

# Deep Learning

---

Romain Tavenard (Université de Rennes)

A course @EDHEC

# Administrative details

---

- 15 hours
- Instructor: Romain Tavenard  
[romain.tavenard@univ-rennes2.fr](mailto:romain.tavenard@univ-rennes2.fr)
- Webpage:  
[rtavenar.github.io/teaching/deep\\_edhec/](https://rtavenar.github.io/teaching/deep_edhec/)
- Tool: Deepnote (create an account)
- Evaluation
  - Multiple Choice Questionnaire + Lab session

# Contents

---

- Intro to deep learning
- Fully-connected models
- Images & ConvNets
- (Sequences, Generative models, ...)

# Some slides are more important than others...

---

- Slides marked with this symbol:



Are considered basic knowledge required to pass the exams

# An introduction to deep learning

---

Romain Tavenard (Université de Rennes)

A course @EDHEC

# What can deep learning do?

- Skin cancer image classification  
130 000 images  
Error rate : 28 % (human expert 34 %)

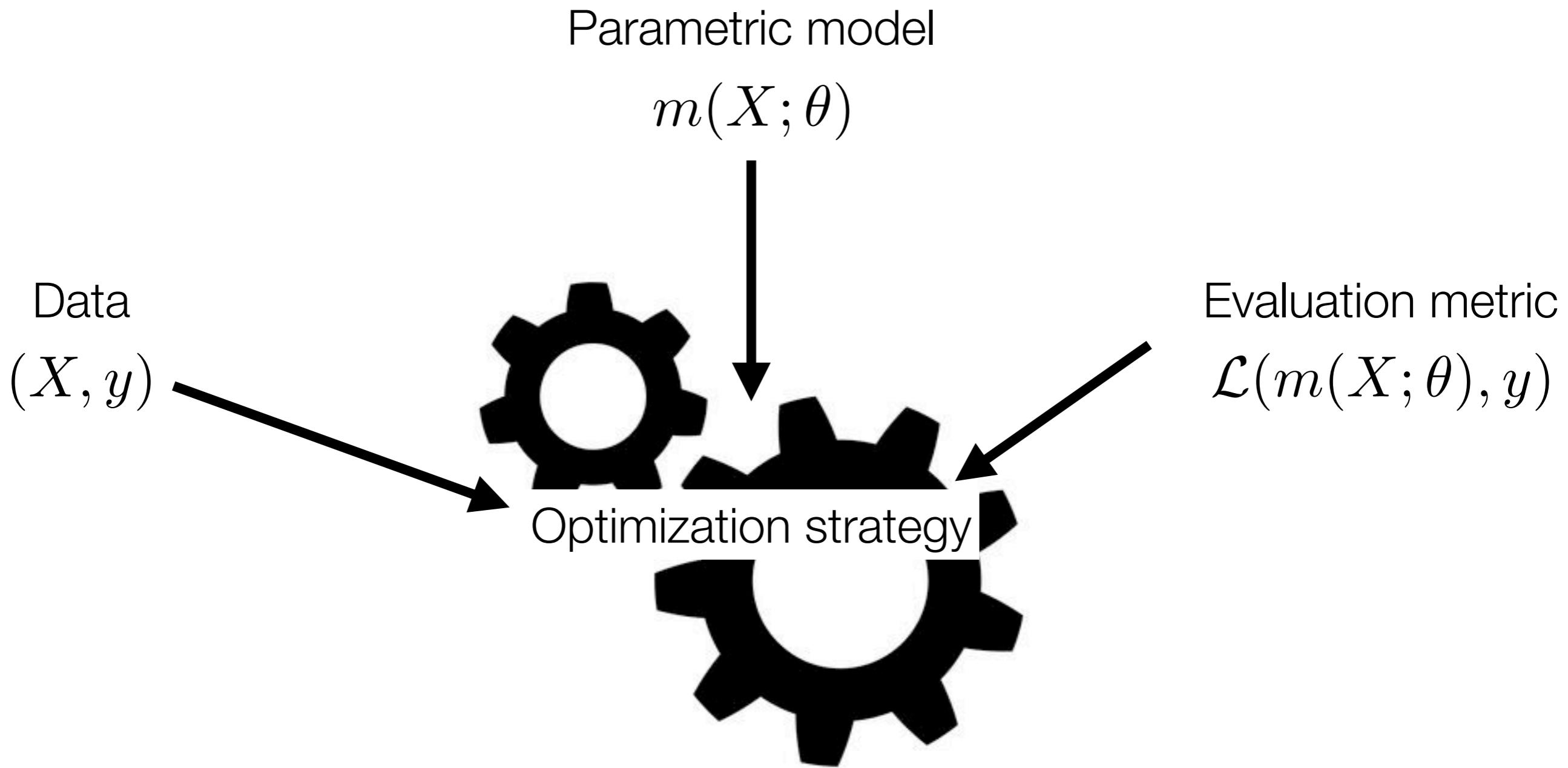


- ECG signal classification  
500 000 ECG  
Precision 92.6 %  
(human expert 80.0 %)



