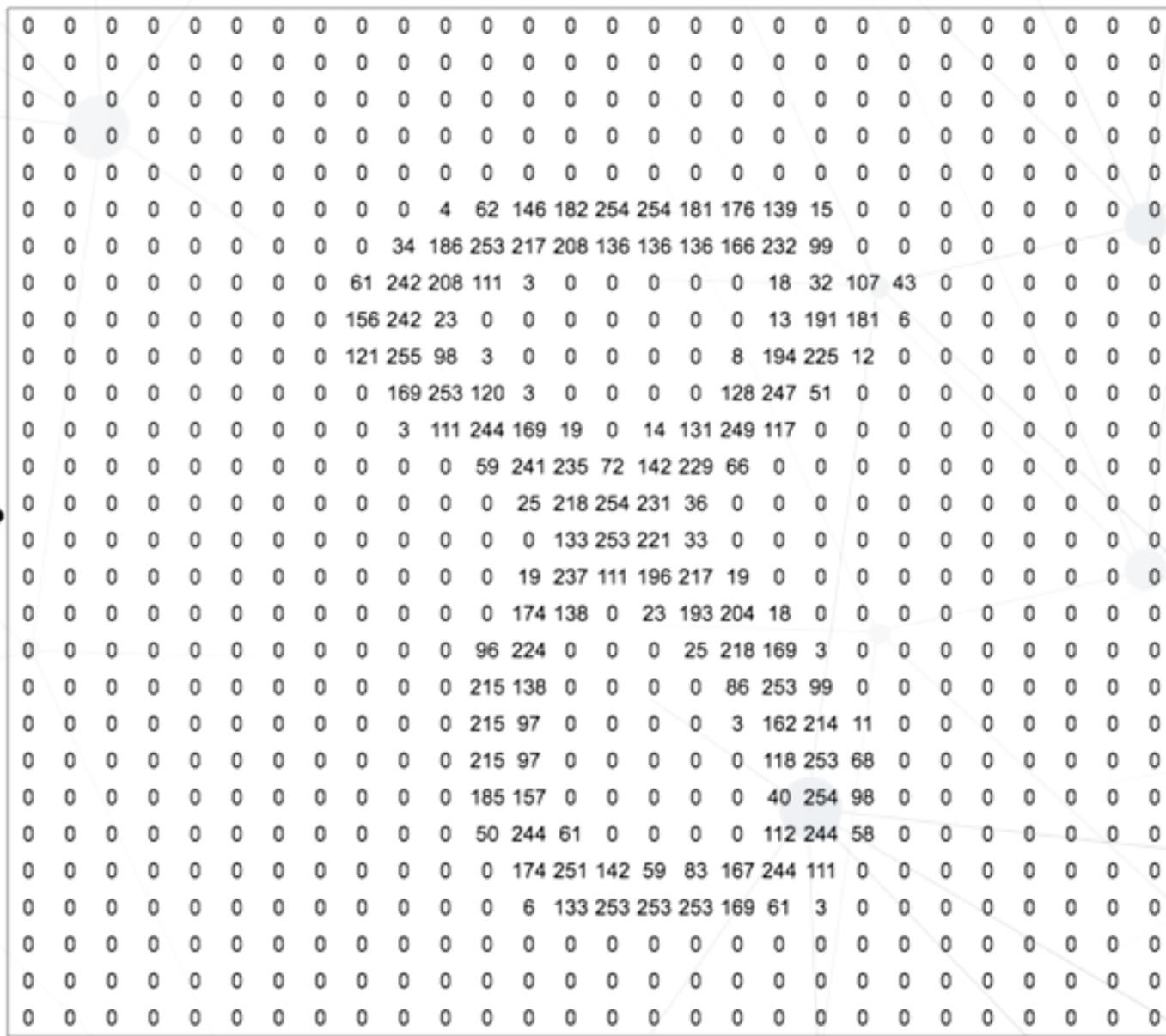
- The goal of a **feed-forward Neural Network** is to find a good enough approximation of the function f that maps inputs into outputs
- for a **classifier** $y = f(\mathbf{x})$: f maps the input \mathbf{x} into the category y
- The Neural Network defines a mapping $f^* = f^*(\mathbf{x}; \theta)$ and finds the value of the parameter θ that results in the best approximation $f^* \sim f$

What is a Neural Network?

$$f(8) = 8$$

A 28x28 pixel grayscale image showing a handwritten digit '4'. The digit is drawn in white on a black background, with some noise pixels visible. It is overlaid on a light gray square grid.

$$f(\alpha) = 2$$



$$f : \mathbb{R}^{n \times m} \rightarrow \mathbb{N}$$

What is a Neural Network?

- Neural networks can, in principle, be used every time there is a unique relation between a input and output
- They are a very powerful tool especially when this relation is unknown
-but it exists a set of data "large enough" for which the output associated at each input is known (training set)

Supervised Learning

What is a Neural Network?

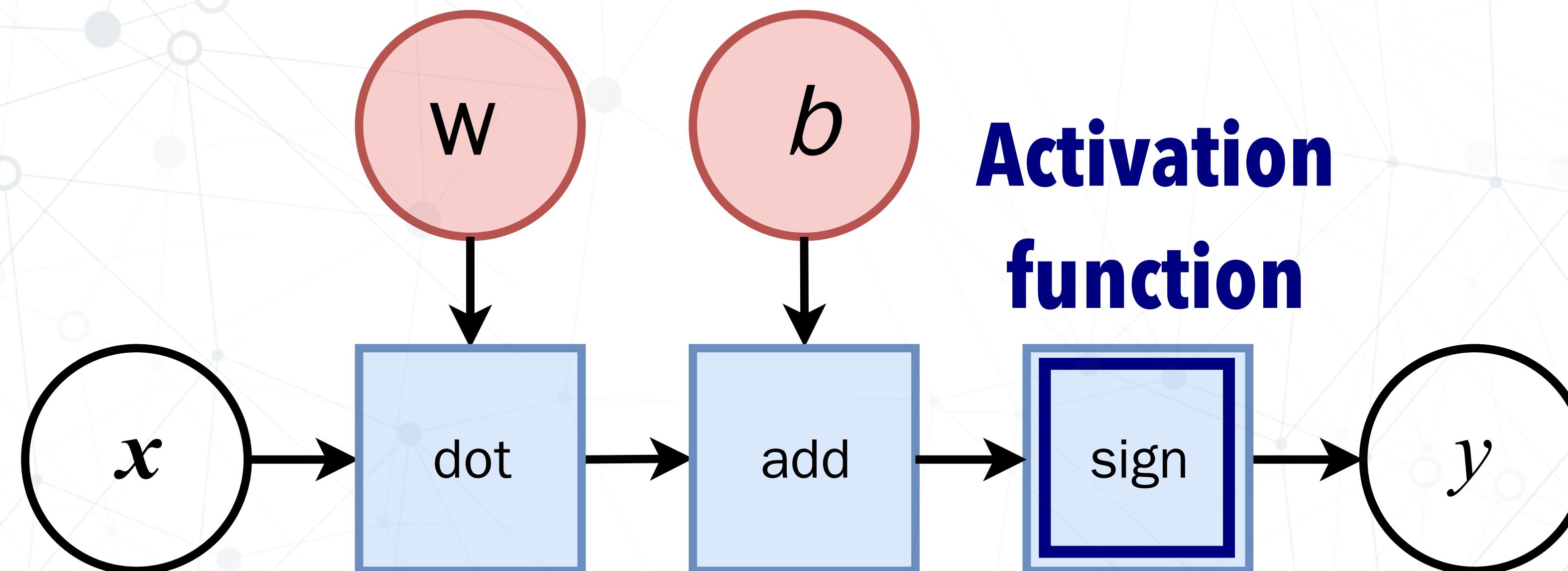
Q: How can a NN approximate very complex unknown functions?

A: By recursively apply **non-linear** activation
functions to a linear combination of input elements

Perceptron

see also previous lectures

- First model of artificial neuron
- Introduced in the late 50s by Frank Rosenbalt



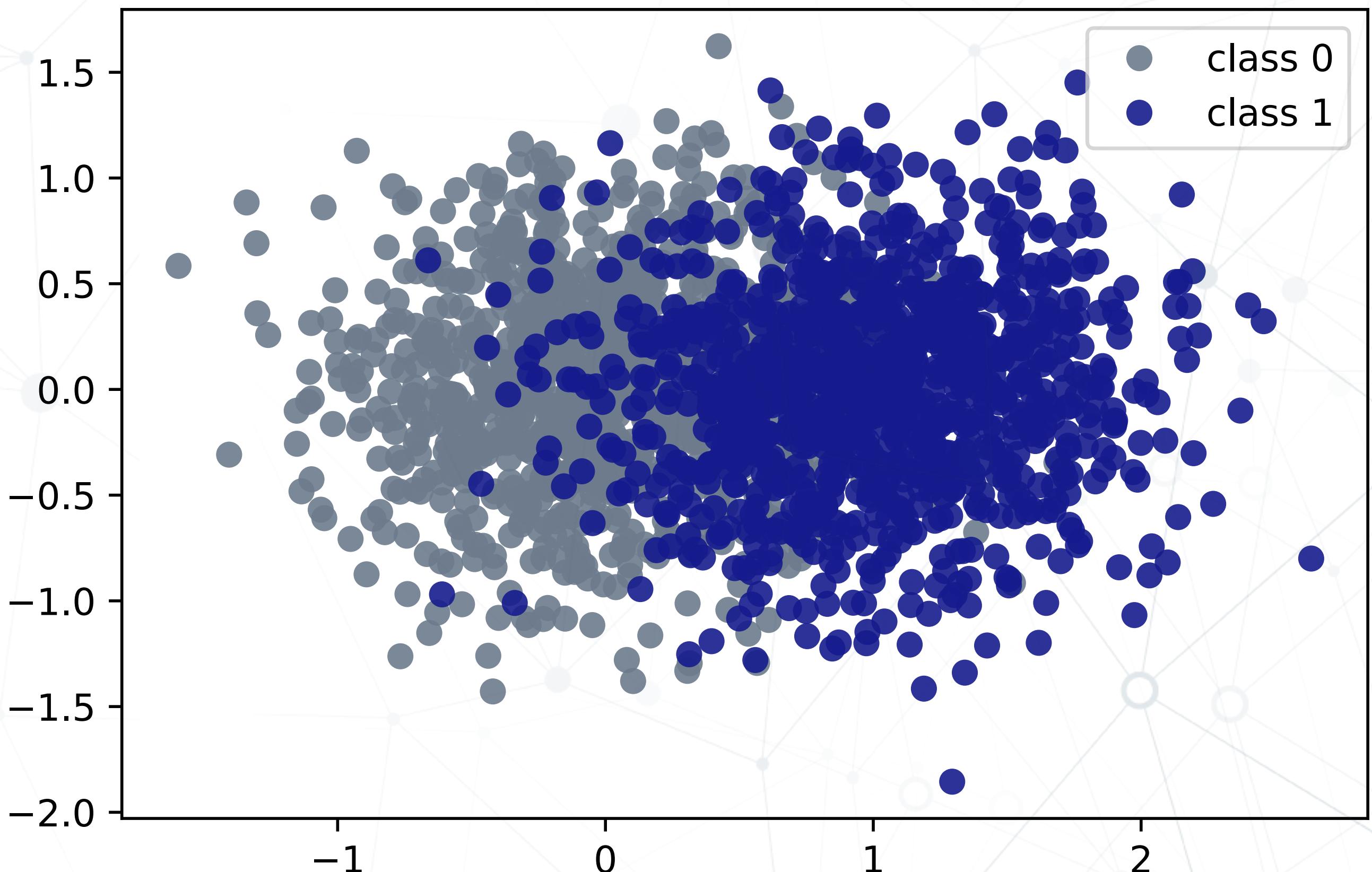
$$\text{sign}(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Sigmoid activation function

see also previous lectures

Training data (\mathbf{x}, y) with:

- $\mathbf{x} \in \mathbb{R}$
- $y \in \{0, 1\}$



Class populations are Gaussian, with covariance Σ

$$P(\mathbf{x}|y) = \frac{1}{\sqrt{(2\pi)^p |\Sigma|}} \exp \left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_y)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_y) \right)$$

Sigmoid activation function

see also previous lectures

With Bayes' theorem we have:

$$\begin{aligned} P(Y = 1|\mathbf{x}) &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x})} \\ &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x}|Y = 0)P(Y = 0) + P(\mathbf{x}|Y = 1)P(Y = 1)} \\ &= \frac{1}{1 + \frac{P(\mathbf{x}|Y=0)P(Y=0)}{P(\mathbf{x}|Y=1)P(Y=1)}}. \end{aligned}$$

and introducing the **sigmoid function** σ

$$\sigma(x) = \frac{1}{1 + \exp(-x)},$$

we get:

$$P(Y = 1|\mathbf{x}) = \sigma \left(\log \frac{P(\mathbf{x}|Y = 1)}{P(\mathbf{x}|Y = 0)} + \log \frac{P(Y = 1)}{P(Y = 0)} \right).$$

Sigmoid activation function

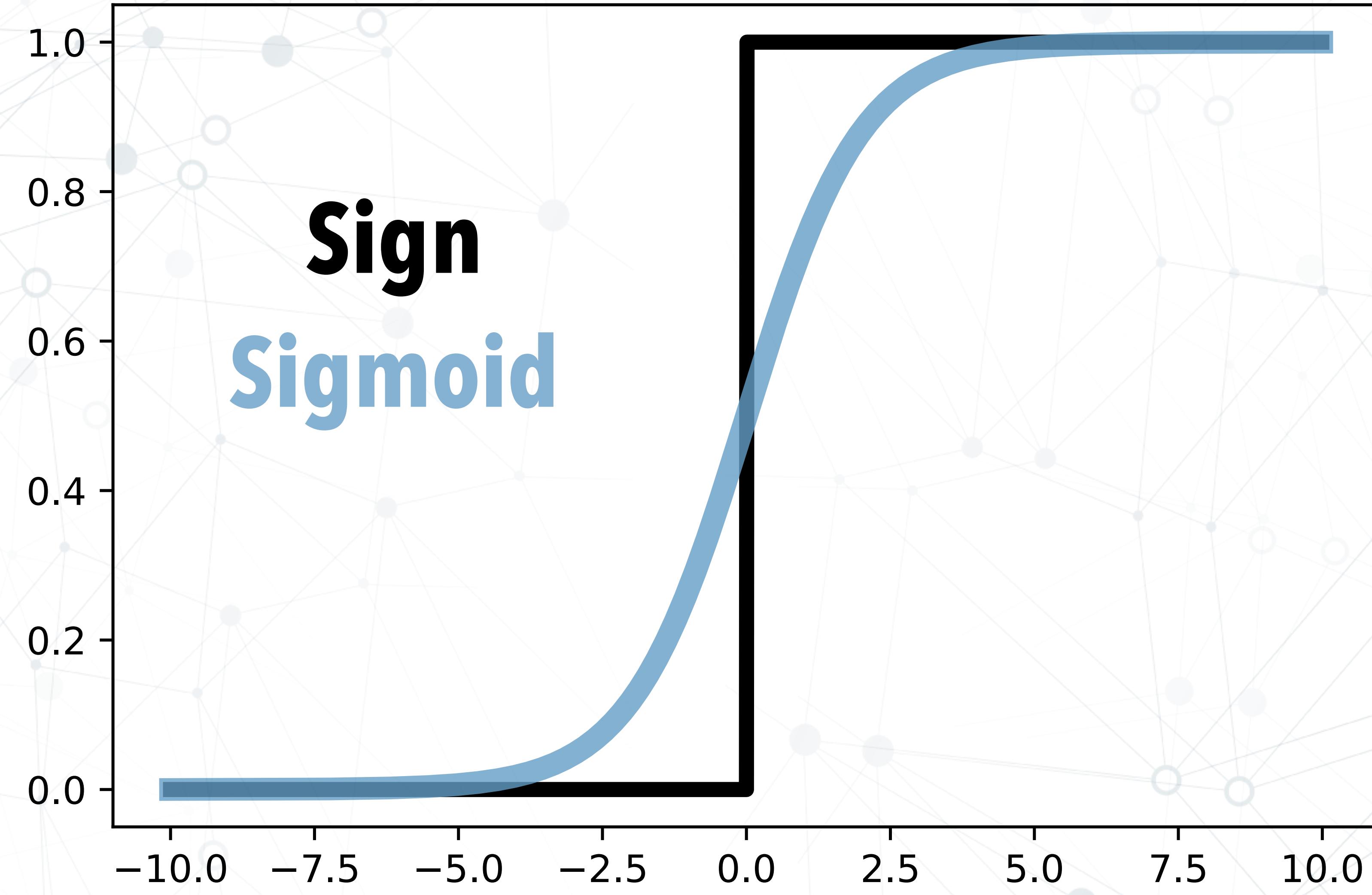
see also previous lectures

$$\begin{aligned} P(Y = 1 | \mathbf{x}) &= \sigma \left(\log \frac{P(\mathbf{x}|Y = 1)}{P(\mathbf{x}|Y = 0)} + \underbrace{\log \frac{P(Y = 1)}{P(Y = 0)}}_a \right) \\ &= \sigma (\log P(\mathbf{x}|Y = 1) - \log P(\mathbf{x}|Y = 0) + a) \\ &= \sigma \left(-\frac{1}{2}(\mathbf{x} - \mu_1)^T \Sigma^{-1} (\mathbf{x} - \mu_1) + \frac{1}{2}(\mathbf{x} - \mu_0)^T \Sigma^{-1} (\mathbf{x} - \mu_0) + a \right) \\ &= \sigma \left(\underbrace{(\mu_1 - \mu_0)^T \Sigma^{-1} \mathbf{x}}_{\mathbf{w}^T} + \underbrace{\frac{1}{2}(\mu_0^T \Sigma^{-1} \mu_0 - \mu_1^T \Sigma^{-1} \mu_1)}_b + a \right) \\ &= \sigma (\mathbf{w}^T \mathbf{x} + b) \end{aligned}$$

The probability that data belong to a given class can be exactly written with the **sigmoid function** σ

