

TSIA-SD 210 - Machine Learning

Introduction to Neural Networks and Deep Learning

Matthieu Labeau

matthieu.labeau@telecom-paris.fr

Outline

- Introduction
- The formal neuron and Perceptron
- Probabilistic Learning
- From Perceptron to Multi-Layer Perceptron
- Deep Learning

Machine Learning

Reminder: Tom Mitchell's definition (*Machine Learning*, 1997)

A computer program is said to learn

- from **experience** E
- with respect to some class of **tasks** T
- and **performance** measure P

if its performance at tasks in T , as measured by P , improves with experience E .

Machine Learning

Reminder: Tom Mitchell's definition (*Machine Learning*, 1997)

A computer program is said to learn

- from **experience** E
- with respect to some class of **tasks** T
- and **performance** measure P

if its performance at tasks in T , as measured by P , improves with experience E .

- Tasks: *Classification, Regression...* and many others

Machine Learning

Reminder: Tom Mitchell's definition (*Machine Learning*, 1997)

A computer program is said to learn

- from **experience** E
- with respect to some class of **tasks** T
- and **performance** measure P

if its performance at tasks in T , as measured by P , improves with experience E .

- Tasks: *Classification, Regression...* and many others
- Performance: For example, *Accuracy* - but can be tricky to choose !

Supervised vs Unsupervised Learning

Experience: What kind of *dataset* do you have ?

Supervised vs Unsupervised Learning

Dataset X : collection of n *data points* $(\mathbf{x}_i)_{i=1}^n \in \mathbb{R}^d$

Supervised vs Unsupervised Learning

Dataset X : collection of n *data points* $(\mathbf{x}_i)_{i=1}^n \in \mathbb{R}^d$

→ Each consisting of d *features* $(x_i^j)_{j=1}^d \in \mathbb{R}$

Supervised vs Unsupervised Learning

Dataset X : collection of n *data points* $(\mathbf{x}_i)_{i=1}^n \in \mathbb{R}^d$

With *Supervised* learning, each data point \mathbf{x}_i is associated with a *label* or *target* y_i .

- We learn a function that can predict y_i from \mathbf{x}_i .

Supervised vs Unsupervised Learning

Dataset X : collection of n *data points* $(\mathbf{x}_i)_{i=1}^n \in \mathbb{R}^d$

With *Supervised* learning, each data point \mathbf{x}_i is associated with a *label* or *target* y_i .

- We learn a function that can predict y_i from \mathbf{x}_i .

With *Unsupervised* learning, we don't have labels.

- We try to make sense of the \mathbf{x}_i
- It can be with *clustering*, or by learning the *probability distribution* that generated them

Supervised vs Unsupervised Learning

Supervised vs Unsupervised Learning

Supervised and *Unsupervised* learning are rough categories, and are **not formal**.

- Other variants are possible: for example, *Semi-supervised* learning
- Datasets are not necessarily fixed: for example, with *Reinforcement* learning

Supervised vs Unsupervised Learning

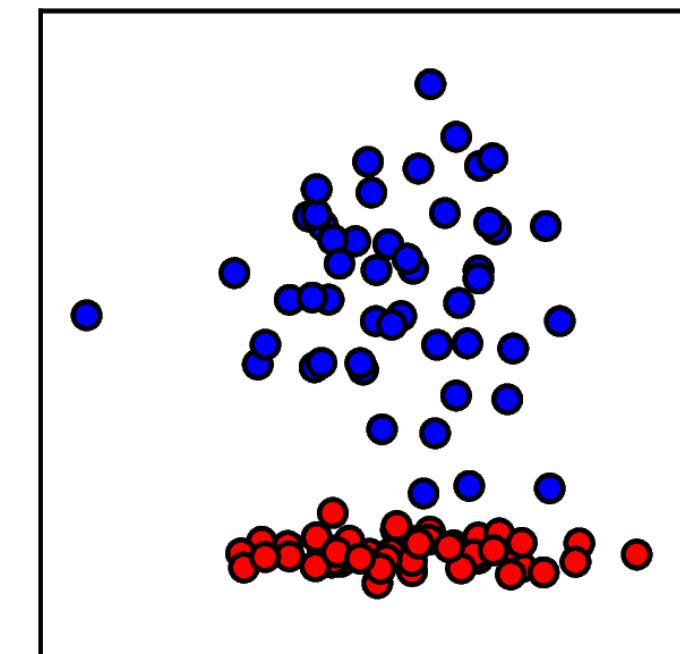
Supervised and *Unsupervised* learning are rough categories, and are **not formal**.

- Other variants are possible: for example, *Semi-supervised* learning
 - Datasets are not necessarily fixed: for example, with *Reinforcement* learning
- Neural networks and Deep learning can be used for all these variants !

Back to (supervised) binary classification

Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

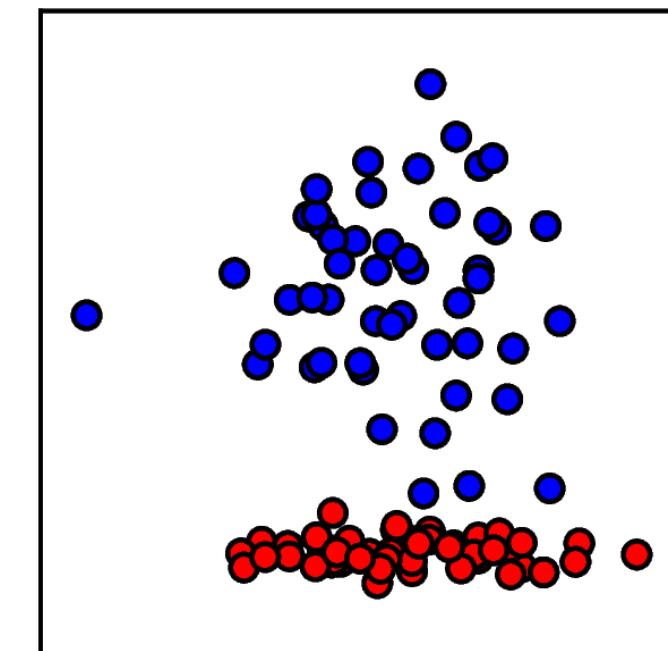
→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Back to (supervised) binary classification

Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



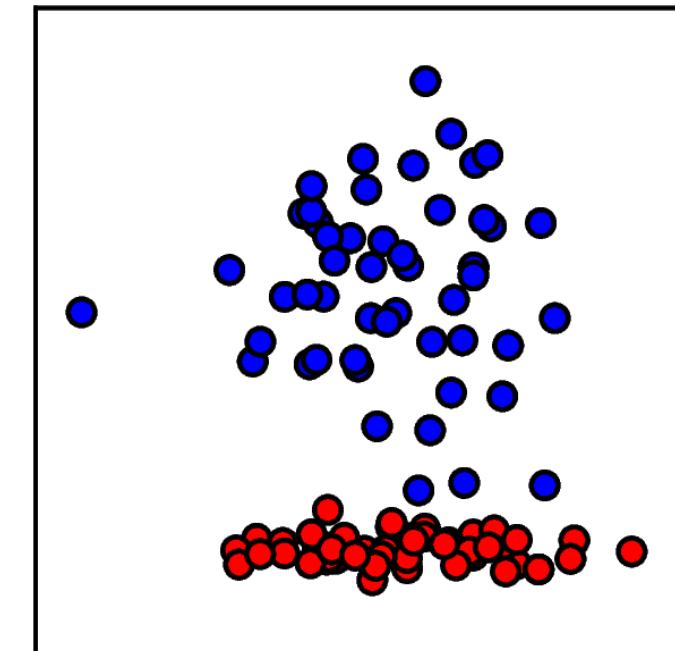
Reminder: **Generative approach**

- How can we model the joint distribution $p(\mathbf{x}_i, y_i)$ of the features and classes ?

Back to (supervised) binary classification

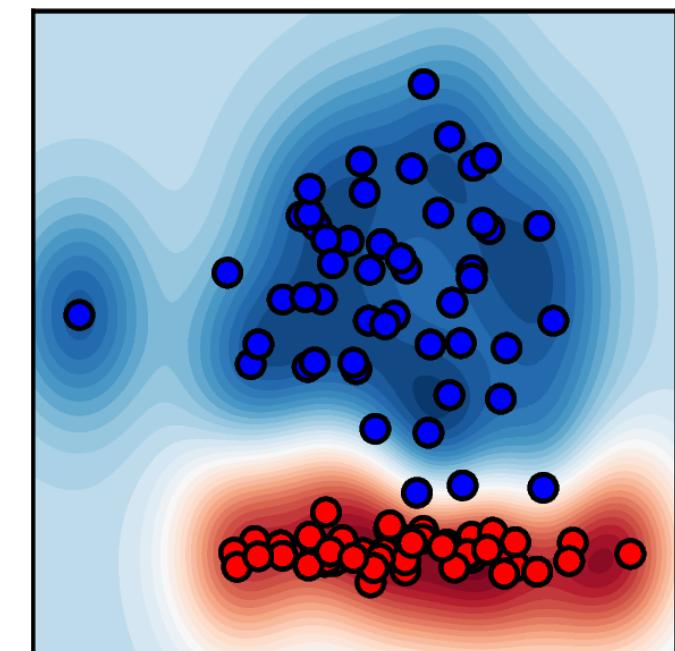
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Reminder: **Generative approach**

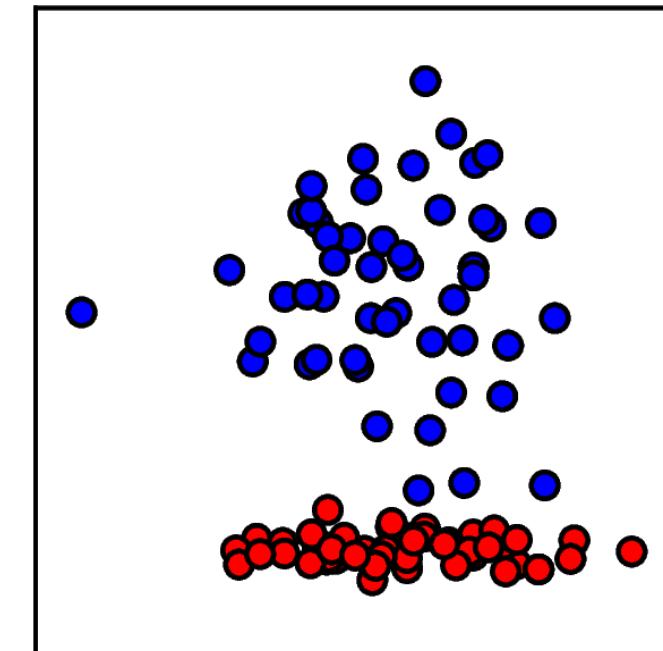
- How can we model the joint distribution $p(\mathbf{x}_i, y_i)$ of the features and classes ?
- Using the estimated $\hat{p}(\mathbf{x}_i | y_i)$ and prior probabilities !



Back to (supervised) binary classification

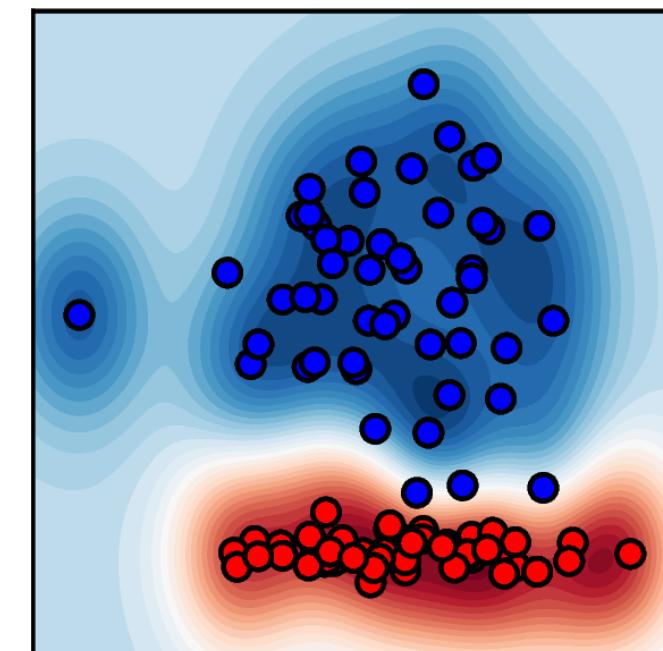
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Reminder: **Generative approach**

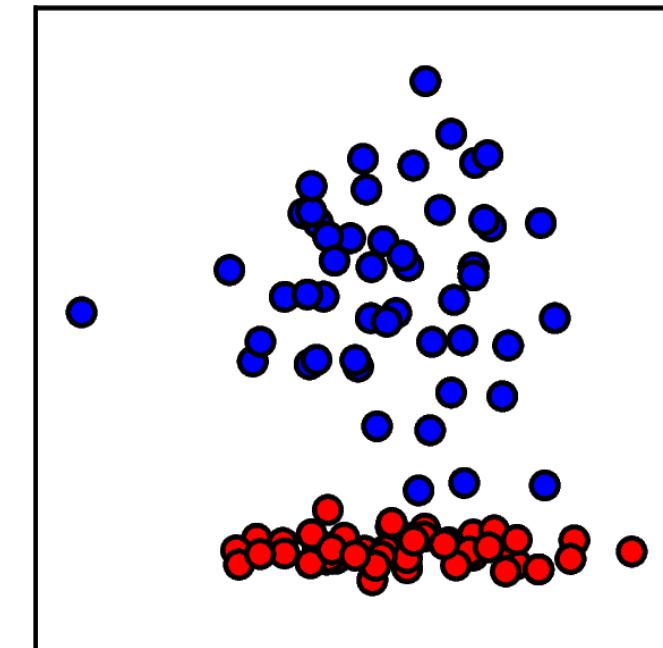
- How can we model the joint distribution $p(\mathbf{x}_i, y_i)$ of the features and classes ?
- Examples: Naïve Bayes, Generative/Hidden Markov Models



Back to (supervised) binary classification

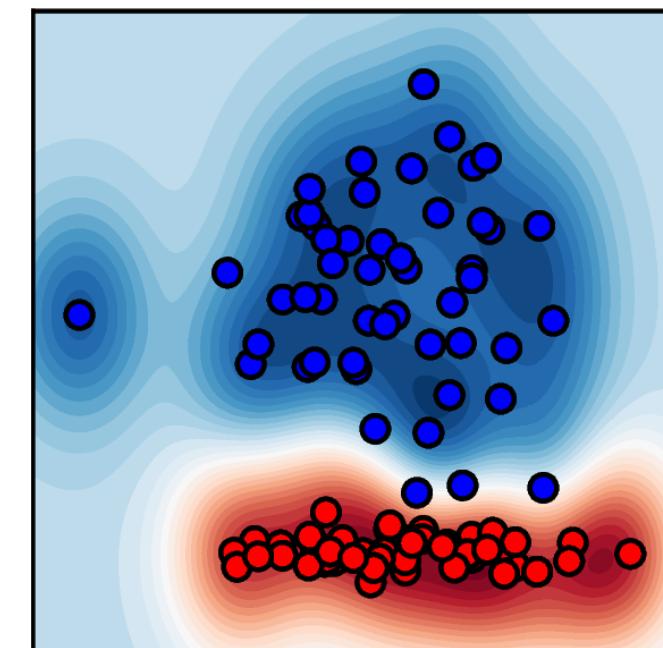
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Reminder: **Generative approach**

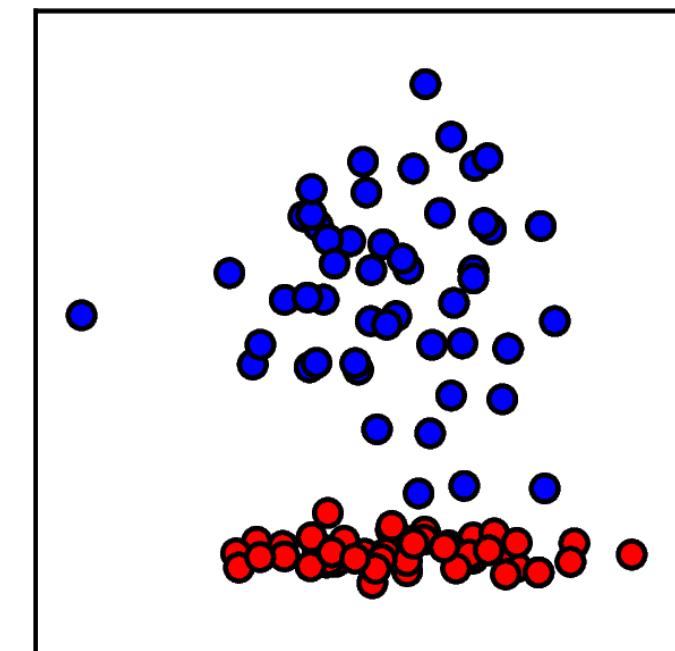
- How can we model the joint distribution $p(\mathbf{x}_i, y_i)$ of the features and classes ?
- Examples: Naïve Bayes, Generative/Hidden Markov Models



Back to (supervised) binary classification

Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

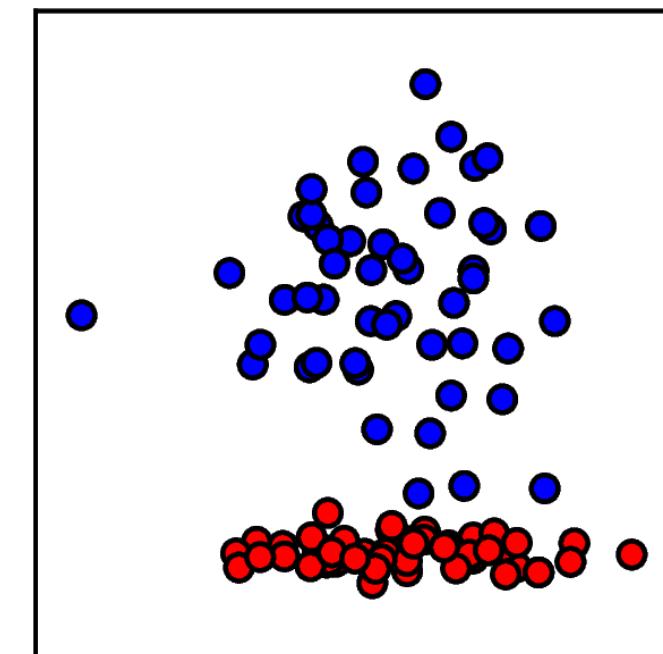
→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Back to (supervised) binary classification

Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



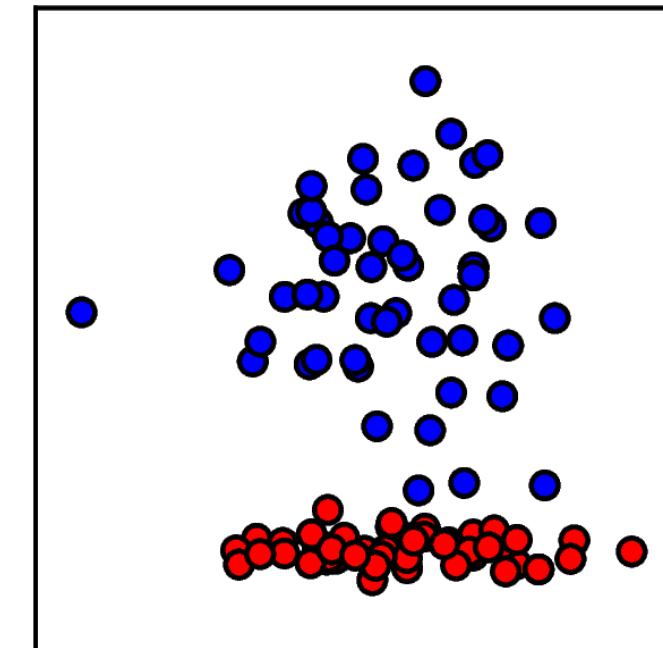
Reminder: **Discriminative approach**

- Which features x_i^j can allow us to distinguish between the classes ?

Back to (supervised) binary classification

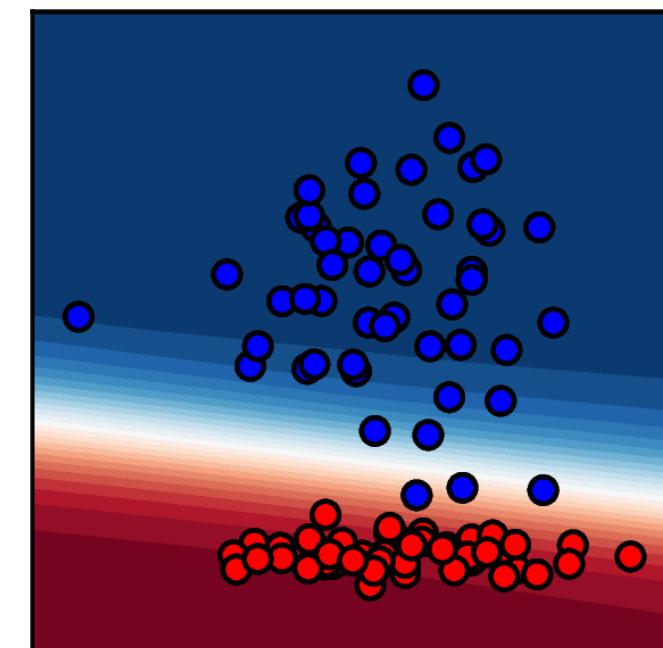
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Reminder: **Discriminative approach**

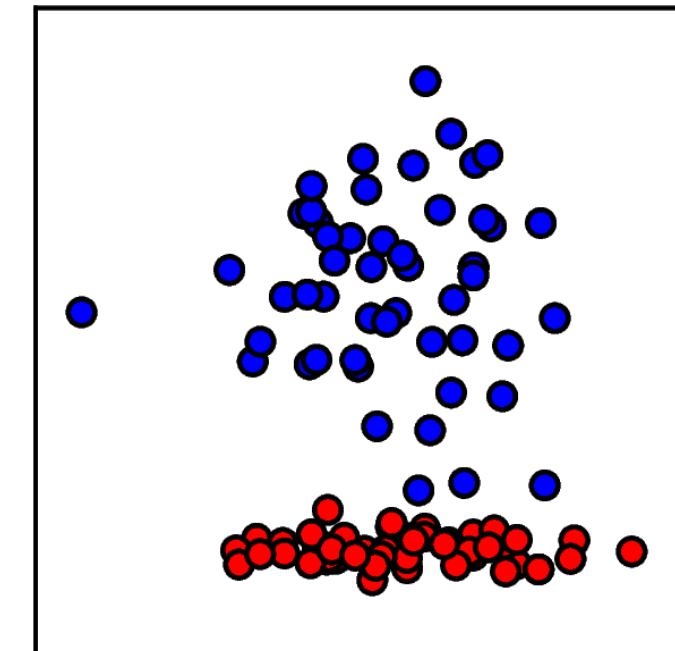
- Which features x_i^j can allow us to distinguish between the classes ?
- It means finding a boundary between the classes (which can be by computing $p(y_i | \mathbf{x}_i)$, but not necessarily !)



Back to (supervised) binary classification

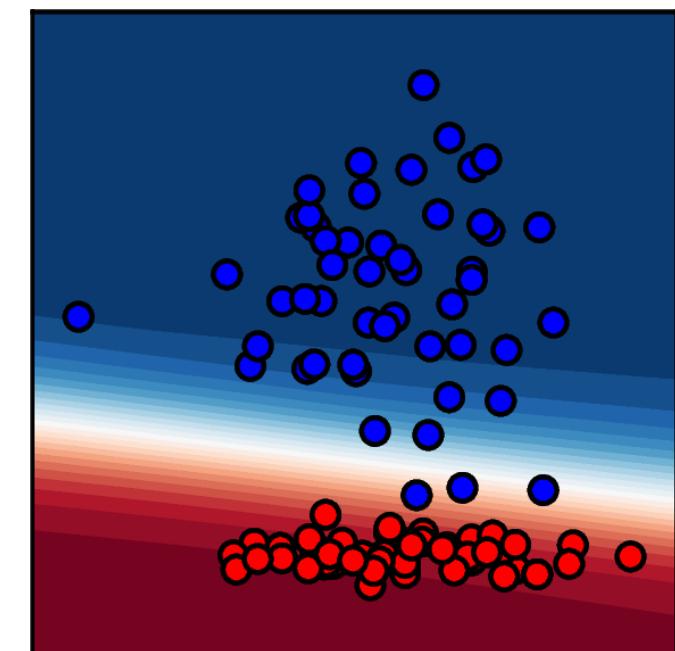
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



Reminder: **Discriminative approach**

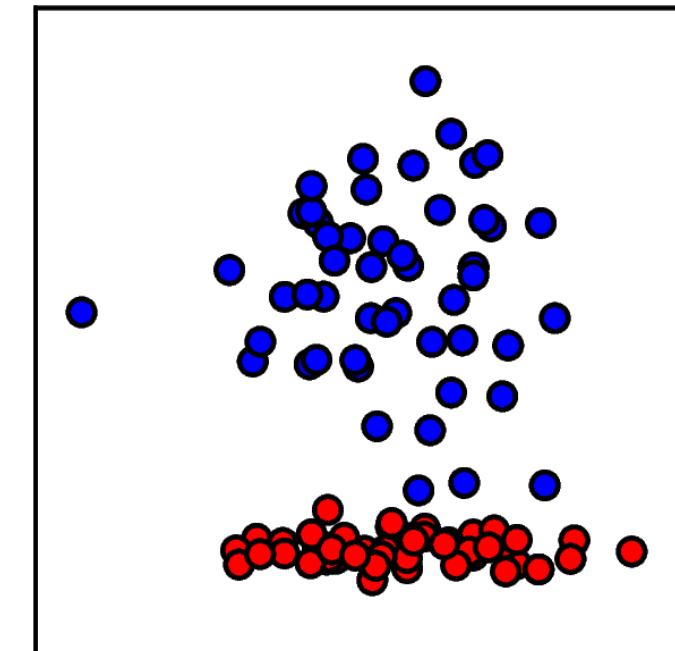
- Which features x_i^j can allow us to distinguish between the classes ?
- Examples: Decision trees, SVM



Back to (supervised) binary classification

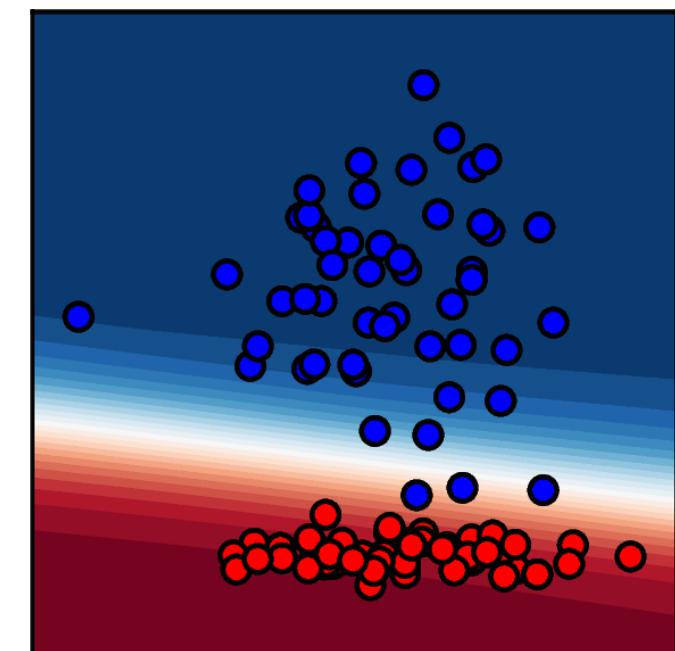
Data points $\mathbf{x}_i \in \mathbb{R}^2$ with labels $y_i \in \{0, 1\}$

→ Learn a function $f(\mathbf{x}_i)$ able to predict y_i



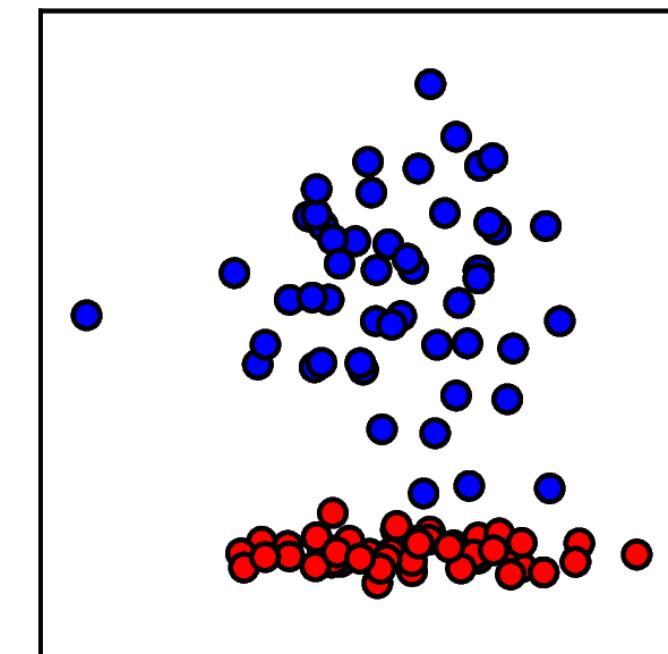
Reminder: **Discriminative approach**

- Which features x_i^j can allow us to distinguish between the classes ?
- Examples: Decision trees, SVM



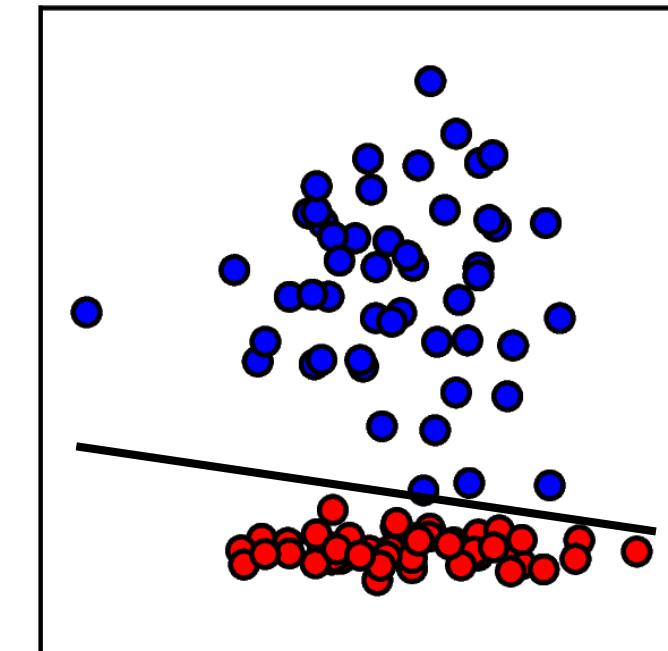
Simple Discriminative classification : Perceptron

Reminder: When the classes are linearly separable, we are looking for a **separating hyperplane** !



Simple Discriminative classification : Perceptron

Reminder: When the classes are linearly separable, we are looking for a **separating hyperplane** !