# Deep learning in a nutshell

---



# A first example: Linear / logistic regression

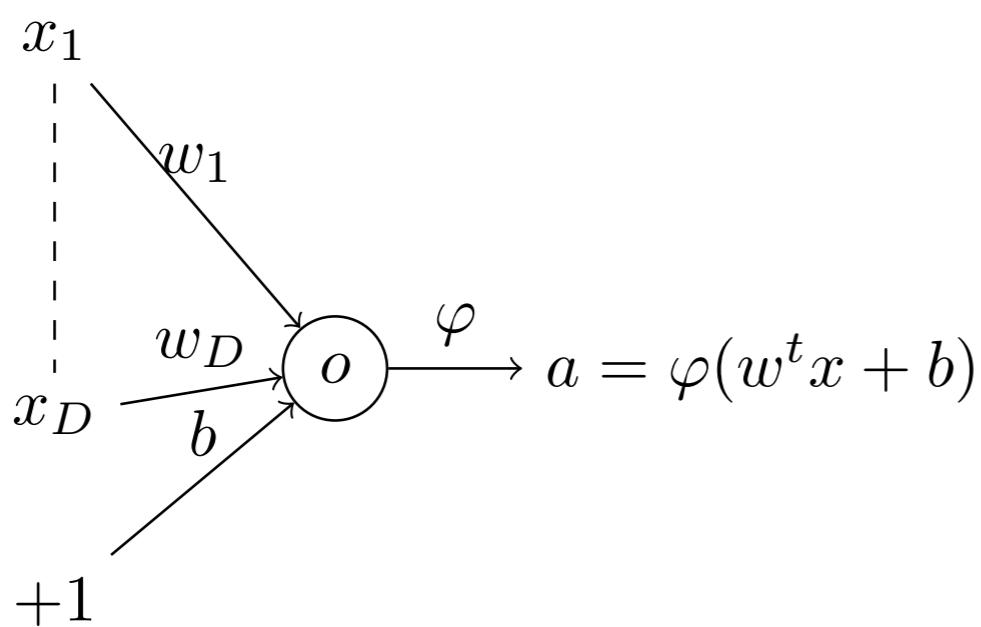
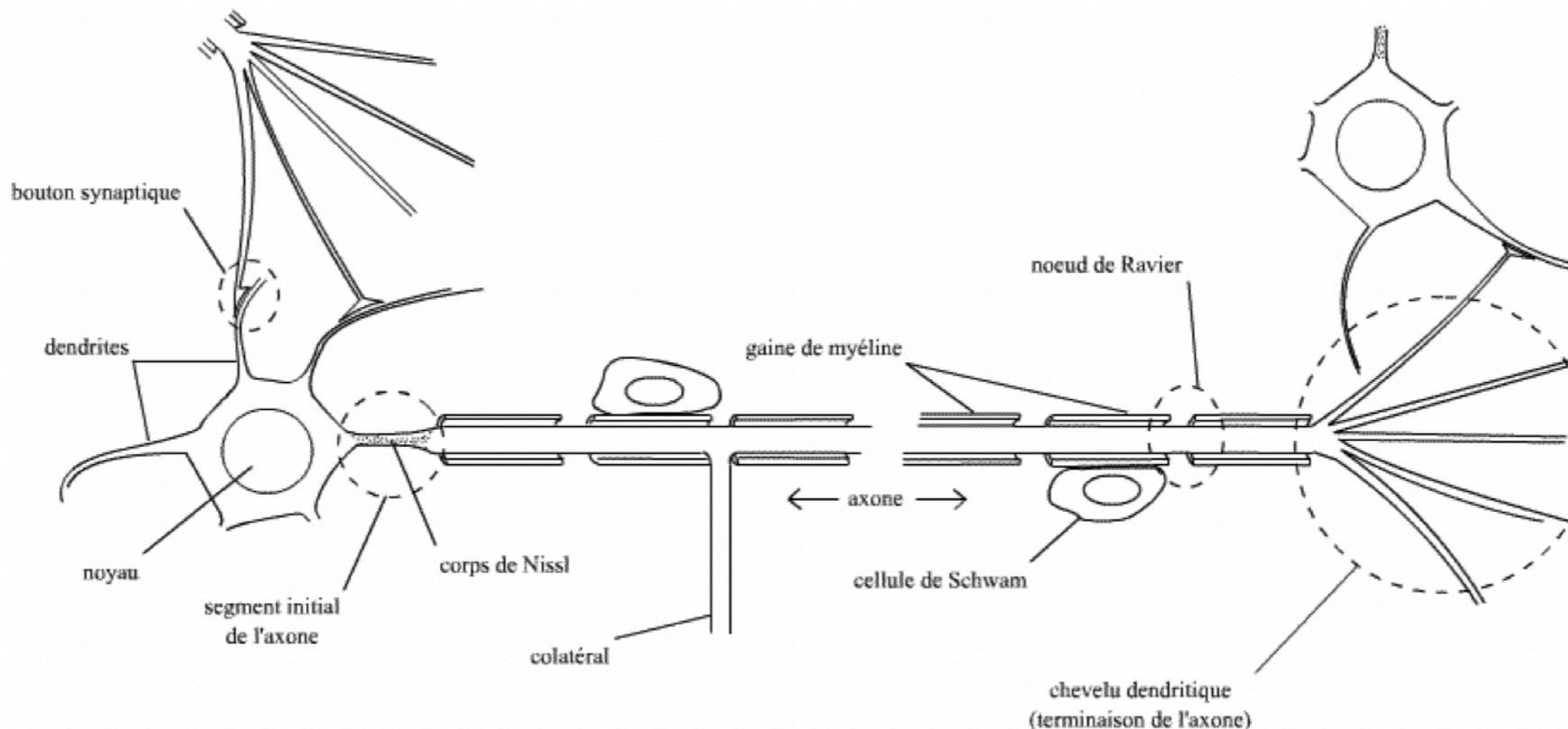
---

- Linear regression
  - **Data:** tabular data with features and targets
  - **Model:** predict output as linear combination of inputs
  - **Loss:** Mean Squared Error
- Logistic regression
  - **Data:** categorical targets
  - **Model:** linear + activation function
  - **Loss:** Cross-entropy (aka logistic loss)

# Our first model: the Perceptron

## Formal neuron by (McCulloch & Pitts, 1943)

---



$\varphi$  activation function

$a$  neuron response

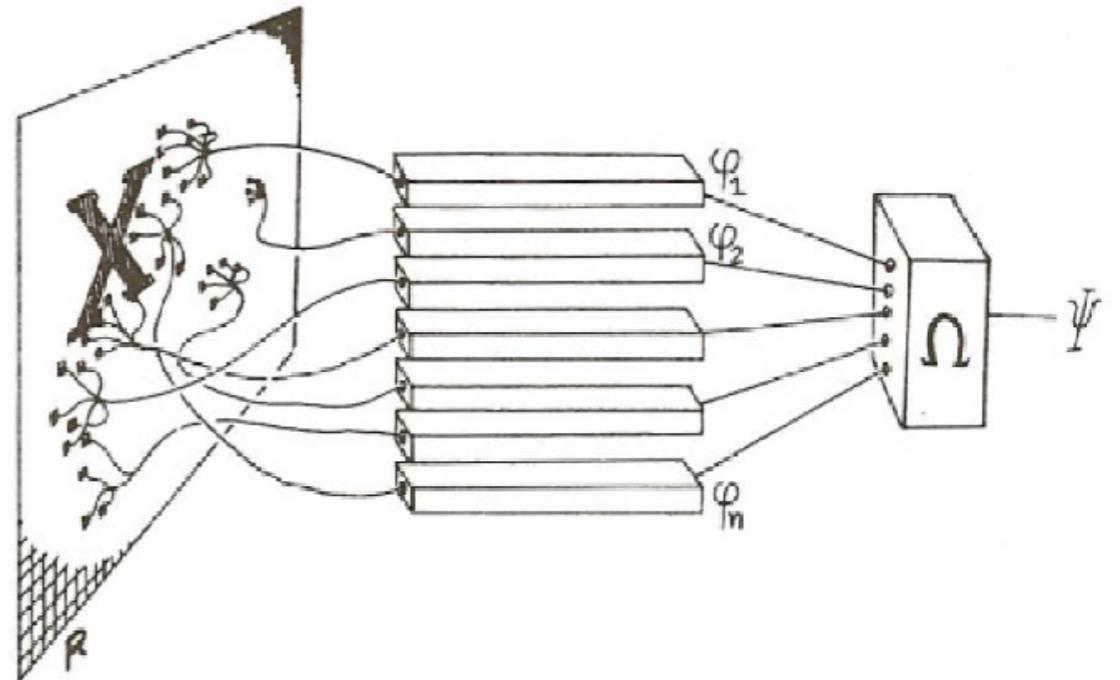
$w, b$  weight, bias

# Learning with the Perceptron

(Rosenblatt, 1957)