Sigmoid activation function

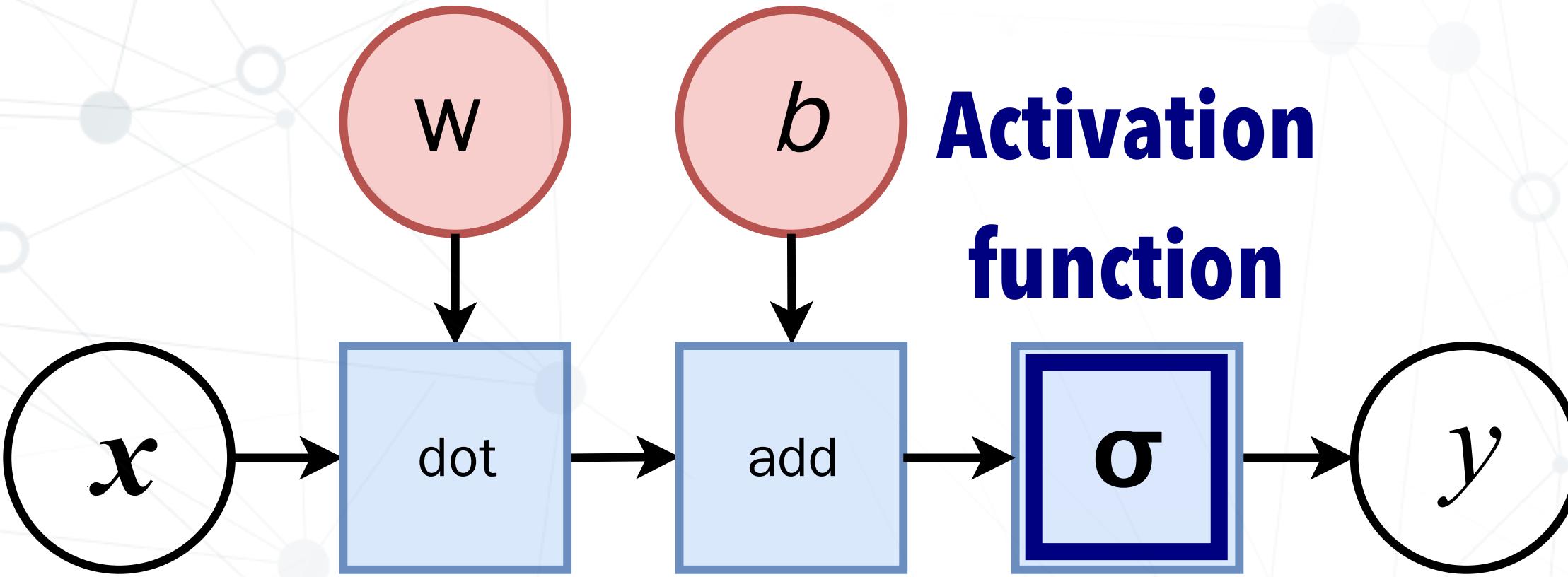
see also previous lectures



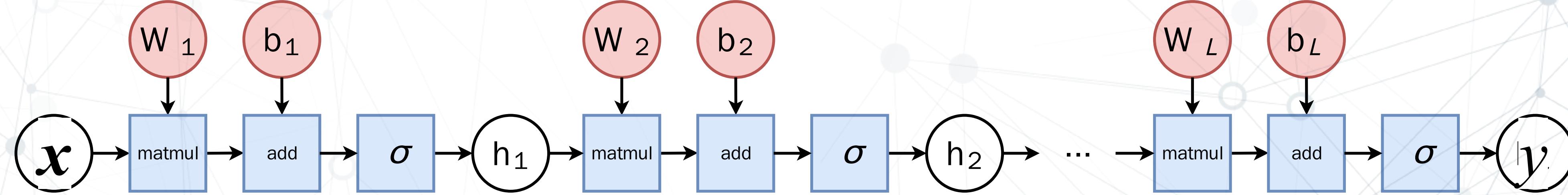
Sigmoid activation function

see also previous lectures

- we can write the same computational graph as for perception but with the sigmoid activation function

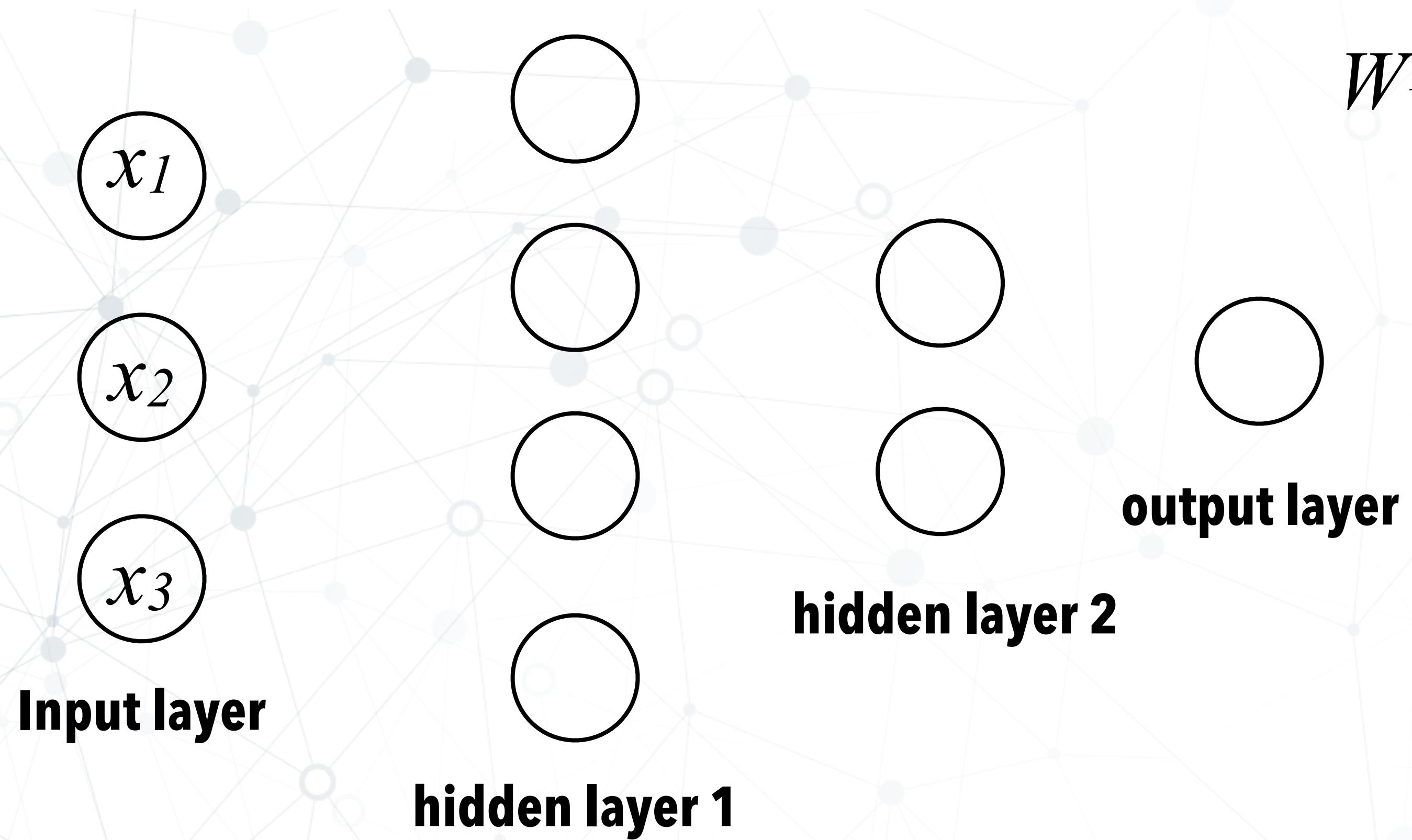


- this represents the **building block of a fully connected feed-forward Neural Network**



see previous lectures

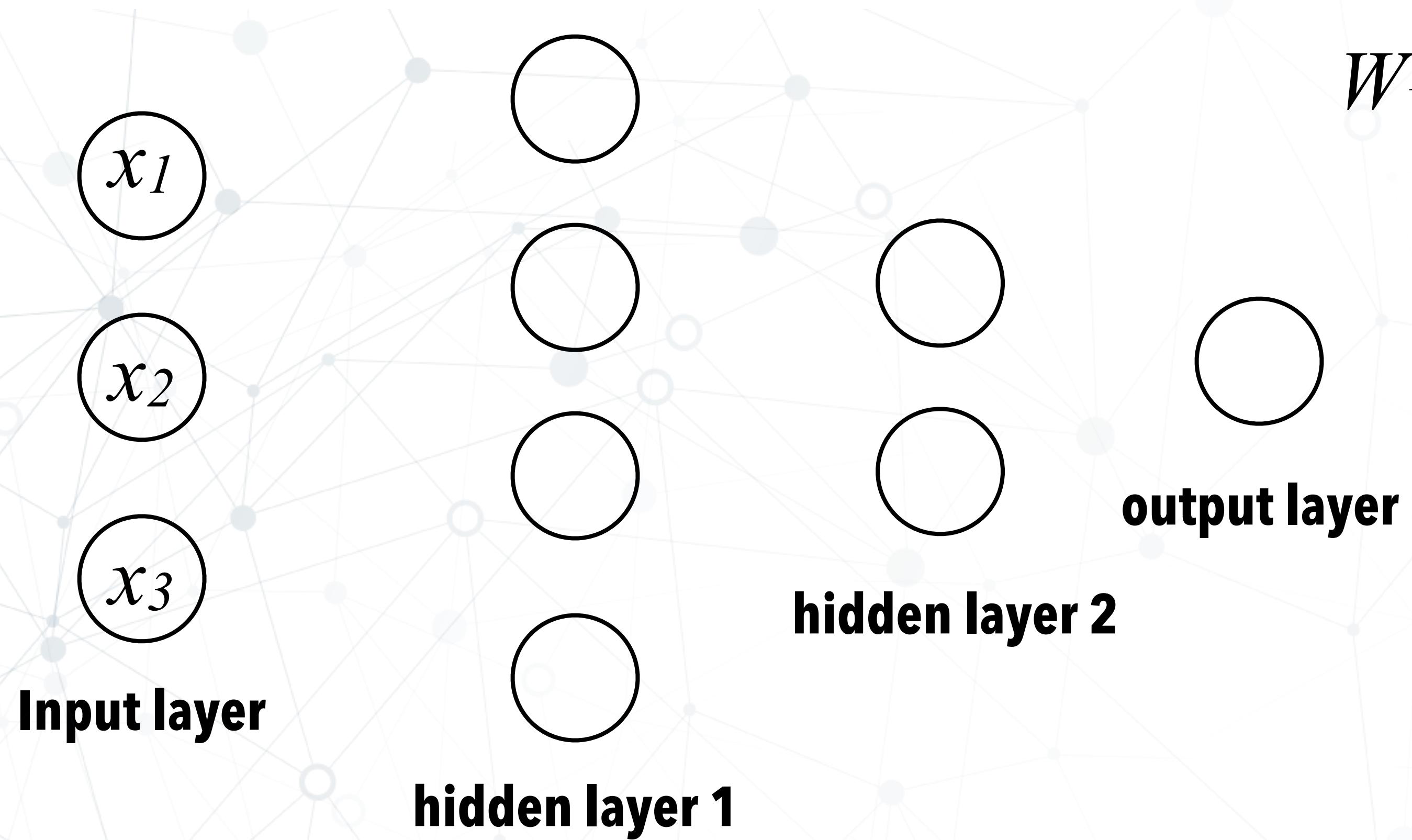
Multi-layer feedforward NN



$$W^1 = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

$$b^1 = \begin{bmatrix} & \\ & \\ & \end{bmatrix}$$

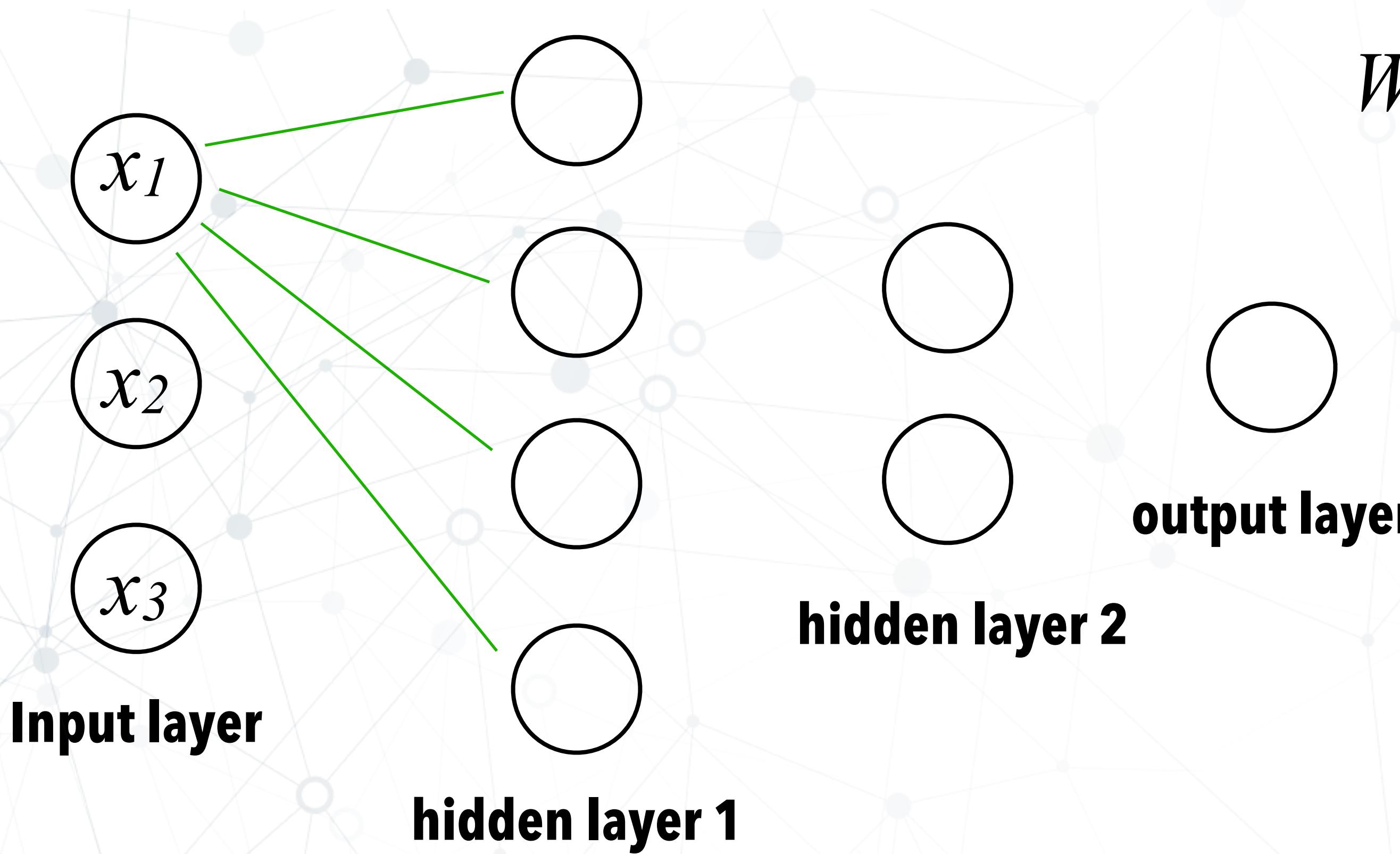
Multi-layer feedforward NN



$$W^1 = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$
$$b^1 = \begin{bmatrix} & \\ & \end{bmatrix}$$

- Each line represents a weight w

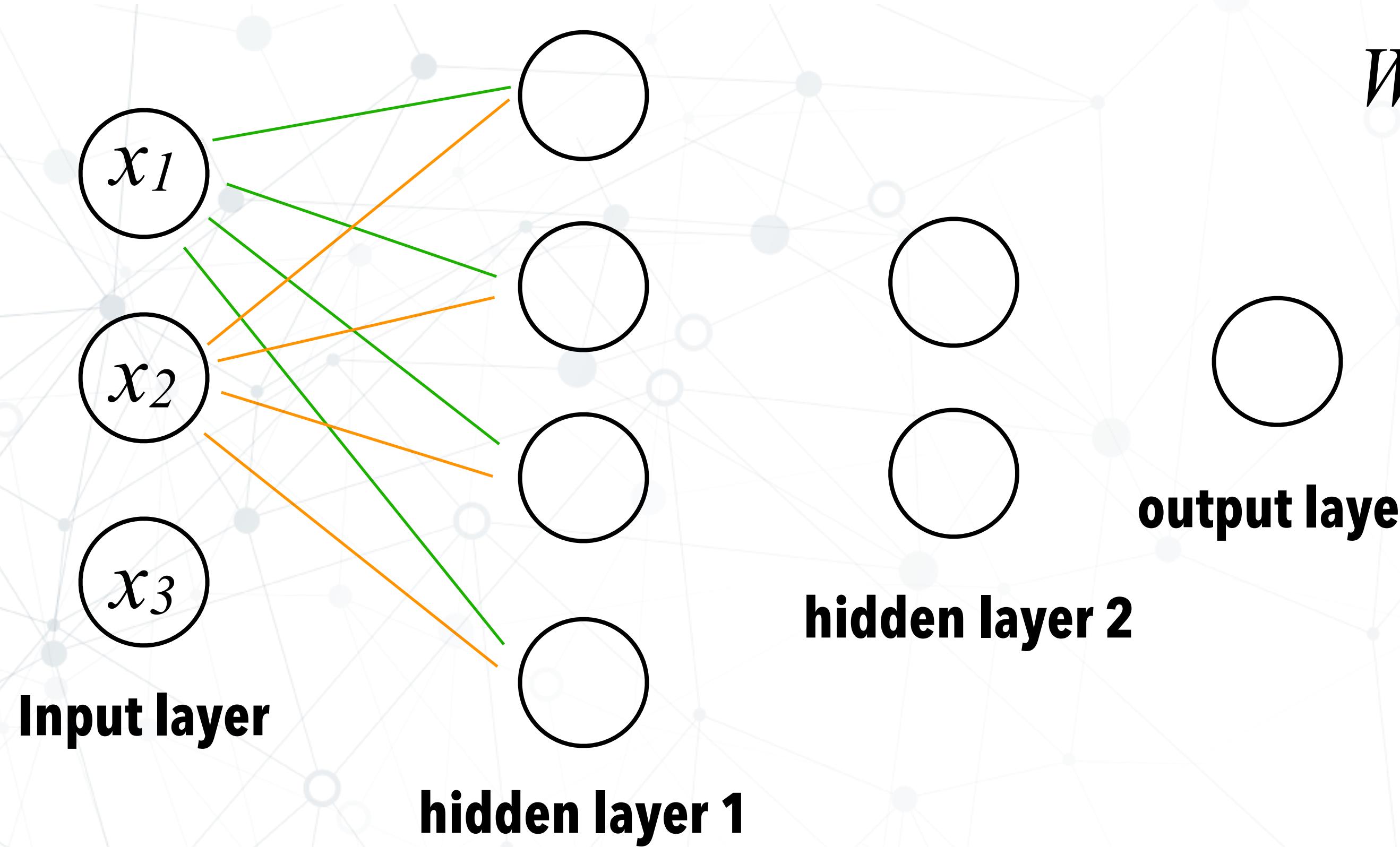
Multi-layer feedforward NN



- Each line represents a weight w

$$W^1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \end{bmatrix}$$
$$b^1 = \begin{bmatrix} \end{bmatrix}$$

Multi-layer feedforward NN



$W^1 =$

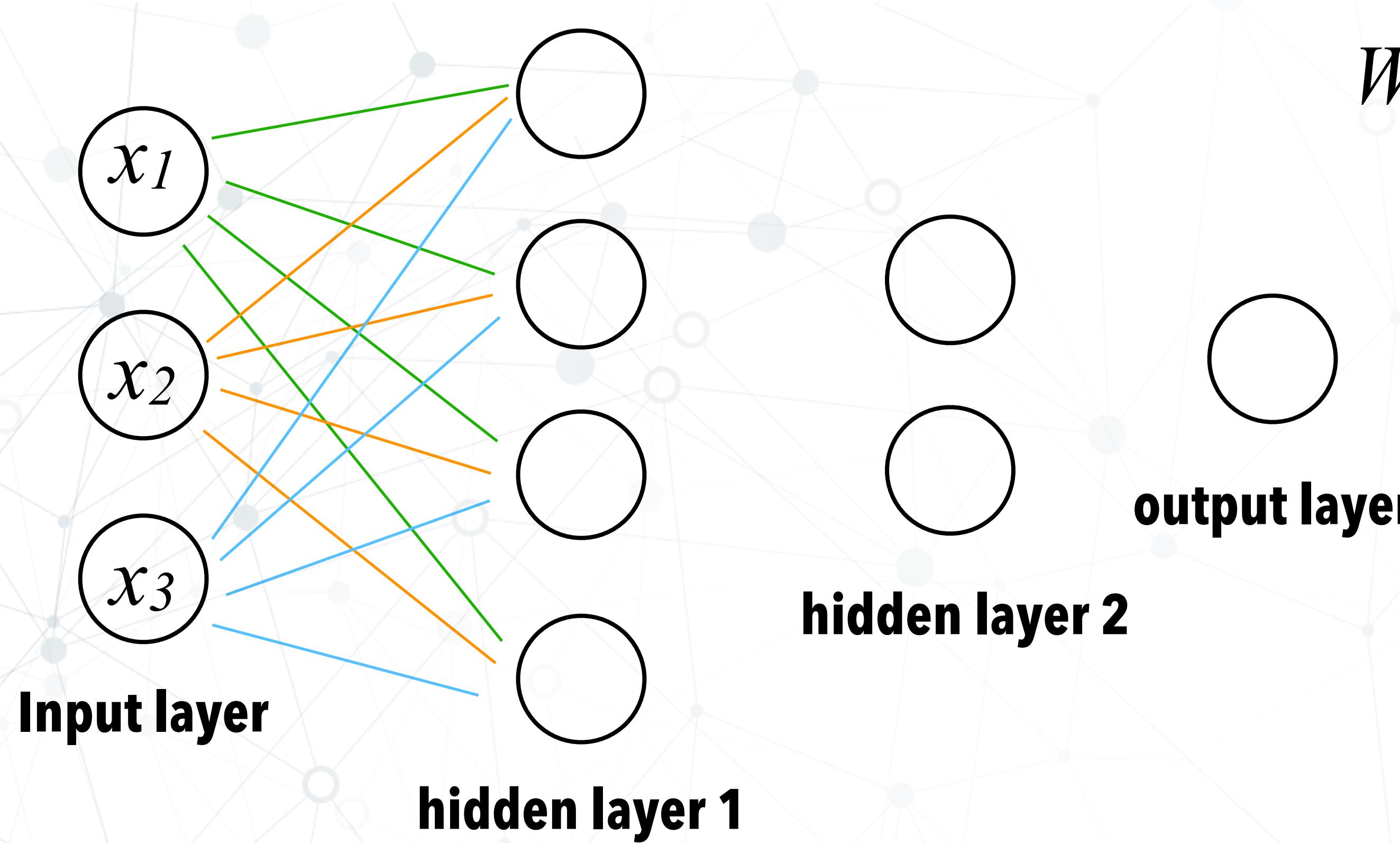
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}$$