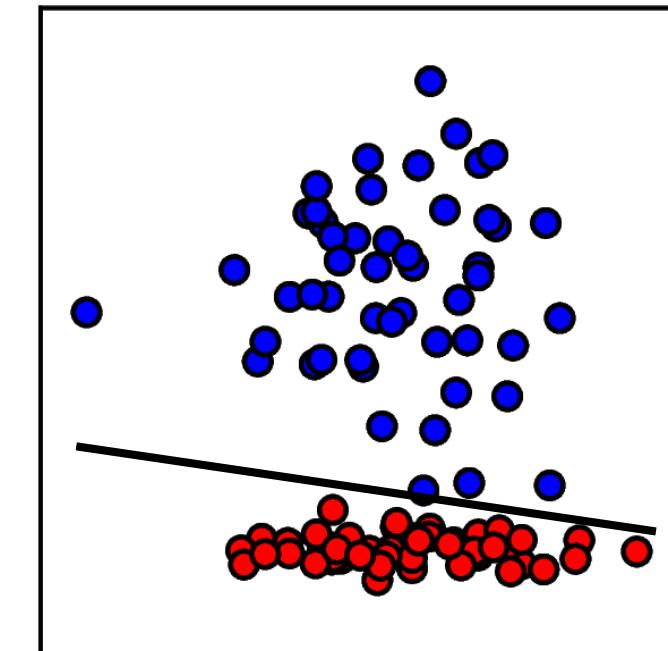


Such an hyperplane is defined by the equation:

$$\mathbf{w}^T \mathbf{x} + b = 0$$

Simple Discriminative classification : Perceptron

Reminder: When the classes are linearly separable, we are looking for a **separating hyperplane** !



Such an hyperplane is defined by the equation:

$$\mathbf{w}^T \mathbf{x} + b = 0$$

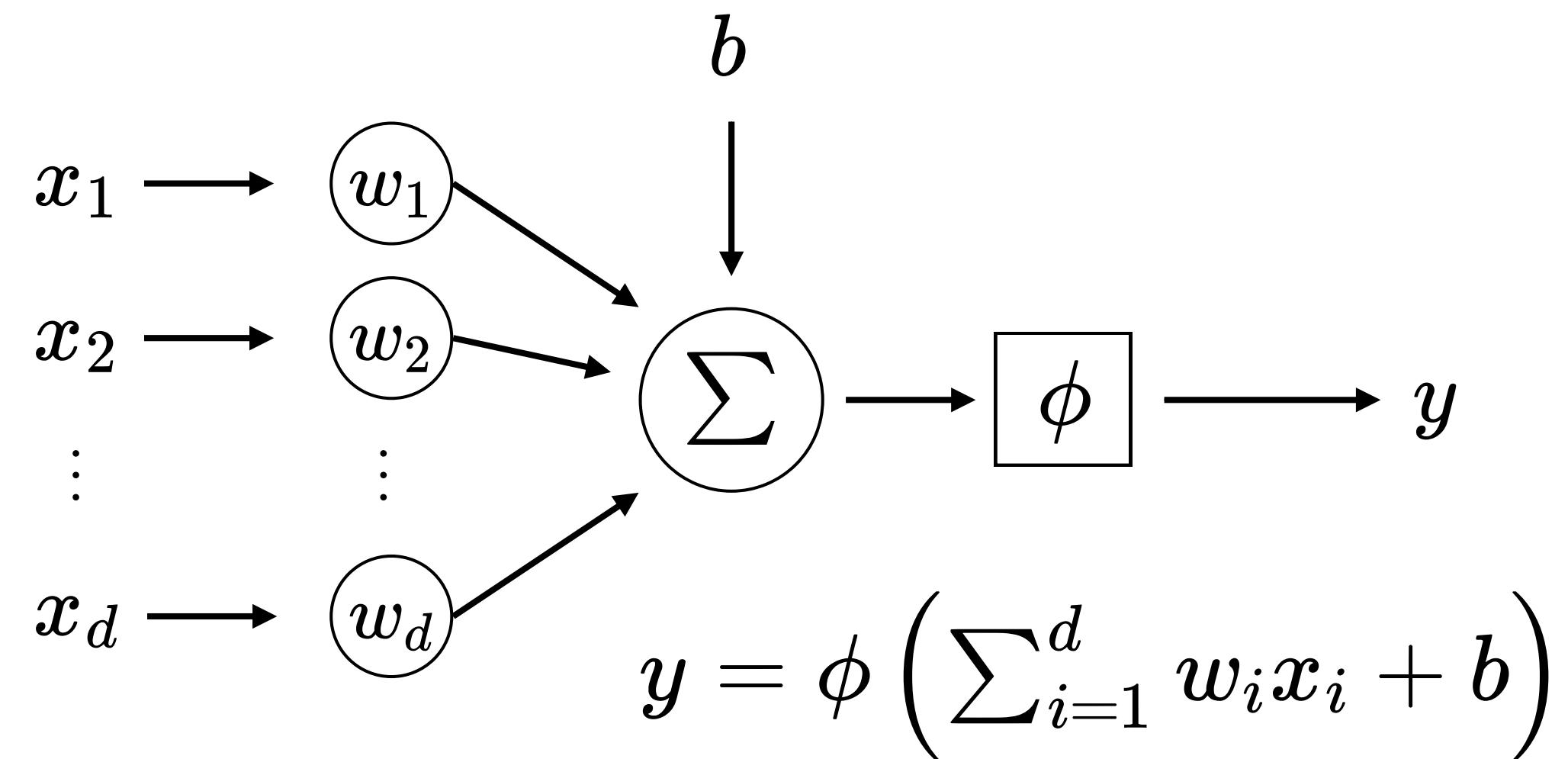
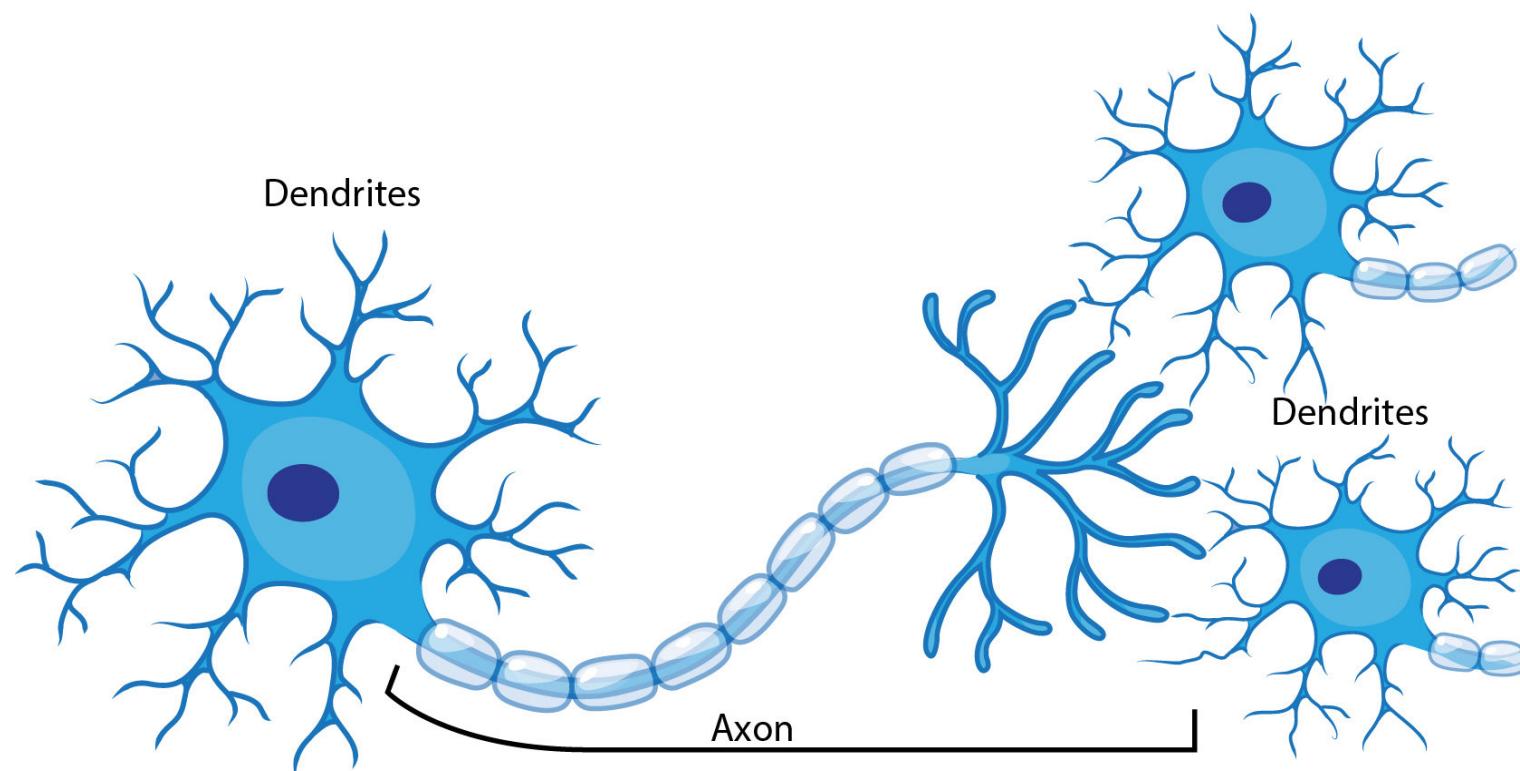
We then have a simple model, called the **Perceptron**, giving the class y :

$$\hat{y}_i = f(\mathbf{x}_i, \theta) = \text{sign}(\mathbf{w}^T \mathbf{x}_i + b)$$

where $\theta = \{\mathbf{w}, b\}$ are the parameters of the model.

Note: here, $y_i \in -1, 1$ by convenience - for as long as we use the *sign* function.

Artificial and biological Neurons



- First proposed to model the **activity of a neuron** by physiologists McCulloch and Pitts (*A logical calculus of the ideas immanent in nervous activity*, 1943)
- The input signal is processed by **weighted** synapses, firing if the weighted sum of the components is above a **threshold**

Perceptron: Learning Rule

Proposed by Rosenblatt (*The Perceptron—a perceiving and recognizing automaton*, 1958), with a **learning rule**

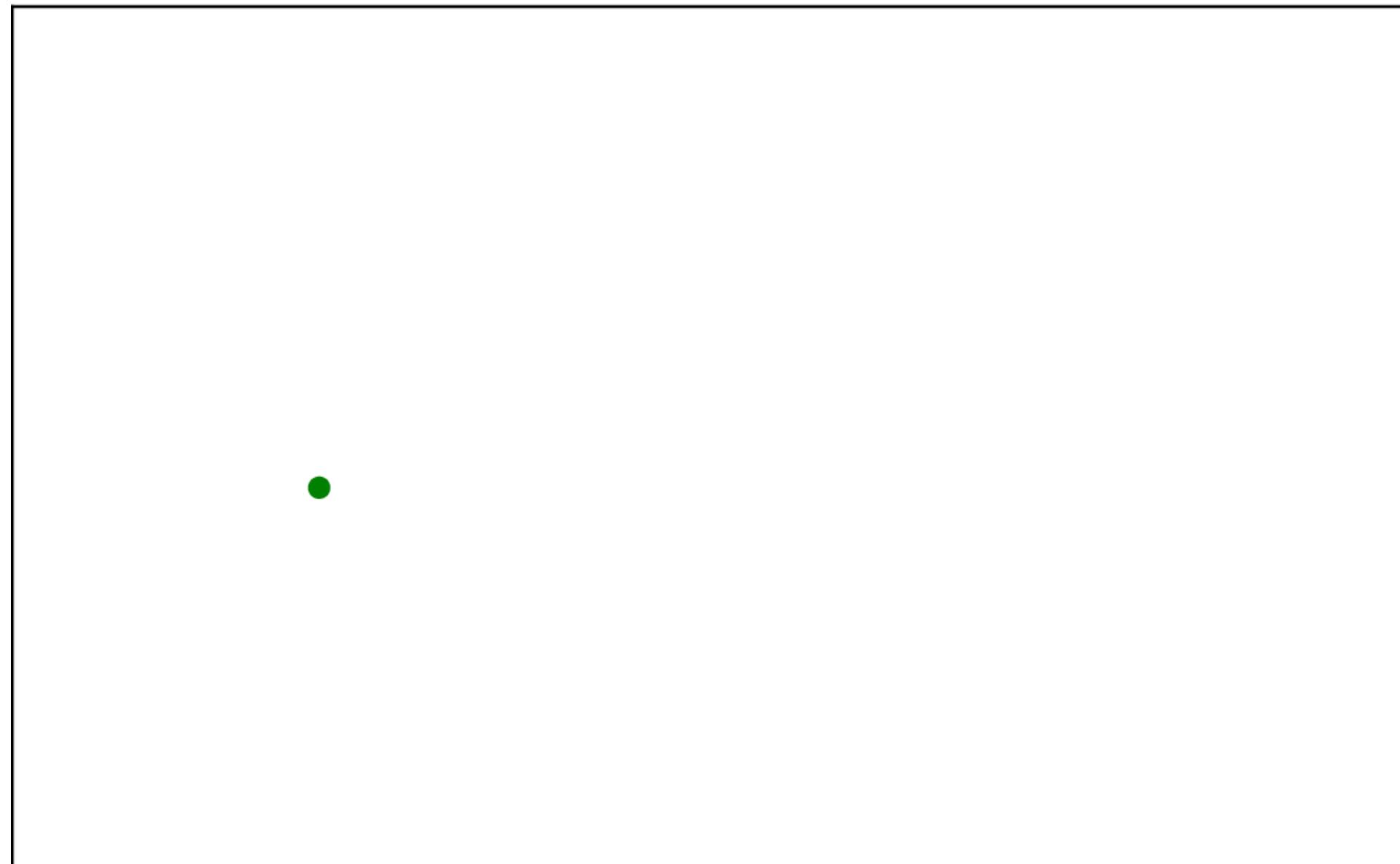
→ Allows to **update** the boundary when new examples are introduced

Perceptron learning procedure:

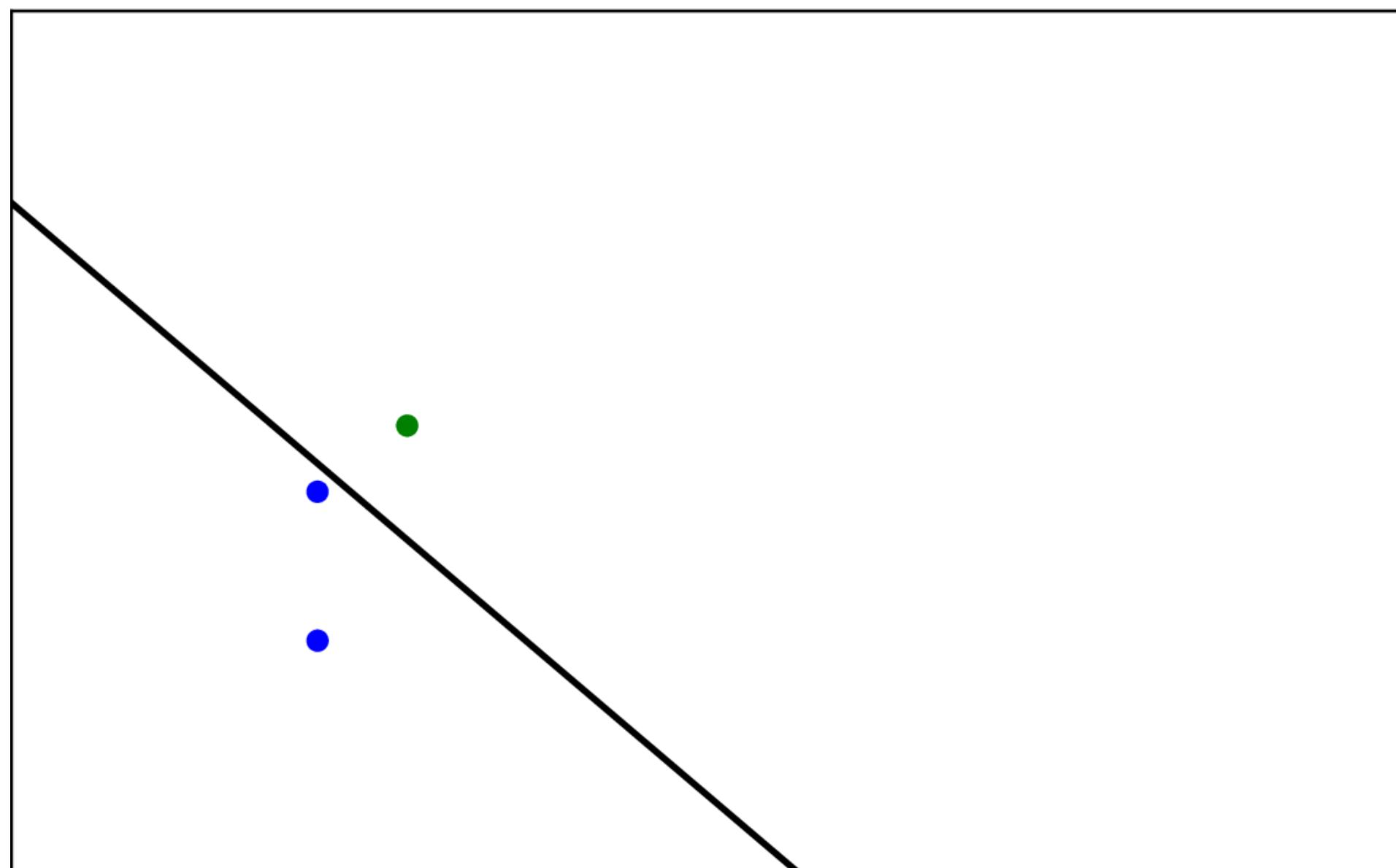
- Start with $t = 1$, $\mathbf{w}_t = 0$, and examples $(\mathbf{x}_i, y_i)_{i=1}^n$ such as $\|\mathbf{x}_i\| = 1$
 - While the model makes mistakes:
 - For $i \in [1, n]$:
 - Predict $\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x}_i)$
 - If $y_i \neq \hat{y}_i$ do $\mathbf{w}_{t+1} = \mathbf{w}_t + \text{sign}(y_i)\mathbf{x}_i$
- Note: we include the bias b in \mathbf{w}

Perceptron: Learning Rule

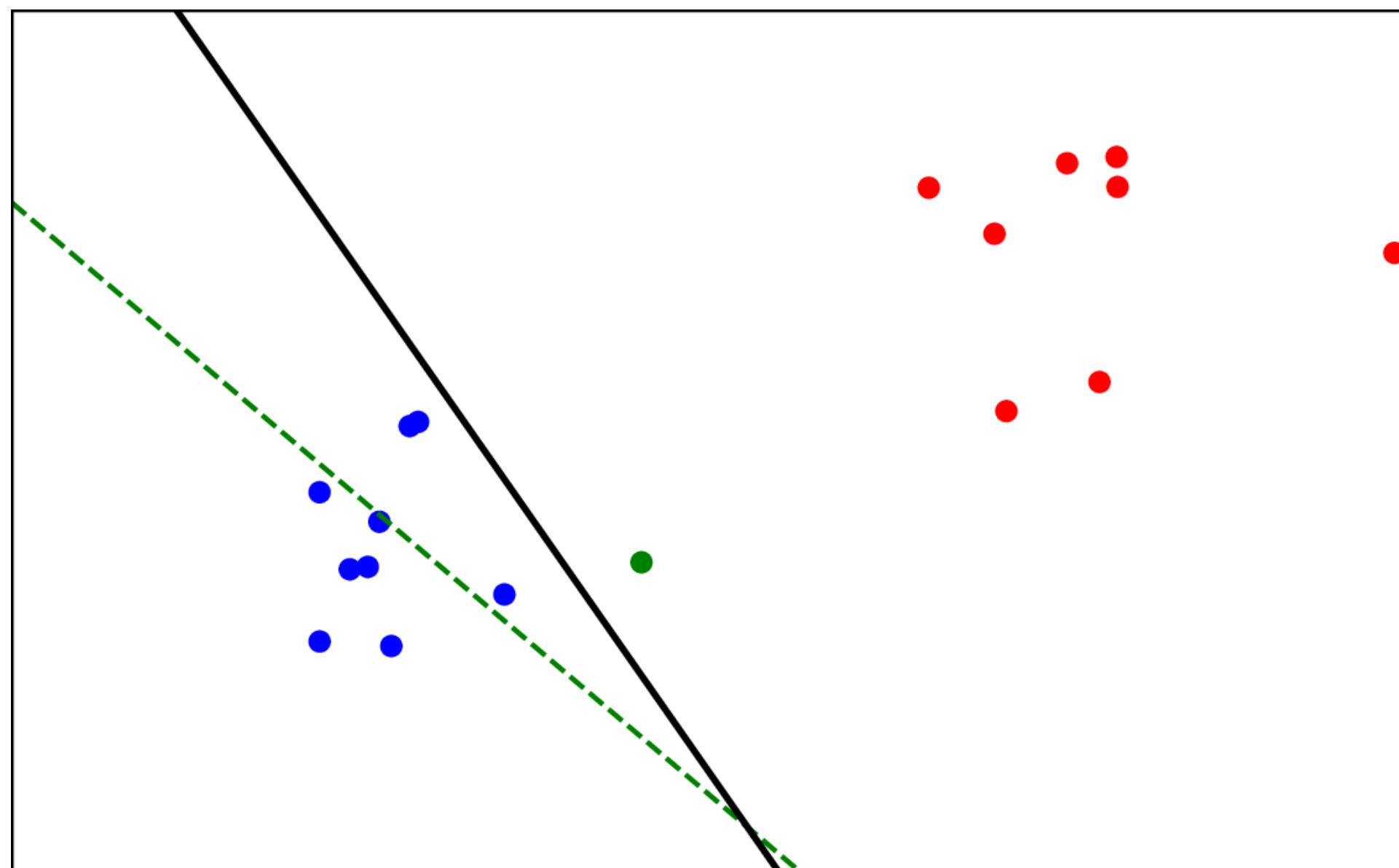
Perceptron: Learning Rule



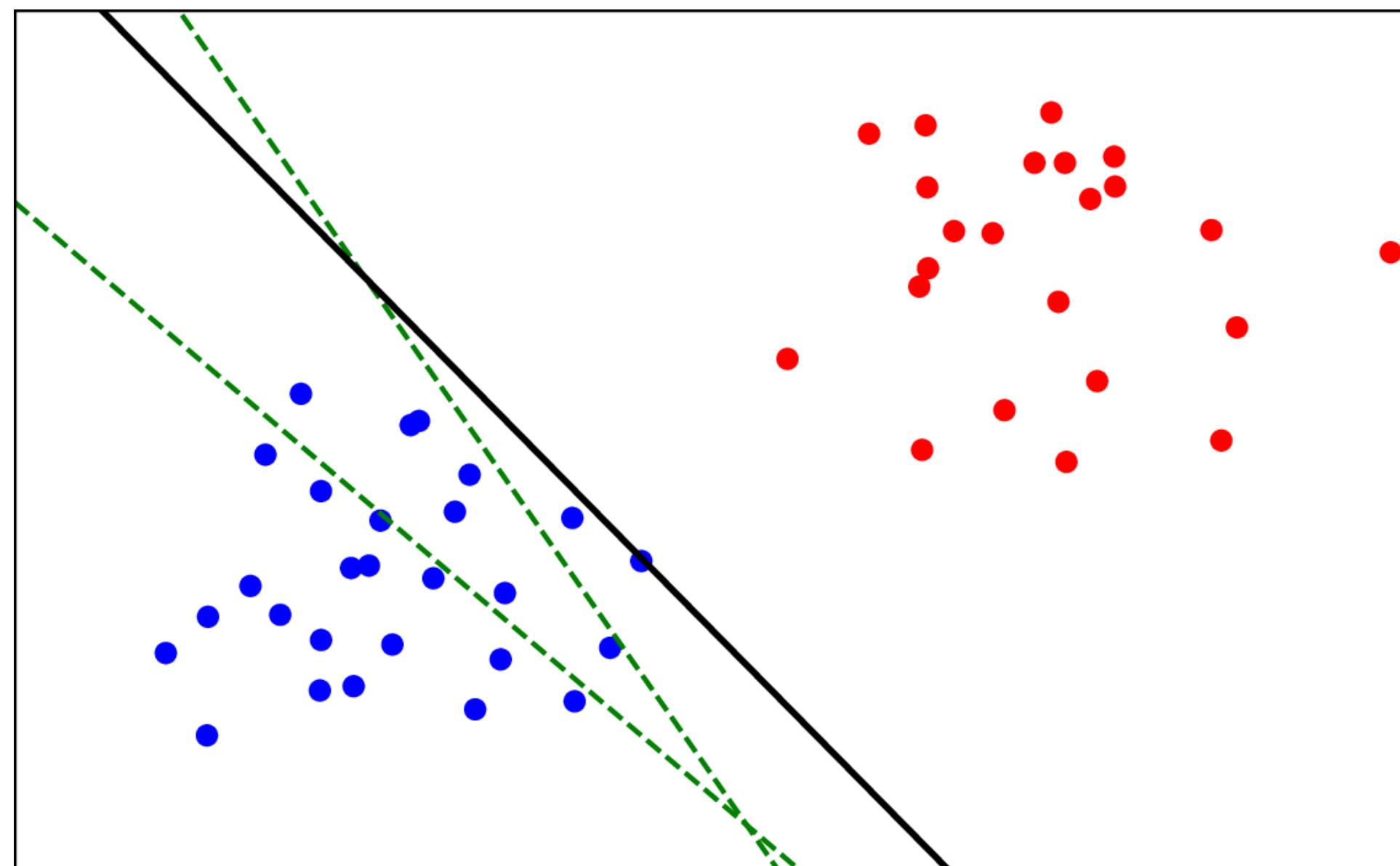
Perceptron: Learning Rule



Perceptron: Learning Rule



Perceptron: Learning Rule



Perceptron: Convergence

The algorithm converges if the classes are exactly **linearly separable**. More formally:

Convergence theorem:

Let us assume that there exists a parameter \mathbf{w}^* such that $\|\mathbf{w}^*\| = 1$ and note $\gamma > 0$ such that:

$$\forall i \in [1, n], \quad y_i \times (\mathbf{x}_i^T \mathbf{w}^*) \geq \gamma$$

and $\exists R > 0$ such that:

$$\|\mathbf{x}_i\| \leq R$$

Then the perceptron algorithm converges in at most $\frac{R^2}{\gamma^2}$ iterations

Perceptron: Proof of convergence

Main ideas:

- Linear separability implies the existence of \mathbf{w}^*
- γ represents the margin
- R is the maximum norm of an input vector

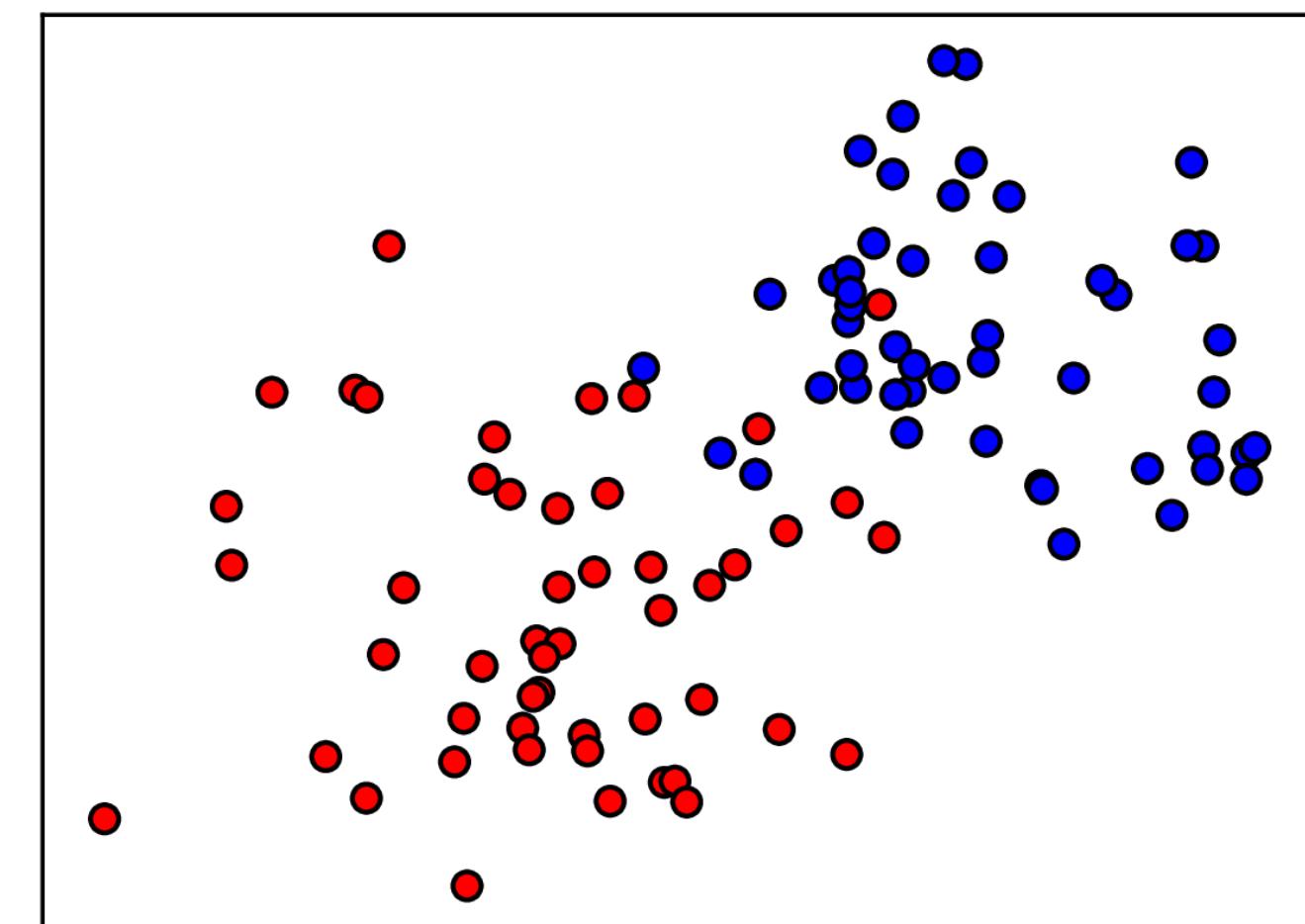
→ With both these quantities, we can bound the amount of change \mathbf{w}^t undergoes after t updates, and therefore obtain a maximum number of iterations after which it separates the classes.
→ The proof is made by recurrence.