---

- Problem statement
  - Given pairs of input-output data  $x_i, y_i$
  - Find  $w$  such that:
$$\forall i, \varphi(w^t x_i) \approx y_i$$



Source : (Minsky & Papert, 1969)

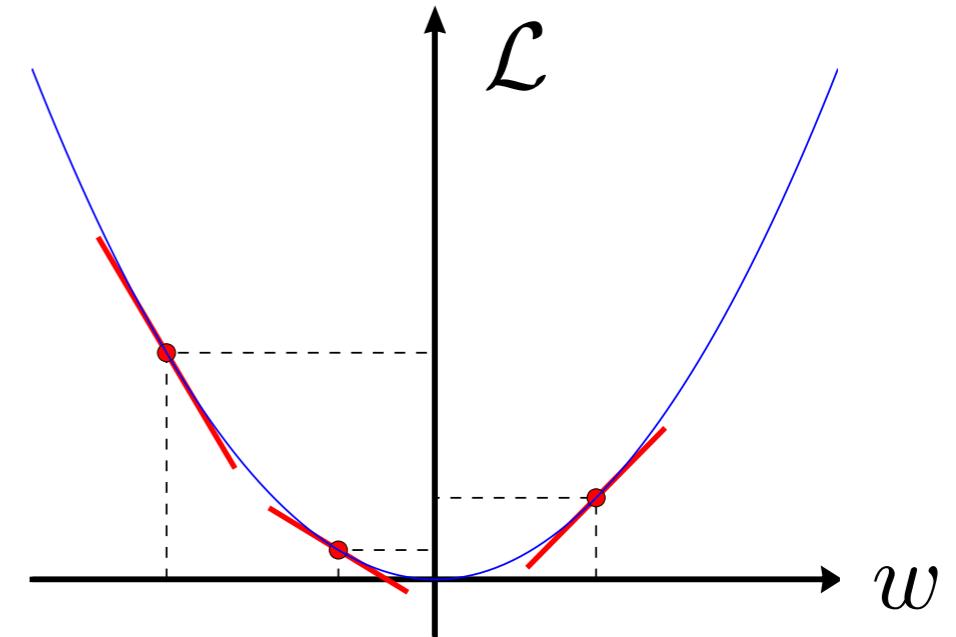
- To do so:
  - Gradient descent

# General optimization strategy: Gradient descent



1. Pick a (differentiable) loss function to be minimized

$$\begin{aligned}\text{eg. } \mathcal{L}(w, \{x_i, y_i\}) &= \frac{1}{n} \sum_{i=1}^n \mathcal{L}_i(w, x_i, y_i) \\ &= \frac{1}{n} \sum_{i=1}^n (\varphi(w^t x_i) - y_i)^2\end{aligned}$$



2. Use gradient descent

---

#### Algorithm 1: Gradient Descent

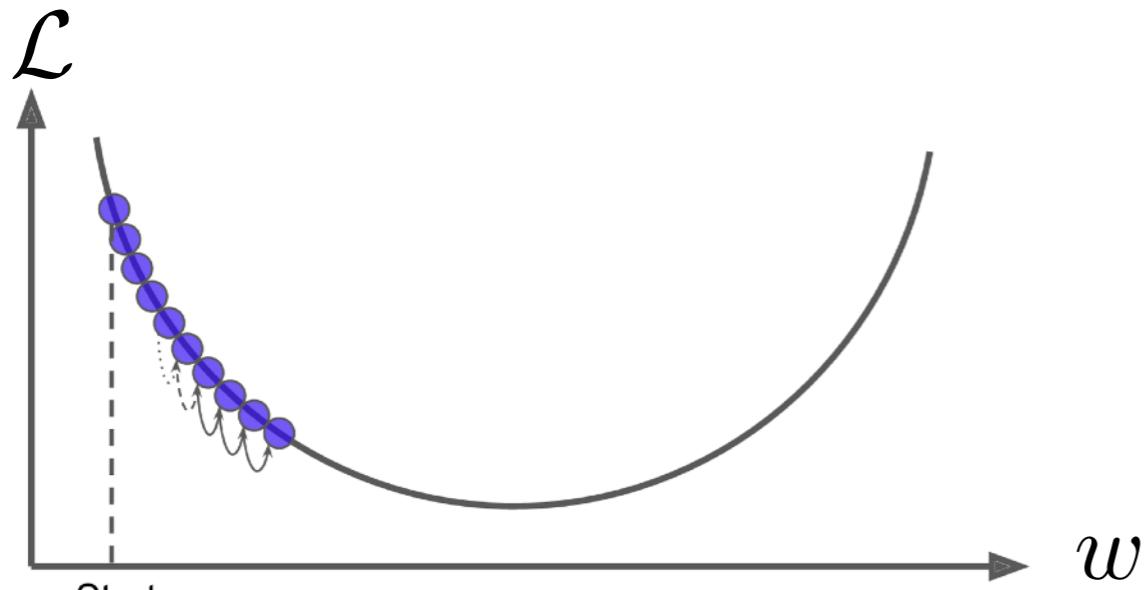
---

```
Data:  $\mathcal{D}$ : a dataset  
Initialize weights  
for  $e = 1..E$  do  
    // e is called an epoch  
    for  $(x_i, y_i) \in \mathcal{D}$  do  
        | Compute prediction  $\hat{y}_i = h(x_i)$   
        | Compute gradient  $\nabla_w \mathcal{L}_i$   
    end  
    | Compute overall gradient  $\nabla_w \mathcal{L} = \frac{1}{n} \sum_i \nabla_w \mathcal{L}_i$   
    | Update parameter  $w$  using  $\nabla_w \mathcal{L}$   
end
```