$b^1 =$

$$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

- Each line represents a weight w

Multi-layer feedforward NN



$W1 =$

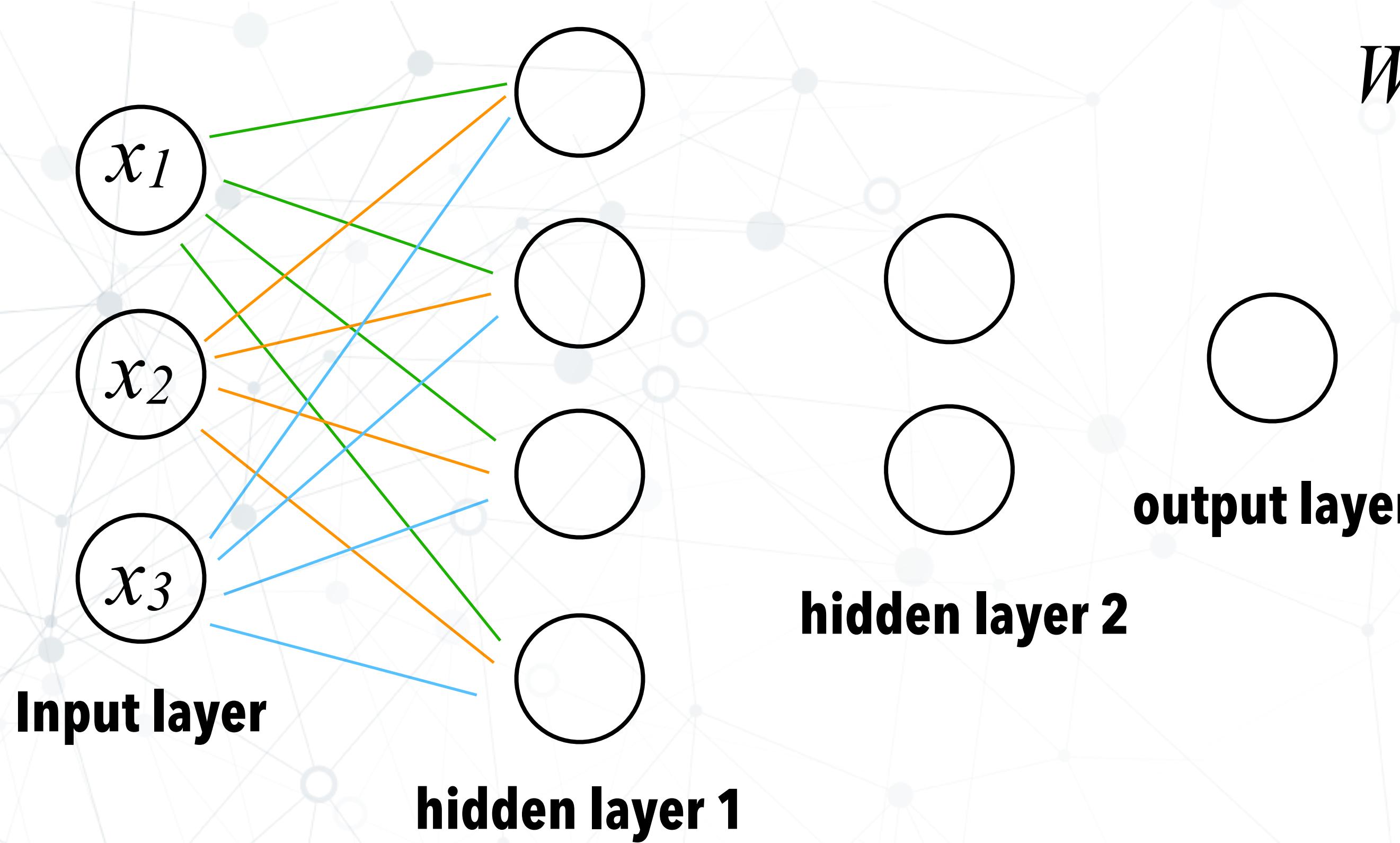
$$W1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

$b1 =$

$$b1 = \begin{bmatrix} \quad \end{bmatrix}$$

- Each line represents a weight w

Multi-layer feedforward NN



$W1 =$

$$W1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

$b^1 =$

$$b^1 = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

output layer

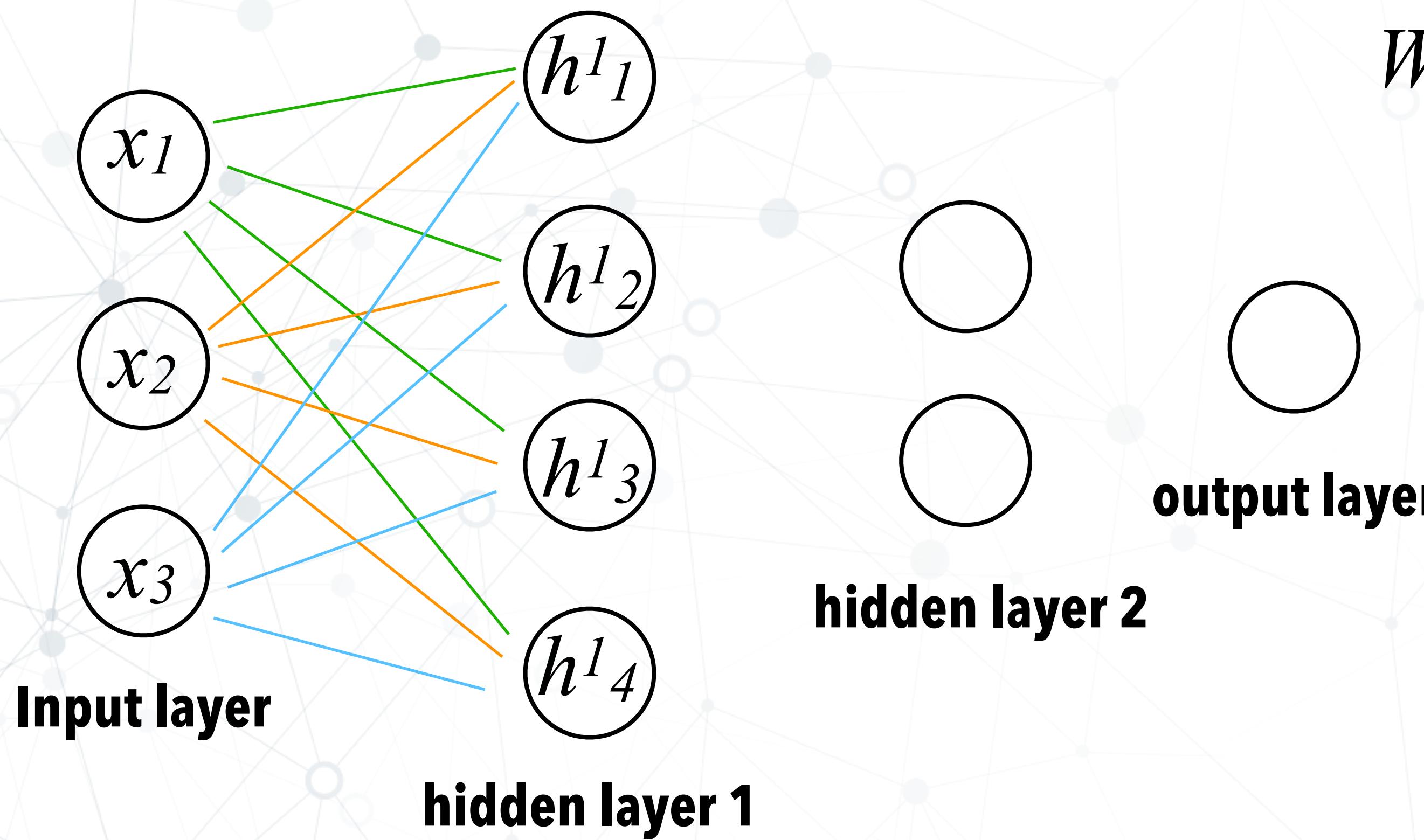
hidden layer 2

Input layer

hidden layer 1

- Each line represents a weight w

Multi-layer feedforward NN

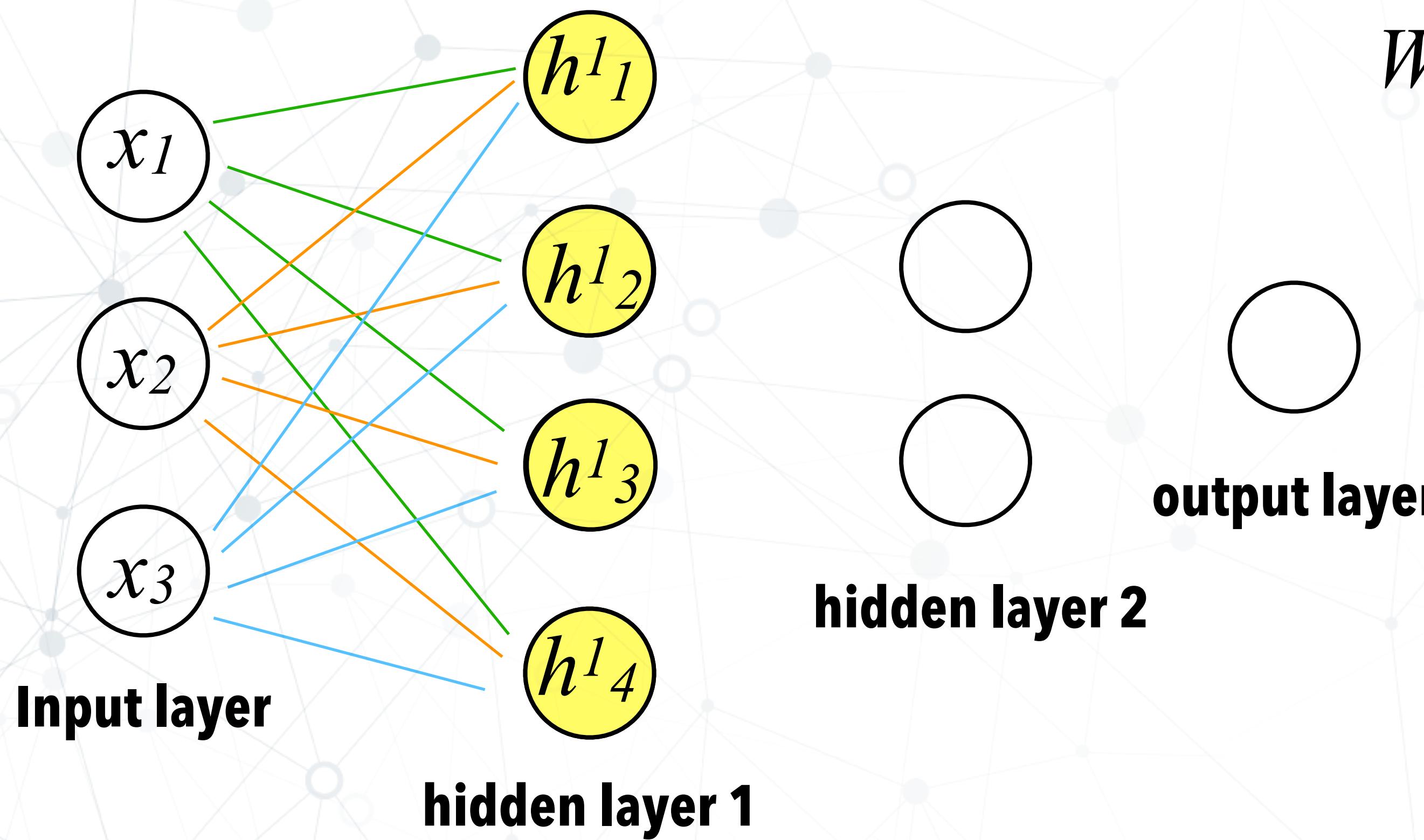


- Each line represents a weight w
- In each neuron a linear combination of the inputs is computed

$$W^1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$
$$b^1 = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

Multi-layer feedforward NN



$W1 =$

$$W1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

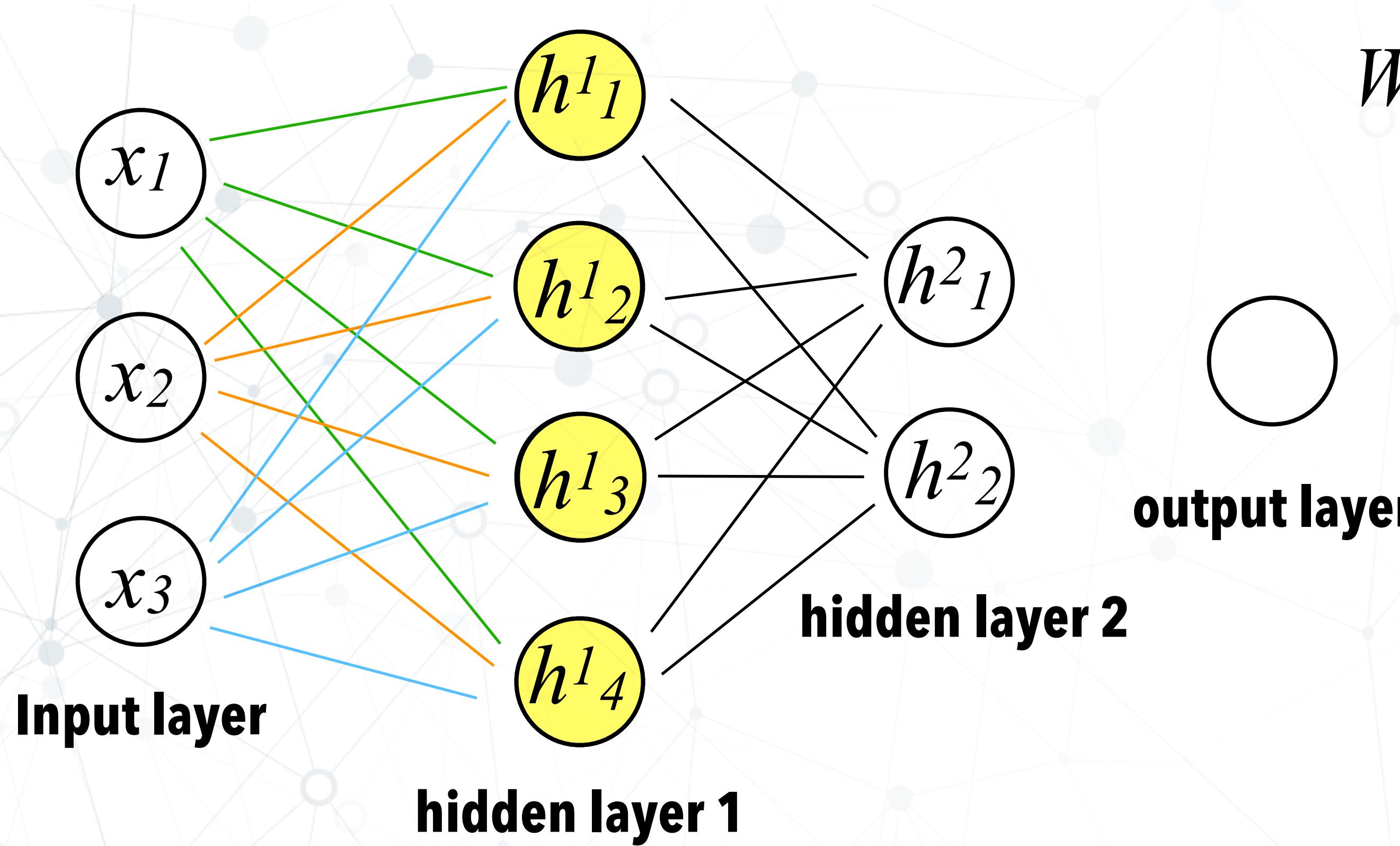
$$b^1 = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

- Each line represents a weight w
- In each neuron a linear combination of the inputs is computed
- The result is then activated with a non-linearity and become one of the inputs of the following layer

Multi-layer feedforward NN



$W^1 =$

$$W^1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

$b^1 =$

$$b^1 = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

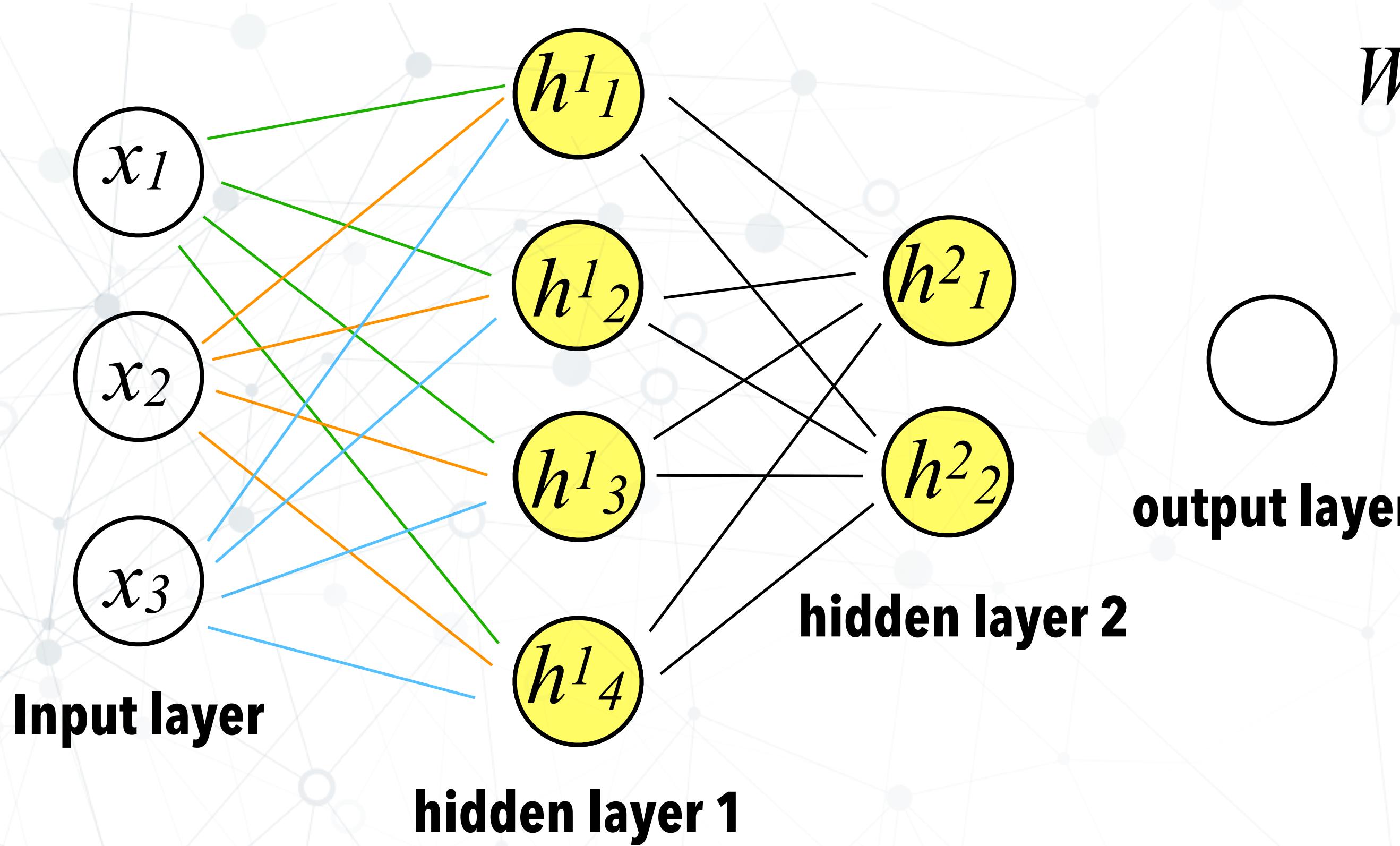
$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