Perceptron: Limitations

- **Quality of the boundary:** That's what SVM are made for (They are designed around maximizing a *margin* !)
- **Case of Non-linear separability:**
 - The data is *almost* linearly separable: for example, because of noise.
→ The *Bayes classifier* (*classifier minimizing the probability of misclassification*) is still an hyperplane

Perceptron: Limitations

- **Quality of the boundary:** That's what SVM are made for (They are designed around maximizing a *margin* !)
- **Case of Non-linear separability:**
 - The data is *almost* linearly separable: for example, because of noise.
→ The *Bayes classifier* (*classifier minimizing the probability of misclassification*) is still an hyperplane

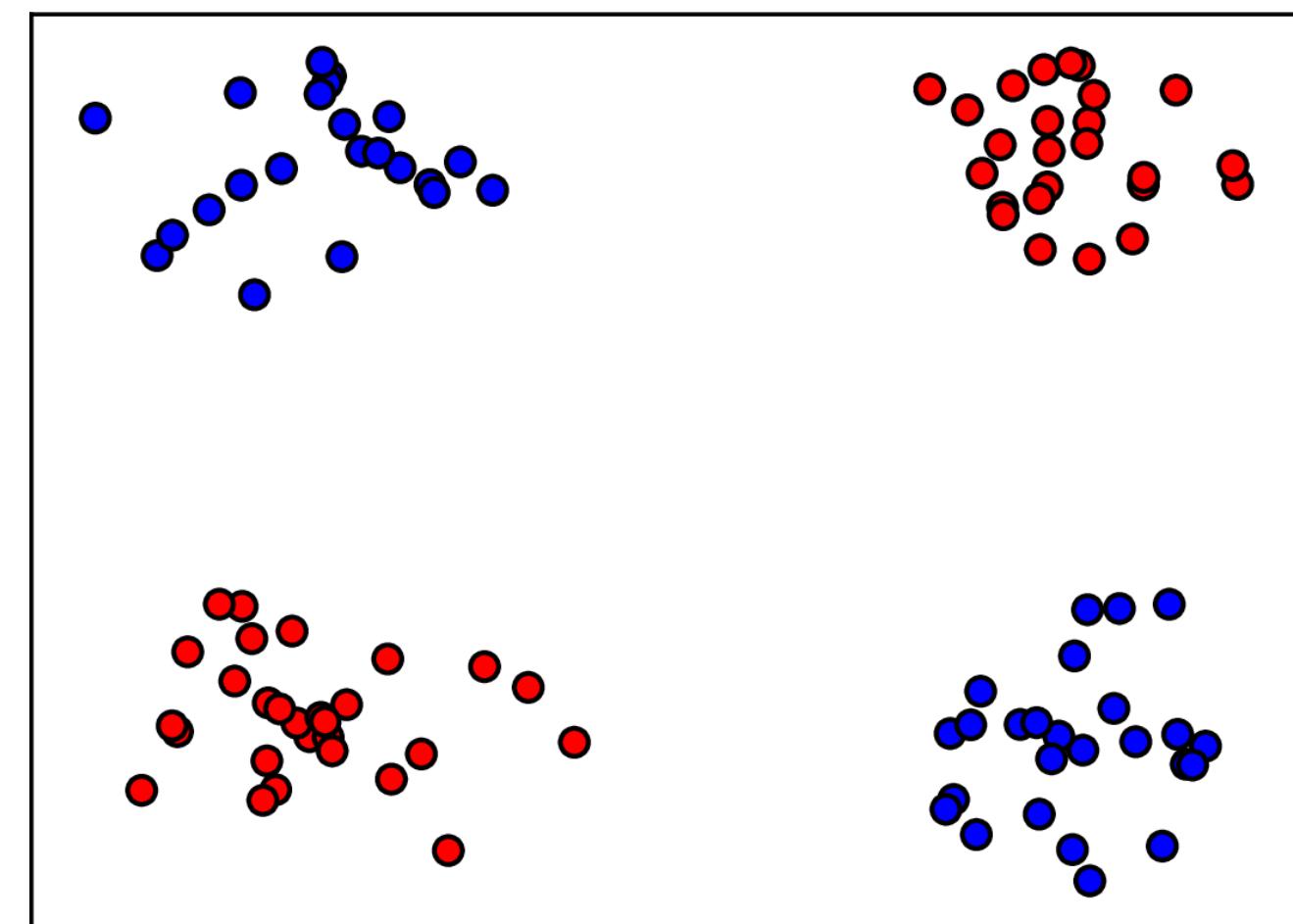


Perceptron: Limitations

- **Quality of the boundary:** That's what SVM are made for (They are designed around maximizing a *margin* !)
- **Case of Non-linear separability:**
 - The data is *almost* linearly separable: for example, because of noise.
→ The *Bayes classifier* (*classifier minimizing the probability of misclassification*) is still an hyperplane
 - The data is not linearly separable.
→ The *Bayes classifier* is not an hyperplane

Perceptron: Limitations

- **Quality of the boundary:** That's what SVM are made for (They are designed around maximizing a *margin* !)
- **Case of Non-linear separability:**
 - The data is *almost* linearly separable: for example, because of noise.
→ The *Bayes classifier* (*classifier minimizing the probability of misclassification*) is still an hyperplane
 - The data is not linearly separable.
→ The *Bayes classifier* is not an hyperplane



Perceptron: Loss function

What do we need ?

Perceptron: Loss function

What do we need ?

- Be able to deal with uncertainty: quantify how happy we are with the scores given by the model across the dataset with a **loss function**

Perceptron: Loss function

What do we need ?

- Be able to deal with uncertainty: quantify how happy we are with the scores given by the model across the dataset with a **loss function**
- Efficiently find the parameters that minimize this loss function (*optimization*)

Perceptron: Loss function

What do we need ?

- Be able to deal with uncertainty: quantify how happy we are with the scores given by the model across the dataset with a **loss function**
- Efficiently find the parameters that minimize this loss function (*optimization*)

Given a dataset $(\mathbf{x}_i, y_i)_{i=1}^n$, a loss function l tells us how good our classifier is by being averaged over all examples:

$$l((\mathbf{x}_i, y_i)_{i=1}^n | \mathbf{w}) = \frac{1}{n} \sum_{i=1}^n l_i(\hat{y}_i, y_i) = \frac{1}{n} \sum_{i=1}^n l_i(f(\mathbf{x}_i, \mathbf{w}), y_i)$$

Perceptron: Loss function

What do we need ?

- Be able to deal with uncertainty: quantify how happy we are with the scores given by the model across the dataset with a **loss function**
- Efficiently find the parameters that minimize this loss function (*optimization*)

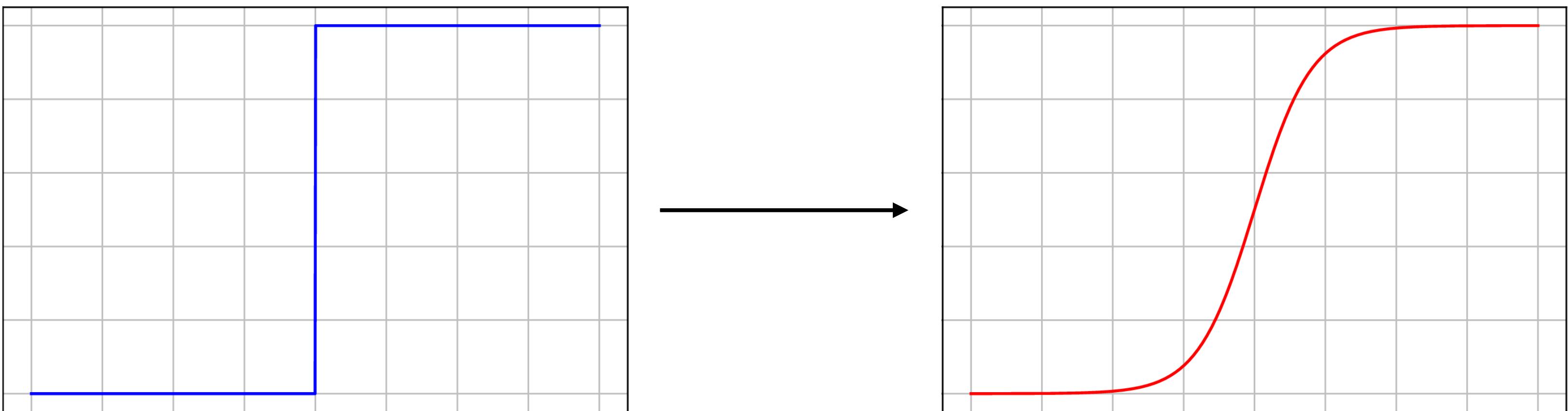
Given a dataset $(\mathbf{x}_i, y_i)_{i=1}^n$, a loss function l tells us how good our classifier is by being averaged over all examples:

$$l((\mathbf{x}_i, y_i)_{i=1}^n | \mathbf{w}) = \frac{1}{n} \sum_{i=1}^n l_i(\hat{y}_i, y_i) = \frac{1}{n} \sum_{i=1}^n l_i(f(\mathbf{x}_i, \mathbf{w}), y_i)$$

We have seen:

- The zero-one loss: $|y - sign(f(\mathbf{x}, \mathbf{w}))|$
- The hinge loss: $\max(0, 1 - y \cdot f(\mathbf{x}, \mathbf{w}))$
- The MSE loss: $(y - f(\mathbf{x}, \mathbf{w}))^2$

Probabilistic Learning



First, replace the *sign* function by the *differentiable sigmoid* function:

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

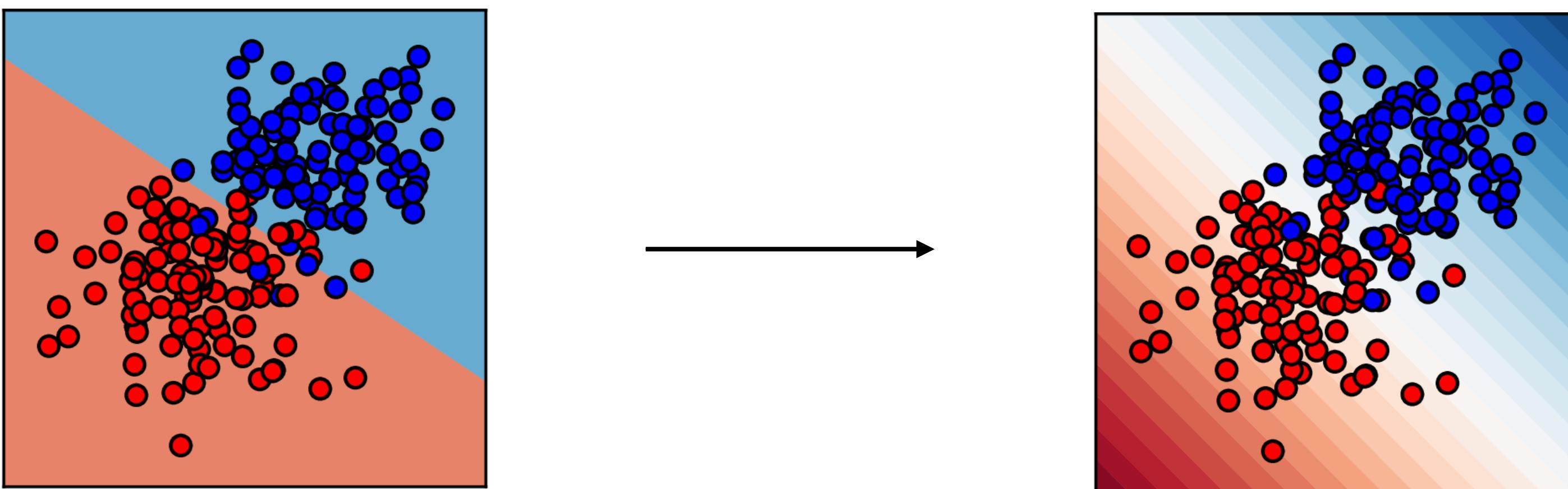
Probabilistic Learning



First, replace the *sign* function by the *differentiable sigmoid* function:

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

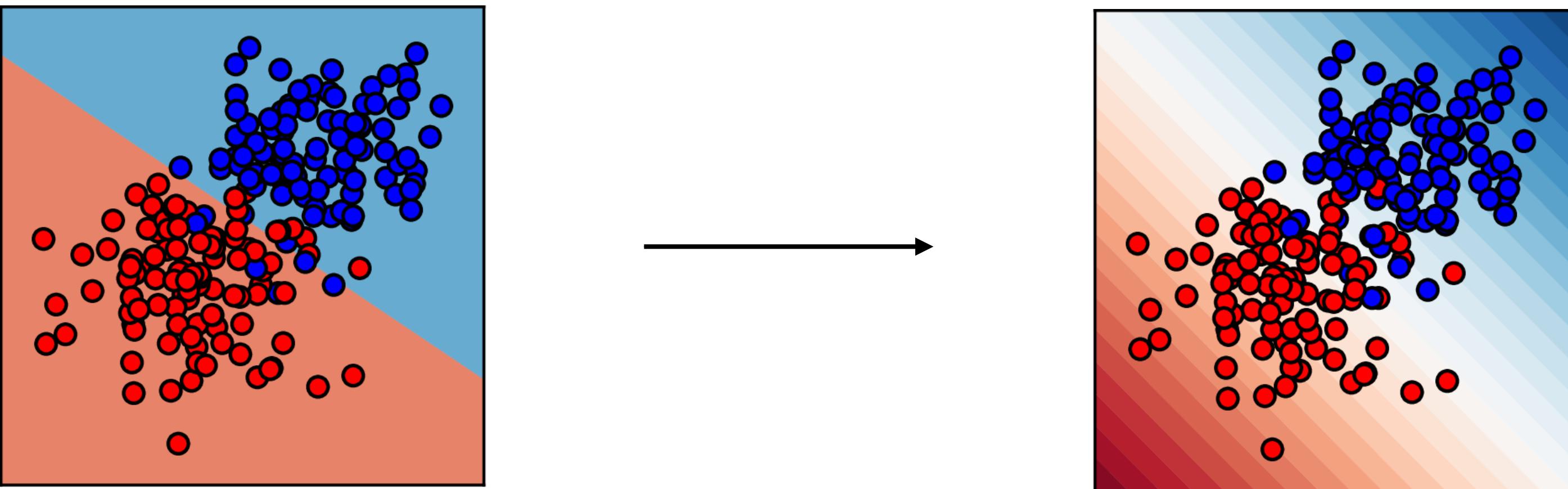
Probabilistic Learning



First, replace the *sign* function by the *differentiable sigmoid* function:

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

Probabilistic Learning



First, replace the *sign* function by the *differentiable sigmoid* function:

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

We now have the following function:

$$\hat{y}_i = f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}_i)$$

where $\hat{y}_i \in [0, 1]$ has a *probabilistic interpretation* as $\hat{P}(y_i = 1 | \mathbf{x}_i, \theta)$.

(Remember: θ is the set of *parameters* of the model, here being \mathbf{w})

Now, how do we get the **probabilistic loss function** ?

Probabilistic Learning with MLE

→ **Maximum Likelihood Estimation (MLE)**

Probabilistic Learning with MLE

→ Maximum Likelihood Estimation (MLE)

Under our interpretation, the (conditional) *likelihood* of our data $(\mathbf{x}_i, y_i)_{i=1}^n$ according to our model parameters θ is written:

$$\mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \prod_{i=1}^n \hat{P}(y_i = 1 | \mathbf{x}_i, \theta)^{y_i} \hat{P}(y_i = 0 | \mathbf{x}_i, \theta)^{1-y_i}$$

Probabilistic Learning with MLE

→ Maximum Likelihood Estimation (MLE)

Under our interpretation, the (conditional) *likelihood* of our data $(\mathbf{x}_i, y_i)_{i=1}^n$ according to our model parameters θ is written:

$$\begin{aligned}\mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) &= \prod_{i=1}^n \hat{P}(y_i = 1 | \mathbf{x}_i, \theta)^{y_i} \hat{P}(y_i = 0 | \mathbf{x}_i, \theta)^{1-y_i} \\ &= \prod_{i=1}^n f(\mathbf{x}_i, \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}\end{aligned}$$

Probabilistic Learning with MLE

→ Maximum Likelihood Estimation (MLE)

Under our interpretation, the (conditional) *likelihood* of our data $(\mathbf{x}_i, y_i)_{i=1}^n$ according to our model parameters θ is written:

$$\begin{aligned}\mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) &= \prod_{i=1}^n \hat{P}(y_i = 1 | \mathbf{x}_i, \theta)^{y_i} \hat{P}(y_i = 0 | \mathbf{x}_i, \theta)^{1-y_i} \\ &= \prod_{i=1}^n f(\mathbf{x}_i, \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}\end{aligned}$$

You can compare this to the likelihood of success of a sequence of n Bernouilli trials with a varying probability p_i :

$$\prod_{i=1}^n p_i^{y_i} (1 - p_i)^{1-y_i}$$

but the **continuity** of our function f will force similar inputs \mathbf{x}_i to have similar output probabilities !

Probabilistic Learning with MLE

→ We are looking for the θ that will **maximize this likelihood**:

$$\hat{\theta} = \operatorname{argmax}_{\theta} \mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \operatorname{argmax}_{\theta} \prod_{i=1}^n f(\mathbf{x}_i, \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}$$

Probabilistic Learning with MLE

→ We are looking for the θ that will **maximize this likelihood**:

$$\hat{\theta} = \operatorname{argmax}_{\theta} \mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \operatorname{argmax}_{\theta} \prod_{i=1}^n f(\mathbf{x}_i, \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}$$

The equivalent but more convenient optimization problem is to maximize the **log-likelihood**:

$$\log \mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \sum_{i=1}^n y_i \log f(\mathbf{x}_i, \mathbf{w}) + (1 - y_i) \log(1 - f(\mathbf{x}_i, \mathbf{w}))$$

Probabilistic Learning with MLE

→ We are looking for the θ that will **maximize this likelihood**:

$$\hat{\theta} = \operatorname{argmax}_{\theta} \mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \operatorname{argmax}_{\theta} \prod_{i=1}^n f(\mathbf{x}_i, \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}$$

The equivalent but more convenient optimization problem is to maximize the **log-likelihood**:

$$\log \mathcal{L}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = \sum_{i=1}^n y_i \log f(\mathbf{x}_i, \mathbf{w}) + (1 - y_i) \log(1 - f(\mathbf{x}_i, \mathbf{w}))$$

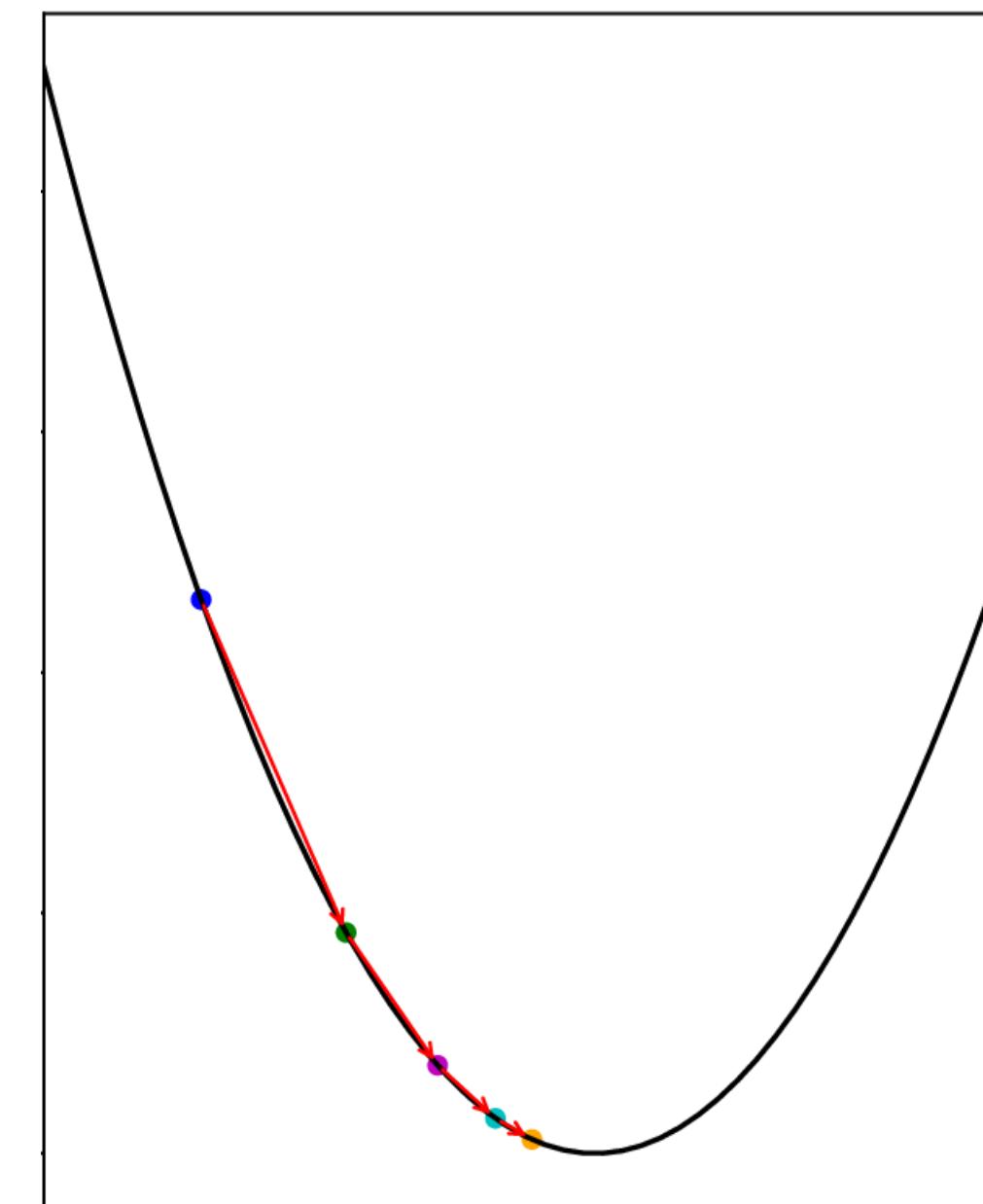
We note the *negative log-likelihood* $l_{MLE}(\theta) = -\log \mathcal{L}(\theta)$ as the loss function to minimize.

It corresponds to the *Kullback-Leibler divergence* between the data distribution \hat{p}_{data} and the model distribution $p_\theta = f(., \mathbf{w})$:

$$l_{MLE}(\theta) = D_{KL}(\hat{p}_{data}, p_\theta) + C$$

Gradient-based Learning

→ We are looking for the parameters θ^* that will minimize the negative log-likelihood.

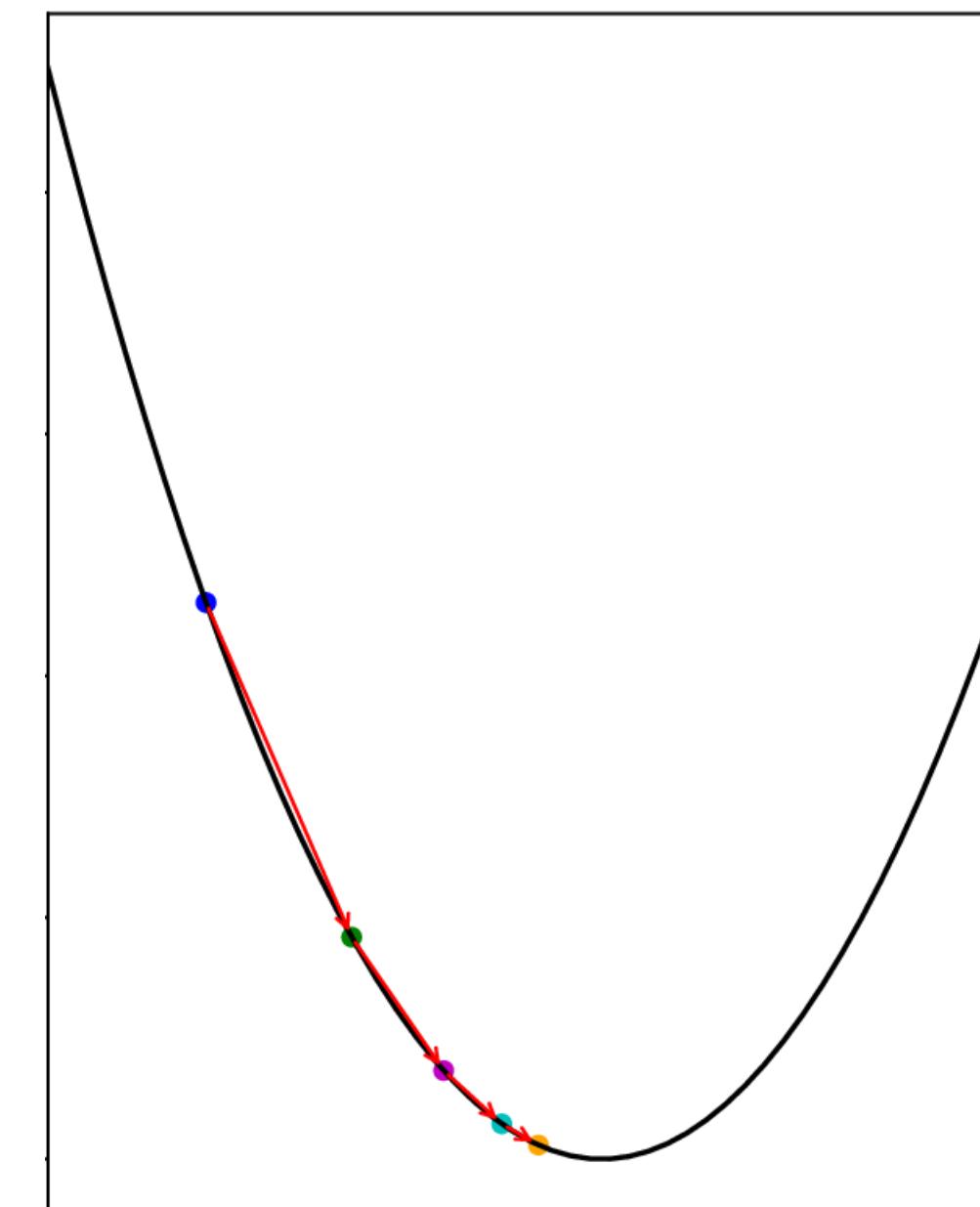


Gradient-based Learning

→ We are looking for the parameters θ^* that will minimize the negative log-likelihood.

We use the **gradient descent**, which is based on the following update rule:

$$\theta^{t+1} = \theta^t + \epsilon \nabla_{\theta} l(\theta)$$



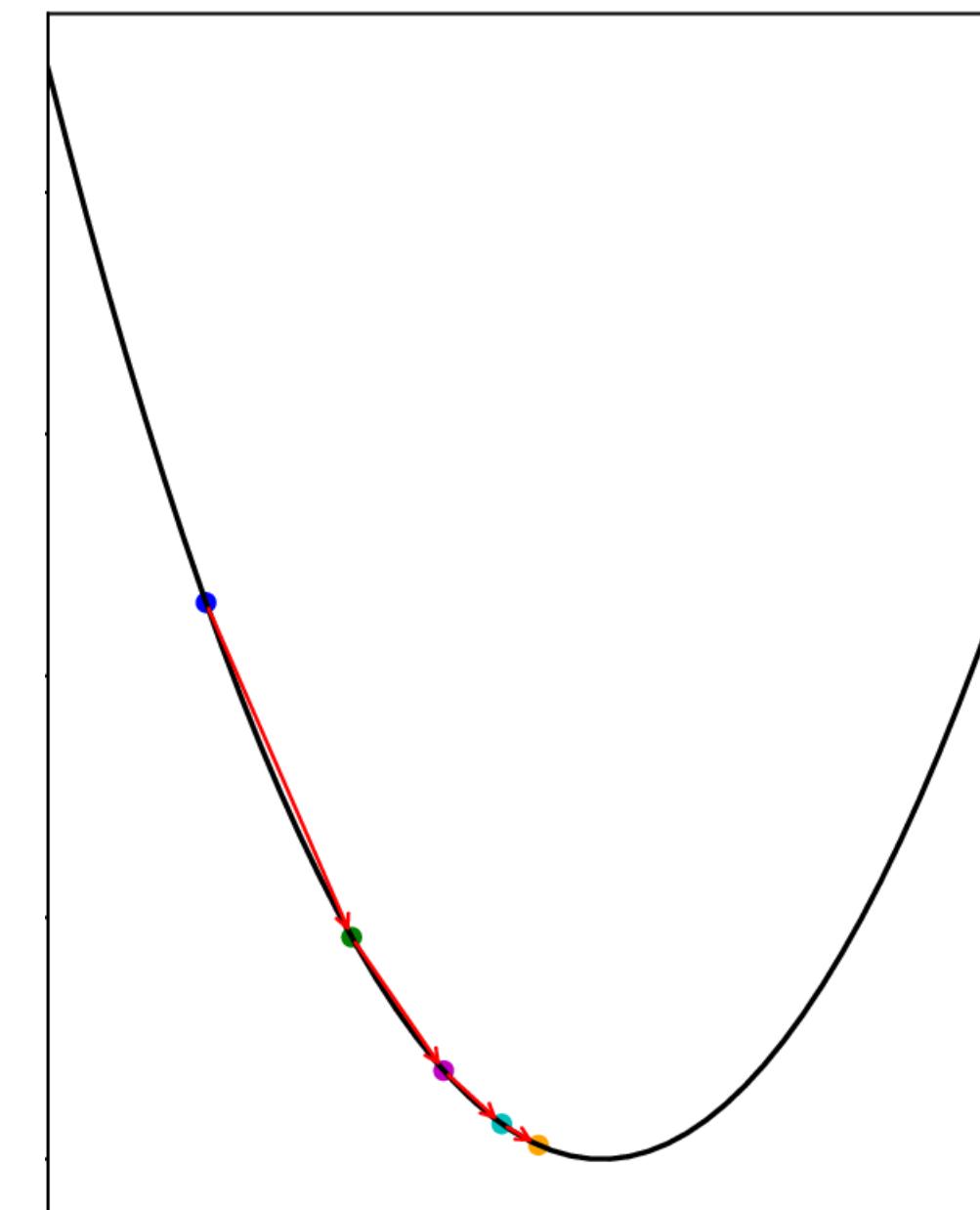
Gradient-based Learning

→ We are looking for the parameters θ^* that will minimize the negative log-likelihood.

We use the **gradient descent**, which is based on the following update rule:

$$\theta^{t+1} = \theta^t + \epsilon \nabla_{\theta} l(\theta)$$

The learning rate ϵ dictates how fast we follow the direction of steepest descent



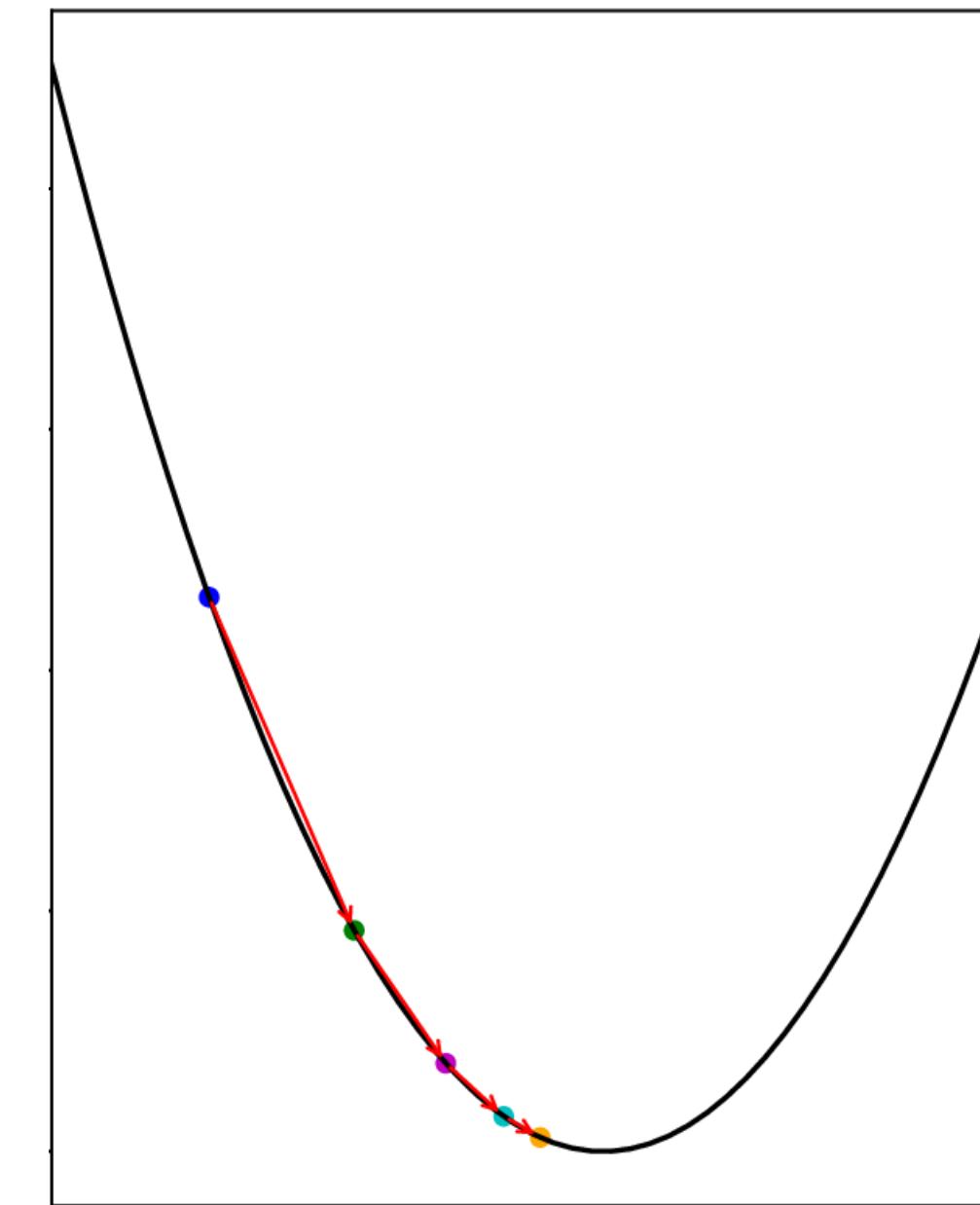
Gradient-based Learning

→ We are looking for the parameters θ^* that will minimize the negative log-likelihood.