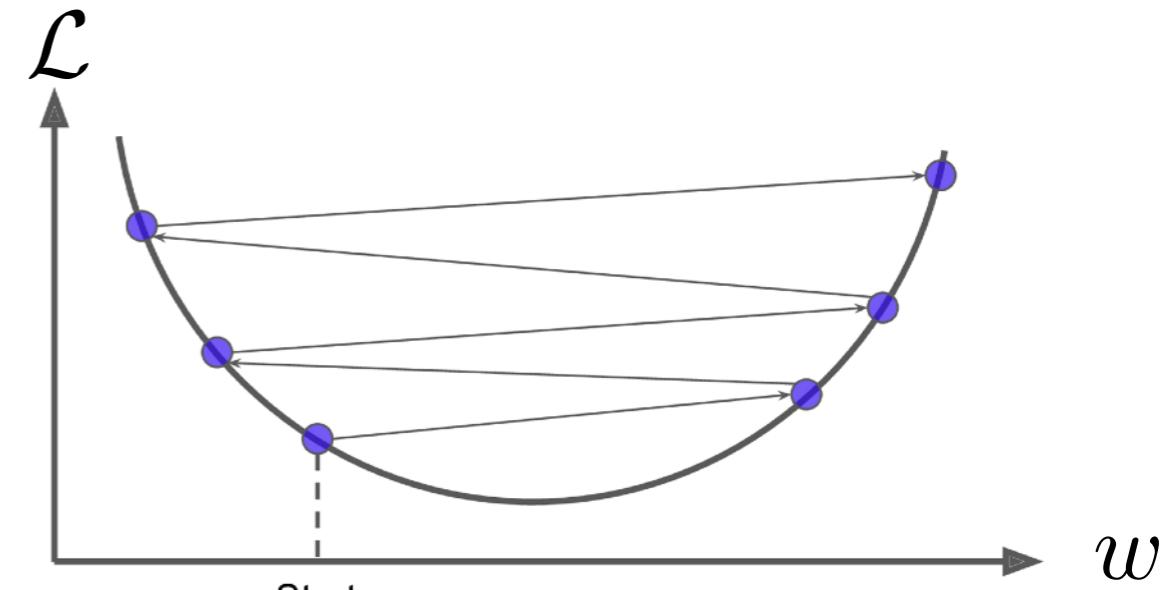
---

# Optimization

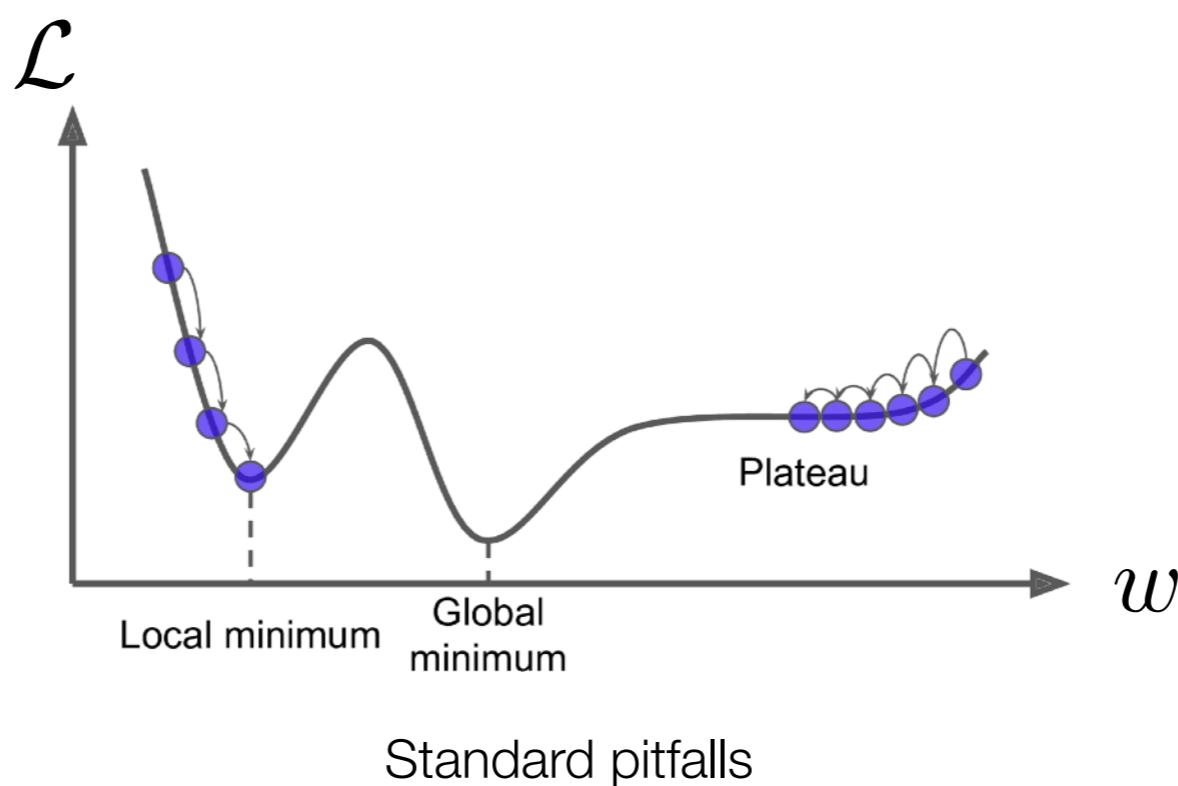
## Gradient descent in Real Life



Learning rate is too small



Learning rate is too large



Source: "Hands-On Machine Learning with Scikit-Learn and TensorFlow", A. Géron

# Optimization

## Stochastic Gradient Descent



---

**Algorithm 1:** Gradient Descent

---

**Data:**  $\mathcal{D}$ : a dataset  
Initialize weights  
**for**  $e = 1..E$  **do**  
    // e is called an epoch  
    **for**  $(x_i, y_i) \in \mathcal{D}$  **do**  
        Compute prediction  $\hat{y}_i = h(x_i)$   
        Compute gradient  $\nabla_w \mathcal{L}_i$   
    **end**  
    Compute overall gradient  $\nabla_w \mathcal{L} = \frac{1}{n} \sum_i \nabla_w \mathcal{L}_i$   
    Update parameter  $w$  using  $\nabla_w \mathcal{L}$   
**end**