- Each line represents a weight w
- In each neuron a linear combination of the inputs is computed
- The result is then activated with a non-linearity and become one of the inputs of the following layer

Multi-layer feedforward NN



- Each line represents a weight w
- In each neuron a linear combination of the inputs is computed
- The result is then activated with a non-linearity and become one of the inputs of the following layer

$$W^1 =$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

$$b^1 =$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

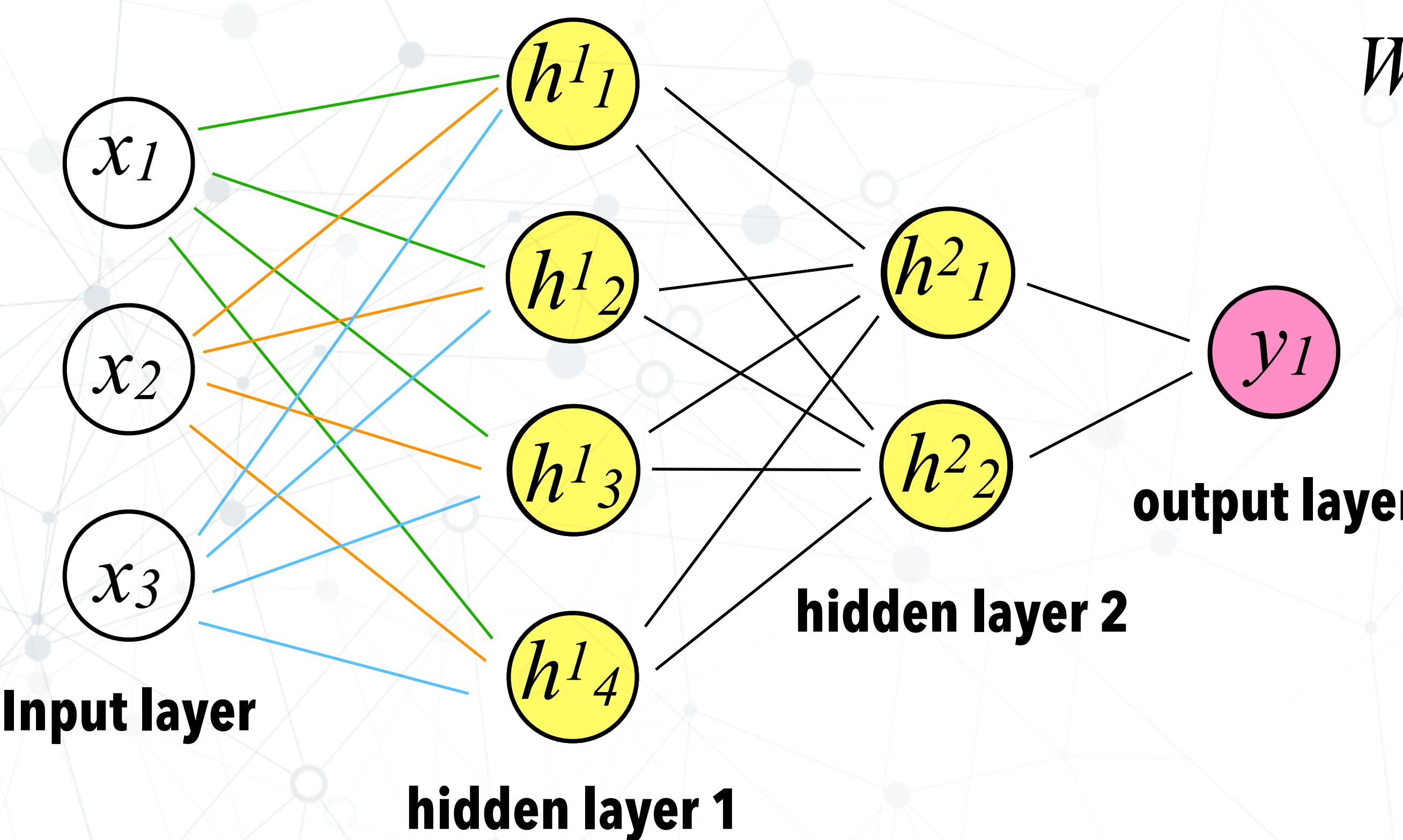
$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

$$h^{[2]} = \sigma(\tilde{h}^{[2]})$$

Multi-layer feedforward NN



- Each line represents a weight w
- In each neuron a linear combination of the inputs is computed
- The result is then activated with a non-linearity and become one of the inputs of the following layer

$$W^1 =$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

$$b^1 =$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

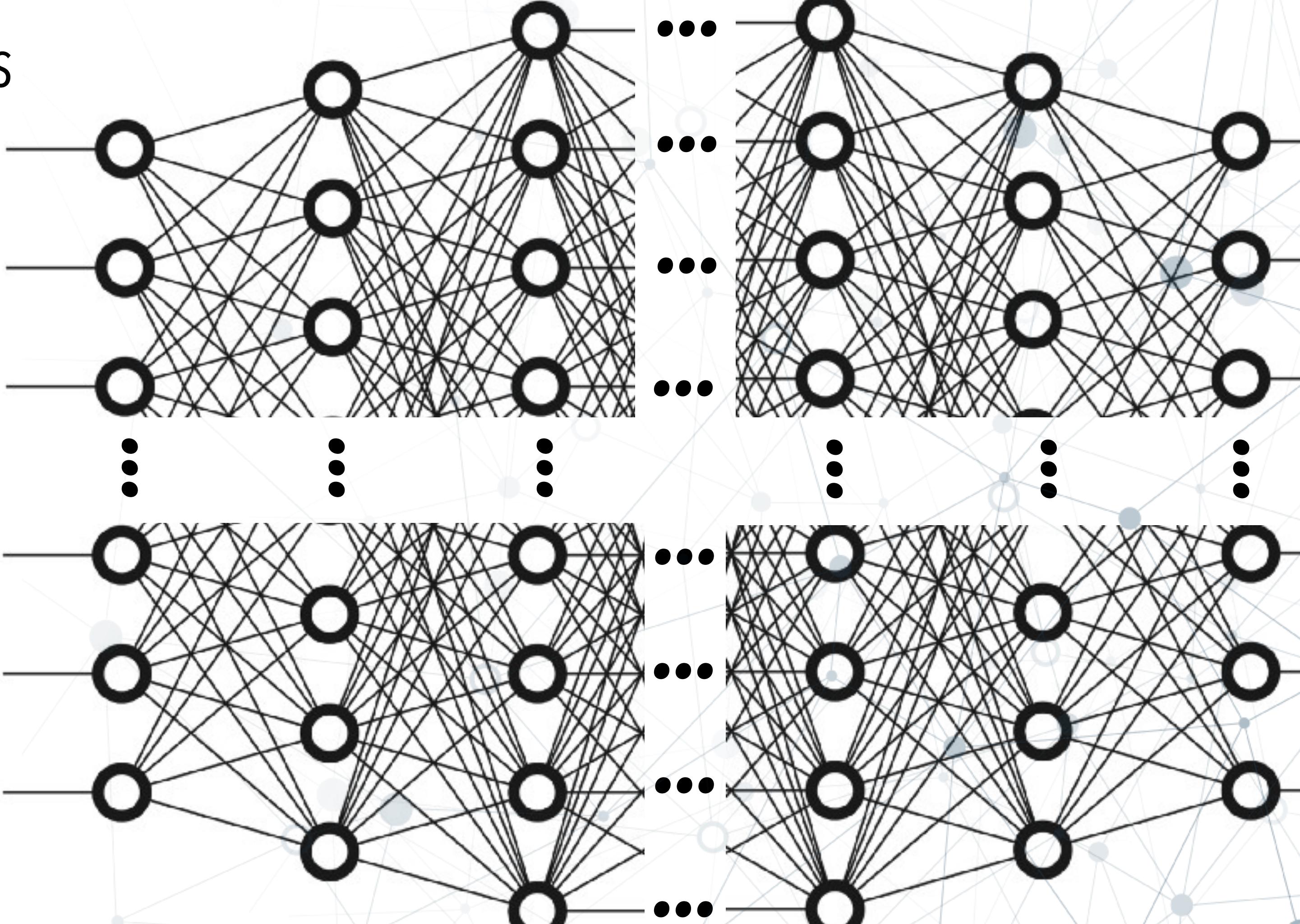
$$h^{[2]} = \sigma(\tilde{h}^{[2]})$$

$$\tilde{y} = W^{[3]T} \cdot h^{[2]} + b^{[3]}$$

$$y = \sigma^*(\tilde{y})$$

Multi-layer feedforward Neural Network

- This type of NN are called **dense** or **fully connected**
- very deep NNs can have thousands of layers
- and $O(10^7)$ parameters
- The number of layers and neurons in each layer define the architecture of the NN are represent **hyperparameters**



Loss function

- Through the feed-forward propagation the NN produces the output (values of the neurons in the output layer)
- During training the **output is compared with the ground truth**
- This is done by computing a **loss function**

$$\mathcal{L}(y^i, \hat{y}^i)$$

NN output Ground truth

Cost function

- The **cost function** is the average of the loss over the training set

$$\mathcal{J} = \frac{1}{N} \sum_{i=0}^N \mathcal{L}(y^i, \hat{y}^i)$$

- it is a function of all the NN's parameters (weights and biases):

$$\mathcal{J} = \mathcal{J}(w, b)$$

The goal of training is to find the parameters

w and b that minimize \mathcal{J}

Gradient descent

see also previous lectures

- usually the minimum of the cost function is found with a **gradient descent algorithm**
- at each iteration the parameters are updated as:

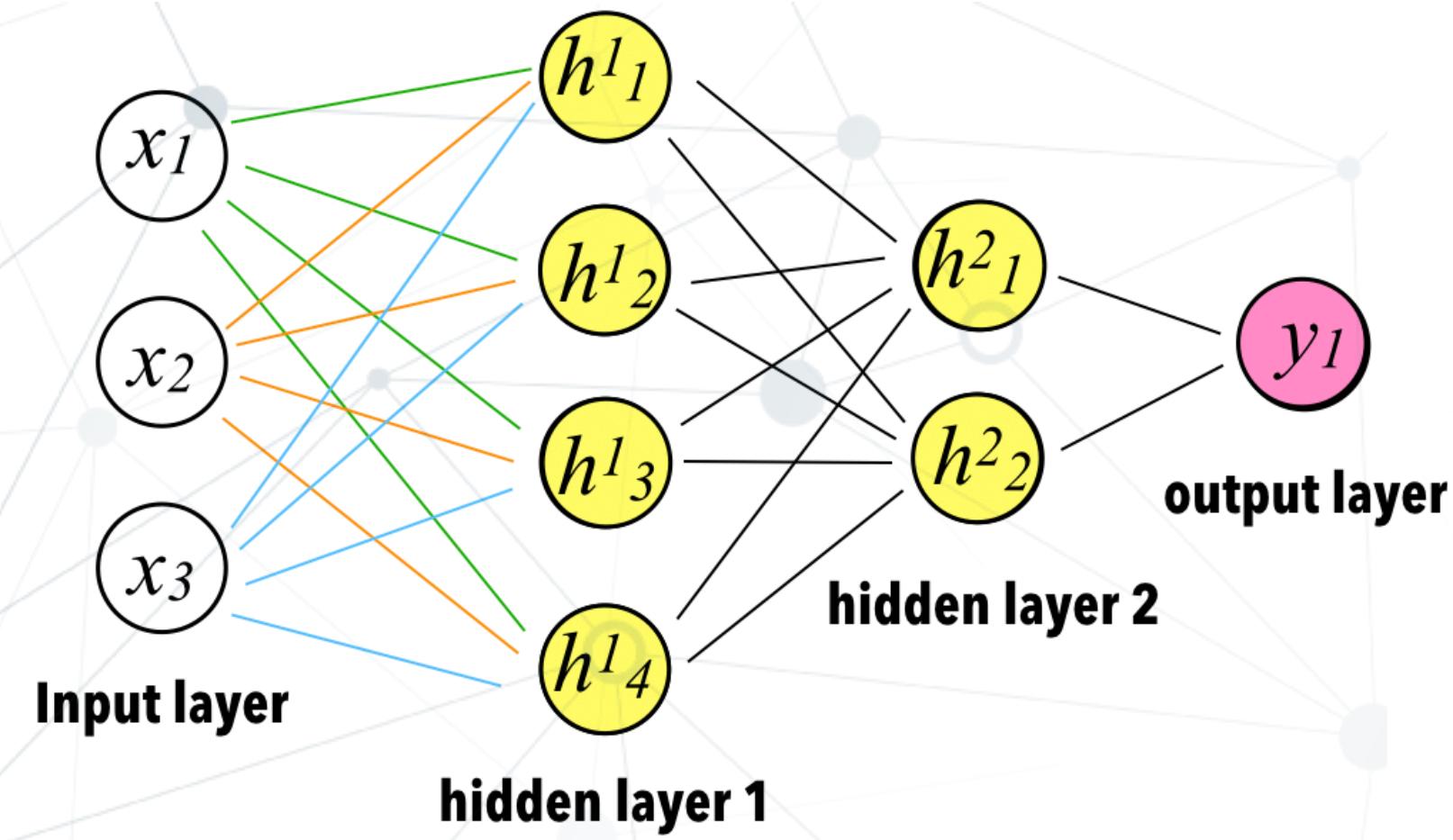
$$w := w - \alpha \frac{\partial \mathcal{J}}{\partial w}$$

$$b := b - \alpha \frac{\partial \mathcal{J}}{\partial b}$$

- α is the **learning rate** and is an **hyperparameter** of the NN

(hyperparameter are set before training the network and are not optimized during training)

Backpropagation



$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

$$h^{[2]} = \sigma(\tilde{h}^{[2]})$$

$$\tilde{y} = W^{[3]T} \cdot h^{[2]} + b^{[3]}$$

$$y = \sigma^*(\tilde{y})$$

$$\mathcal{L} = \mathcal{L}(y)$$

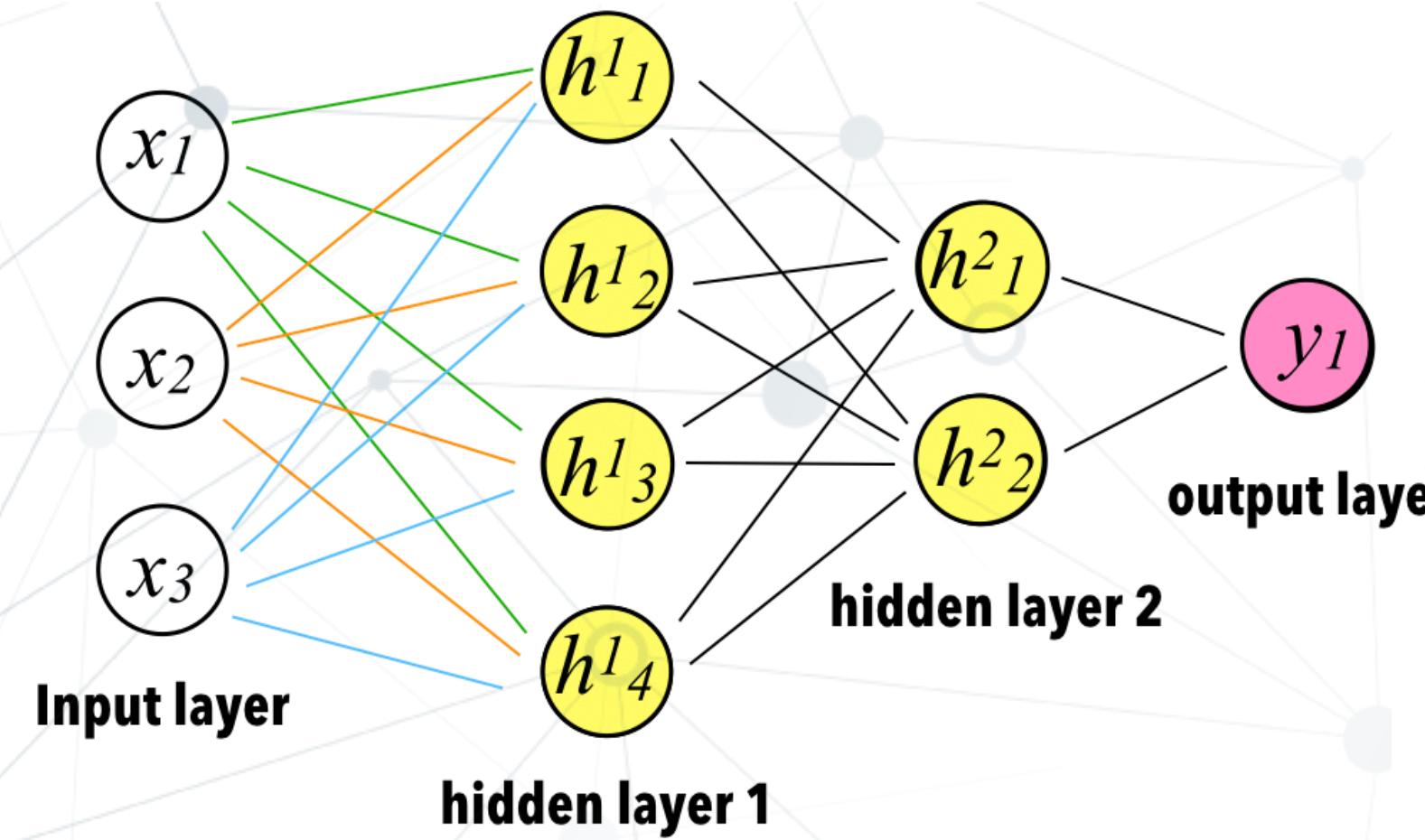
$$\mathcal{J} = \frac{1}{N} \sum \mathcal{L}$$

Derivatives are computed with the **chain rule**
see also previous lectures

$$\frac{\partial \mathcal{J}}{\partial W^{[3]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial W^{[3]}}$$

$$\frac{\partial \mathcal{J}}{\partial b^{[3]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial b^{[3]}}$$