We use the **gradient descent**, which is based on the following update rule:

$$\theta^{t+1} = \theta^t + \epsilon \nabla_{\theta} l(\theta)$$

The learning rate ϵ dictates how fast we follow the direction of steepest descent



Computing the gradient on all data points is expensive: in practice, we use only a small batch of randomly selected points for an update.

→ This is **Stochastic Gradient Descent (SGD)**.

Gradient-based Learning

→ Coming back to our model: the gradient of the loss $l_{MLE}(\theta)$.

The derivative with respect to the j^{th} component of \mathbf{w} , w^j is:

$$\frac{\delta l_{MLE}}{\delta w^j} = - \sum_{i=1}^n [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$$

Gradient-based Learning

→ Coming back to our model: the gradient of the loss $l_{MLE}(\theta)$.

The derivative with respect to the j^{th} component of \mathbf{w} , w^j is:

$$\frac{\delta l_{MLE}}{\delta w^j} = - \sum_{i=1}^n [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$$

Gradient-based learning procedure:

Gradient-based Learning

→ Coming back to our model: the gradient of the loss $l_{MLE}(\theta)$.

The derivative with respect to the j^{th} component of \mathbf{w} , w^j is:

$$\frac{\delta l_{MLE}}{\delta w^j} = - \sum_{i=1}^n [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$$

Gradient-based learning procedure:

- Start with $t = 1$, \mathbf{w}_t initialized appropriately;

Number of iterations n_{iter} , Learning rate $\epsilon > 0$, and $\delta > 0$

Examples $(\mathbf{x}_i, y_i)_{i=1}^n$;

Gradient-based Learning

→ Coming back to our model: the gradient of the loss $l_{MLE}(\theta)$.

The derivative with respect to the j^{th} component of \mathbf{w} , w^j is:

$$\frac{\delta l_{MLE}}{\delta w^j} = - \sum_{i=1}^n [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$$

Gradient-based learning procedure:

- Start with $t = 1$, \mathbf{w}_t initialized appropriately;

Number of iterations n_{iter} , Learning rate $\epsilon > 0$, and $\delta > 0$

Examples $(\mathbf{x}_i, y_i)_{i=1}^n$;

- While $||l_{ML}(\mathbf{w}_{t+1}) - l_{ML}(\mathbf{w}_t)|| > \delta$ and $t > n_{iter}$:
 - For $j \in [1, d]$ (on a set of d randomly sampled examples in the data):
 - Do $w_{t+1}^j = w_t^j - \epsilon \sum_{i=1}^d [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$

Gradient-based Learning

→ Coming back to our model: the gradient of the loss $l_{MLE}(\theta)$.

The derivative with respect to the j^{th} component of \mathbf{w} , w^j is:

$$\frac{\delta l_{MLE}}{\delta w^j} = - \sum_{i=1}^n [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$$

Gradient-based learning procedure:

- Start with $t = 1$, \mathbf{w}_t initialized appropriately;

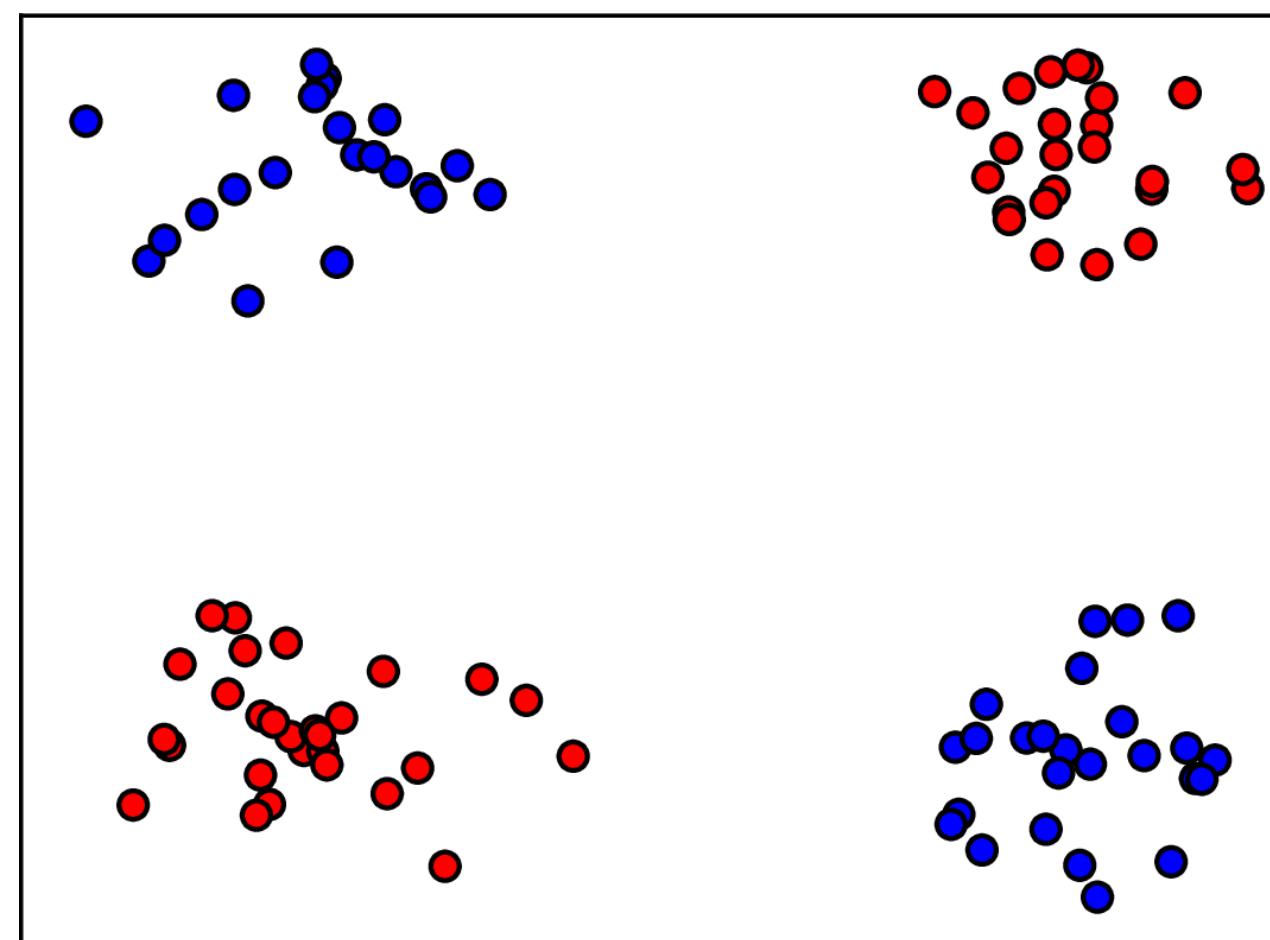
Number of iterations n_{iter} , Learning rate $\epsilon > 0$, and $\delta > 0$

Examples $(\mathbf{x}_i, y_i)_{i=1}^n$;

- While $||l_{ML}(\mathbf{w}_{t+1}) - l_{ML}(\mathbf{w}_t)|| > \delta$ and $t > n_{iter}$:
 - For $j \in [1, d]$ (on a set of d randomly sampled examples in the data):
 - Do $w_{t+1}^j = w_t^j - \epsilon \sum_{i=1}^d [y_i - f(\mathbf{x}_i, \mathbf{w})] x_i^j$
- We now have a well defined **probabilistic** model for binary classification, and **an efficient way to learn it !**

Non-linear Learning

→ Reminder: the **XOR** issue !



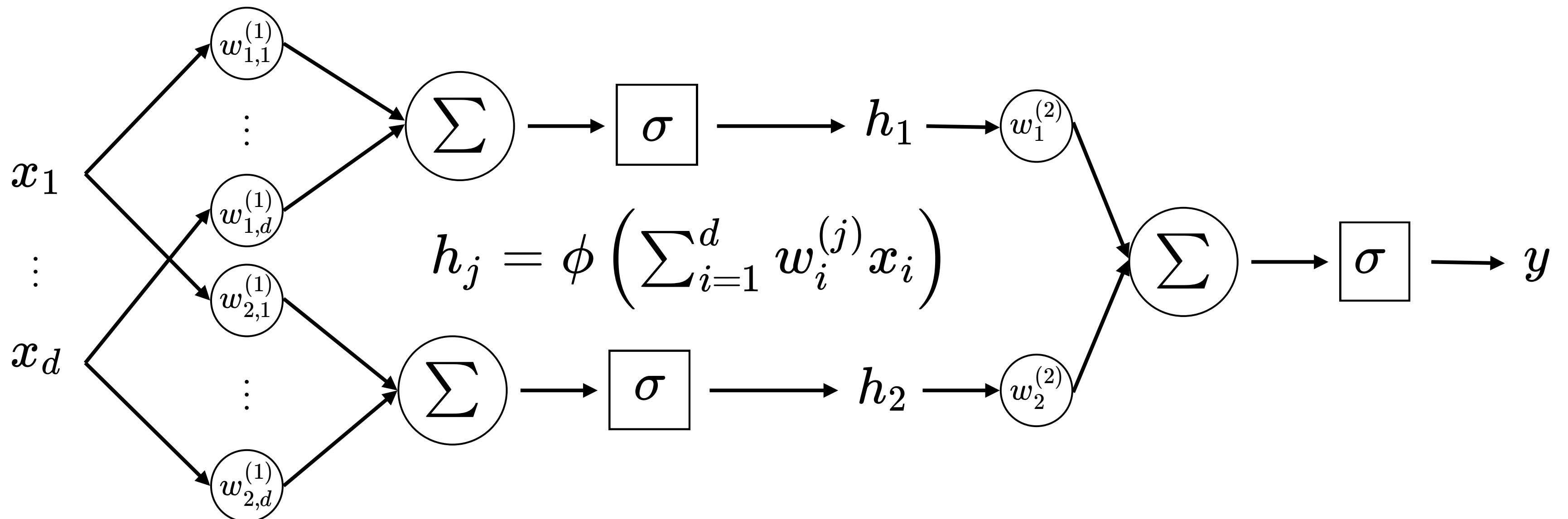
Non-linear Learning

→ Reminder: the **XOR** issue !

Non-linear Learning

→ Reminder: the **XOR** issue !

Solution: *combining several outputs as inputs to a new neuron, forming a Multi-Layer Perceptron (MLP)*:



Multi-Layer Perceptron

Intuitions:

- Features are mapped into an *intermediate space*: the model uses an **internal representation** that's the input for the output neuron

Multi-Layer Perceptron

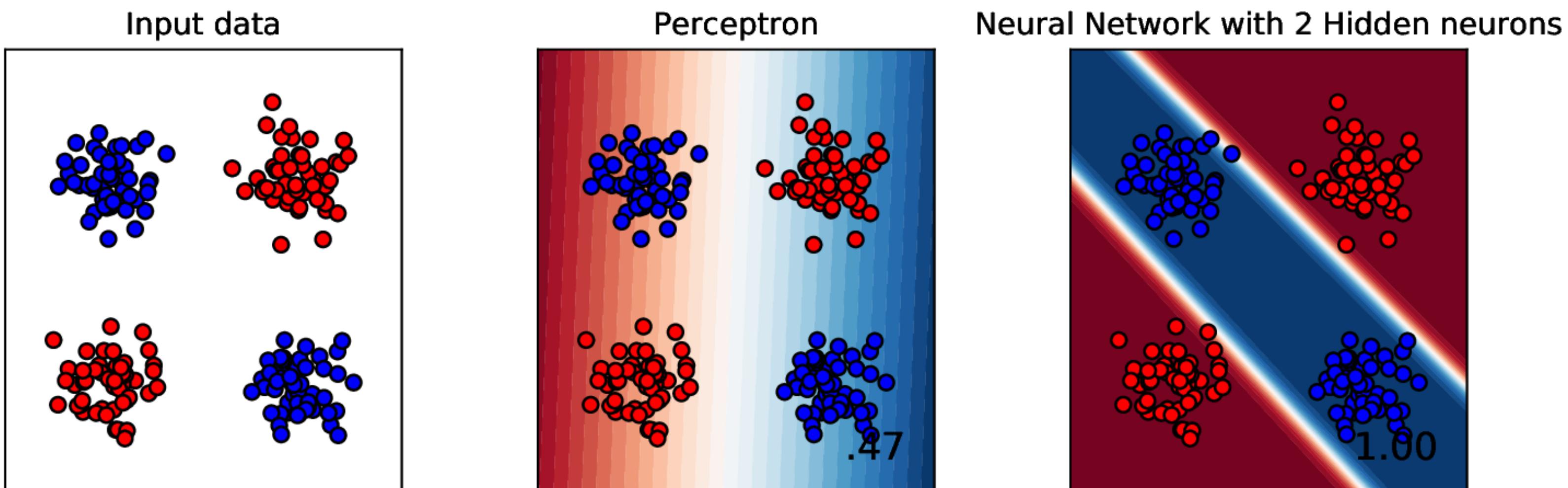
Intuitions:

- Features are mapped into an *intermediate space*: the model uses an **internal representation** that's the input for the output neuron
- The input representation \mathbf{x} is transformed/compressed in a new representation \mathbf{h}

Multi-Layer Perceptron

Intuitions:

- Features are mapped into an *intermediate space*: the model uses an **internal representation** that's the input for the output neuron
- The input representation \mathbf{x} is transformed/compressed in a new representation \mathbf{h}



Multi-Layer Perceptron: Architecture

Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

Multi-Layer Perceptron: Architecture

Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

$$h_1^{(1)} = \phi^{(1)} \left(\mathbf{w}_1^{(1)T} \mathbf{x} + b_1^{(1)} \right)$$

$$h_2^{(1)} = \phi^{(1)} \left(\mathbf{w}_2^{(1)T} \mathbf{x} + b_2^{(1)} \right)$$

⋮

$$h_{d_h}^{(1)} = \phi^{(1)} \left(\mathbf{w}_{d_h}^{(1)T} \mathbf{x} + b_{d_h}^{(1)} \right)$$

Multi-Layer Perceptron: Architecture

Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

.... Obtaining a **fully connected** layer:

Multi-Layer Perceptron: Architecture

Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

.... Obtaining a **fully connected** layer:

$$\mathbf{h}^{(1)} = \sigma \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

Multi-Layer Perceptron: Architecture

Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

.... Obtaining a **fully connected** layer:

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$\phi^{(1)}$ can be any **non-linear activation function**

Multi-Layer Perceptron: Architecture

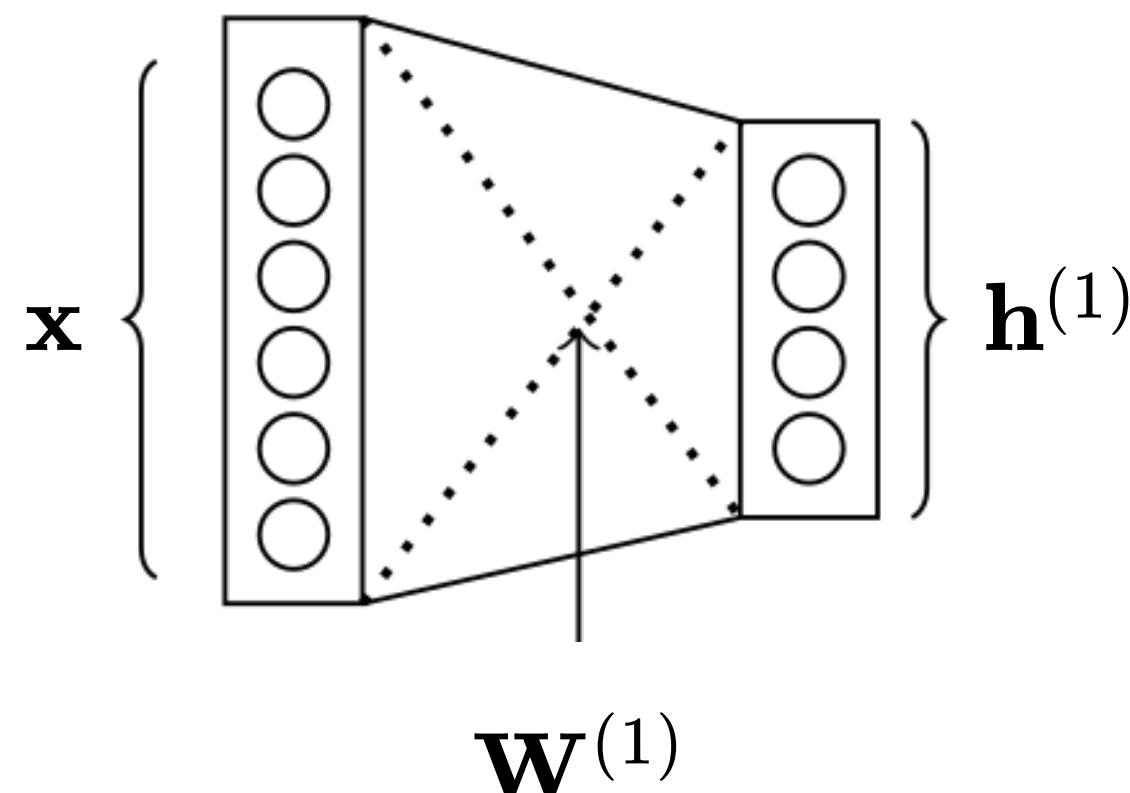
Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

.... Obtaining a **fully connected** layer:

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$\phi^{(1)}$ can be any **non-linear activation function**



Multi-Layer Perceptron: Architecture

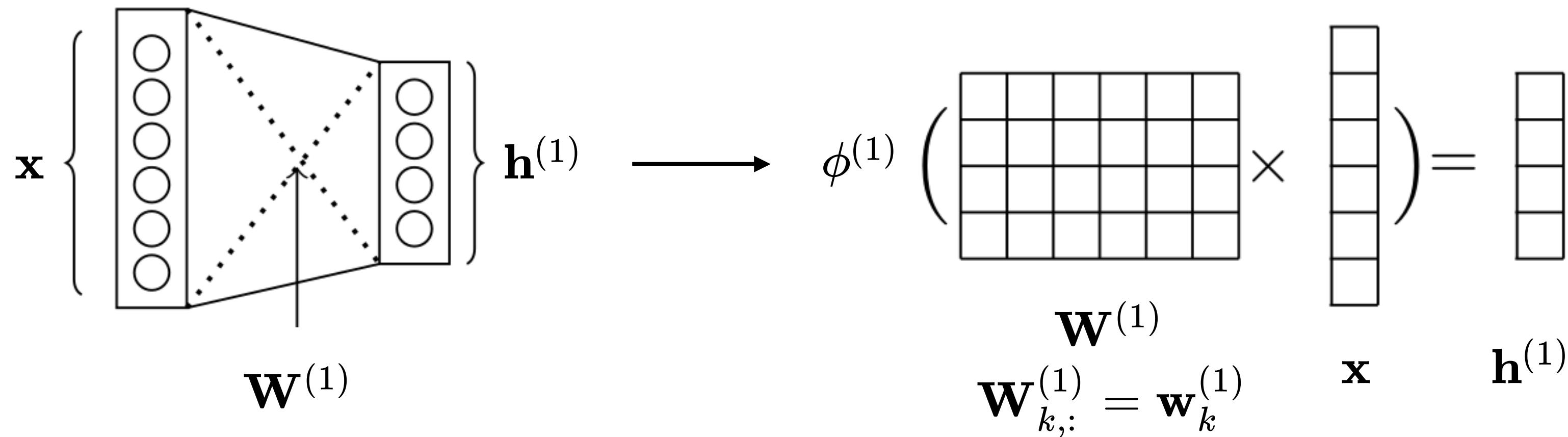
Usually, neurons are grouped into **layers**:

→ We *vectorize* the neurons

.... Obtaining a **fully connected** layer:

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$\phi^{(1)}$ can be any **non-linear activation function**



Multi-Layer Perceptron: Architecture

Layers are organized in a **chain structure**:

→ Each layer is a function of the previous one

Multi-Layer Perceptron: Architecture

Layers are organized in a **chain structure**:

→ Each layer is a function of the previous one

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

Multi-Layer Perceptron: Architecture

Layers are organized in a **chain structure**:

→ Each layer is a function of the previous one

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$$\mathbf{h}^{(2)} = \phi^{(2)} \left(\mathbf{W}^{(2)T} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right)$$

Multi-Layer Perceptron: Architecture

Layers are organized in a **chain structure**:

→ Each layer is a function of the previous one

$$\mathbf{h}^{(1)} = \phi^{(1)} \left(\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$$\mathbf{h}^{(2)} = \phi^{(2)} \left(\mathbf{W}^{(2)T} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right)$$

And so on, until the layer l :

$$\mathbf{h}^{(l)} = \phi^{(l)} \left(\mathbf{W}^{(l)T} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \right)$$

Multi-Layer Perceptron: Activations

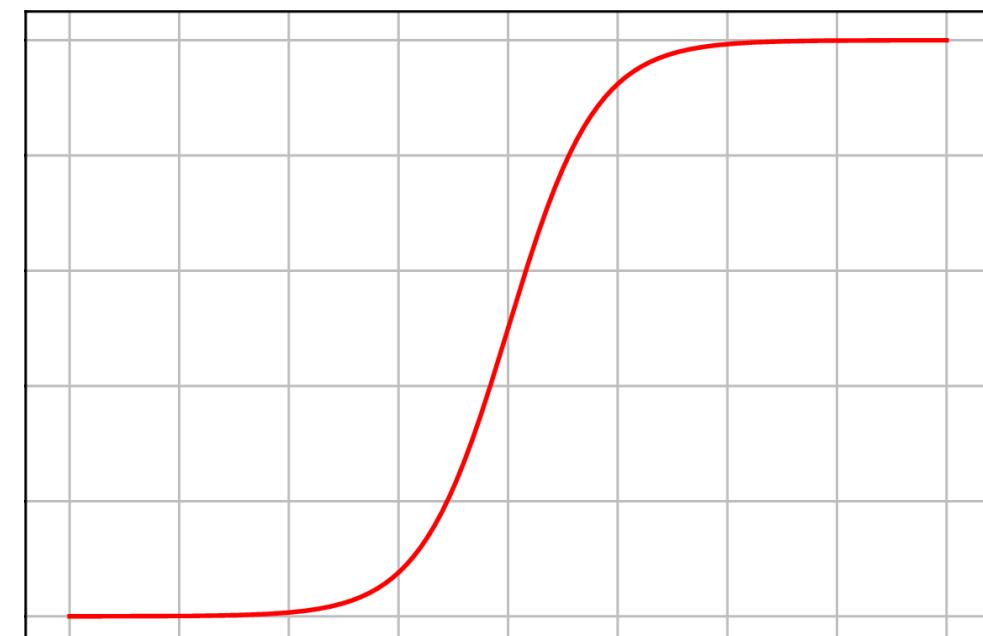
Activations for the hidden layers have various (dis)advantages:

→ Which one to choose depends on the data and task !

Multi-Layer Perceptron: Activations

Activations for the hidden layers have various (dis)advantages:

→ Which one to choose depends on the data and task !

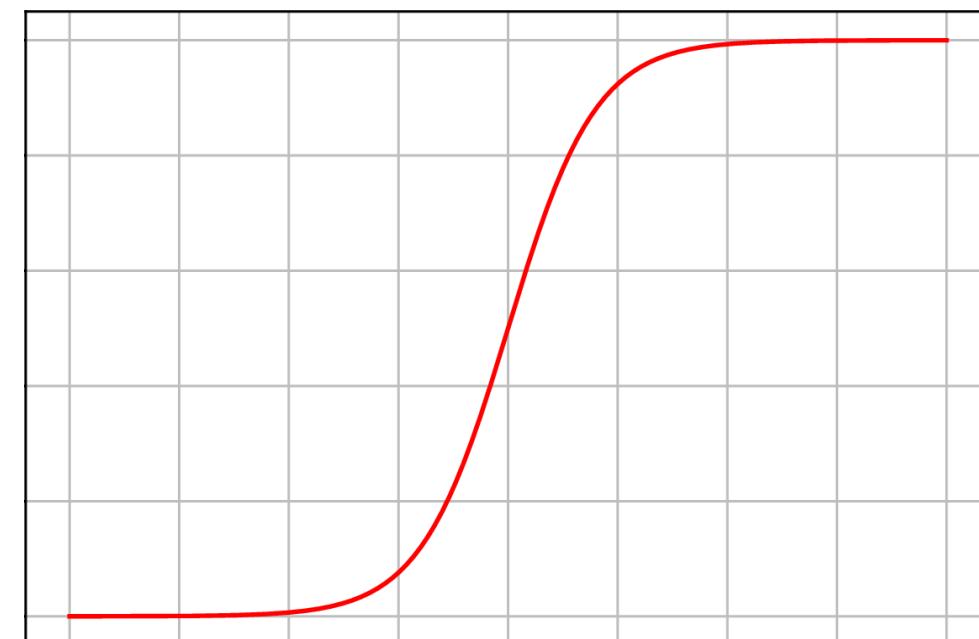


$$\sigma(x) = \frac{1}{1+e^{-x}}$$

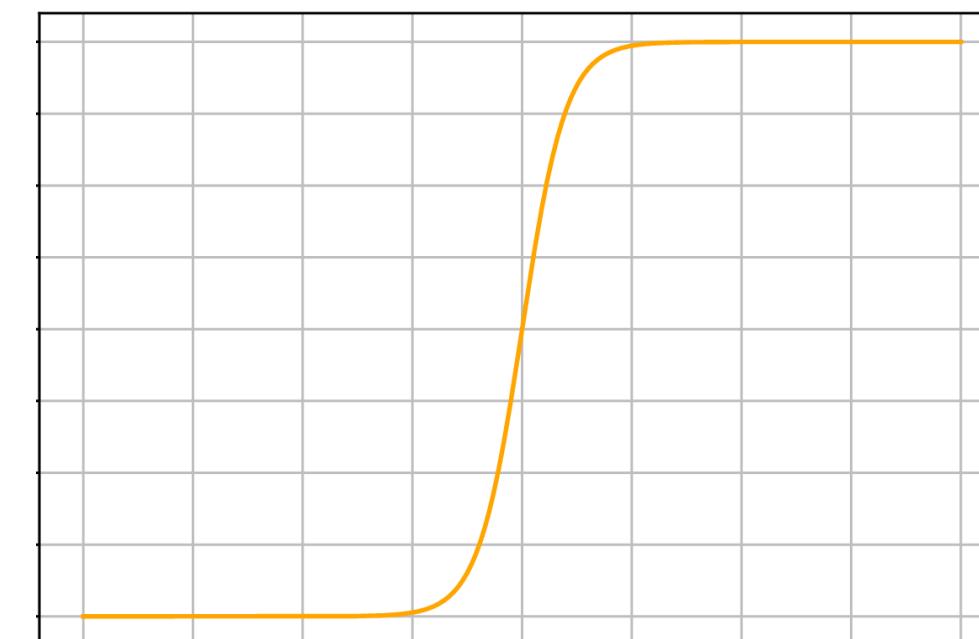
Multi-Layer Perceptron: Activations

Activations for the hidden layers have various (dis)advantages:

→ Which one to choose depends on the data and task !



$$\sigma(x) = \frac{1}{1+e^{-x}}$$



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Multi-Layer Perceptron: Multi-outputs

We can also have an arbitrary number of output neurons: for multi-class classification with K classes, we have $y_i \in [1, K]$.

Equivalently, we write the corresponding output vector $\mathbf{o}_i \in \mathbb{R}^K$:

$$y_i = k \iff \mathbf{o}_i = [0, \dots, 1, \dots, 0]$$

\uparrow
 k^{th} component

Multi-Layer Perceptron: Multi-outputs

We can also have an arbitrary number of output neurons: for multi-class classification with K classes, we have $y_i \in [1, K]$.

Equivalently, we write the corresponding output vector $\mathbf{o}_i \in \mathbb{R}^K$:

$$y_i = k \iff \mathbf{o}_i = [0, \dots, 1, \dots, 0]$$

\uparrow
 k^{th} component

→ We use the **softmax** activation function, which generalizes the sigmoid to k outputs:

$$\hat{\mathbf{o}}_i = softmax(\mathbf{h}^{(l)}) = \left[\frac{e^{\mathbf{h}^{(l),1}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l),k}}}, \dots, \frac{e^{\mathbf{h}^{(l),K}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l),k}}} \right]$$

Multi-Layer Perceptron: Multi-outputs

We can also have an arbitrary number of output neurons: for multi-class classification with K classes, we have $y_i \in [1, K]$.