---

---

**Algorithm 2:** Mini-Batch Stochastic Gradient Descent

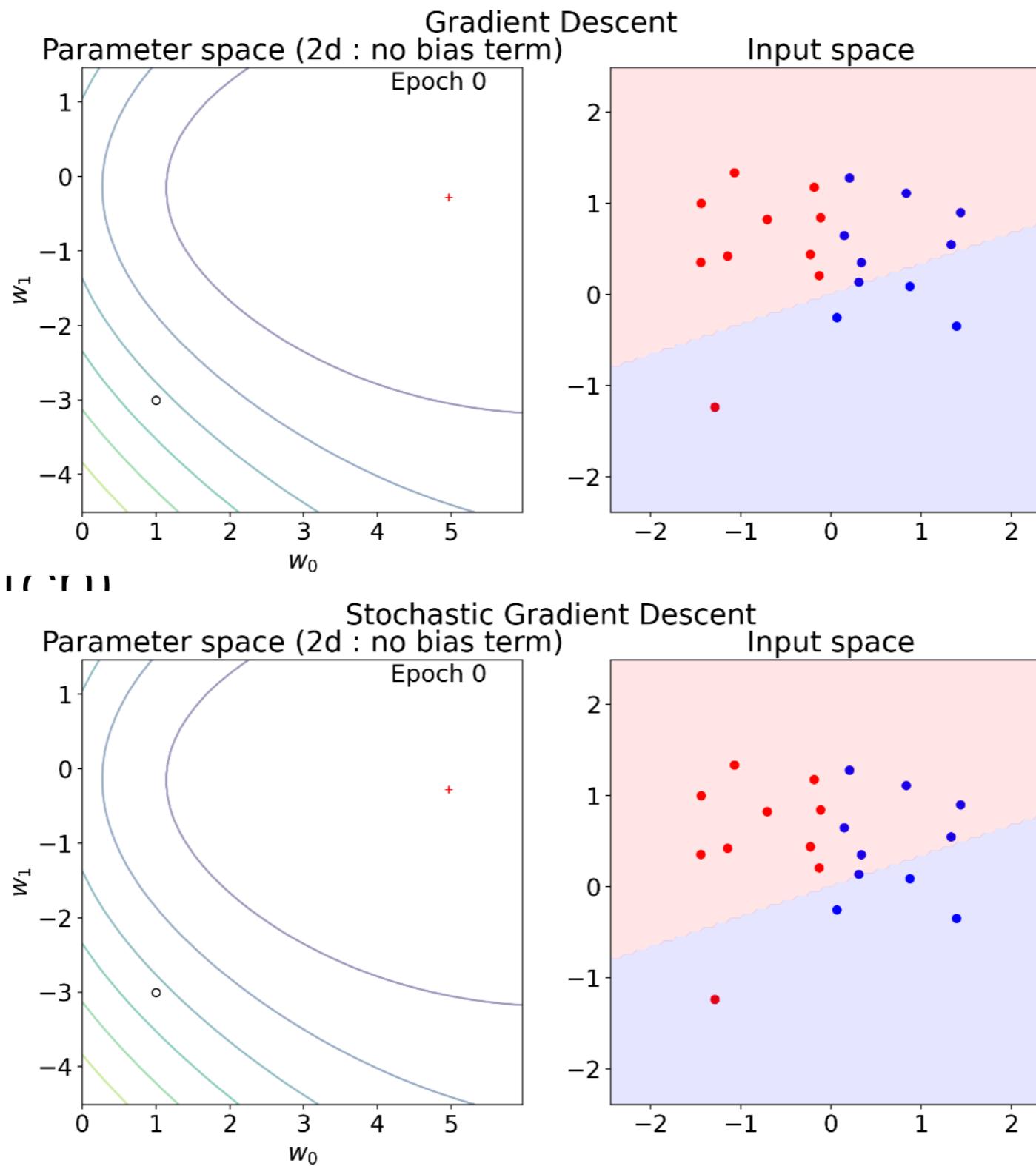
---

**Data:**  $\mathcal{D}$ : a dataset  
Initialize weights  
**for**  $e = 1..E$  **do**  
    // e is called an epoch  
    **for**  $t = 1..n_b$  **do**  
        // t is called an iteration  
        **for**  $i = 1..m$  **do**  
            Draw  $(x_i, y_i)$  without replacement from  $t$ -th minibatch of  $\mathcal{D}$   
            Compute prediction  $\hat{y}_i = h(x_i)$   
            Compute gradient  $\nabla_w \mathcal{L}_i$   
        **end**  
        Compute gradient for the  $t$ -th minibatch  $\nabla_w \mathcal{L}_{(t)} = \frac{1}{m} \sum_i \nabla_w \mathcal{L}_i$   
        Update parameter  $w$  using  $\nabla_w \mathcal{L}_{(t)}$   
    **end**  
**end**

---

# Optimization: Gradient Descent vs Stochastic Gradient Descent (1/2)

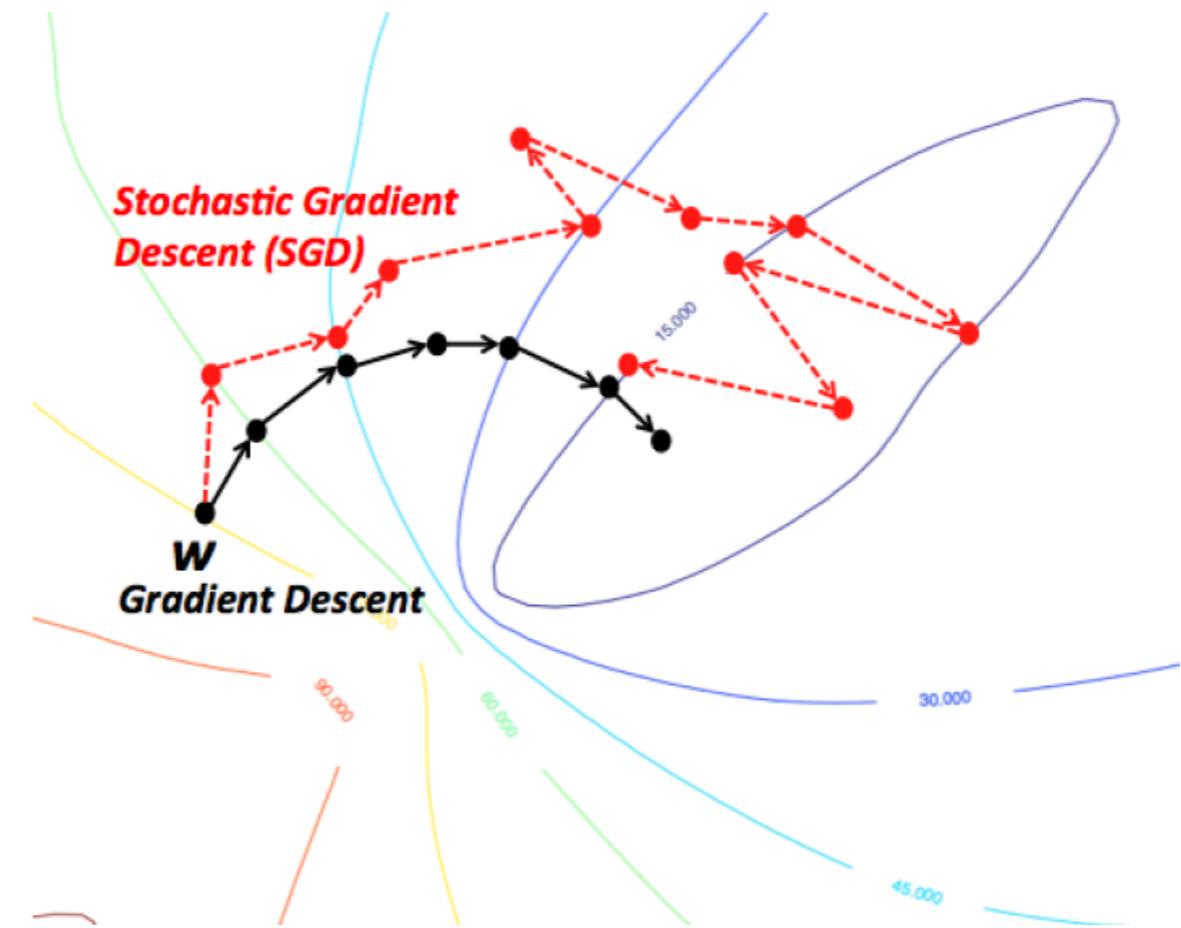
- Cons
  - Subject to local minima
- Pros
  - Faster (each mini batch update)
- Escapes non-convexity