Backpropagation



$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

$$h^{[2]} = \sigma(\tilde{h}^{[2]})$$

$$\tilde{y} = W^{[3]T} \cdot h^{[2]} + b^{[3]}$$

$$y = \sigma^*(\tilde{y})$$

$$\mathcal{L} = \mathcal{L}(y)$$

$$\mathcal{J} = \frac{1}{N} \sum \mathcal{L}$$

Derivatives are computed with the **chain rule**
see also previous lectures

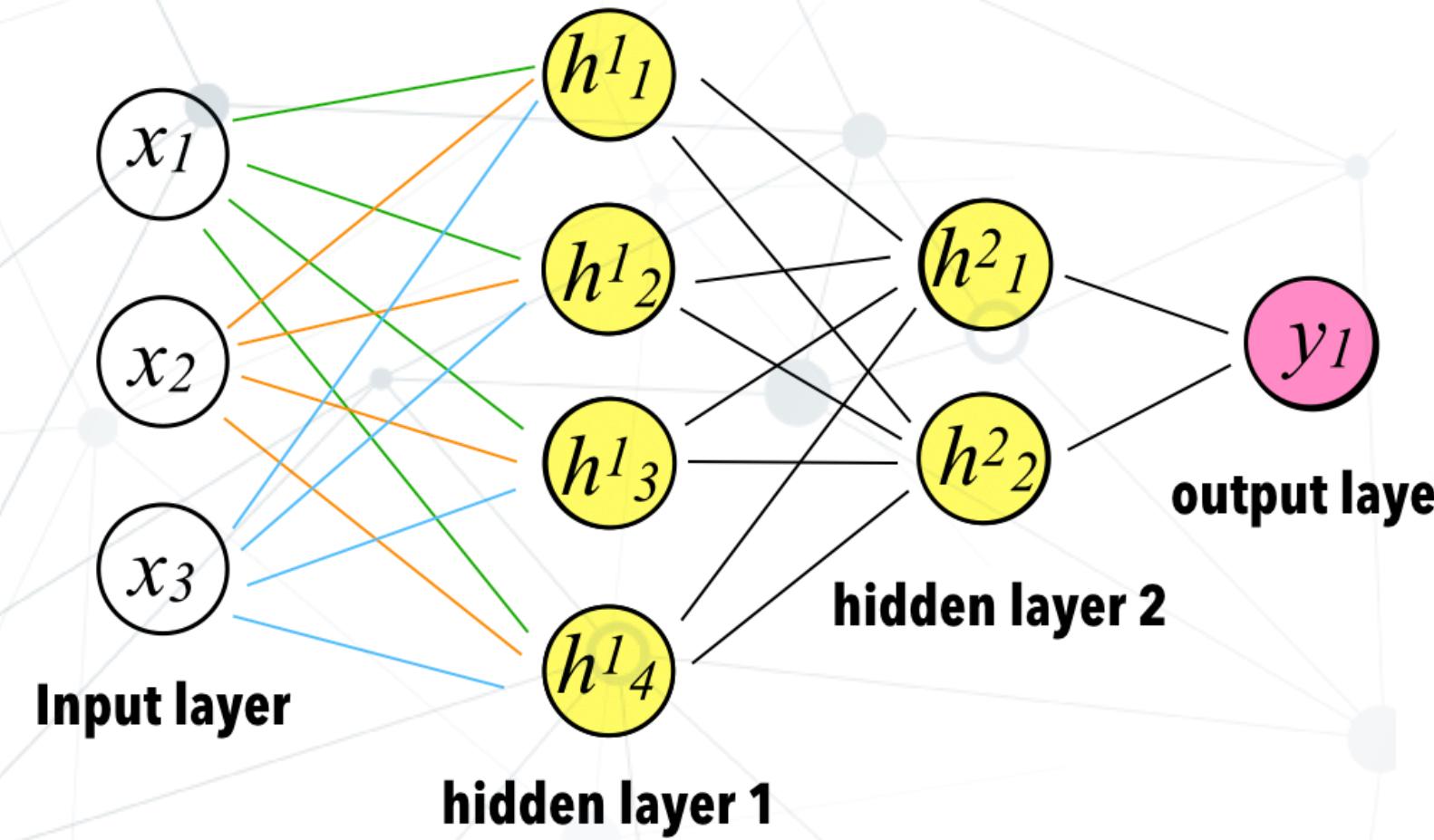
$$\frac{\partial \mathcal{J}}{\partial W^{[3]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial W^{[3]}}$$

$$\frac{\partial \mathcal{J}}{\partial W^{[2]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial \sigma}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial W^{[2]}}$$

$$\frac{\partial \mathcal{J}}{\partial b^{[3]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial b^{[3]}}$$

$$\frac{\partial \mathcal{J}}{\partial b^{[2]}} = \boxed{\frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial \sigma}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial b^{[2]}}$$

Backpropagation



$$\tilde{h}^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

$$h^{[1]} = \sigma(\tilde{h}^{[1]})$$

$$\tilde{h}^{[2]} = W^{[2]T} \cdot h^{[1]} + b^{[2]}$$

$$h^{[2]} = \sigma(\tilde{h}^{[2]})$$

$$\tilde{y} = W^{[3]T} \cdot h^{[2]} + b^{[3]}$$

$$y = \sigma^*(\tilde{y})$$

$$\mathcal{L} = \mathcal{L}(y)$$

$$\mathcal{J} = \frac{1}{N} \sum \mathcal{L}$$

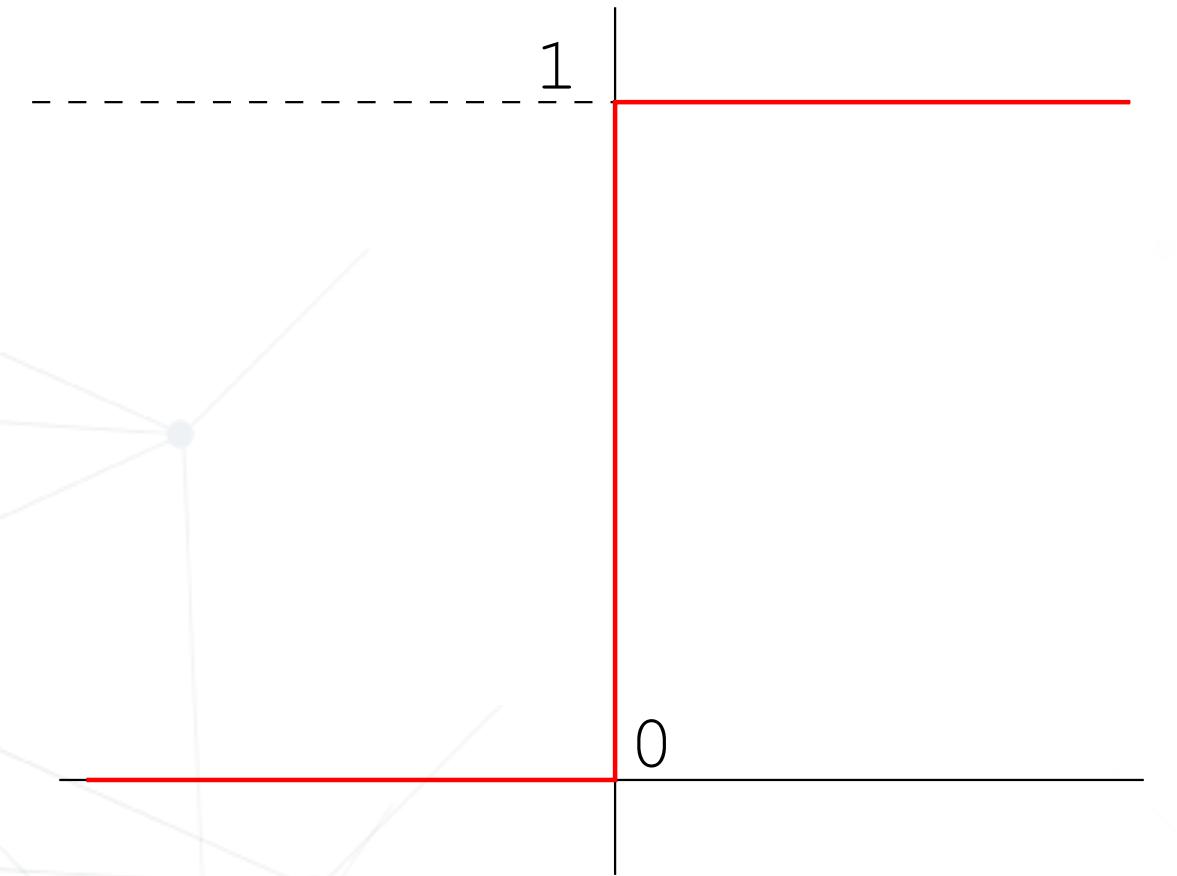
Derivatives are computed with the **chain rule**
see also previous lectures

$$\frac{\partial \mathcal{J}}{\partial W^{[3]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial W^{[3]}}$$
$$\frac{\partial \mathcal{J}}{\partial W^{[2]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial h^{[2]}}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial W^{[2]}}$$
$$\frac{\partial \mathcal{J}}{\partial W^{[1]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial h^{[2]}}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial h^{[1]}} \frac{\partial h^{[1]}}{\partial \tilde{h}^{[1]}} \frac{\partial \tilde{h}^{[1]}}{\partial W^{[1]}}$$

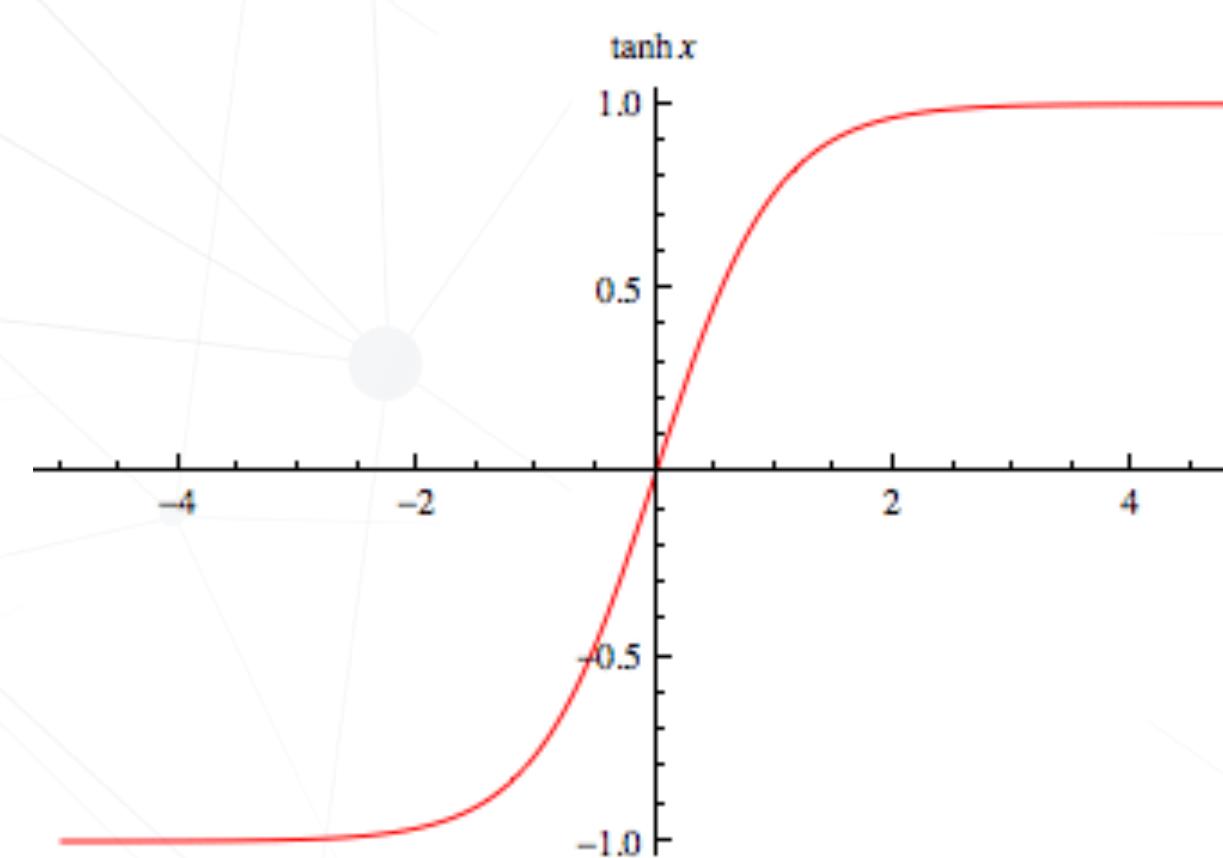
$$\frac{\partial \mathcal{J}}{\partial b^{[3]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial b^{[3]}}$$
$$\frac{\partial \mathcal{J}}{\partial b^{[2]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial h^{[2]}}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial b^{[2]}}$$
$$\frac{\partial \mathcal{J}}{\partial b^{[1]}} = \frac{\partial \mathcal{J}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} \frac{\partial \sigma^*}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial h^{[2]}} \frac{\partial h^{[2]}}{\partial \tilde{h}^{[2]}} \frac{\partial \tilde{h}^{[2]}}{\partial h^{[1]}} \frac{\partial h^{[1]}}{\partial \tilde{h}^{[1]}} \frac{\partial \tilde{h}^{[1]}}{\partial b^{[1]}}$$

Activation functions

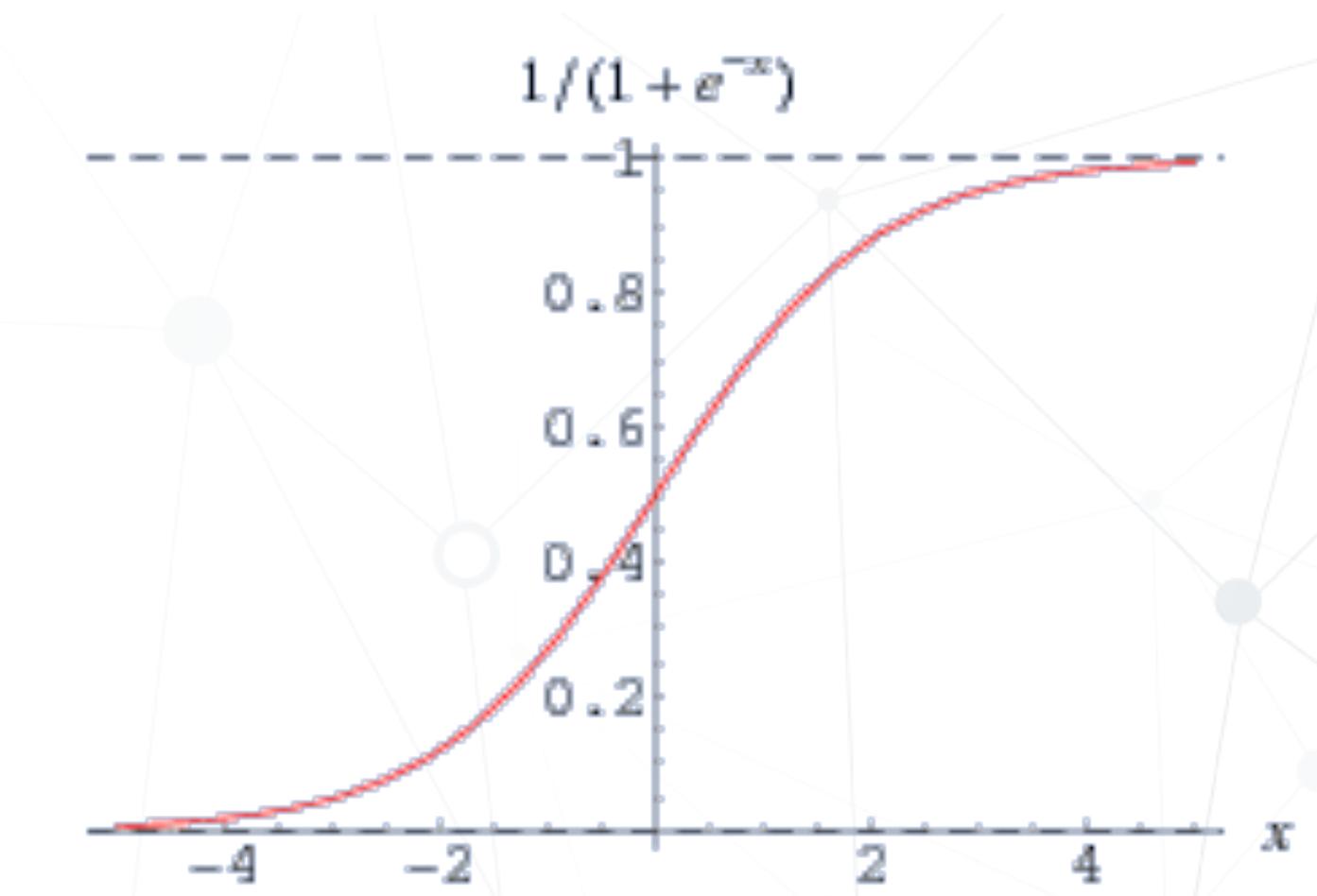
The **ReLU** function is currently the most used, preventing the problem of vanishing gradient during backpropagation