Equivalently, we write the corresponding output vector $\mathbf{o}_i \in \mathbb{R}^K$:

$$y_i = k \iff \mathbf{o}_i = [0, \dots, 1, \dots, 0]$$

\uparrow
 k^{th} component

→ We use the **softmax** activation function, which generalizes the sigmoid to k outputs:

$$\hat{\mathbf{o}}_i = softmax(\mathbf{h}^{(l)}) = \left[\frac{e^{\mathbf{h}^{(l),1}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l),k}}}, \dots, \frac{e^{\mathbf{h}^{(l),K}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l),k}}} \right]$$

The components of $\hat{\mathbf{o}}_i$ sum to 1 and can be interpreted as probabilities over a **multinomial** distribution

Multi-Layer Perceptron: Optimization

The parameters of the MLP are $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(l)}, \mathbf{b}^{(l)})$

Multi-Layer Perceptron: Optimization

The parameters of the MLP are $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(l)}, \mathbf{b}^{(l)})$

As for the binary case, we use the negative log-likelihood as loss to minimize, following a **probabilistic interpretation of the softmax**:

$$l_{MLE}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = - \sum_{i=1}^n \log \prod_{k=1}^K \hat{P}(y_i = k | \mathbf{x}_i, \theta)^{o_i^k} = - \sum_{i=1}^n \sum_{k=1}^K o_i^k \log \hat{o}_i^k$$

Multi-Layer Perceptron: Optimization

The parameters of the MLP are $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(l)}, \mathbf{b}^{(l)})$

As for the binary case, we use the negative log-likelihood as loss to minimize, following a **probabilistic interpretation of the softmax**:

$$l_{MLE}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = - \sum_{i=1}^n \log \prod_{k=1}^K \hat{P}(y_i = k | \mathbf{x}_i, \theta)^{o_i^k} = - \sum_{i=1}^n \sum_{k=1}^K o_i^k \log \hat{o}_i^k$$

Since $o_i^k = 1 \iff y_i = k$, we end up with:

$$l_{MLE}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = - \sum_{i=1}^n \log \hat{o}_i^{y_i} = - \sum_{i=1}^n \log \left(\frac{e^{\mathbf{h}^{(l), y_i}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l), k}}} \right)$$

Multi-Layer Perceptron: Optimization

The parameters of the MLP are $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(l)}, \mathbf{b}^{(l)})$

As for the binary case, we use the negative log-likelihood as loss to minimize, following a **probabilistic interpretation of the softmax**:

$$l_{MLE}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = - \sum_{i=1}^n \log \prod_{k=1}^K \hat{P}(y_i = k | \mathbf{x}_i, \theta)^{o_i^k} = - \sum_{i=1}^n \sum_{k=1}^K o_i^k \log \hat{o}_i^k$$

Since $o_i^k = 1 \iff y_i = k$, we end up with:

$$l_{MLE}((\mathbf{x}_i, y_i)_{i=1}^n | \theta) = - \sum_{i=1}^n \log \hat{o}_i^{y_i} = - \sum_{i=1}^n \log \left(\frac{e^{\mathbf{h}^{(l), y_i}}}{\sum_{k=1}^K e^{\mathbf{h}^{(l), k}}} \right)$$

But here, it is not as easy to compute the derivative $\frac{\delta l_{MLE}}{\delta w}$ for all parameters $w \in \theta$...

→ To do so, we will use **Backpropagation**

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

→ It is based on the **chain rule**:

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y)$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

→ It is based on the **chain rule**:

In the scalar case:
 $y = g(x)$ and $z = f(g(x)) = f(y)$

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

→ It is based on the **chain rule**:

In the scalar case:
 $y = g(x)$ and $z = f(g(x)) = f(y)$

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

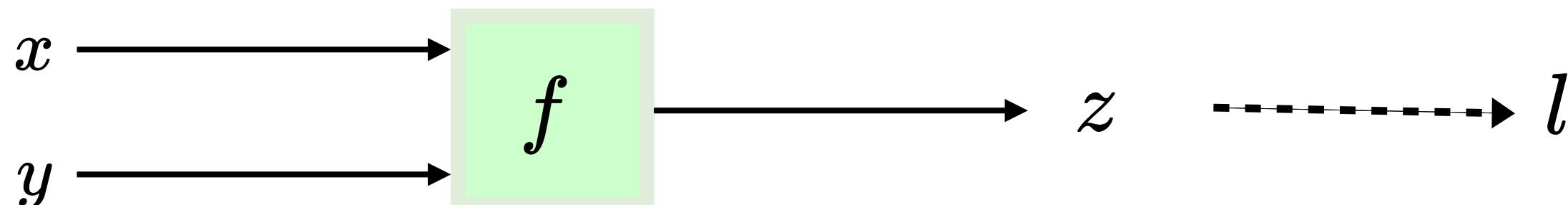
→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)



$$z = f(x, y)$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

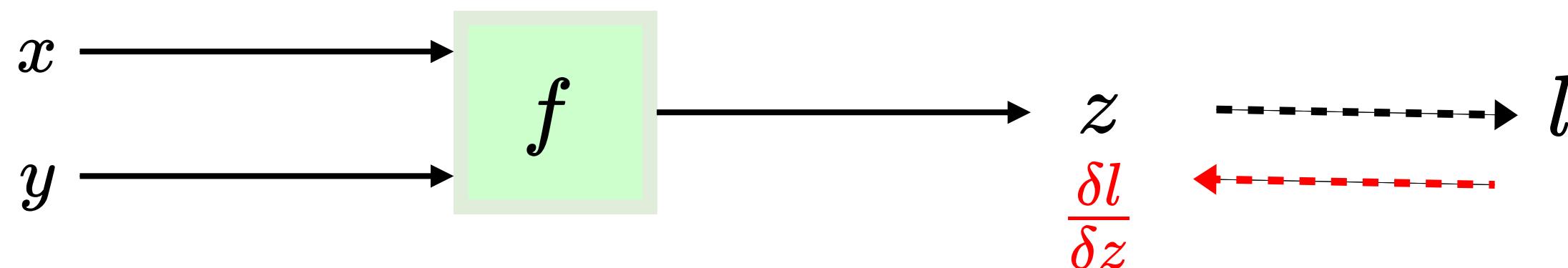
→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)



$$z = f(x, y)$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

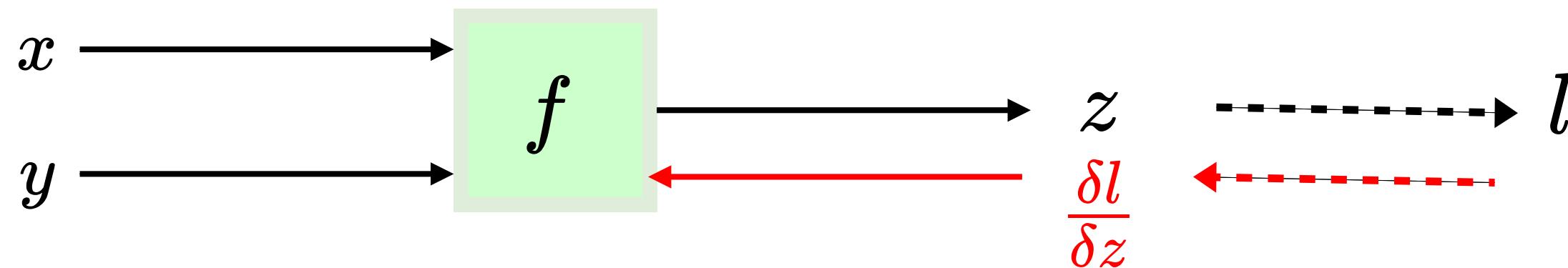
→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)



$$z = f(x, y)$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

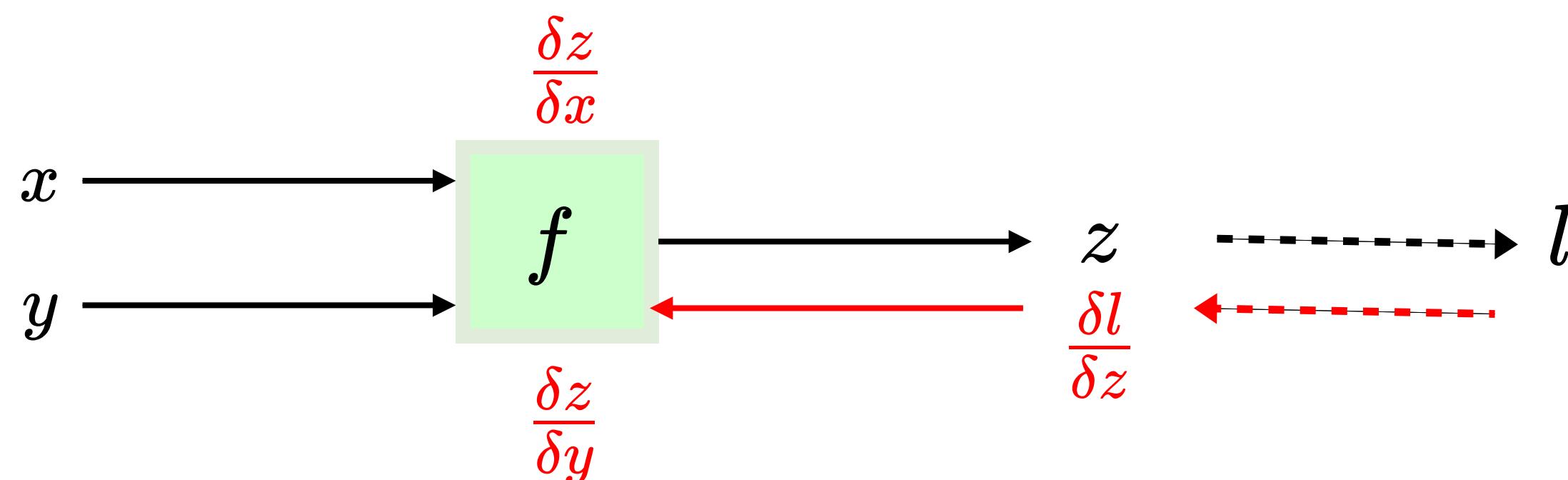
→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)



$$z = f(x, y)$$

MLP: Backpropagation

Backpropagation was proposed by Rumelhart (*Learning representations by back-propagating errors*, 1986)

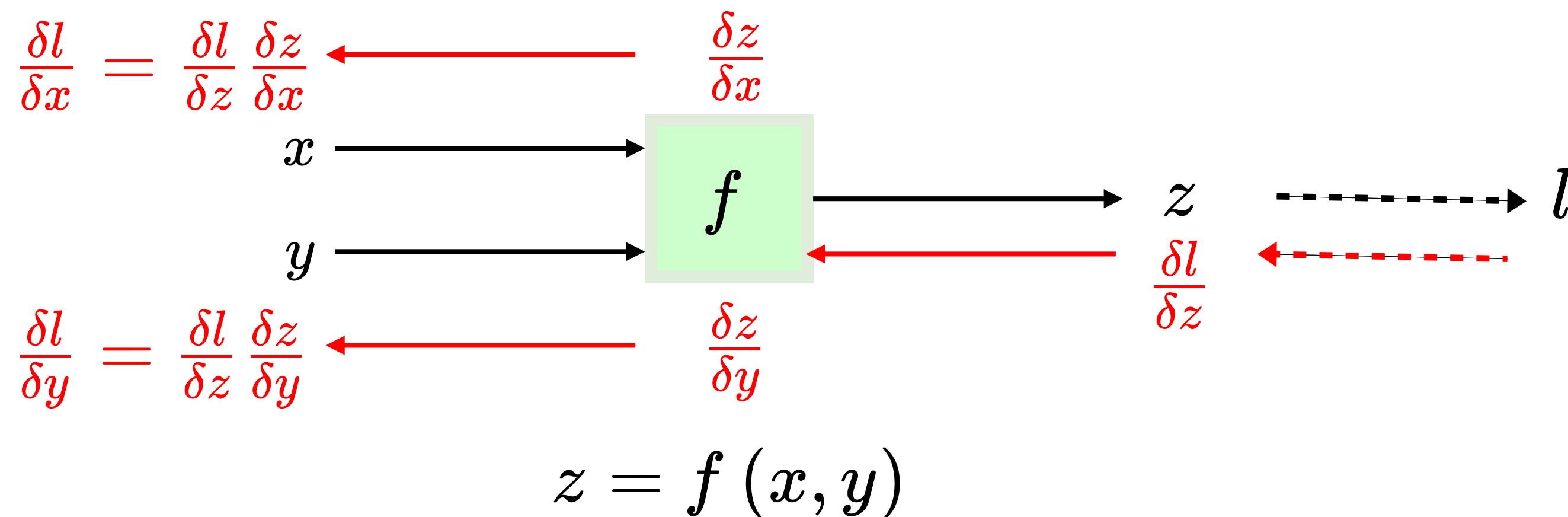
→ It is based on the **chain rule**:

In the scalar case:

$$y = g(x) \text{ and } z = f(g(x)) = f(y) \quad \frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \frac{\delta y}{\delta x}$$

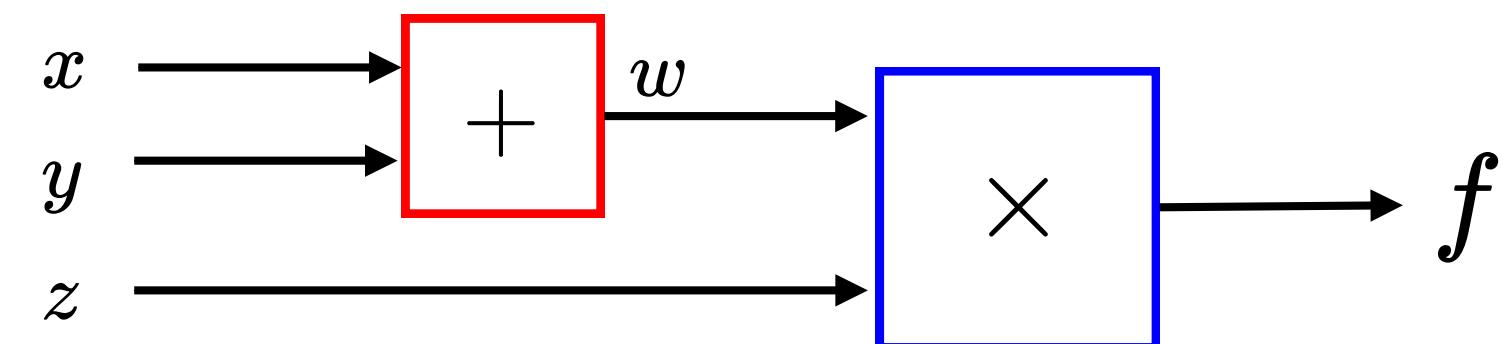
→ Allows to **recursively** compute gradients for all the parameters in the network from a **computational graph**

(But parameters are vectors and tensors: it quickly becomes complicated)



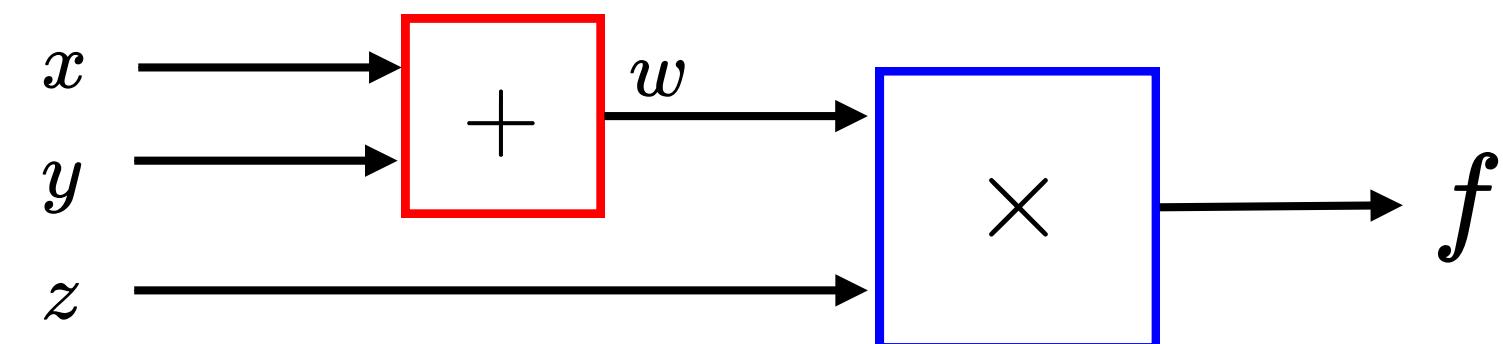
Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$



Backpropagation : simple example

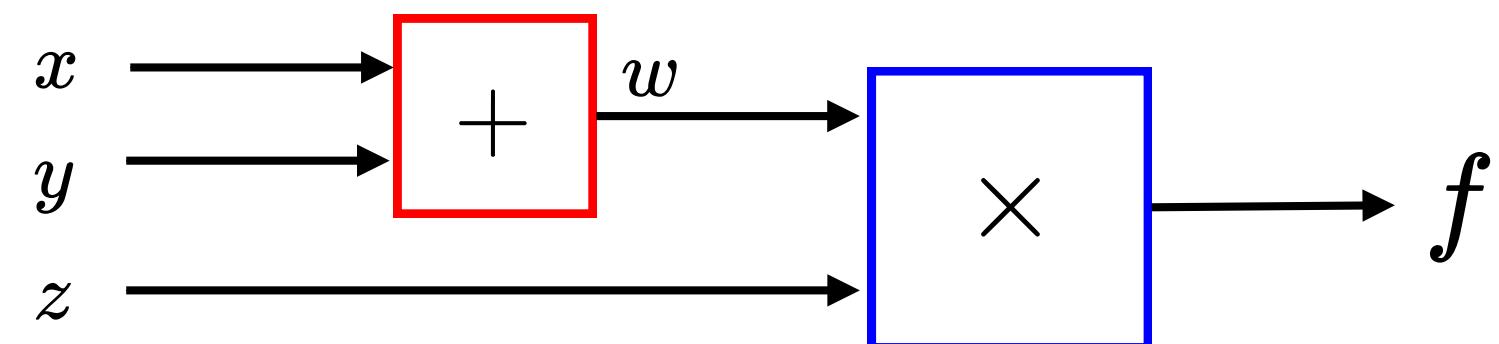
$$f(x, y, z) = (x + y)z$$



$$w = x + y$$

Backpropagation : simple example

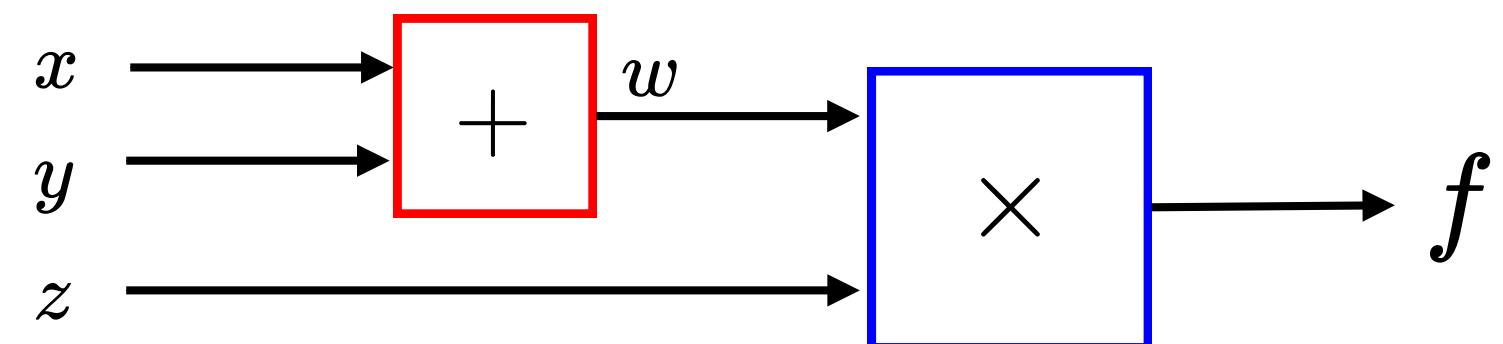
$$f(x, y, z) = (x + y)z$$



$$w = x + y \quad \rightarrow \quad \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$

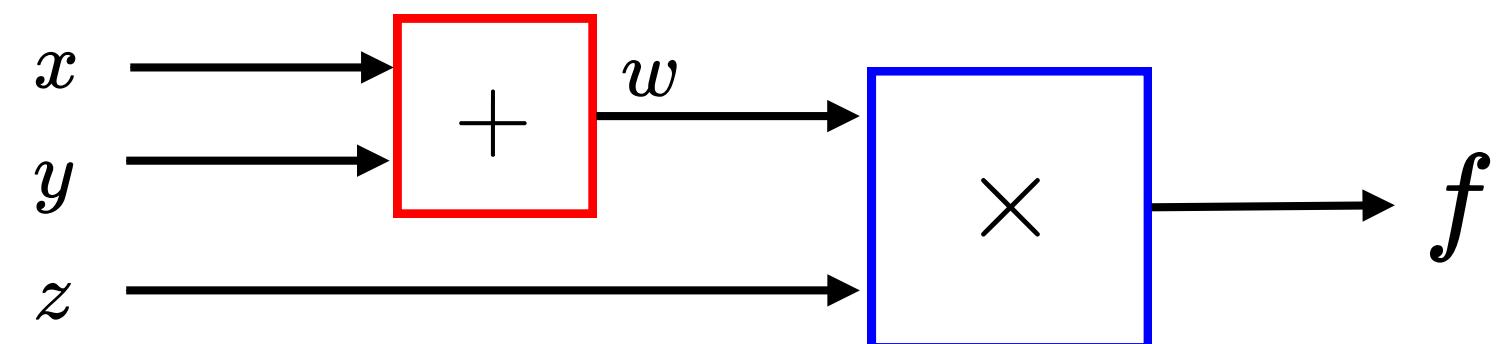


$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

$$f = wz$$

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$

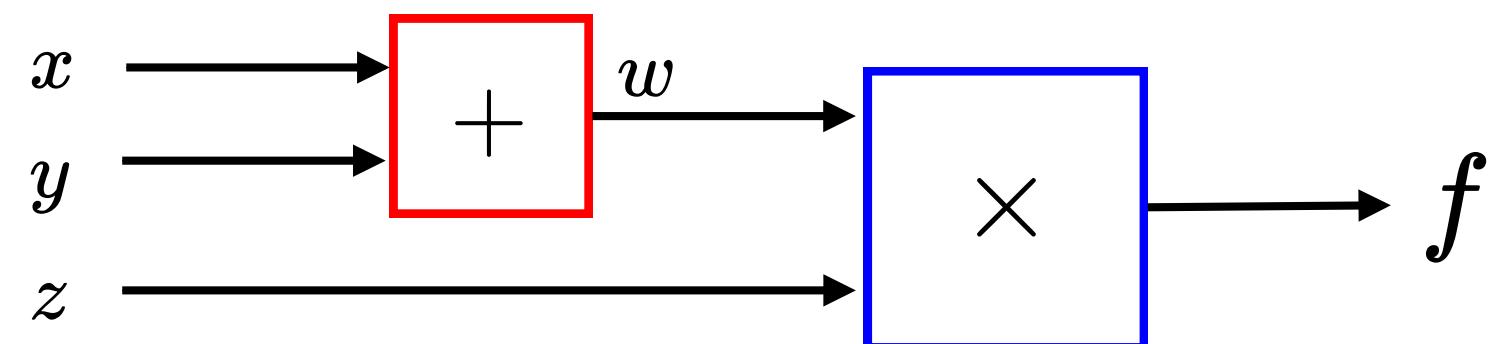


$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

$$f = wz \rightarrow \frac{\delta f}{\delta w} = z \quad \frac{\delta f}{\delta z} = w$$

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$



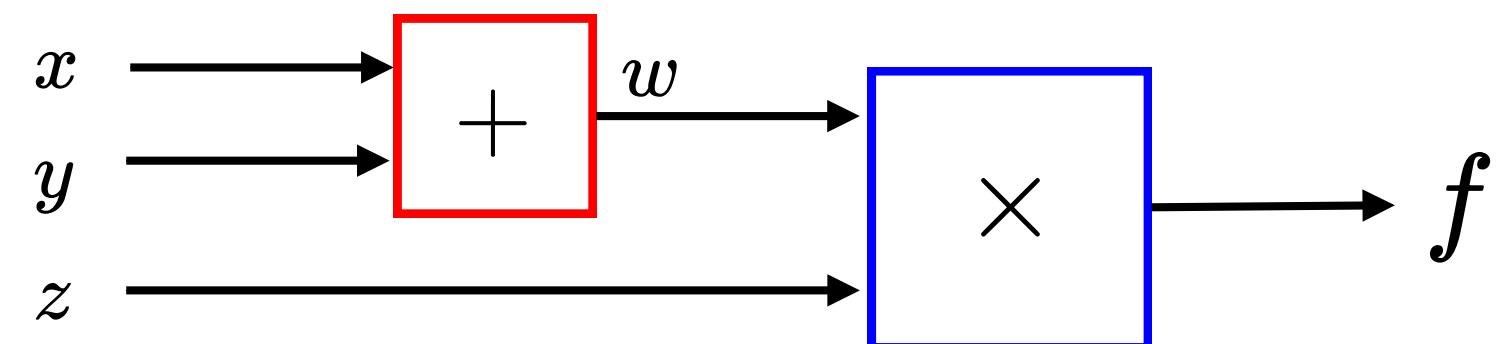
$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

$$f = wz \rightarrow \frac{\delta f}{\delta w} = z \quad \frac{\delta f}{\delta z} = w$$

What are the gradients
 $\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}$?

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$



$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

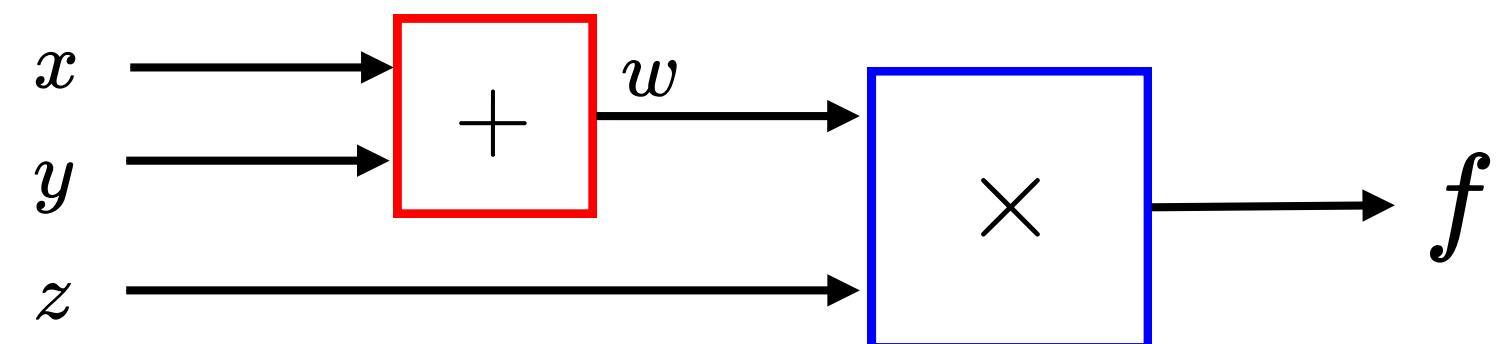
$$f = wz \rightarrow \frac{\delta f}{\delta w} = z \quad \frac{\delta f}{\delta z} = w$$

What are the gradients
 $\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}$?

→ **Forward:** with $x = -2, y = 5, z = -4$: → $w = 3$ and $f = -12$

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$



$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

$$f = wz \rightarrow \frac{\delta f}{\delta w} = z \quad \frac{\delta f}{\delta z} = w$$

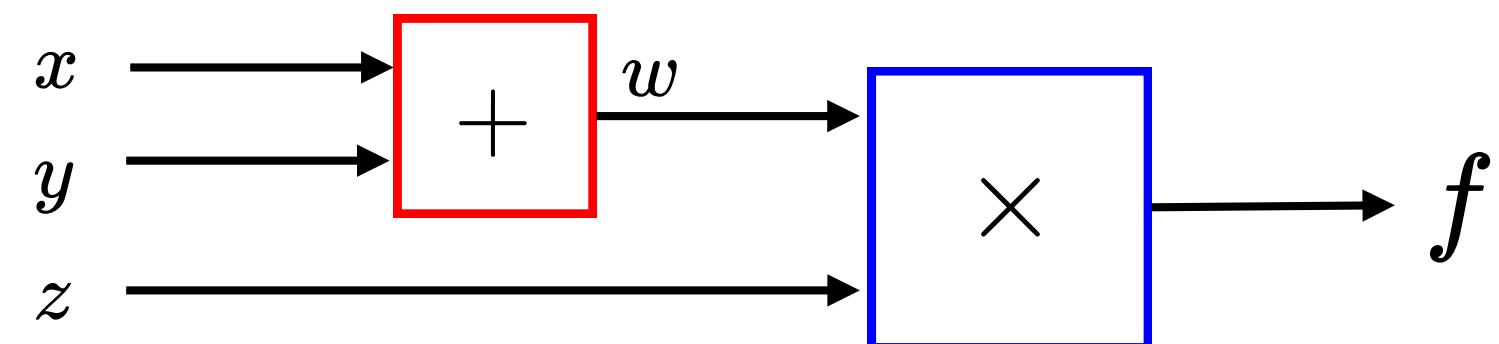
What are the gradients
 $\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}$?