# Optimization: Gradient Descent vs Stochastic Gradient Descent (2/2)



- Cons
  - Subject to high variance
- Pros
  - Faster weight update (each sample, or each mini batch)
  - Escape local minima in non-convex settings



Source: [wikidocs.net/3413](http://wikidocs.net/3413)

# Optimization

## SGD variants: a focus on Adam

---

- Adam uses ideas from
  - Momentum [[link to distill](#)]
  - AdaGrad

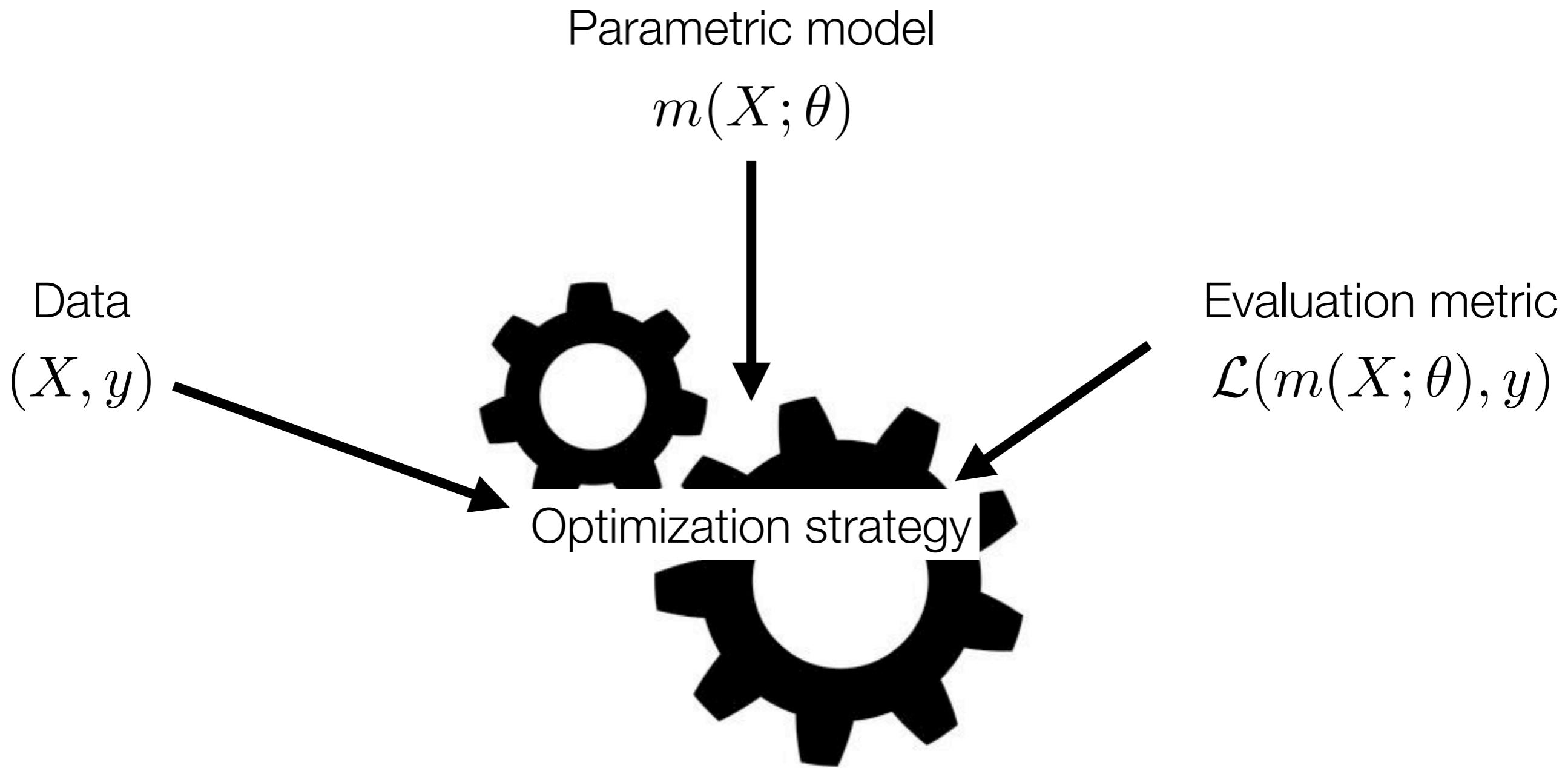
$$\mathbf{m}^{(t+1)} \propto \beta_1 \mathbf{m}^{(t)} + (1 - \beta_1) \nabla_w \mathcal{L}$$

$$\mathbf{s}^{(t+1)} \propto \beta_2 \mathbf{s}^{(t)} + (1 - \beta_2) \nabla_w \mathcal{L} \otimes \nabla_w \mathcal{L}$$

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \rho \mathbf{m}^{(t+1)} \oslash \sqrt{\mathbf{s}^{(t+1)} + \epsilon}$$