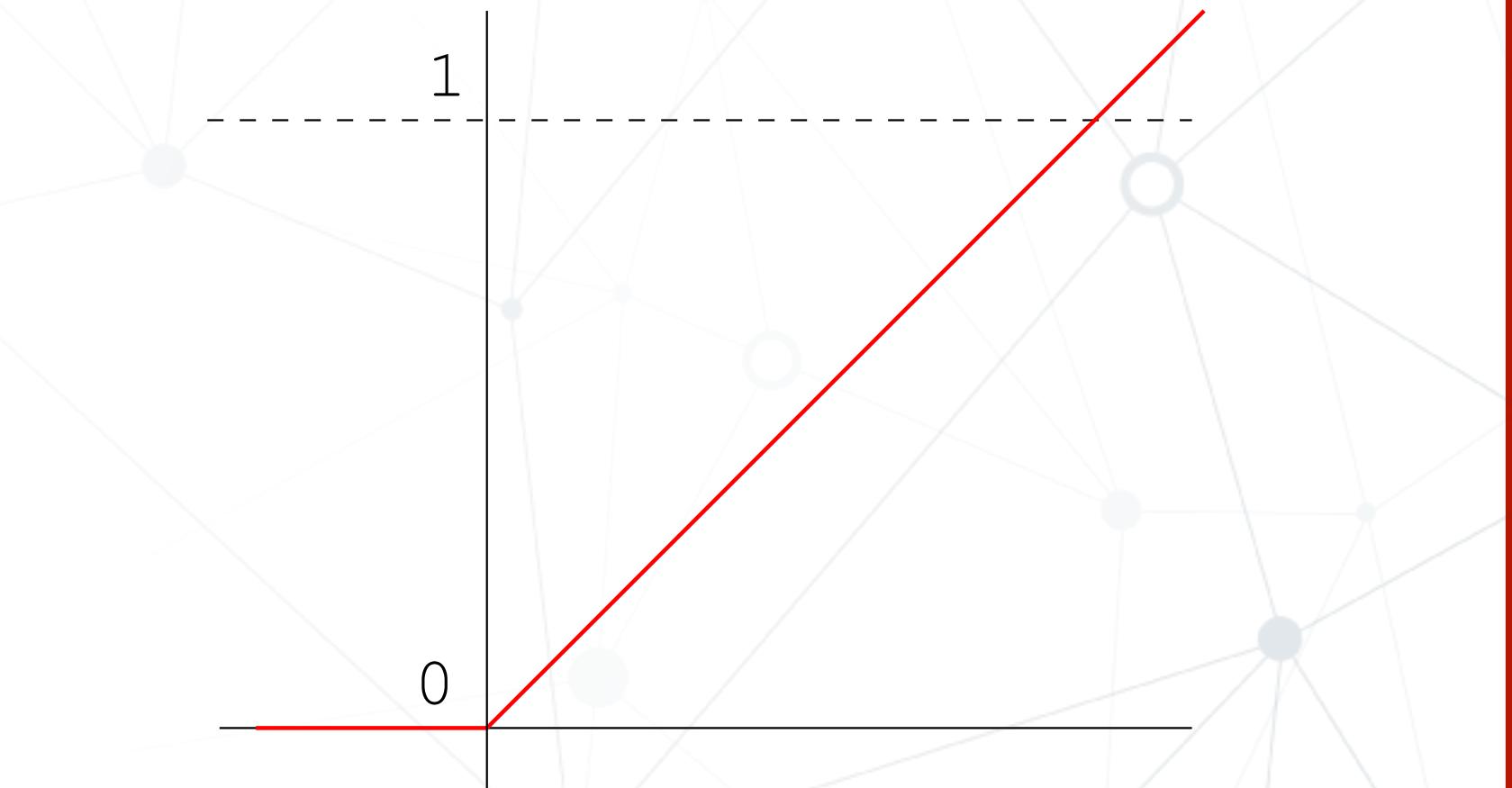
threshold: if $x > 0$ then 1 else 0



hyperbolic tangent: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$



logistic function: $\frac{1}{1+e^{-x}}$



rectified linear (ReLU): if $x > 0$ then x else 0

see also previous lectures

Tutorial 1

Coding a simple NN from scratch

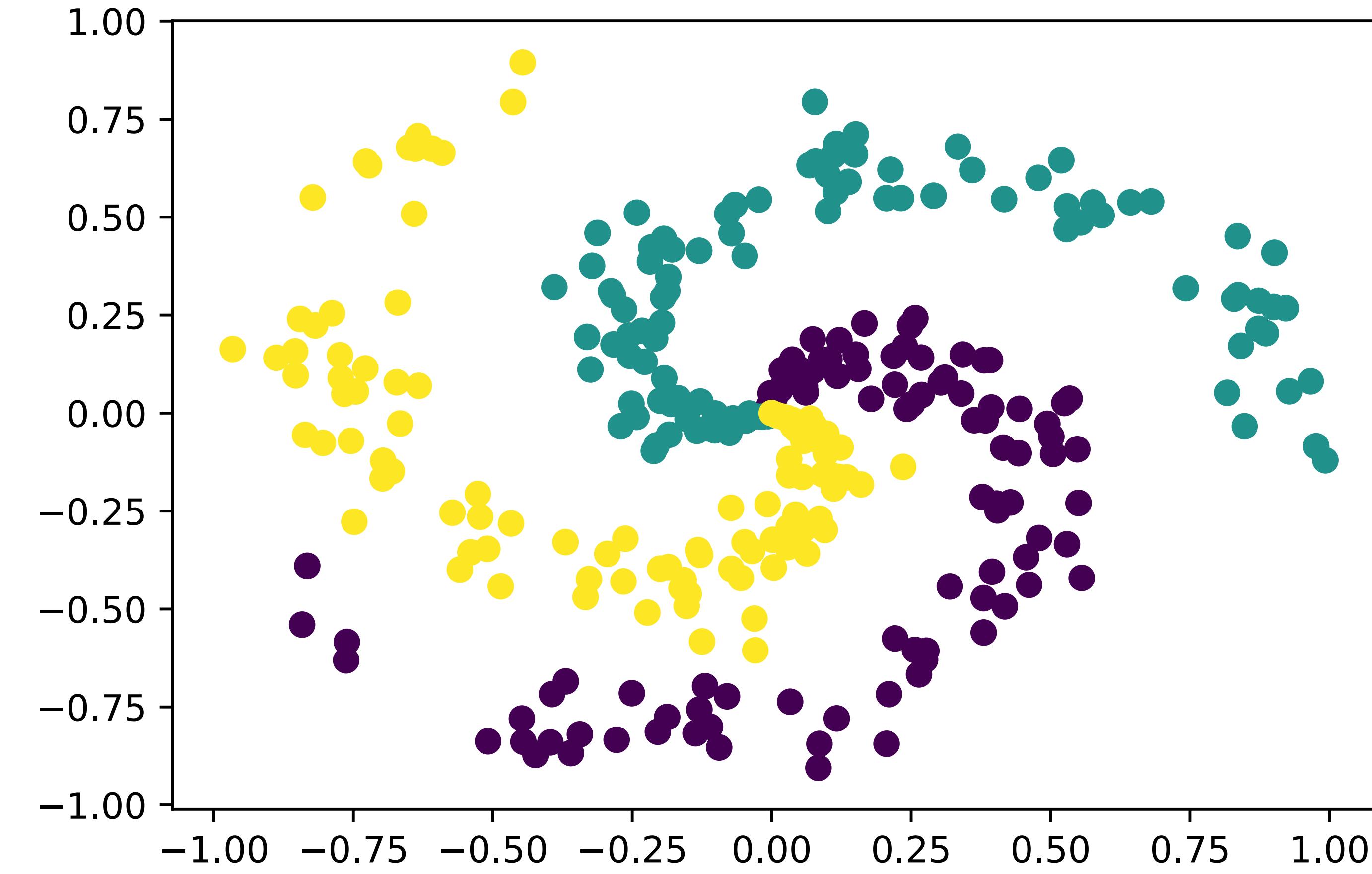
taken from: <http://cs231n.stanford.edu>

Coding a simple NN from scratch

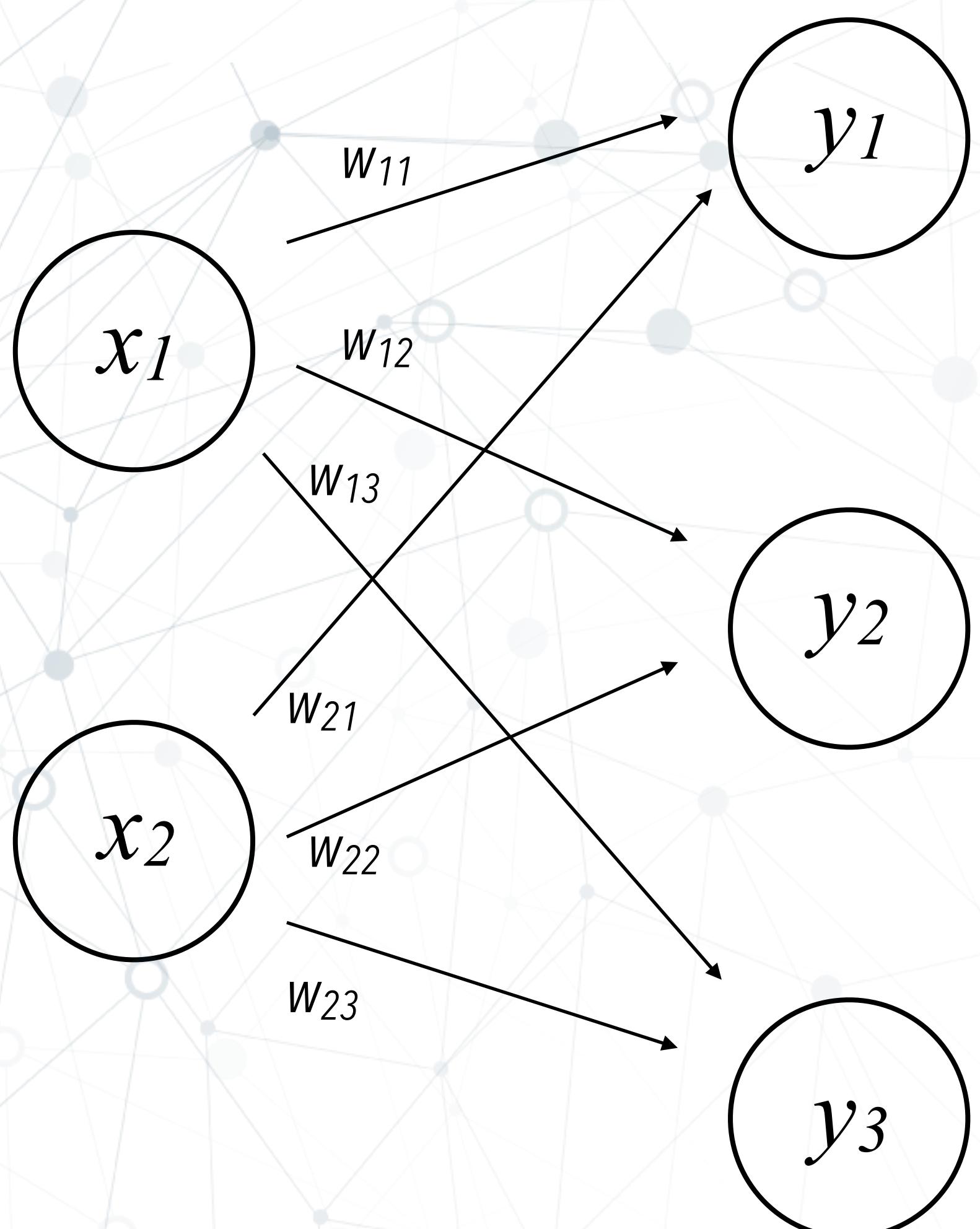
Toy model: assign each particle to the correct class (arm)

- **step one:** linear classifier

- **step two:** fully connected NN



Coding a simple NN from scratch

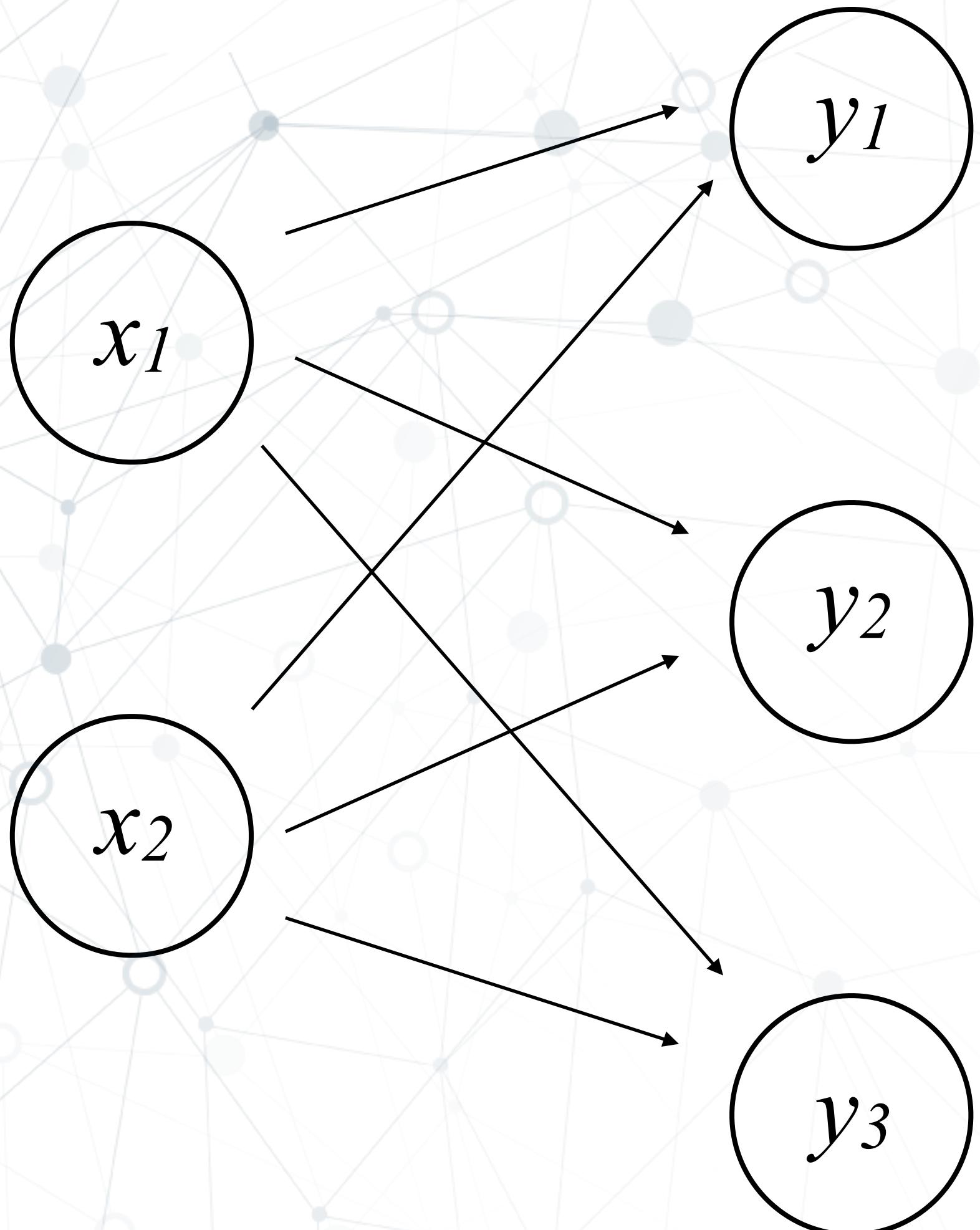


- The input vector x has 2 elements, representing the coordinates of each point
- The NN has just input and output layers, without any non-linear activation function

$$y = W^T \cdot x + b$$

$$W: (2, 3)$$

Coding a simple NN from scratch



In [1]: `D = 2 # dimensionality
K = 3 # number of classes (arms)`

Initialize parameters:

In [4]: `w = np.random.randn(D,K)
b = np.zeros((1,K))`

Compute output:

In [8]: `print('Input shape: ', X.shape)
num_examples = X.shape[0]
y = np.dot(X, w) + b
print('Output shape: ', y.shape)`

Input shape: (300, 2)
Output shape: (300, 3)

Coding a simple NN from scratch

- The **loss function** is the negative log of the normalized exponential output of the true class (y_t) that can be interpret as a probability

$$\mathcal{L}_i = -\log\left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}}\right)_i = -\log[p_i(y_t)]$$

- The **cost function** is the sum over the number of elements in training set

$$\mathcal{J} = \frac{1}{N} \sum_{i=0}^N \mathcal{L}_i$$

Compute loss and cost:

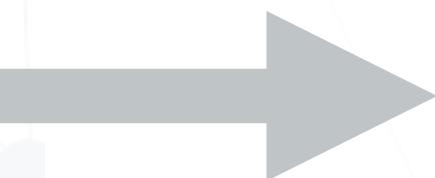
```
In [9]: exp_y = np.exp(y)
probs = exp_y / np.sum(exp_y, axis=1, keepdims=True)

loss = -np.log(probs[range(num_examples), y_true])
cost = np.sum(loss)/num_examples
```

Coding a simple NN from scratch

- to implement the **backpropagation** we need to compute the derivatives of the loss function w.r.t. the parameters W and b
- we use the chain rule and first we compute

$$\mathcal{L}_i = -\log \left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \right)_i = -\log [p_i(y_t)]$$



$$\frac{\partial \mathcal{L}_i}{\partial y_j} = -\frac{1}{p_i(y_t)} \frac{\partial}{\partial y_j} \left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \right)_i$$

when $j = t$

$$\frac{\partial}{\partial y_j} \left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \right) = \frac{e^{y_t} \sum_{k=1}^3 e^{y_k} - e^{y_t} e^{y_j}}{\left[\sum_{k=1}^3 e^{y_k} \right]^2} = \frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \left[1 - \frac{e^{y_j}}{\sum_{k=1}^3 e^{y_k}} \right] = p(y_t)[1 - p(y_j)]$$

$$\left(\frac{g}{h} \right)' = \frac{g'h - gh'}{h^2}$$

when $j \neq t$

$$\frac{\partial}{\partial y_j} \left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \right) = -\frac{e^{y_t} e^{y_j}}{\sum_{k=1}^3 e^{y_k}} = -p(y_t)p(y_j)$$

Coding a simple NN from scratch

therefore:
$$\frac{\partial \mathcal{L}}{\partial y_j} = -\frac{1}{p(y_t)} \frac{\partial}{\partial y_j} \left(\frac{e^{y_t}}{\sum_{k=1}^3 e^{y_k}} \right)_i = p(y_j) - 1_{j=t}$$

```
In [13]: # derivative w.r.t y  
dy = probs  
dy[range(num_examples),y_true] -= 1  
dy /= num_examples
```

we continue the backpropagation and compute the derivatives w.r.t. the parameters

remind that $y = x \cdot W + b$ and therefore $\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial W}$ and similarly for b

```
In [12]: dW = np.dot(X.T, dy)  
db = np.sum(dy, axis=0, keepdims=True)
```

Coding a simple NN from scratch

- We have now everything needed to update the value of the parameters
- and we can train the linear classifier by iterate in order to find the minimum of the cost function

In [87]:

```
step_size = 1e-0
W += -step_size * dw
b += -step_size * db
```

[go to notebook](#)

Coding a simple NN from scratch

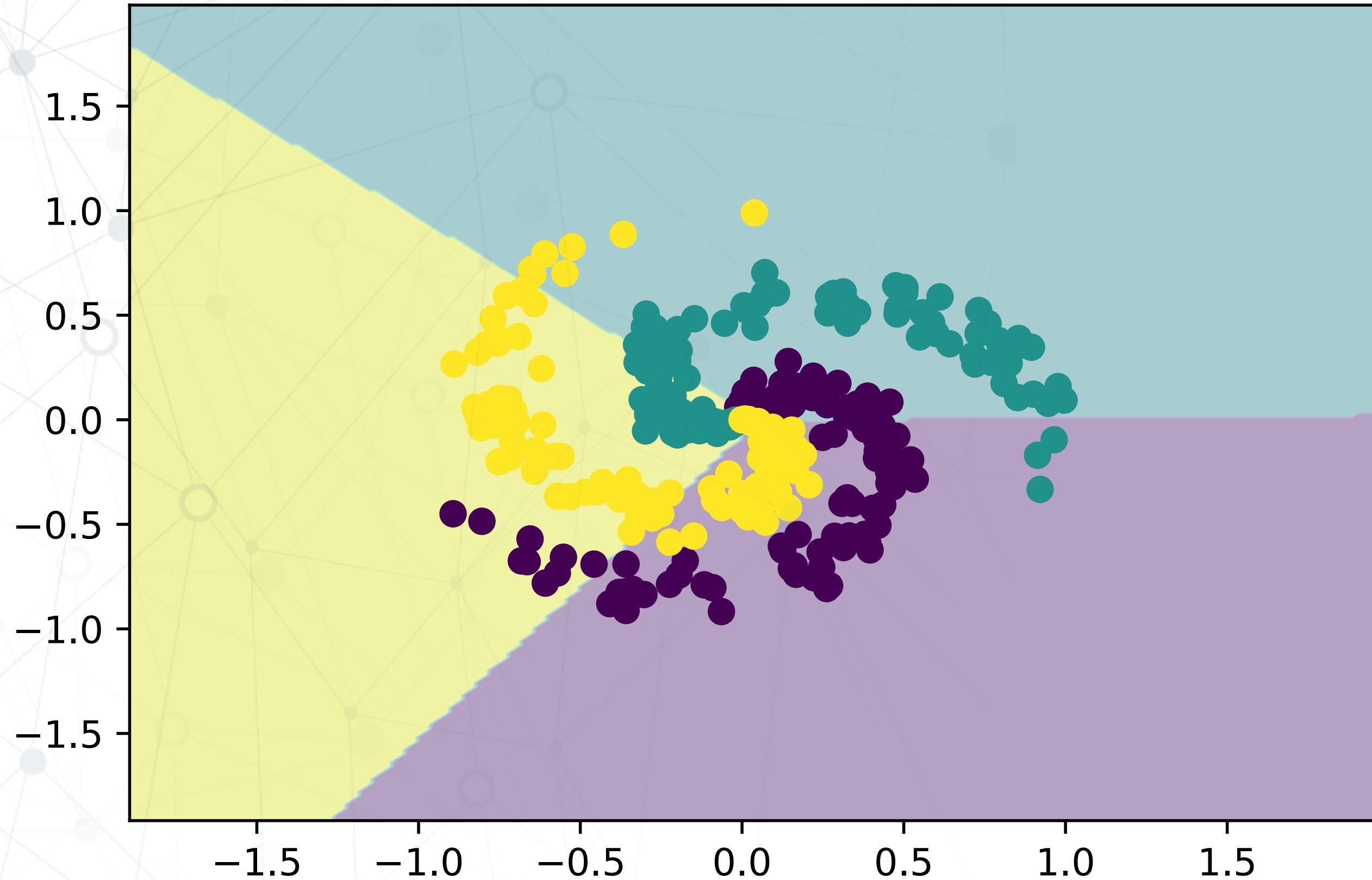
- We have now everything needed to update the value of the parameters