→ **Forward:** with $x = -2, y = 5, z = -4$: → $w = 3$ and $f = -12$

→ **Backward:** $\frac{\delta f}{\delta z} = w = 3$

Backpropagation : simple example

$$f(x, y, z) = (x + y)z$$



$$w = x + y \rightarrow \frac{\delta w}{\delta x} = 1 \quad \frac{\delta w}{\delta y} = 1$$

What are the gradients
 $\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}$?

$$f = wz \rightarrow \frac{\delta f}{\delta w} = z \quad \frac{\delta f}{\delta z} = w$$

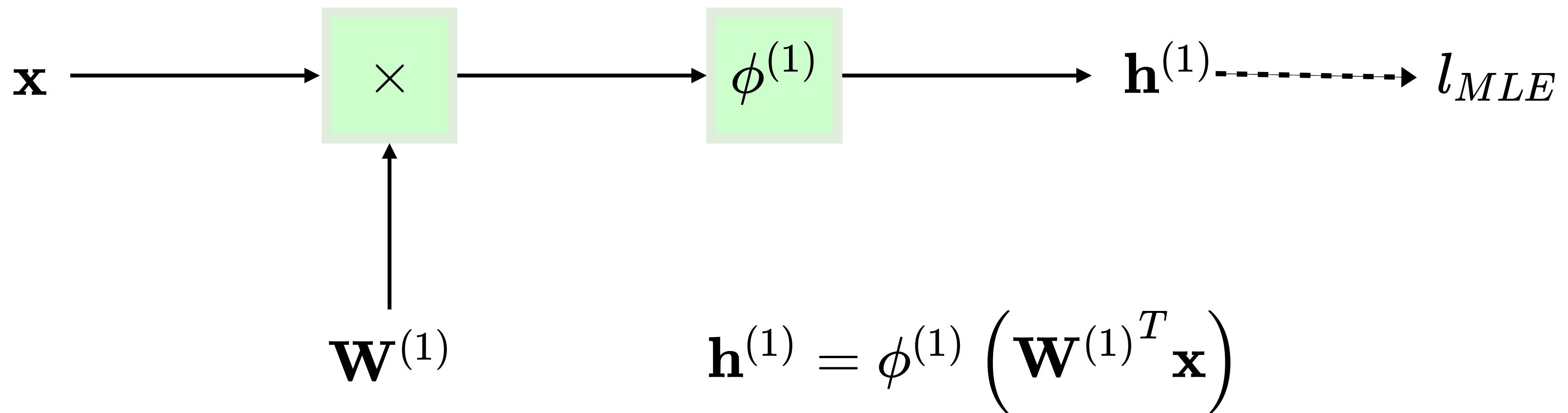
→ **Forward:** with $x = -2, y = 5, z = -4$: → $w = 3$ and $f = -12$

→ **Backward:** $\frac{\delta f}{\delta z} = w = 3$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta w} \frac{\delta w}{\delta x} = z \times 1 = -4$$

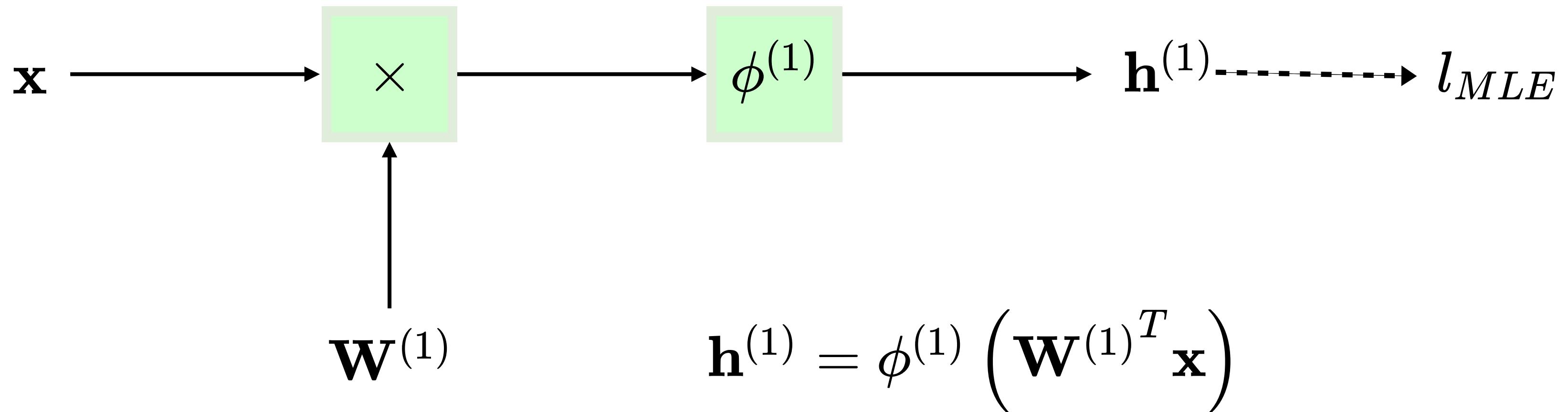
$$\frac{\delta f}{\delta y} = \frac{\delta f}{\delta x} = -4$$

MLP: Computational graph



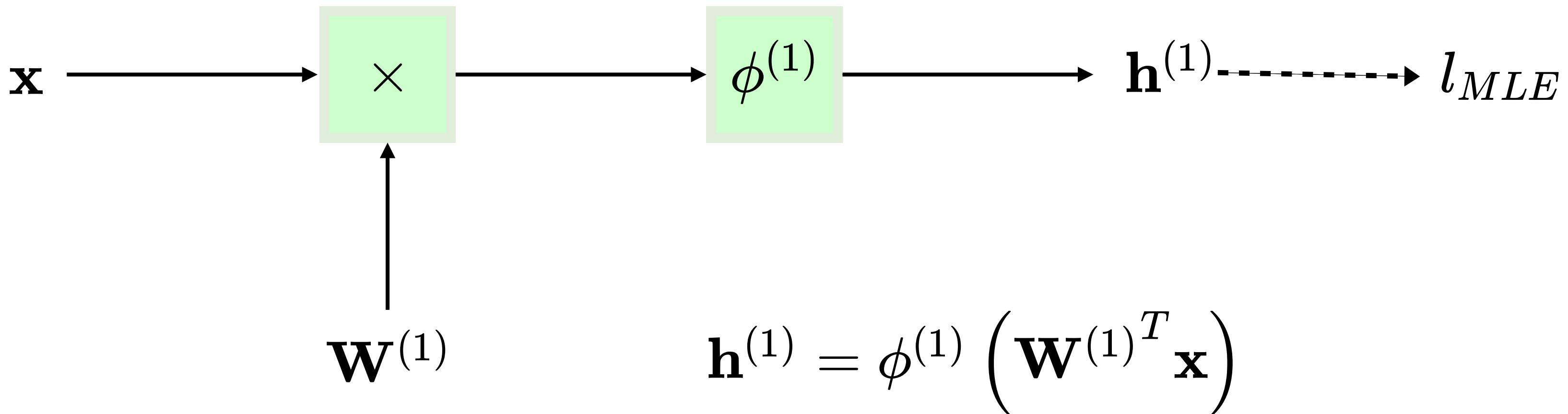
- Each node is an **operation** or a **variable**

MLP: Computational graph



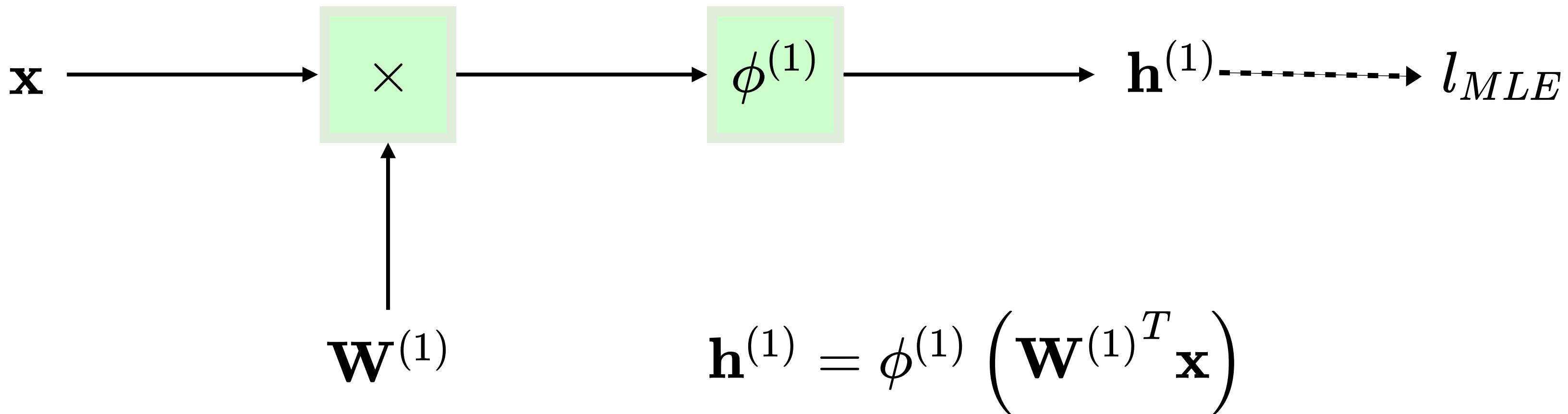
- Each node is an **operation** or a **variable**
- An *operation* has some inputs/outputs made of *variables*

MLP: Computational graph



- Each node is an **operation** or a **variable**
- An *operation* has some inputs/outputs made of *variables*
- The **forward pass** computes the output for a given input

MLP: Computational graph



- Each node is an **operation** or a **variable**
- An *operation* has some inputs/outputs made of *variables*
- The **forward pass** computes the output for a given input
- The **backward pass** computes the gradient of the loss with respect to the parameters (here, $\mathbf{W}^{(1)}$)

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

→ **For example:**

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

→ **For example:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, $z \in \mathbb{R}$:

$\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

→ **For example:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, $z \in \mathbb{R}$:

$\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$

Then $\frac{\delta z}{\delta \mathbf{x}} \in \mathbb{R}^n$, and

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}}y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

→ **For example:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, $z \in \mathbb{R}$:

$\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$

Then $\frac{\delta z}{\delta \mathbf{x}} \in \mathbb{R}^n$, and $\frac{\delta z}{\delta \mathbf{x}} = \left[\frac{\delta z}{\delta x_i} \right]_{i=1}^n$ with $\frac{\delta z}{\delta x_i} = \sum_{j=1}^m \frac{\delta z}{\delta y_j} \frac{\delta y_j}{\delta x_i}$

Backpropagation : vector derivatives

→ As mentionned before, computing gradients rapidly becomes complicated when we work with **vectors**:

→ **Vector to scalar:**

If $\mathbf{x} \in \mathbb{R}^n$, $y \in \mathbb{R}$:

$\frac{\delta y}{\delta \mathbf{x}}$ is a vector noted $\nabla_{\mathbf{x}} y(\mathbf{x}) \in \mathbb{R}^n$
that we call a **gradient**.

→ **Vector to vector:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$:

$\frac{\delta \mathbf{y}}{\delta \mathbf{x}}$ is a matrix noted $\mathbf{J}_{\mathbf{y}}(\mathbf{x}) \in \mathbb{R}^{n \times m}$
that we call a **jacobian**.

→ **For example:**

If $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, $z \in \mathbb{R}$:

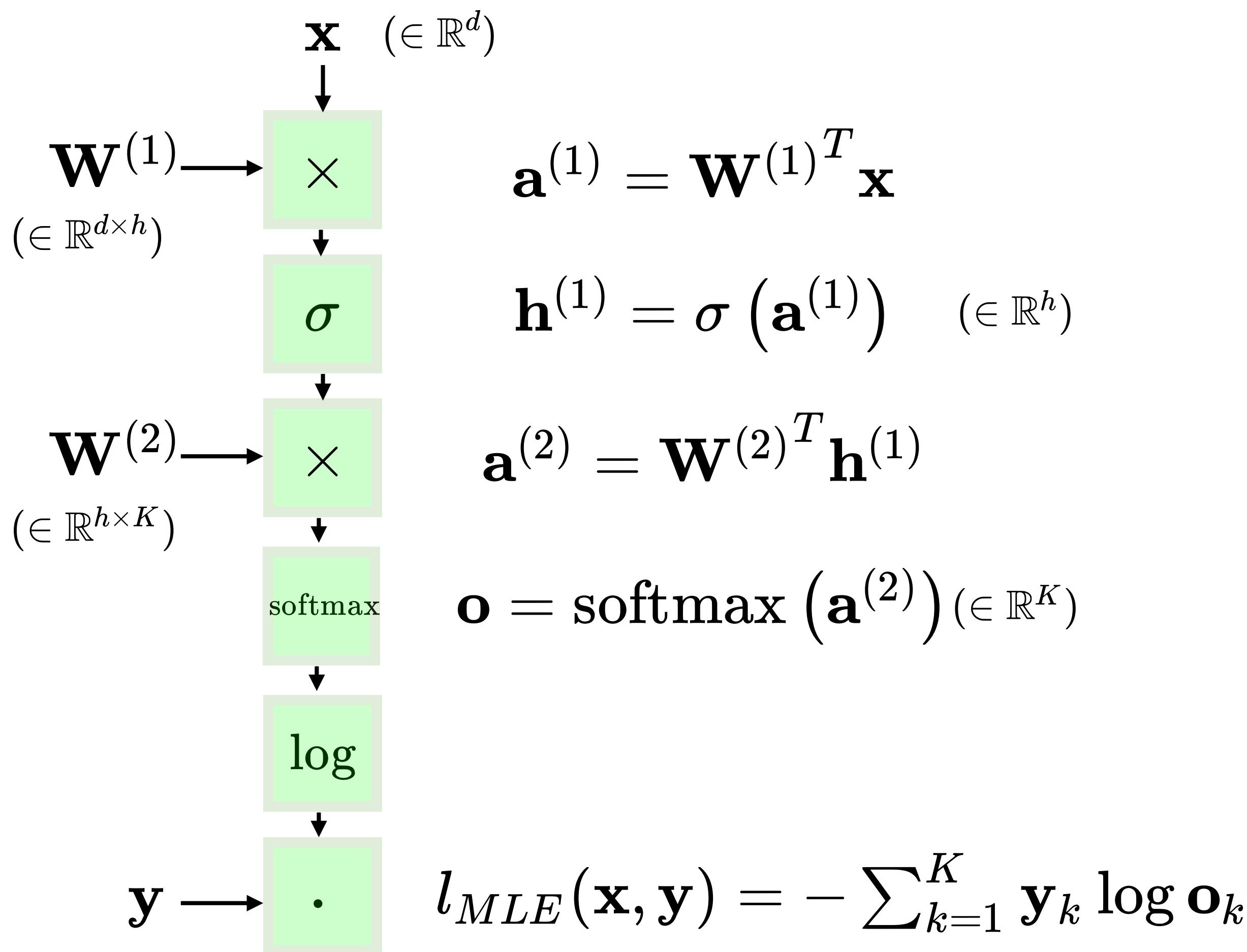
$\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$

Then $\frac{\delta z}{\delta \mathbf{x}} \in \mathbb{R}^n$, and $\frac{\delta z}{\delta \mathbf{x}} = \left[\frac{\delta z}{\delta x_i} \right]_{i=1}^n$ with $\frac{\delta z}{\delta x_i} = \sum_{j=1}^m \frac{\delta z}{\delta y_j} \frac{\delta y_j}{\delta x_i}$

Hence, $\frac{\delta z}{\delta \mathbf{x}} = \frac{\delta z}{\delta \mathbf{y}} \times \frac{\delta \mathbf{y}}{\delta \mathbf{x}}^T \leftarrow \text{Product of matrices !}$
 $[1 \times n] = [1 \times m] \times [m \times n]$

Backpropagation for a MLP

Example with a **one-hidden layer MLP** which outputs a prediction over K classes: **compute the gradients** of the weights ?



Backpropagation for a MLP

First, compute $\frac{\delta l_{MLE}}{\delta \mathbf{W}^{(2)}}$! For each individual weight,

$$\begin{aligned}\frac{\delta l_{MLE}}{\delta \mathbf{W}_{ij}^{(2)}} &= \frac{\delta l_{MLE}}{\delta \mathbf{a}_j^{(2)}} \times \frac{\delta \mathbf{a}_j^{(2)}}{\delta \mathbf{W}_{ij}^{(2)}} \longrightarrow = \mathbf{h}_i^{(1)} \\ &= \left(\sum_{k=1}^K \frac{\delta l_{MLE}}{\delta \mathbf{o}_k} \times \frac{\delta \mathbf{o}_k}{\delta \mathbf{a}_j^{(2)}} \right) \mathbf{h}_i^{(1)} \\ &\quad \downarrow \qquad \qquad \qquad \longrightarrow \left\{ \begin{array}{l} = -\mathbf{o}_j \mathbf{o}_k \text{ if } k \neq j \\ = -\mathbf{o}_j (1 - \mathbf{o}_j) \text{ if } k = j \end{array} \right. \\ &\quad \longrightarrow = -\frac{\mathbf{y}_k}{\mathbf{o}_k} \\ &= \left(-\mathbf{y}_j (1 - \mathbf{o}_j) + \sum_{k \neq j} \mathbf{y}_k \mathbf{o}_j \right) \mathbf{h}_i^{(1)} \\ &= (\mathbf{y}_j - \mathbf{o}_j) \mathbf{h}_i^{(1)}\end{aligned}$$

Backpropagation for a MLP

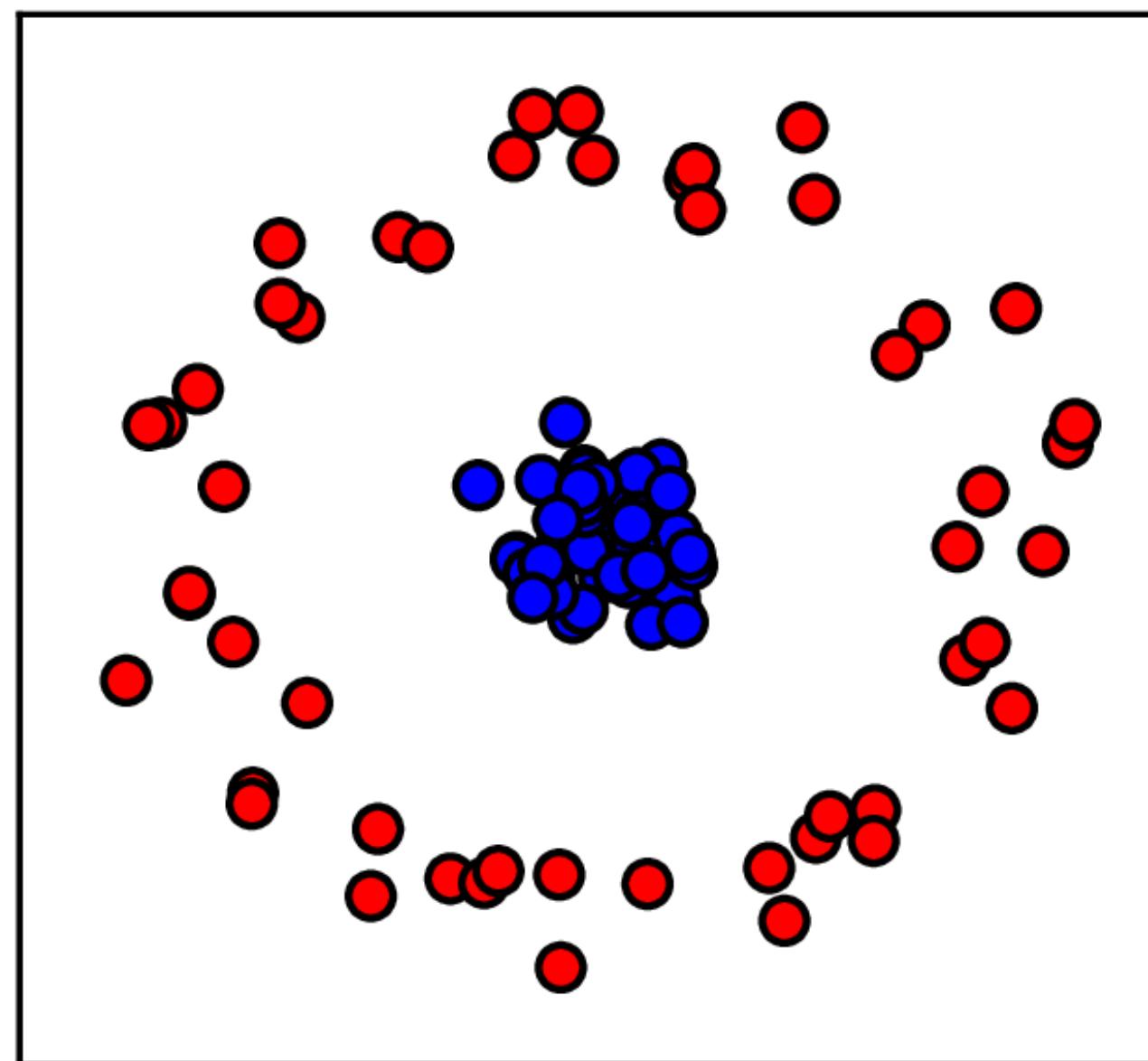
Now, compute $\frac{\delta l_{MLE}}{\delta \mathbf{W}^{(1)}}$! For each individual weight,

$$\begin{aligned}\frac{\delta l_{MLE}}{\delta \mathbf{W}_{ij}^{(1)}} &= \frac{\delta l_{MLE}}{\delta \mathbf{a}_j^{(1)}} \times \frac{\delta \mathbf{a}_j^{(1)}}{\delta \mathbf{W}_{ij}^{(1)}} \longrightarrow = \mathbf{x}_i \\ &= \left(\sum_{k=1}^K \frac{\delta l_{MLE}}{\delta \mathbf{a}_k^{(2)}} \times \frac{\delta \mathbf{a}_k^{(2)}}{\delta \mathbf{a}_j^{(1)}} \right) \mathbf{x}_i \\ &\quad \begin{array}{c} \text{L} \\ \text{L} \end{array} \longrightarrow = \mathbf{W}_{jk}^{(2)} \times \mathbf{h}_j^{(1)} (1 - \mathbf{h}_j^{(1)}) \\ &= \left(\sum_{k=1}^K (\mathbf{y}_k - \mathbf{o}_k) \mathbf{W}_{jk}^{(2)} \right) \mathbf{h}_j^{(1)} (1 - \mathbf{h}_j^{(1)}) \mathbf{x}_i\end{aligned}$$

→ Then, we just need to make the updates !

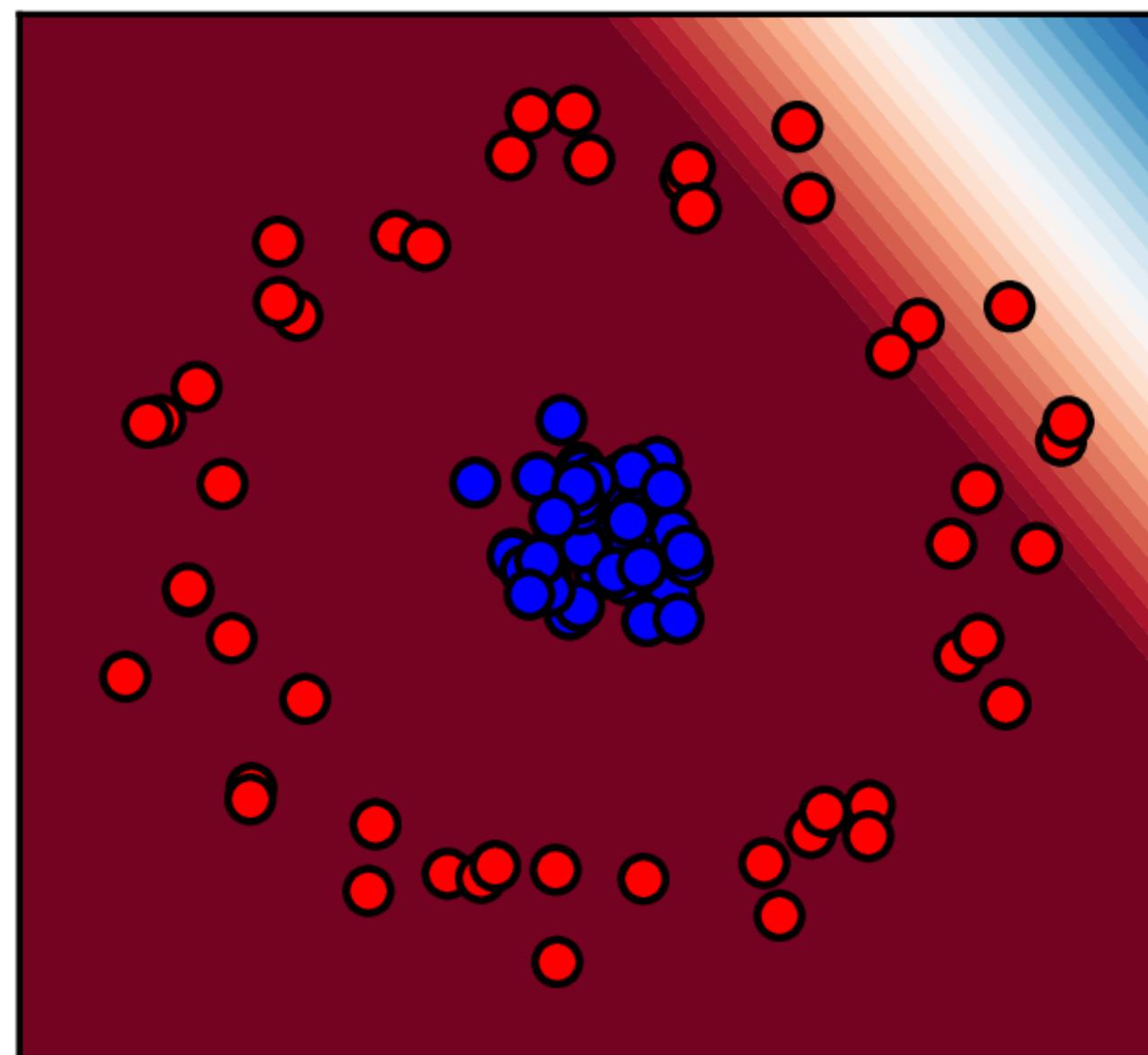
What can Neural Networks do ?

We now have a way to efficiently train neural networks. But **how many neurons** are needed for a one-layer NN to successfully separate these two classes ?



What can Neural Networks do ?

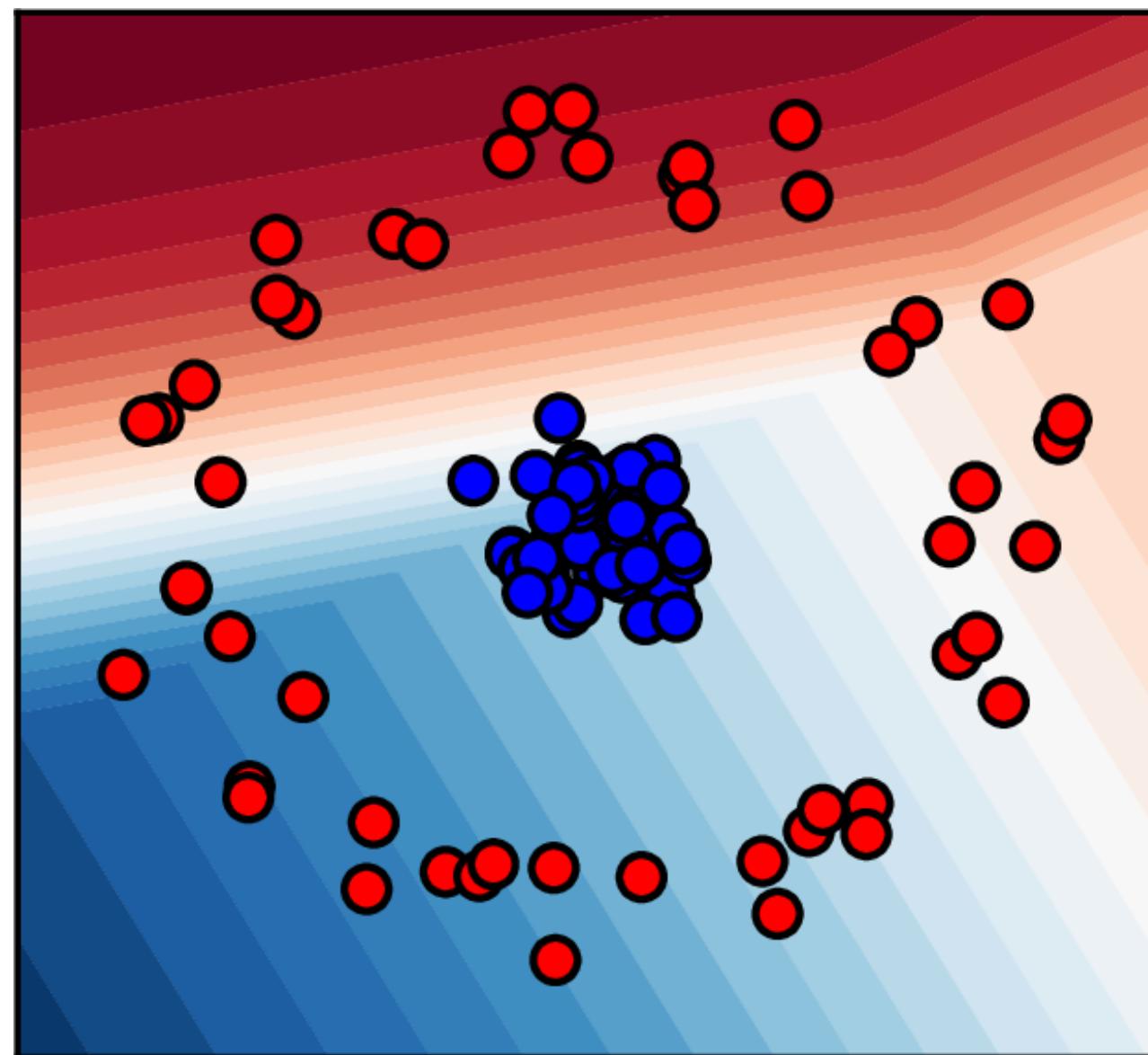
We now have a way to efficiently train neural networks. But **how many neurons** are needed for a one-layer NN to successfully separate these two classes ?



Hidden neuron: 1

What can Neural Networks do ?

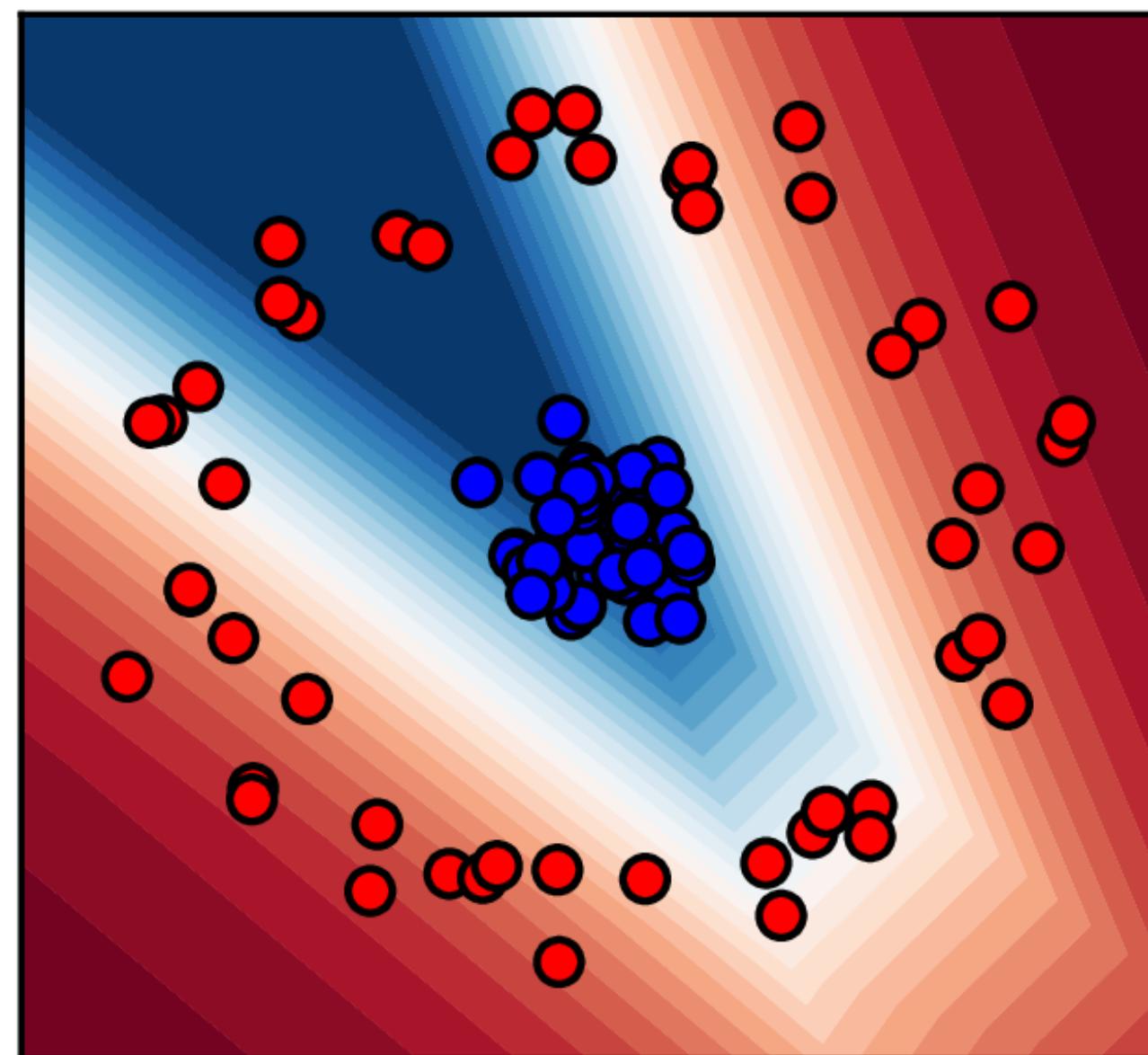
We now have a way to efficiently train neural networks. But **how many neurons** are needed for a one-layer NN to successfully separate these two classes ?



Hidden neuron: 2

What can Neural Networks do ?