# Deep learning in a nutshell

---

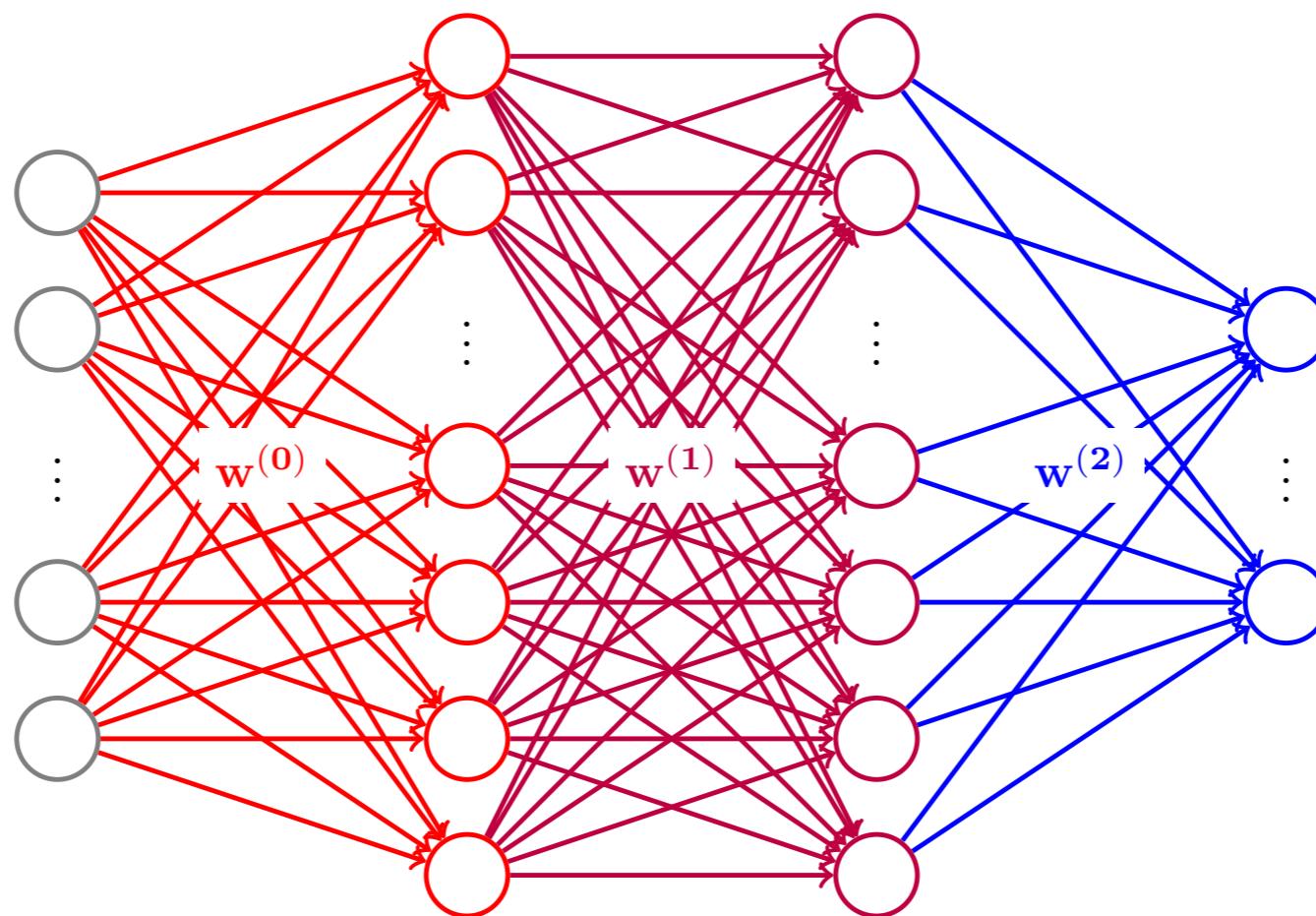


# Multi-Layer Perceptron (MLP) model (Rumelhart, Hinton & Williams, 1985)

---

## Definition

A Multilayer perceptron is an acyclic graph of neurons, where neurons are structured in successive layers, beginning by an input layer and finishing with an output layer.



# Universal approximation theorem (Cybenko, 1989)

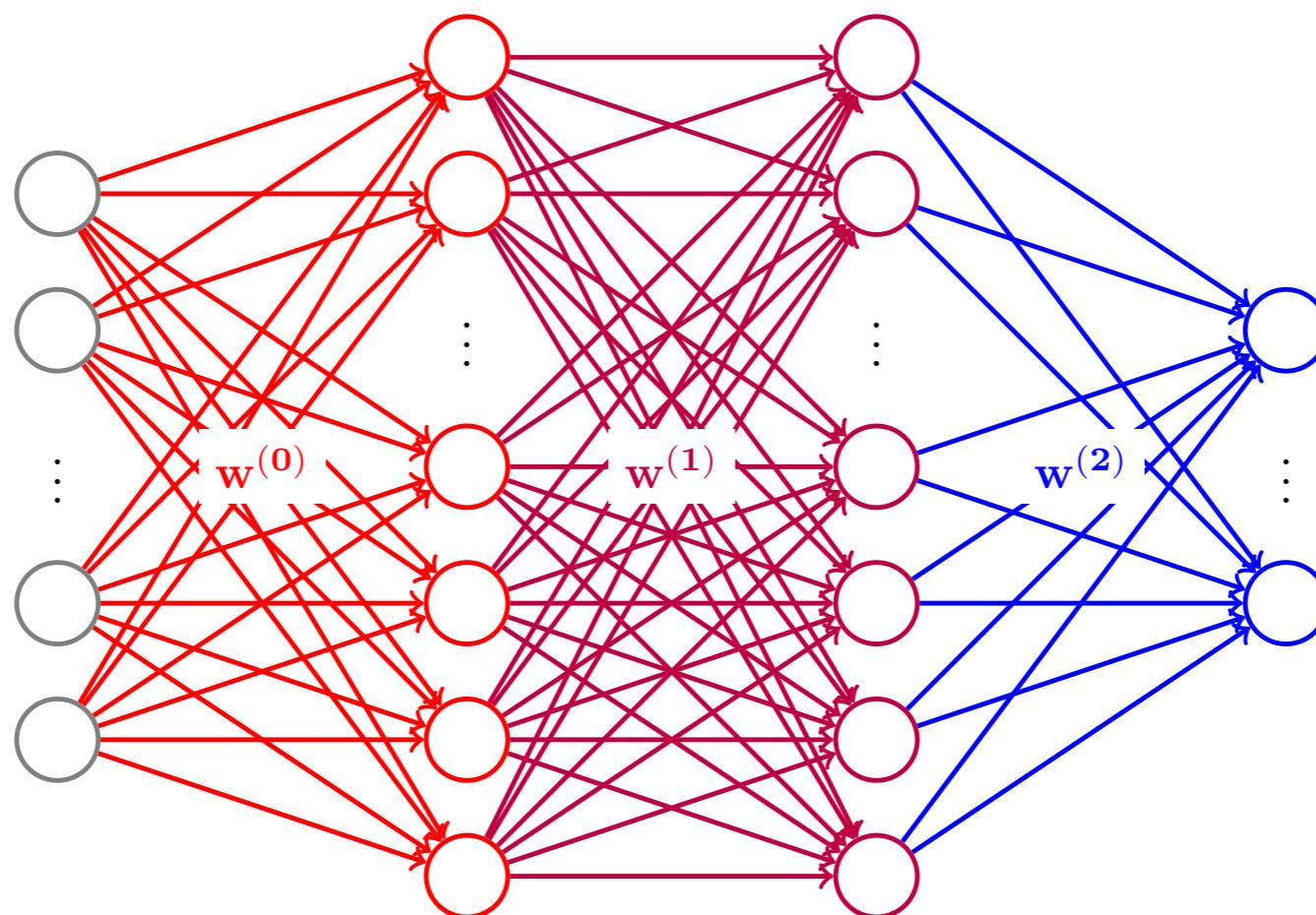
---

- Under reasonable assumptions on the activation function to be used\*
- For any continuous function on a compact  $g$  and any precision threshold  $\epsilon$
- There exists a 1-hidden-layer MLP with a finite number of neurons that can approximate  $g$  at level  $\epsilon$

\*Non-constant, bounded, monotonically increasing, continuous  
also holds for some unbounded functions like ReLU, cf. [Somoda & Murata, 2015]

# Optimizing multi-layer perceptron parameters

- Who wants to compute gradients by hand for such networks (and deeper ones)?