In [87]:

```
step_size = 1e-0
W += -step_size * dw
b += -step_size * db
```

- and we can train the linear classifier by iterate in order to find the minimum of the cost function

[go to notebook](#)



- Not surprisingly, a linear classifier doesn't perform well as our dataset is not linearly separable

Coding a simple NN from scratch

- we add to the NN an hidden layer with 100 neurons activated with a ReLU non-linear function

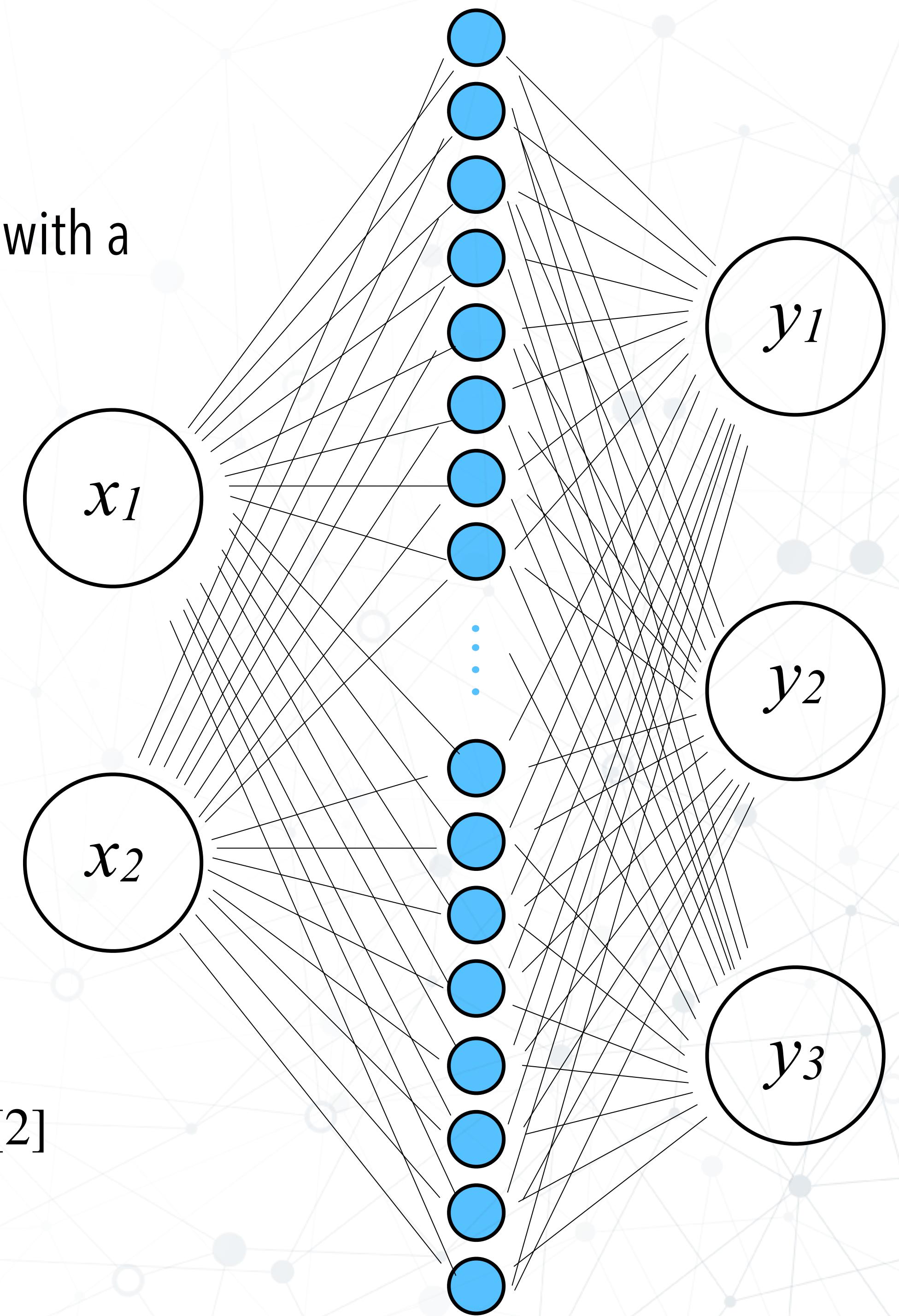
$$W^{[1]} : (2, 100)$$

$$W^{[2]} : (100, 3)$$

$$\tilde{h}^{[1]} = x \cdot W^{[1]} + b^{[1]}$$

$$h = \text{ReLU}(\tilde{h}^{[1]})$$

$$y = h \cdot W^{[2]} + b^{[2]}$$



Coding a simple NN from scratch

```
In [36]: hL = 100 # size of hidden layer  
W1 = 0.01 * np.random.randn(D, hL)  
b1 = np.zeros((1,hL))  
W2 = 0.01 * np.random.randn(hL, K)  
b2 = np.zeros((1,K))
```

```
In [37]: h = np.maximum(0, np.dot(X, W1) + b1) # note, ReLU activation  
print('Hidden layer shape: ', h.shape)  
y = np.dot(h, W2) + b2  
print('Output shape: ', y.shape)
```

Hidden layer shape: (300, 100)
Output shape: (300, 3)

```
In [38]: # compute the class probabilities  
exp_y = np.exp(y)  
probs = exp_y / np.sum(exp_y, axis=1, keepdims=True)  
  
# compute loss and cost  
loss = -np.log(probs[range(num_examples),y_true])  
cost = np.sum(loss)/num_examples
```

Coding a simple NN from scratch

- now we start backpropagation:

$$\tilde{h}^{[1]} = x \cdot W^{[1]} + b^{[1]}$$

$$h = \text{ReLU}(\tilde{h}^{[1]})$$

$$y = h \cdot W^{[2]} + b^{[2]}$$

In [39]:

```
# derivative w.r.t. y
dy = probs
dy[range(num_examples),y_true] -= 1
dy /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(h.T, dy)
db2 = np.sum(dy, axis=0, keepdims=True)

# next backprop into hidden layer
dh = np.dot(dy, W2.T)

# backprop the ReLU non-linearity
dh[h <= 0] = 0

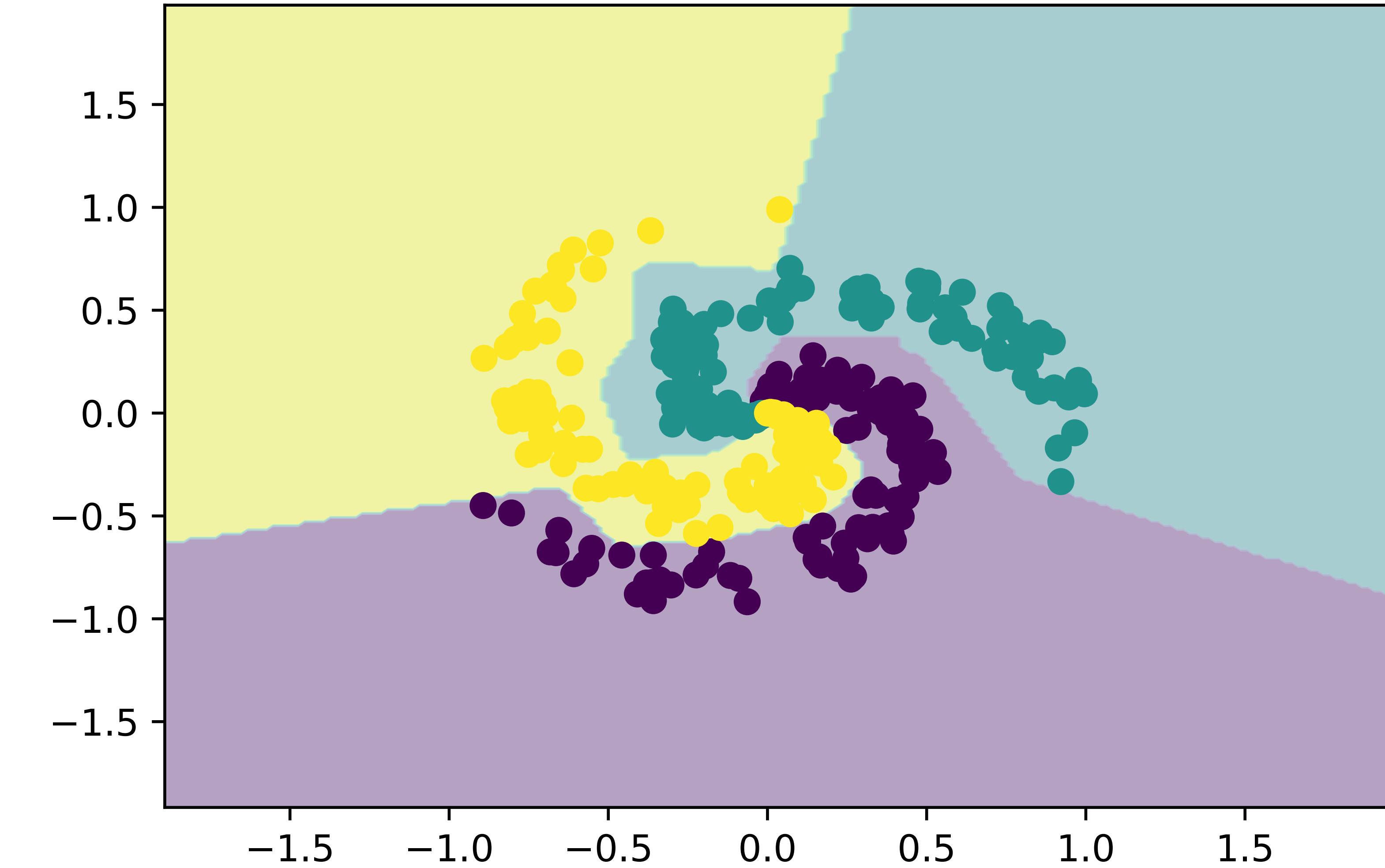
# finally into W,b
dW1 = np.dot(X.T, dh)
db1 = np.sum(dh, axis=0, keepdims=True)

# perform a parameter update
W1 += -step_size * dW1
b1 += -step_size * db1
W2 += -step_size * dW2
b2 += -step_size * db2
```

[go to notebook](#)

Coding a simple NN from scratch

- By implementing this simple fully connected NN we reached an accuracy of ~99% on training set



Libraries for NN development

- In the tutorial we have explicitly implemented all the steps of the forward and back propagations, with the calculation of all the needed derivatives
- Doing that for a deep network with hundreds of layers could easily become a hard job

Libraries for NN development

- In the tutorial we have explicitly implemented all the steps of the forward and back propagations, with the calculation of all the needed derivatives
 - Doing that for a deep network with hundreds of layers could easily become a hard job
-
- There exist several **opensource libraries optimized for tensor computation** which implement the most common kind of layers, activations, loss functions etc, together with their backpropagation
 - Most common: **PyTorch** and **Tensorflow+Keras**, developed in python and GPU accelerated
 - **Tensorflow** is an opensource library developed in python and C++ optimized for machine learning. it has been implemented by Google and released in 2015. Can run on both CPU and GPU (<https://www.tensorflow.org>)
 - **Keras** has a higher level interface specifically used to develop NN and can run with tensorflow backend (<https://keras.io>)

Keras (+ tensorflow)

Losses

Usage of loss functions
Available loss functions
`mean_squared_error`
`mean_absolute_error`
`mean_absolute_percentage_error`
`mean_squared_logarithmic_error`
`squared_hinge`
`hinge`
`categorical_hinge`
`logcosh`
`categorical_crossentropy`
`sparse_categorical_crossentropy`
`binary_crossentropy`
`kullback_leibler_divergence`
`poisson`
`cosine_proximity`

Metrics

Usage of metrics
Arguments
Returns
Available metrics
`binary_accuracy`
`categorical_accuracy`
`sparse_categorical_accuracy`
`top_k_categorical_accuracy`
`sparse_top_k_categorical_accuracy`
Custom metrics

Optimizers

Usage of optimizers
Parameters common to all Keras optimizers
`RMSprop`
`Adagrad`
`Adadelta`
`SGD`
`Adam`
`Adamax`
`Nadam`
`TFOptimizer`

Activations

Usage of activations
Available activations
`elu`
`selu`
`softplus`
`softsign`
`relu`
`tanh`
`sigmoid`
`hard_sigmoid`
`linear`
`softmax`
On "Advanced Activations"

Keras (+ tensorflow)

- 1) **Define a model**, layer by layer, choosing activation functions
- 2) **Compile a model**, setting the optimized, loss function and metric
- 3) **Fit a model** on training set

[go to notebook](#)

Keras (+ tensorflow)

- 1) **Define a model**, layer by layer, choosing activation functions
- 2) **Compile a model**, setting the optimized, loss function and metric
- 3) **Fit a model** on training set

Sequential

```
# For a single-input model with 2 classes (binary classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

Functional