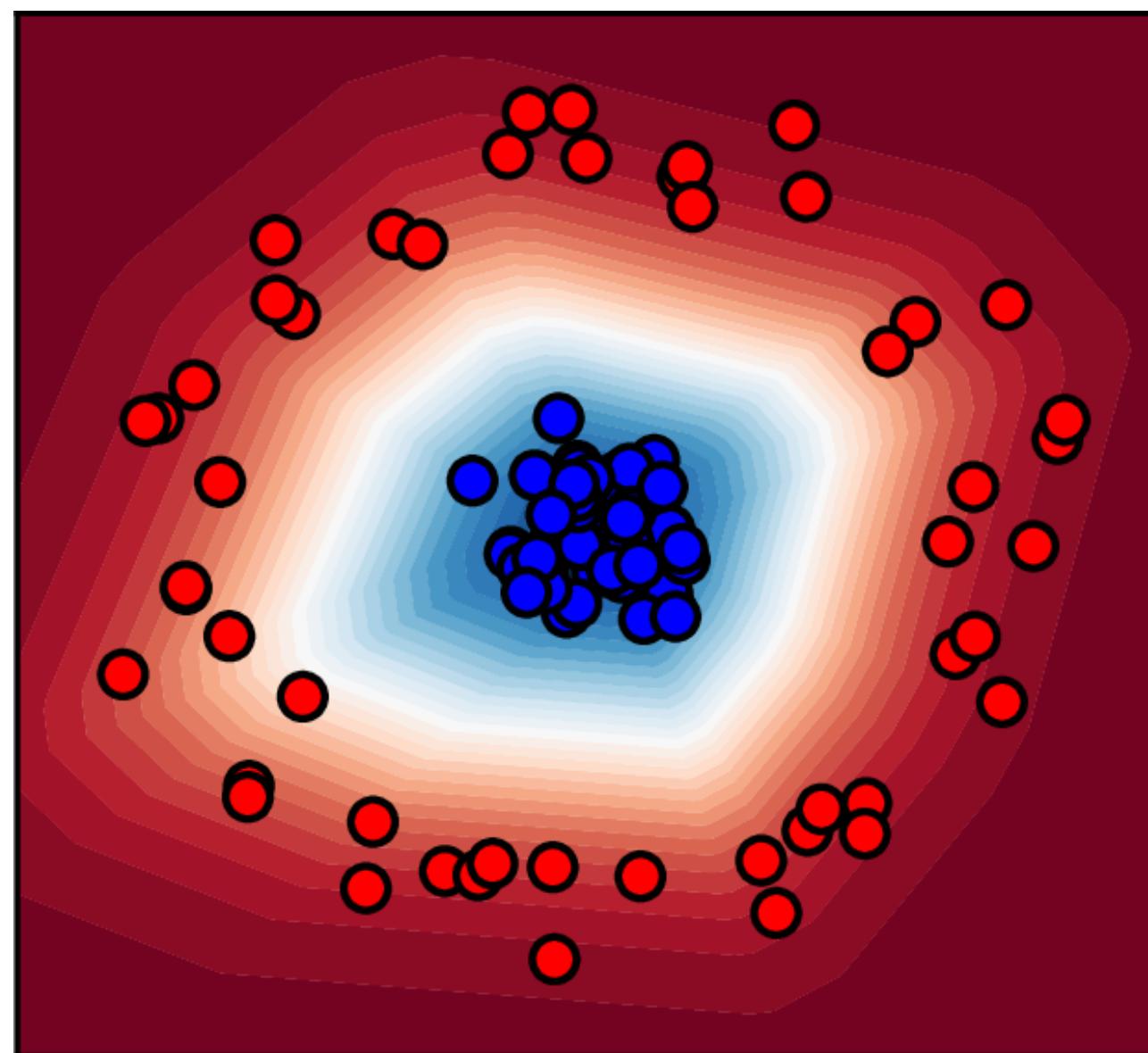
We now have a way to efficiently train neural networks. But **how many neurons** are needed for a one-layer NN to successfully separate these two classes ?



Hidden neuron: 4

What can Neural Networks do ?

We now have a way to efficiently train neural networks. But **how many neurons** are needed for a one-layer NN to successfully separate these two classes ?



Hidden neuron: 20

What can Neural Networks do ?

One-layer NN were proved to be **universal approximators** by Hornik et al. (*Multilayer feedforward networks are universal approximators*, 1989)

What can Neural Networks do ?

One-layer NN were proved to be **universal approximators** by Hornik et al. (*Multilayer feedforward networks are universal approximators*, 1989)

Theorem:

The family of multi-layer perceptrons with one hidden layer of $p + 1$ inputs is **dense** in the space of continuous functions from a compact subset of \mathbb{R}^p to \mathbb{R}

What can Neural Networks do ?

One-layer NN were proved to be **universal approximators** by Hornik et al. (*Multilayer feedforward networks are universal approximators*, 1989)

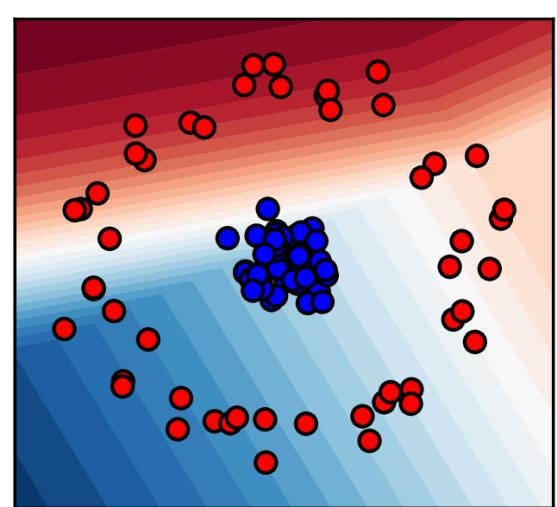
Theorem:

The family of multi-layer perceptrons with one hidden layer of $p + 1$ inputs is **dense** in the space of continuous functions from a compact subset of \mathbb{R}^p to \mathbb{R}

→ For any function f in that space, there exist a number of hidden neurons N and corresponding parameters θ such that the resulting MLP is arbitrarily close to f . ([Read more](#))

Model and hyperparameter selection

Model and hyperparameter selection

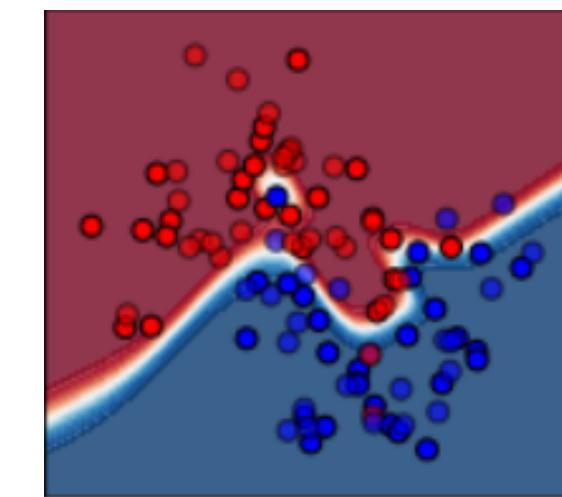


Model not able to obtain a satisfying result on the dataset
→ Not enough *capacity* !
→ **Underfitting**

Model and hyperparameter selection

Model obtaining great results but adapts too much to outliers

- Too much *capacity* !
- **Overfitting**



Model and hyperparameter selection

How to verify in which case we are ?

→ Split data into **train** and **test** data - and choose the *hyperparameters* (here, the number of neurons, determining the capacity of the model) giving the best result on test data.

Model and hyperparameter selection

How to verify in which case we are ?

→ Split data into **train** and **test** data - and choose the *hyperparameters* (here, the number of neurons, determining the capacity of the model) giving the best result on test data.

Not enough ! How will it behave on new data ?

→ Split data into **train**, **validation** and **test** data - use the validation to select the best hyperparameters and verify the performance on test data.

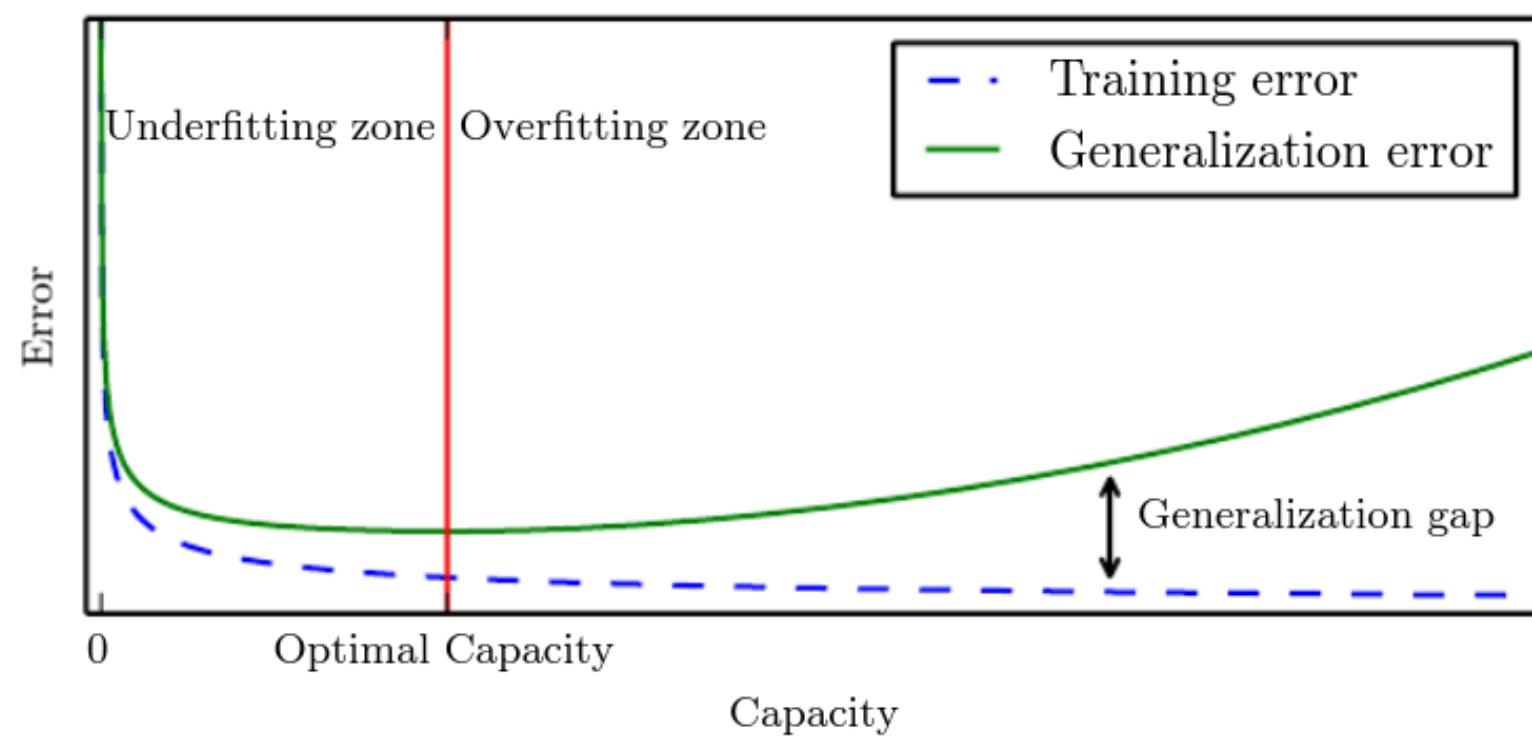
Model and hyperparameter selection

How to verify in which case we are ?

→ Split data into **train** and **test** data - and choose the *hyperparameters* (here, the number of neurons, determining the capacity of the model) giving the best result on test data.

Not enough ! How will it behave on new data ?

→ Split data into **train**, **validation** and **test** data - use the validation to select the best hyperparameters and verify the performance on test data.

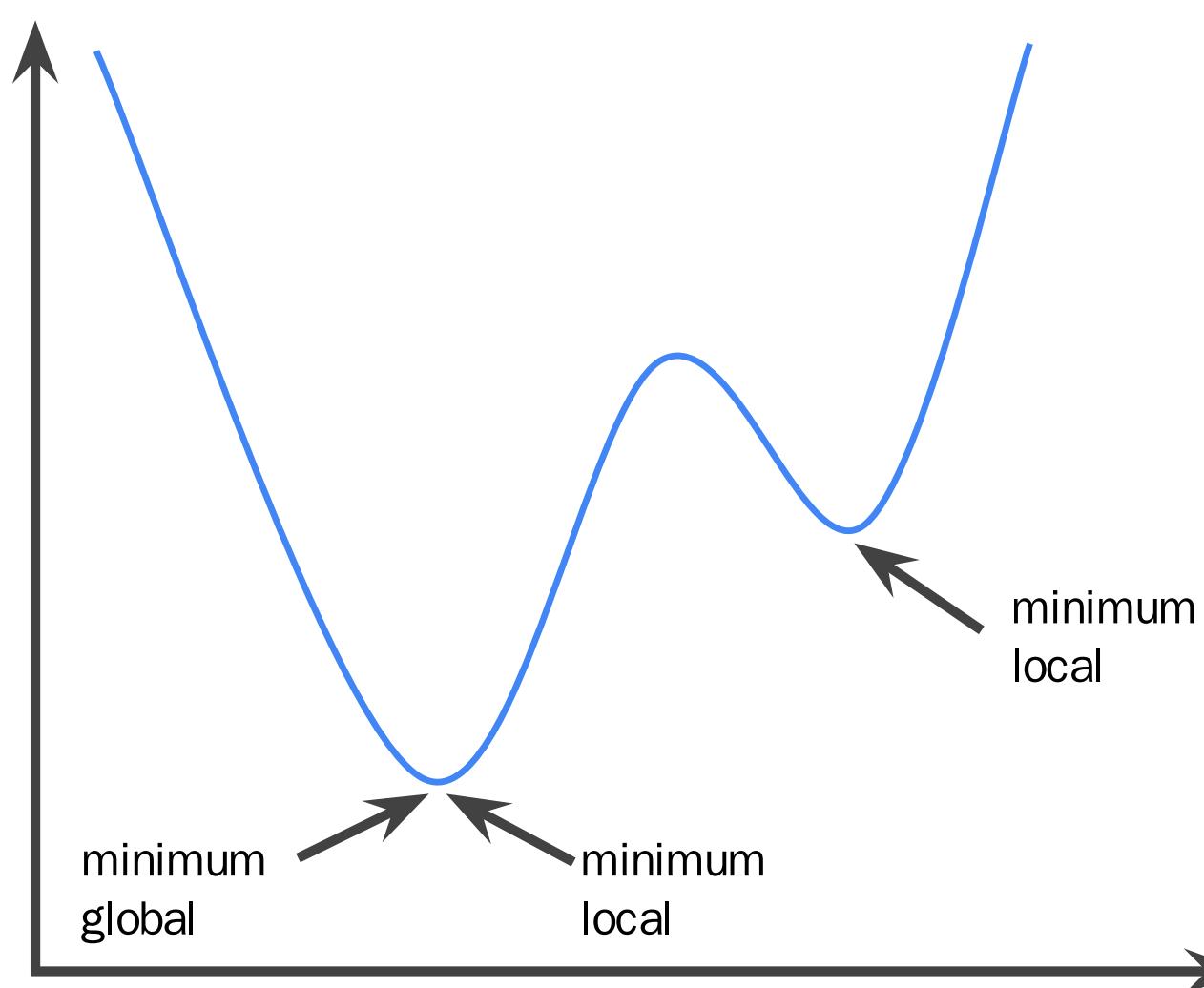


→ **Monitoring the error** on training and validation sets allows you to choose the capacity - but also **other hyperparameters** !

Optimization hyperparameters

Composing non-linear activations makes the loss **non-convex**

→ We will probably end-up on a local minima (which can be good !)



Optimization hyperparameters

Composing non-linear activations makes the loss **non-convex**

→ We will probably end-up on a local minima (which can be good !)

→ How to choose the **learning rate** ϵ - a very important hyperparameter during training ?

- In case of convex optimization, we have exact search strategies.

Optimization hyperparameters

Composing non-linear activations makes the loss **non-convex**

→ We will probably end-up on a local minima (which can be good !)

→ How to choose the **learning rate** ϵ - a very important hyperparameter during training ?

- In case of convex optimization, we have exact search strategies.
- Simple heuristic:
 - Choose a small initial ϵ
 - Monitor the performance on validation data: if it does not increase, decay the learning rate with a value α : $\epsilon = \epsilon * \alpha$

Optimization hyperparameters

Composing non-linear activations makes the loss **non-convex**

→ We will probably end-up on a local minima (which can be good !)

→ How to choose the **learning rate** ϵ - a very important hyperparameter during training ?

- In case of convex optimization, we have exact search strategies.
- Simple heuristic:
 - Choose a small initial ϵ
 - Monitor the performance on validation data: if it does not increase, decay the learning rate with a value α : $\epsilon = \epsilon * \alpha$
- Other strategies:
 - Step decay: $\times \alpha \in [0, 1]$ every N epochs
 - Exponential decay: $\epsilon^{(t)} = \epsilon^{(0)} e^{(-\alpha t)}$

Regularization

Since we can not change the model capacity during training, we use an artificial constraint, **regularization**, to avoid the model *overfitting* the data

→ **Weight decay** adds a tradeoff between fitting the training data and keeping the parameters small - if $\theta = \mathbf{w}$:

$$l_{Reg}(\mathbf{w}) = l_{MLE}(\mathbf{w}) + \lambda \mathbf{w}^T \mathbf{w}$$

The tradeoff is controlled by λ .

→ Constraining the norm of the weights keep the model from adapting too much to the training data !

→ The update rule becomes:

$$\mathbf{w}^{t+1} = (1 - 2\lambda) \mathbf{w}^t - \epsilon \nabla_{\mathbf{w}} l_{MLE}(\mathbf{w})$$

Initialization

Initialization defines the way we choose the initial values of our parameters

→ This choice is highly experimental and what works will depend on the activations and the optimization procedure. Usually,

$$\mathbf{w}^{(i)} \sim \mathcal{N}(0, \frac{1}{\sqrt{d_i}})$$

Another possibility is:

$$\mathbf{w}^{(i)} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{d_i + d_{i+1}}}, \frac{\sqrt{6}}{\sqrt{d_i + d_{i+1}}}\right]$$

→ In practice, most frameworks will nowadays offer alternative solutions brought by progress on other aspects (optimization, architecture, regularization ...)

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

However,

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

However,

- Gradient descent is not easy to execute
 - + Vanishing gradients, saturated gradients

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

However,

- Gradient descent is not easy to execute
 - + Vanishing gradients, saturated gradients
- A lot of model selection ! (Many hyperparameters)

Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

However,

- Gradient descent is not easy to execute
 - + Vanishing gradients, saturated gradients
- A lot of model selection ! (Many hyperparameters)

→ We may have many parameters and not too much data: **Overfitting !**

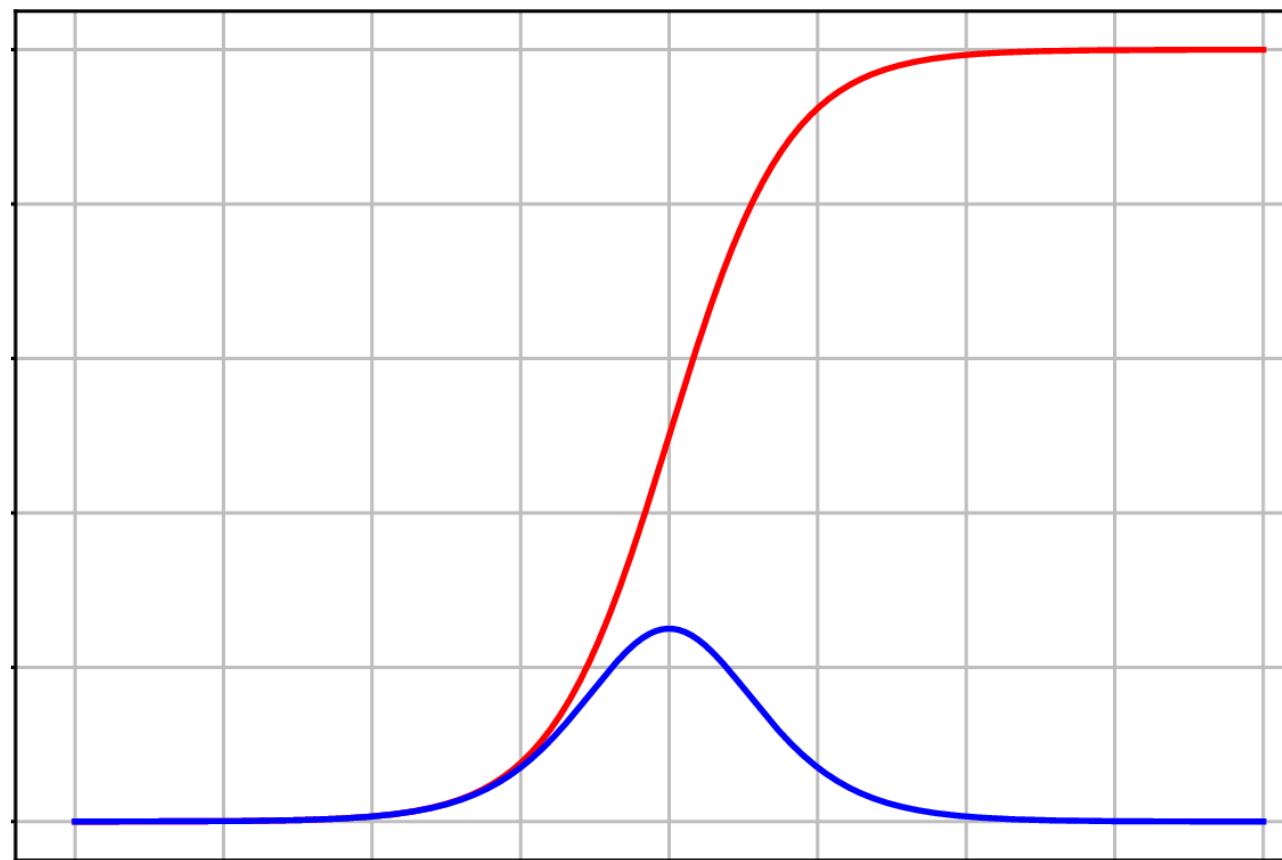
Advantages and difficulties

- Multi-layer perceptrons are very flexible in terms of possible *inputs* and *outputs*
- With **backpropagation**, we can efficiently compute updates for a large number of parameters
- With **stochastic gradient descent**, we have an efficient training procedure that is adapted to a large quantity of data

However,

- Gradient descent is not easy to execute
 - + Vanishing gradients, saturated gradients
 - A lot of model selection ! (Many hyperparameters)
- Optimization is hard: **Underfitting** !
- We may have many parameters and not too much data: **Overfitting** !

Vanishing/saturated gradient: intuition



- The derivative of the activation function is:
$$\sigma' = \sigma(1 - \sigma)$$
- From the chain rule, we can see that gradients will get smaller and smaller as we go back through the layers (**vanishing**)
- When the input values are far from 0, the gradient is very small (**saturated**)
- Higher layers will learn far better than the lower ones, which will often get stuck during training

Deep Learning

Given these issues, what made Neural Networks so prevalent ?

Deep Learning

Deep Learning

- If **underfitting**:
 - Use more data\better optimization !
 - If **overfitting**:
 - Find a better way to regularize !
- **Unsupervised learning**:
 - Initialize hidden layers using *unsupervised learning* to have the network represent the latent structure of the input.
 - Harder task, less overfitting !
 - Hidden layers encode the 'general' structure of the data before the supervised training !
 - **Dropout**:
 - Randomly remove parts of the hidden layers.
 - Stops the network from 'copying' data !

Better optimization: Adaptive Gradient

- Choosing a learning rate is difficult: learning rate schedules adjust the learning rate during training but have to be defined in advance
- The same learning rate applies to all parameter updates
 - Adapt the rate of learning to the gradient - for example, with:

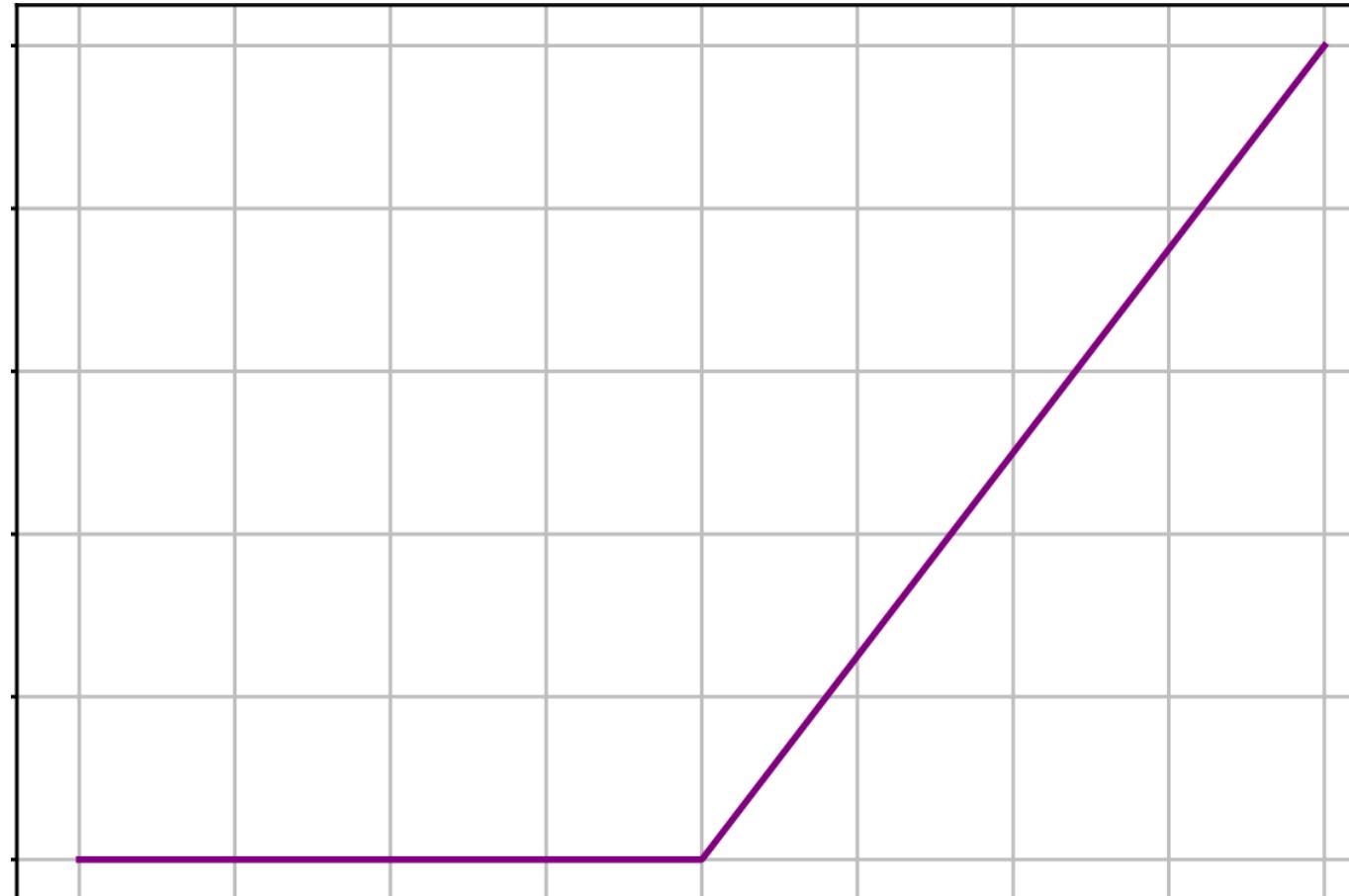
$$\theta^{t+1} = \theta^t - \epsilon \frac{\delta l_{MLE}}{\delta \theta}$$

↓

$$g^{t+1} = g^t + \left(\frac{\delta l_{MLE}}{\delta \theta} \right)^2$$
$$\theta^{t+1} = \theta^t - \frac{\epsilon}{\sqrt{g^{t+1}}} \times \frac{\delta l_{MLE}}{\delta \theta}$$

Which is close to **Adagrad** (Duchi et al, 2011), while **Adadelta** (Zeiler et al, 2012) and especially **Adam** (Kingma et al, 2015) are often used.

Better activation: ReLU

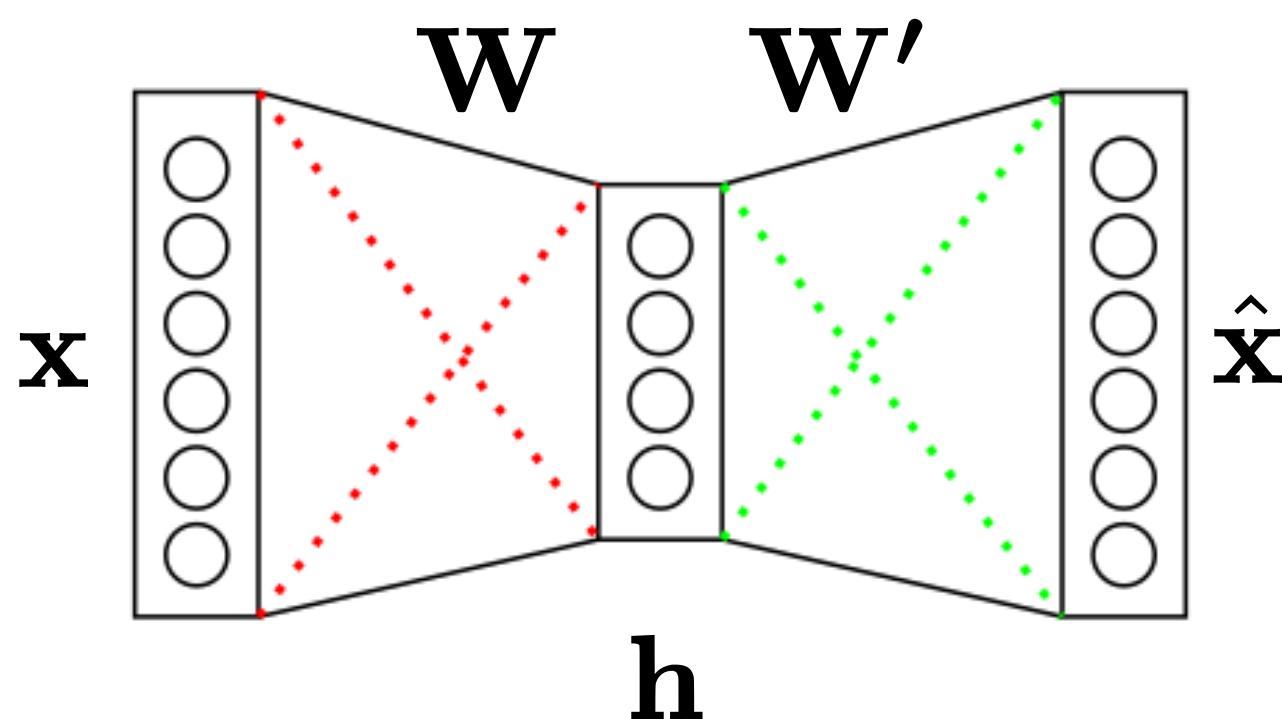


$$relu(x) = \max(0, x)$$

- **ReLU** (Glorot et al. 2011) allows for efficient gradient passing, avoiding saturated and vanishing gradients !
- There exists many more recent variants.

Parenthesis: Autoencoders

- *Autoencoder*: Feed-forward neural network trained to re-build its input through a hidden representation.
→ **Unsupervised learning !**



- Encoder:
$$\mathbf{h} = \phi(\mathbf{W}^T \mathbf{x} + \mathbf{b})$$
- Decoder:
$$\hat{\mathbf{x}} = \mathbf{W}'^T \mathbf{h} + \mathbf{b}'$$

- Loss function: sum of squared differences ($\theta = \{\mathbf{W}, \mathbf{W}', \mathbf{b}, \mathbf{b}'\}$)

$$l_{L2}(\mathbf{x}|\theta) = \frac{1}{2} \sum_k (\hat{x}^k - x^k)^2$$

- Weights \mathbf{W} and \mathbf{W}' can also be tied: $\mathbf{W}' = \mathbf{W}^T$

Parenthesis: Autoencoders

From Hugo Larochelle's class on neural networks, part [6.2]

- Important remark: The loss function of the auto-encoder assume data coming from an **isotropic gaussian** !

Parenthesis: Autoencoders

From Hugo Larochelle's class on neural networks, part [6.2]

- Important remark: The loss function of the auto-encoder assume data coming from an **isotropic gaussian** !
- How to adapt the loss function to new data ?
Assuming this data \mathbf{x} comes from a distribution $p(\mathbf{x}|\mu)$ with parameters μ :
 - Choose the relationship between μ and the hidden representation \mathbf{h} - which will be symmetric.
→ Usually, linear transformation + non-linearity !
 - Use $l(\mathbf{x}) = -\log p(\mathbf{x}|\mu)$ as loss function.

Parenthesis: Autoencoders

From Hugo Larochelle's class on neural networks, part [6.2]

- Important remark: The loss function of the auto-encoder assume data coming from an **isotropic gaussian** !
- How to adapt the loss function to new data ?
Assuming this data \mathbf{x} comes from a distribution $p(\mathbf{x}|\mu)$ with parameters μ :
 - Choose the relationship between μ and the hidden representation \mathbf{h} - which will be symmetric.
→ Usually, linear transformation + non-linearity !
 - Use $l(\mathbf{x}) = -\log p(\mathbf{x}|\mu)$ as loss function.
- So, assuming data from an isotropic gaussian with mean μ :
 - $p(\mathbf{x}|\mu) = \frac{1}{(2\pi)^{D/2}} \exp\left(-\frac{1}{2} \sum_k (x^k - \mu^k)^2\right)$
 - We choose $\mu = \mathbf{W}^T \mathbf{h} + \mathbf{b}$; and obtain the squared error loss !

Parenthesis: Autoencoders

From Hugo Larochelle's class on neural networks, part [6.4]

- Important remark: assuming a linear decoder, the **linear** encoder (no activation function for \mathbf{h}) is **actually optimal** for minimizing the training square error !

Parenthesis: Autoencoders

From Hugo Larochelle's class on neural networks, part [6.4]

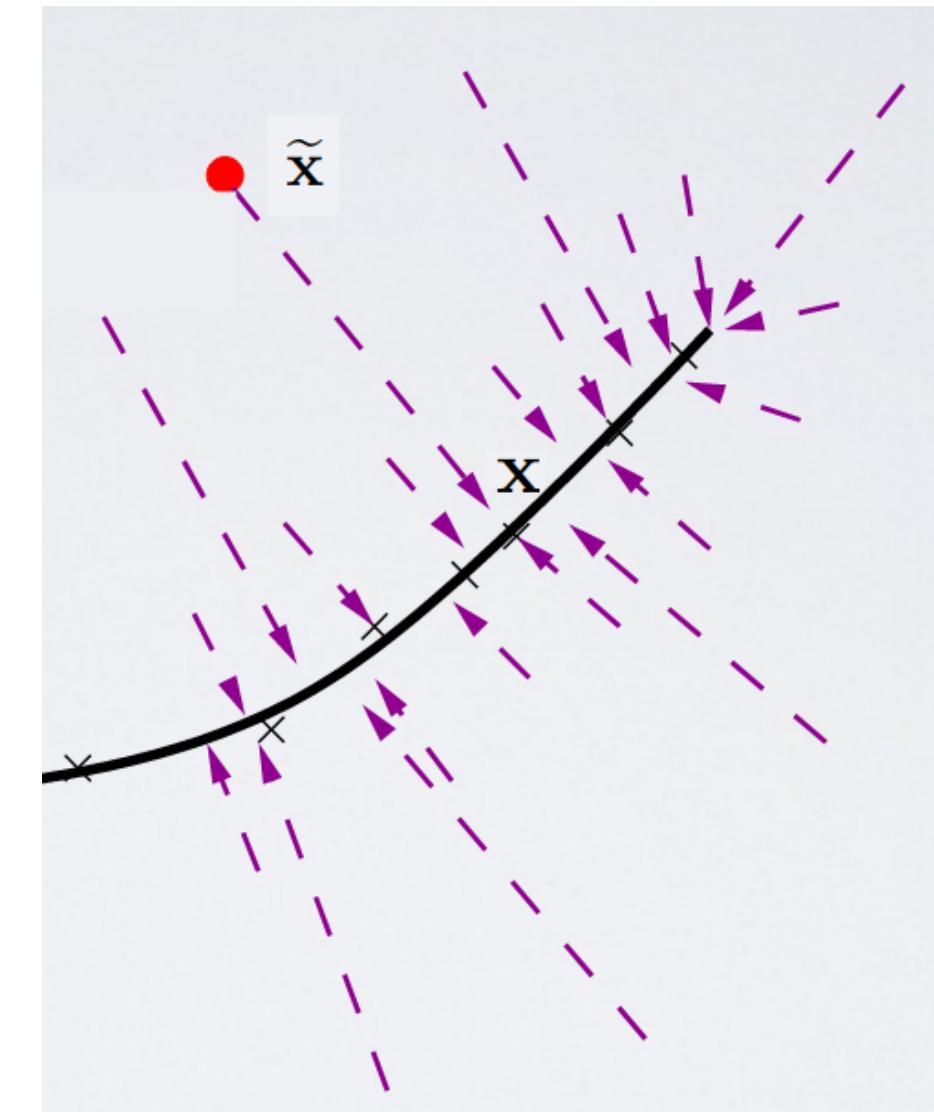
- Important remark: assuming a linear decoder, the **linear** encoder (no activation function for \mathbf{h}) is **actually optimal** for minimizing the training square error !
- The proof relies on the result that the SVD decomposition of a matrix where we keep only the k largest singular values produces the closest matrix of rank k to the original matrix
 - We show that, with k the hidden layer size, **the encoder minimizing the square error comes from the SVD decomposition limited to the k largest singular values**, hence is linear.
 - Intuitive result: if the data is standardized, the encoder corresponds to the Principal Component Analysis (PCA)

Parenthesis: Autoencoders

- Then, what's the point of using an autoencoder ?
 - If we choose a **smaller hidden size**, we compress the data, but that works only for the training distribution
 - If we choose a **larger hidden size**, the autoencoder could just copy each input component and not extract anything

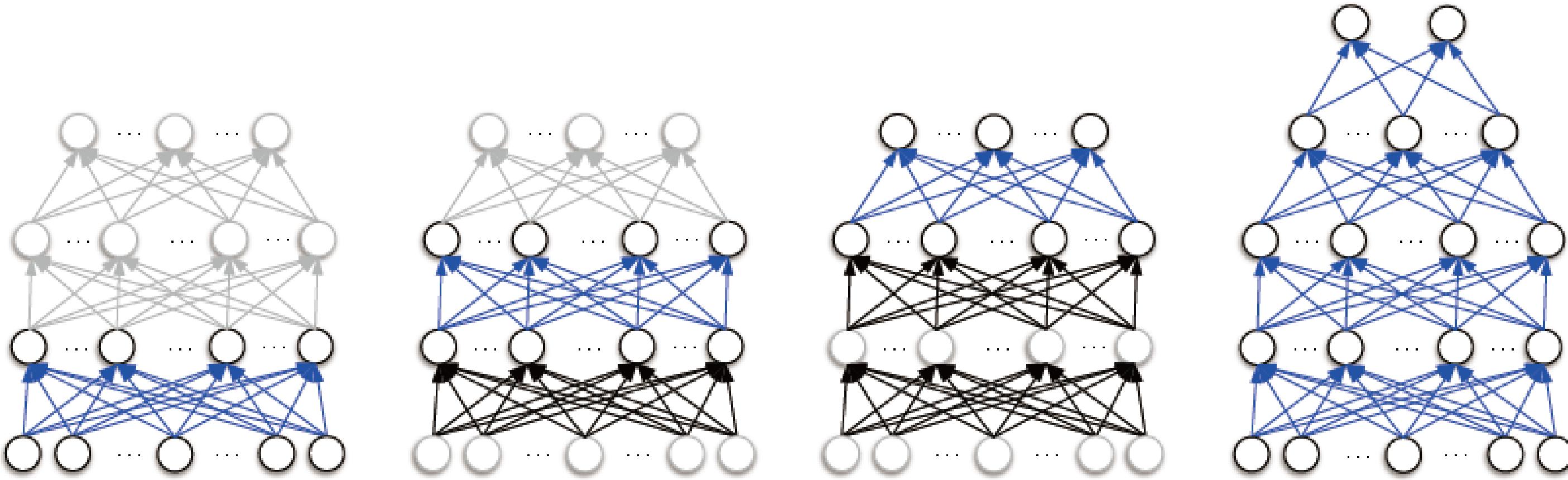
Parenthesis: Autoencoders

- Then, what's the point of using an autoencoder ?
 - If we choose a **smaller hidden size**, we compress the data, but that works only for the training distribution
 - If we choose a **larger hidden size**, the autoencoder could just copy each input component and not extract anything
 - Idea: make the hidden representation robust to noise !
 - Use noise to create corrupted $\tilde{\mathbf{x}}$, and reconstruct $\hat{\mathbf{x}}$.
 - The loss function compares $\hat{\mathbf{x}}$ to the noiseless \mathbf{x} !
- **Denoising autoencoder** (Vincent et al, 2008) : forces the model to learn about the structure of data !



From Hugo Larochelle's class on neural networks, part [6.6]

Better initialization: Unsupervised pre-training

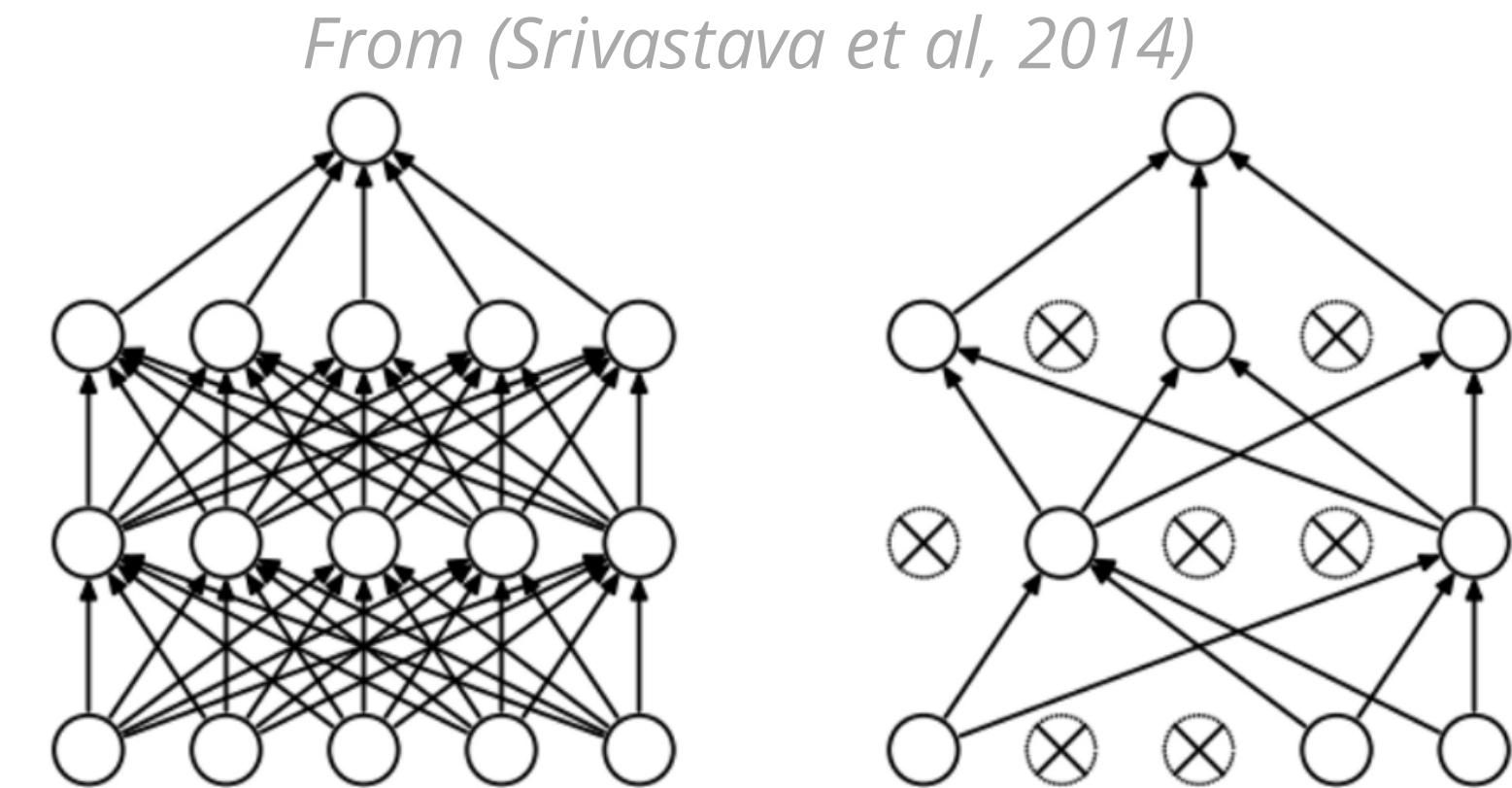


Layer-wise Pre-training followed by **Training the whole network**

- Intuition: when you cannot really escape from the initial (random) point, find a good starting point → **unsupervised** learning !
- This is done by training a **denoising auto-encoder** (but later, other methods !) layer by layer to rebuild the altered input of the layer.
- After unsupervised **pre-training**, add an output layer and train with supervised learning: this is called **fine-tuning**.

Better regularization: Dropout

- For each training example : randomly turn-off hidden units with probability p
 - At test time, use each unit scaled down by p
-
- Intuition: similar idea to denoising autoencoders.
 - In practice: do backpropagation with **random masks** and replace them by their expectation at test time.
 - Dropout aims to **separate effects** from strongly correlated features: each unit needs to be useful on its own.
 - It can be seen as averaging different models that share parameters
→ It acts as powerful **regularization** scheme



Deep Learning

- 1995: **Convolutionnal** Neural Networks (Lecun, Bengio)

Deep Learning

- 1995: **Convolutionnal** Neural Networks (Lecun, Bengio)
- 1997: **Long-Short Term Memory** NN (Hochreiter, Schmidhuber)

Deep Learning

- 1995: **Convolutionnal** Neural Networks (Lecun, Bengio)
- 1997: **Long-Short Term Memory** NN (Hochreiter, Schmidhuber)
- Early 2000s: Development of **GPUs**

Deep Learning

- 1995: **Convolutionnal** Neural Networks (Lecun, Bengio)
- 1997: **Long-Short Term Memory** NN (Hochreiter, Schmidhuber)
- Early 2000s: Development of **GPUs**
- Late 2000s:
 - Very large datasets of Image are created (**Imagenet**)
→ Deeper models can be trained
 - Efficient **initialization** with unsupervised learning (**pre-training**)
→ Deeper models can avoid vanishing gradients

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

.... helped deep neural models to win challenges:

- ImageNet - since 2012 (Krizhevsky et al, 2012)

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

.... helped deep neural models to win challenges:

- ImageNet - since 2012 (Krizhevsky et al, 2012)
- Traffic signs Recognition : in 2011 (Ciresan et al, 2012)

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

.... helped deep neural models to win challenges:

- ImageNet - since 2012 (Krizhevsky et al, 2012)
- Traffic signs Recognition : in 2011 (Ciresan et al, 2012)
- Handwritting recognition - since 2009 (Graves et al, 2009)

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

.... helped deep neural models to win challenges:

- ImageNet - since 2012 (Krizhevsky et al, 2012)
- Traffic signs Recognition : in 2011 (Ciresan et al, 2012)
- Handwritting recognition - since 2009 (Graves et al, 2009)
- Automatic Speech Recognition (Hinton et al, 2012)

Deep Learning

Besides, improvements such as:

- **Dropout** (Regularization technique)
- **Rectified Linear Unit** (activation)

.... helped deep neural models to win challenges:

- ImageNet - since 2012 (Krizhevsky et al, 2012)
- Traffic signs Recognition : in 2011 (Ciresan et al, 2012)
- Handwritting recognition - since 2009 (Graves et al, 2009)
- Automatic Speech Recognition (Hinton et al, 2012)

The expression "*Deep Learning*" was coined around 2006 from work on unsupervised pre-training (Hinton et al, 2006; Bengio et al, 2007)

What else is new ?

Architectures !

- *Convolutional networks*, specialized for processing a grid of values, like time-series and images
- *Recurrent networks*, specialized for processing sequences of values of arbitrary sizes
- Networks adapted to more specific structures: Recursive networks, Tree-Recurrent networks, *Graph Neural Networks*
- *Attention mechanism* and its adaptation to create faster layers for processing sequences: the *transformer* layers.

→ **Idea** : use a model that will:

- Take advantage of the **structure** of the data
- **Capture the invariance** of a problem

Deep Learning

And besides innovations in **Architecture, Regularization, and Optimization** ...

- Tremendous improvement of computational capacity
- Availability of huge datasets

Deep Learning

And besides innovations in **Architecture, Regularization, and Optimization** ...

- Tremendous improvement of computational capacity
- Availability of huge datasets

Why does it work ?

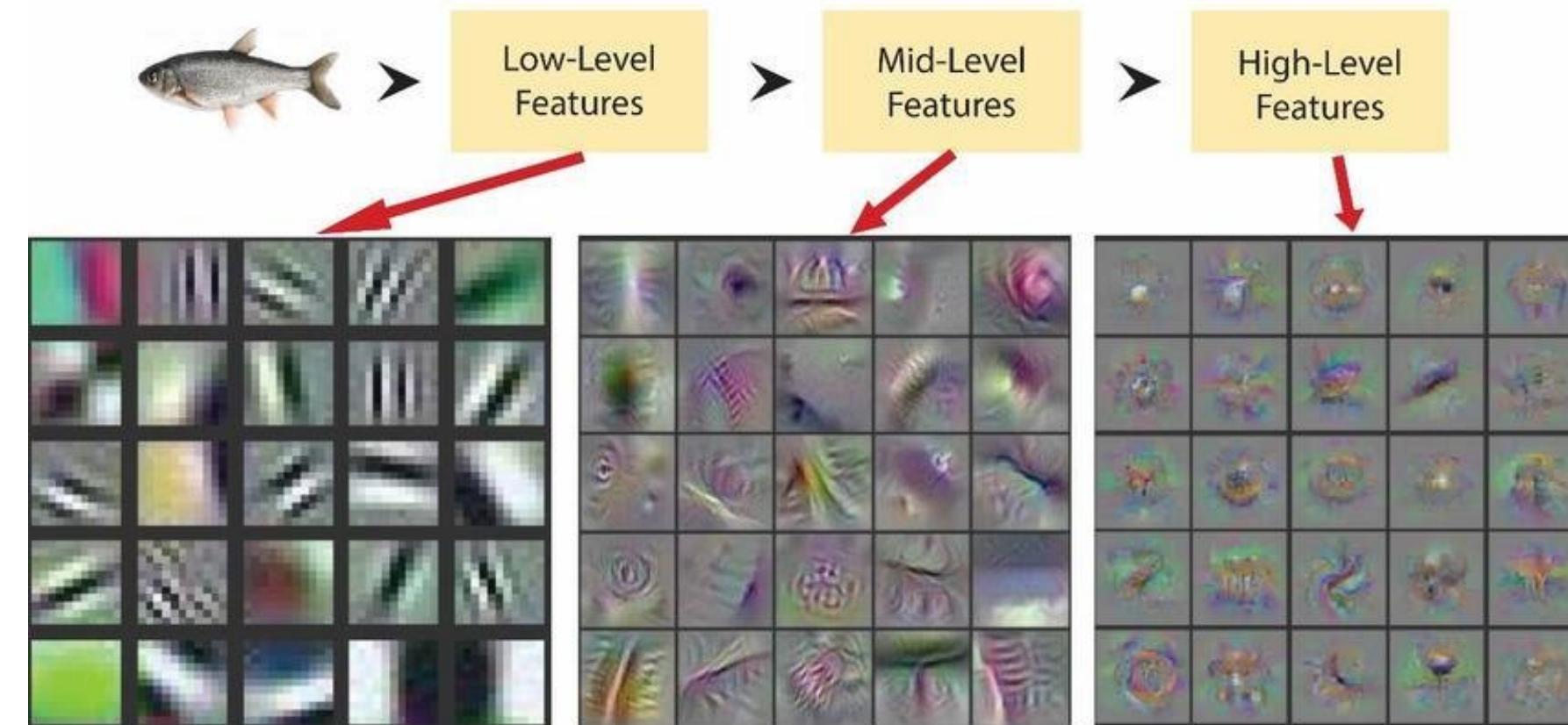
Deep Learning

And besides innovations in **Architecture, Regularization, and Optimization** ...

- Tremendous improvement of computational capacity
- Availability of huge datasets

Why does it work ?

→ Deep learning is especially relevant for complex inputs like *images*, natural language, because it is especially good at learning **better data representation**.



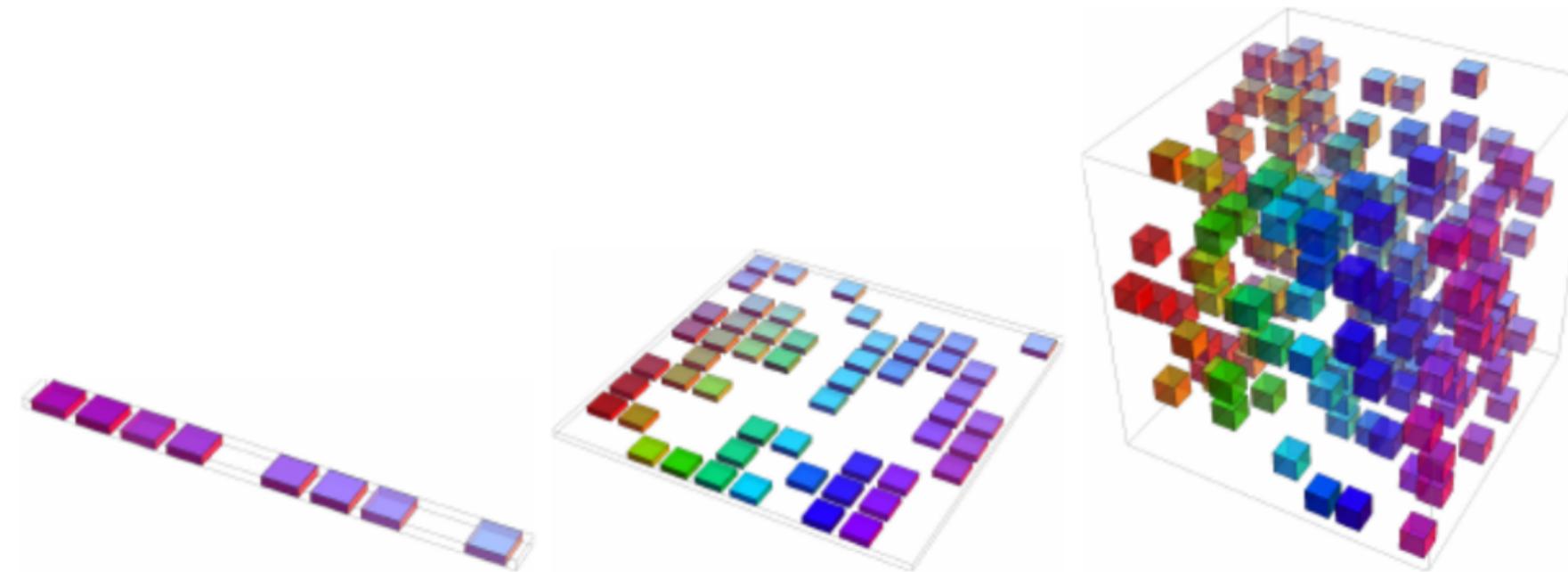
From *Automatic fish species classification in underwater videos*
(Siddiqui et al, 2016)

DL: a kind of representation learning

Various (tied !) ideas explaining the success of deep learning:

- It makes great use of *distributed representations*, allowing **generalization** (through shared features and **smoothness**) : the deep network is able to compactly represent complicated structures using a small number of features automatically !
- Learning to accomplish a task on some data will organize the hidden representation of this data to be easy to process for the task (discovering *explanatory factors*)
- Exponential gain through **depth**: features learned can be organized and re-used **hierarchically**, with more abstract representations on the top.
- All of these ideas are especially relevant when working in very high dimensions

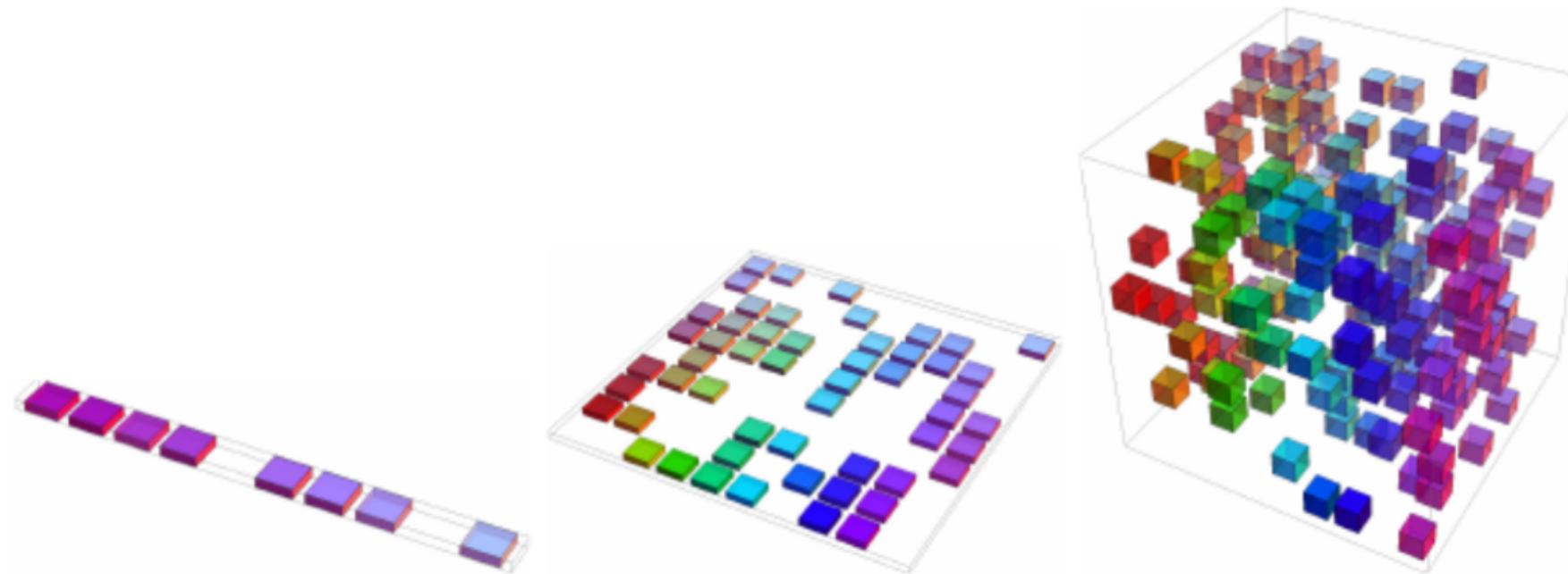
Deep Learning



From *Deep Learning*
(Ian Goodfellow,
Yoshua Bengio and
Aaron Courville, 2016)

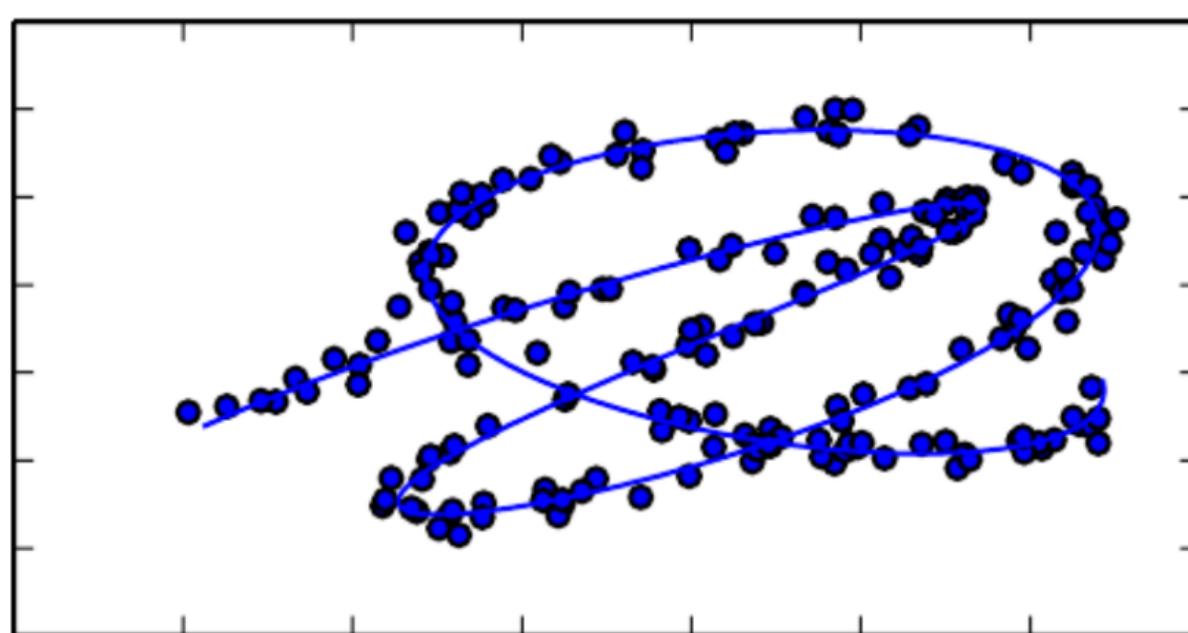
→ As the number of dimension increases, the number of possible relevant configurations increases **exponentially**, and the number of training examples to learn about them increases too !

Deep Learning



From *Deep Learning*
(Ian Goodfellow,
Yoshua Bengio and
Aaron Courville, 2016)

→ As the number of dimension increases, the number of possible relevant configurations increases **exponentially**, and the number of training examples to learn about them increases too !



→ Learning incrementally abstract representations may help us discover the underlying **manifold**

Acknowledgements & References

From the slides by **Florence d'Alché-Buc** and by **Alexandre Allauzen**

Some references:

- *The Deep Learning Book*, Chapters 5, 6, 7, 8, 11, 14 and 15 !
- Hugo Larochelle's class.
- Y. Bengio et al (2007). *Greedy Layer-Wise Training of Deep Networks* - NIPS
- X. Glorot et al (2011). *Deep Sparse Rectifier Neural Networks* - JMLR
- N. Srivastava et al (2014). *Dropout: a simple way to prevent neural networks from overfitting* - JMLR
- Duchi, J., Hazan, E., & Singer, Y. (2011). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization* - JMLR
- Zeiler, M. D. (2012). *ADADELTA: An Adaptive Learning Rate Method*
- Kingma, D. P., & Ba, J. L. (2015). *Adam: a Method for Stochastic Optimization* - ICLR