```
# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels) # starts training
```

[go to notebook](#)

Training a NN

- Training a NN is an empirical process, where **tuning of hyper-parameters** (such as #layers, #neurons, learning rate, etc.) is reached through trials and errors
- usually the available labeled data are divided into **training** (80%), **validation** (10%) and **test** (10%)

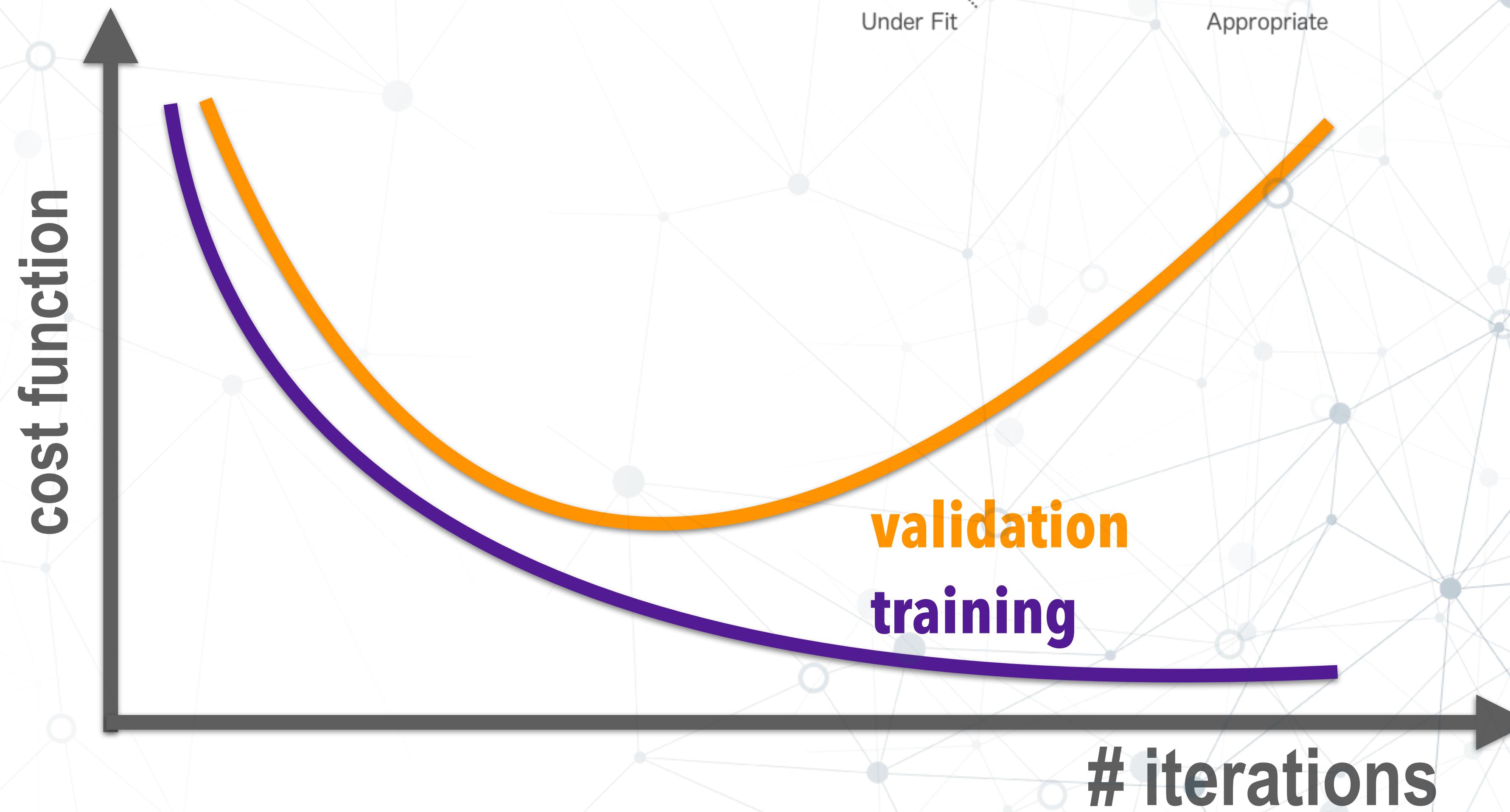
Training: set of data used to optimize NN's weights

Validation: set of data used to tune NN's hyper-parameters

Test: set of data used to test and assess the final NN's performances once hyperparameters are set

Training a NN: Overfitting

- One of the typical problem is **overfitting** which doesn't allow the NN to generalize results



Training a NN: regularization

- **Regularization:** a regularization term R is added to the cost function. This term imposes a penalty on the complexity of \mathcal{L} . the hyperparameter λ controls the importance of the regularization term

$$\mathcal{J} = \frac{1}{N} \sum_{i=0}^N \mathcal{L}_i + \lambda R(\mathcal{L})$$

Training a NN: regularization

- **Regularization:** a regularization term R is added to the cost function. This term imposes a penalty on the complexity of \mathcal{L} . the hyperparameter λ controls the importance of the regularization term

$$\mathcal{J} = \frac{1}{N} \sum_{i=0}^N \mathcal{L}_i + \lambda R(\mathcal{L})$$

- a typical example is the L^2 regularization

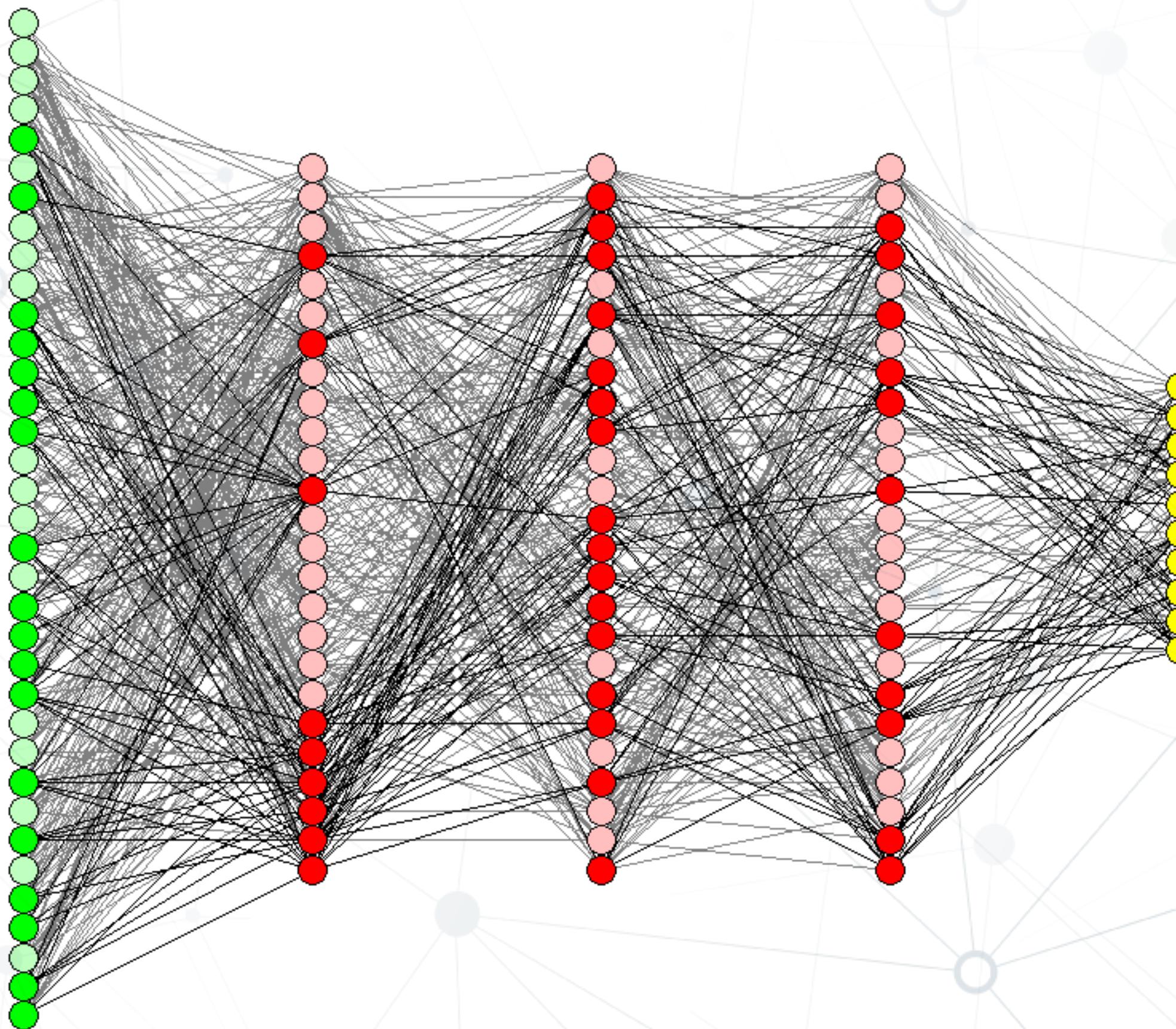
$$\mathcal{J} = \frac{1}{N} \sum_{i=0}^N \mathcal{L}_i + \lambda \sum_{l=1}^L \|W\|^2$$

sum over the layers

it gives preference to smaller values of the NN's weights

Training a NN: dropout

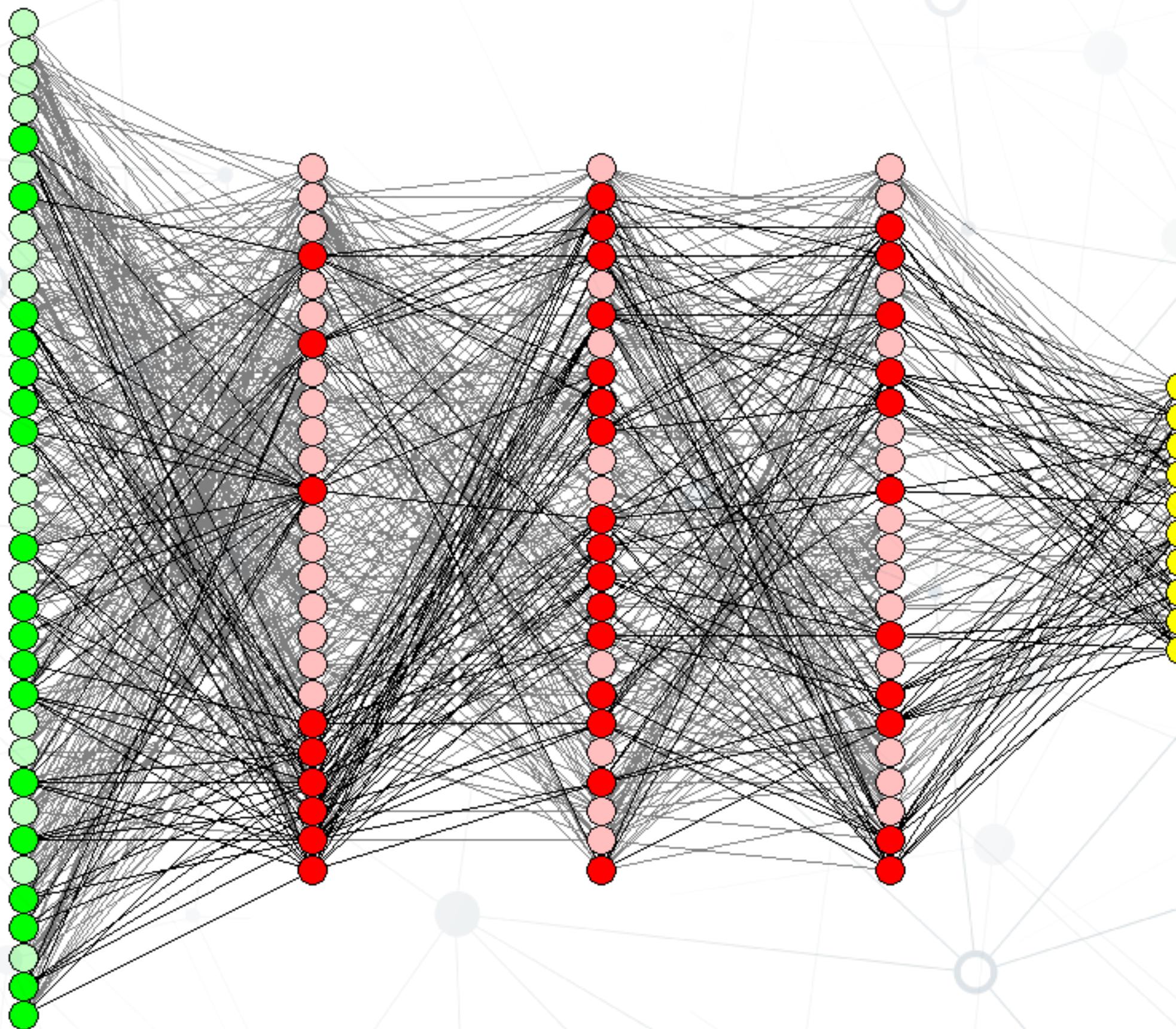
- **Dropout:** during training, at each iteration randomly switch off some neurons during training



- it prevents the NN to rely on the weights of specific neurons
- the number of neurons that are randomly switched off at each iteration is another hyperparameter that needs to be tuned

Training a NN: dropout

- **Dropout:** during training, at each iteration randomly switch off some neurons during training



- it prevents the NN to rely on the weights of specific neurons
- the number of neurons that are randomly switched off at each iteration is another hyperparameter that needs to be tuned

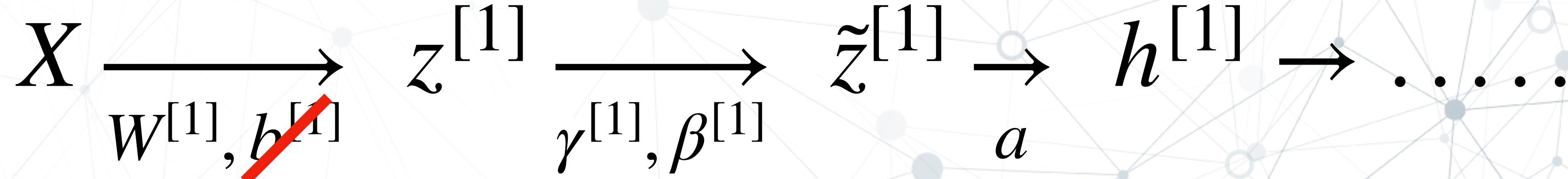
Training a NN: batch normalization

- Inputs of a NN are usually normalized in order to have values in the interval [0, 1] (or [-1, 1]) and small variance
- it has been shown that, in practice, **apply a normalization to each layer of the NN** also improves the speed and efficiency of learning

$$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]}$$

for each neuron $z_i^{[1]}$ the mean and variance over the training set are computed (μ_i and σ_i^2) and:

$$z_{i,normed}^{[1]} = \frac{z_i - \mu_i}{\sqrt{\sigma_i^2 - \epsilon}} \quad \tilde{z}_i^{[1]} = \gamma^{[1]} z_{i,normed}^{[1]} + \beta^{[1]}$$



γ and β are learnable parameters of the NN

Training a NN: mini-batches

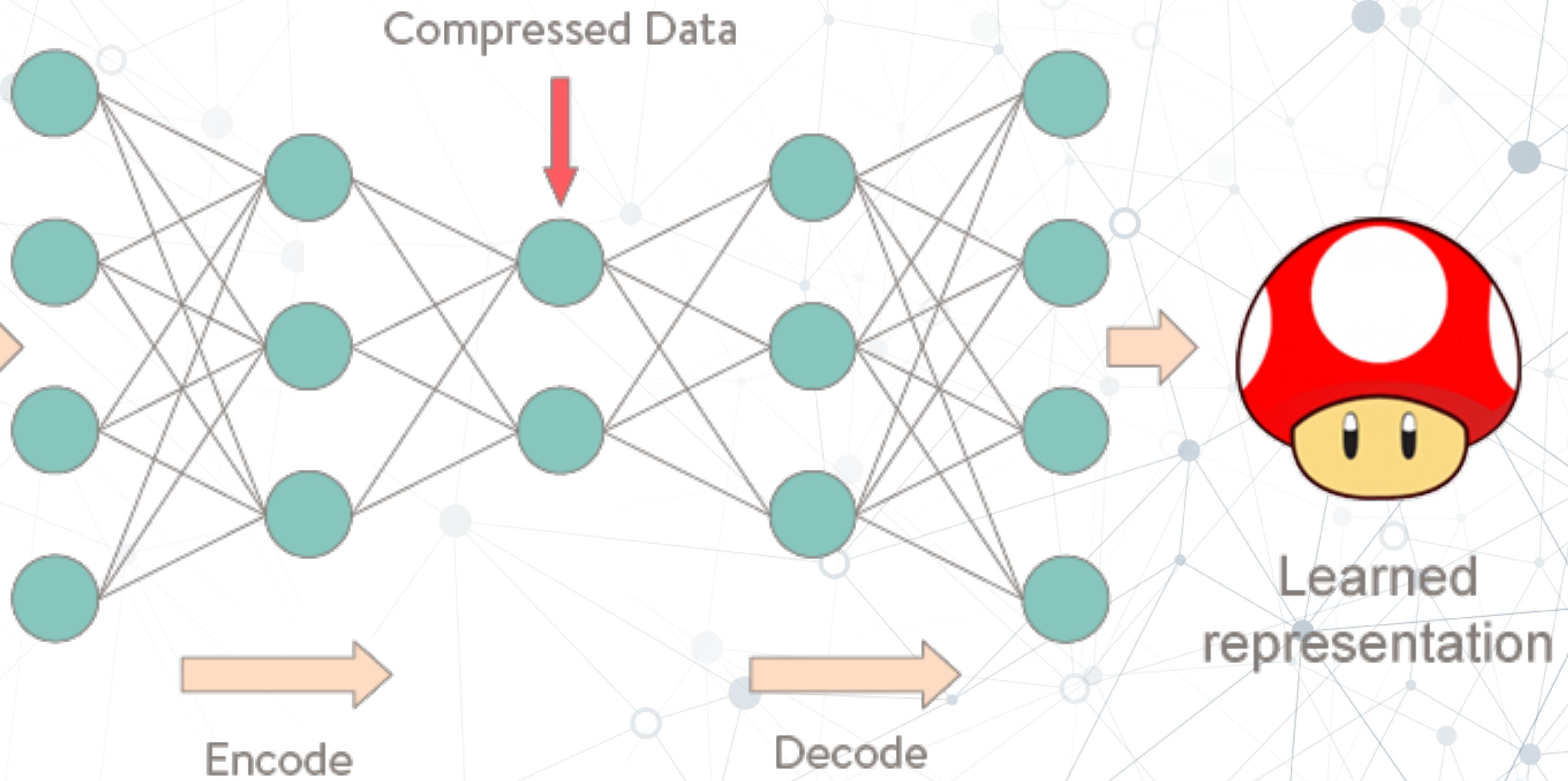
- In our example of spiral arms classification, all the elements of the training set were passed at the same time to the NN, and weights were updated at each iteration
- Usually the training set is instead divided into **mini-batches of few elements** (typically 32-64-128) . A different batch is passed to the NN at each iteration and weights are updated.
- When all the training set has passed through the network and **Epoch** has passed
- The number of elements in each mini-batch is another hyperparameter to be tuned

Autoencoders

- NN used to learn efficient representation of data by compressing dimensionality



Original
mushroom

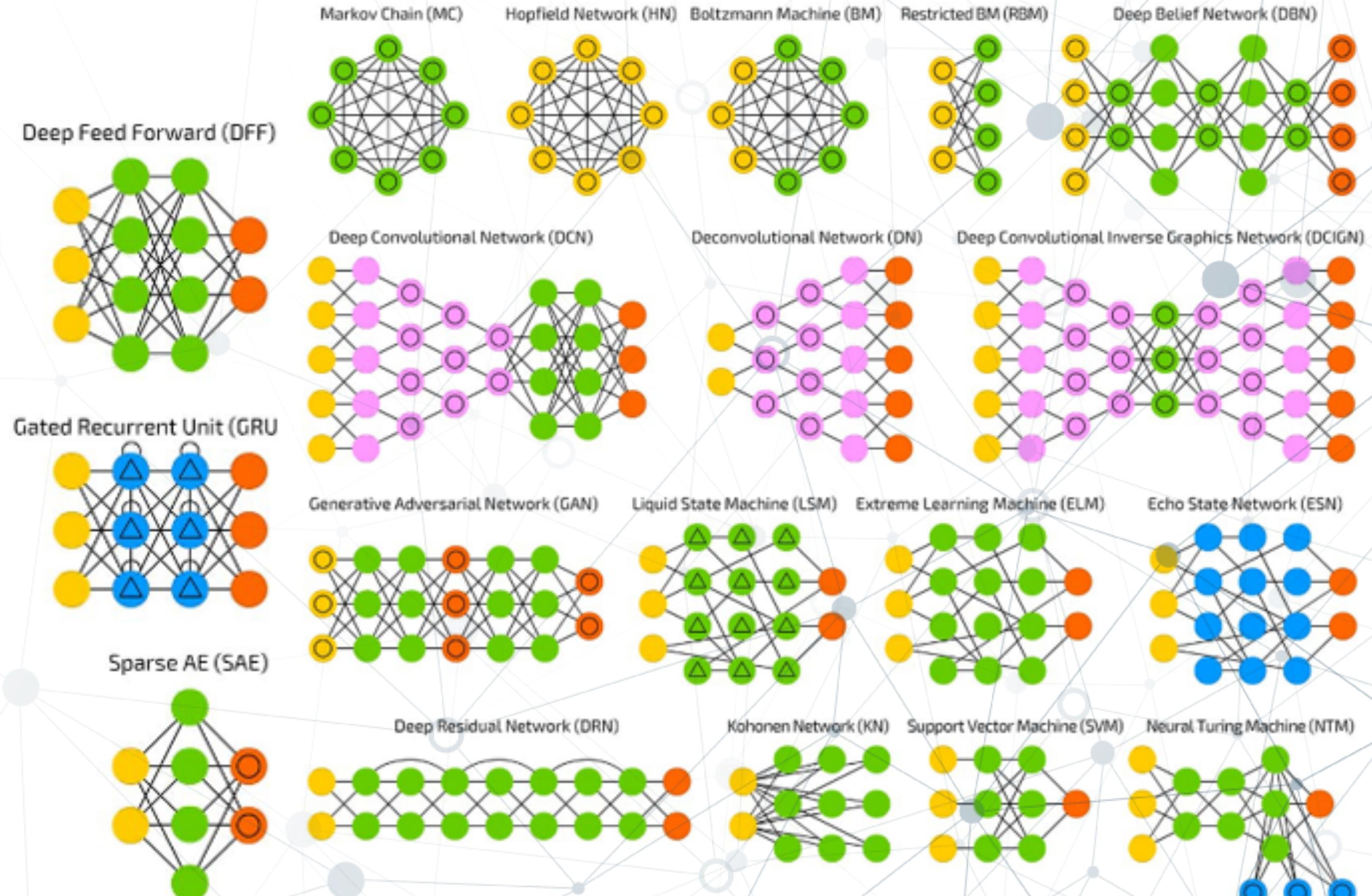
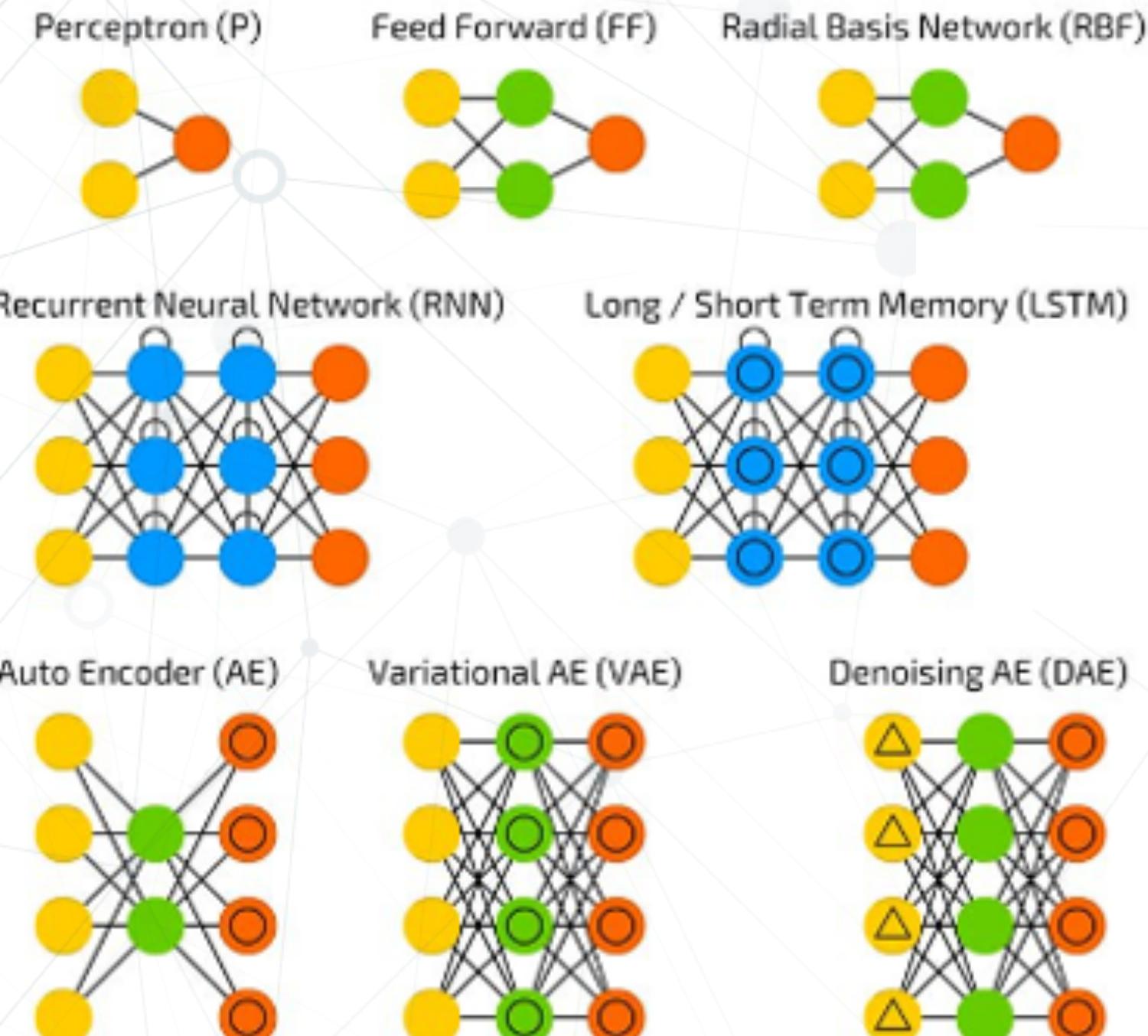


A mostly complete chart of

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool





End of Lecture 1