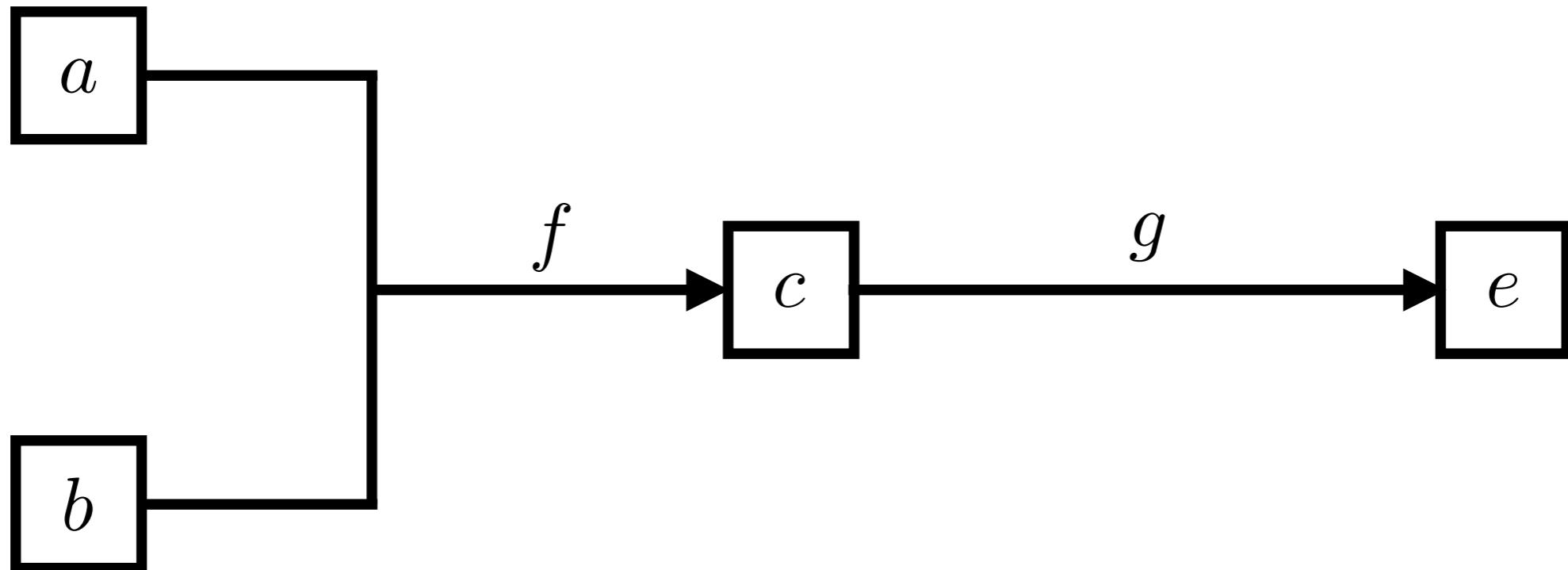


$$\hat{\mathbf{y}} = \varphi \left[ \mathbf{w}^{(2)} \varphi \left( \mathbf{w}^{(1)} \varphi(\mathbf{w}^{(0)} \mathbf{x} + b^{(0)}) + b^{(1)} \right) + b^{(2)} \right]$$

# Optimizing multi-layer perceptron parameters

## Automatic differentiation to the rescue!

---

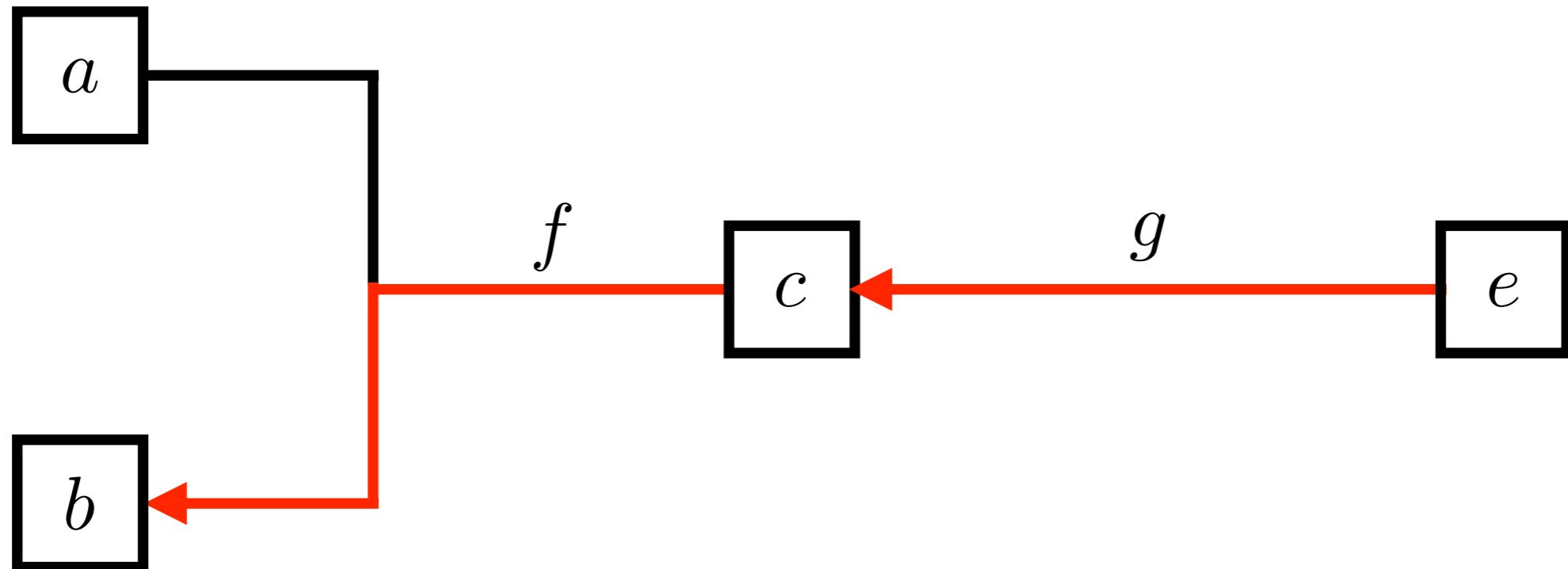


$$c = f(a, b)$$

$$e = g(c)$$

# Optimizing multi-layer perceptron parameters

## Automatic differentiation to the rescue!



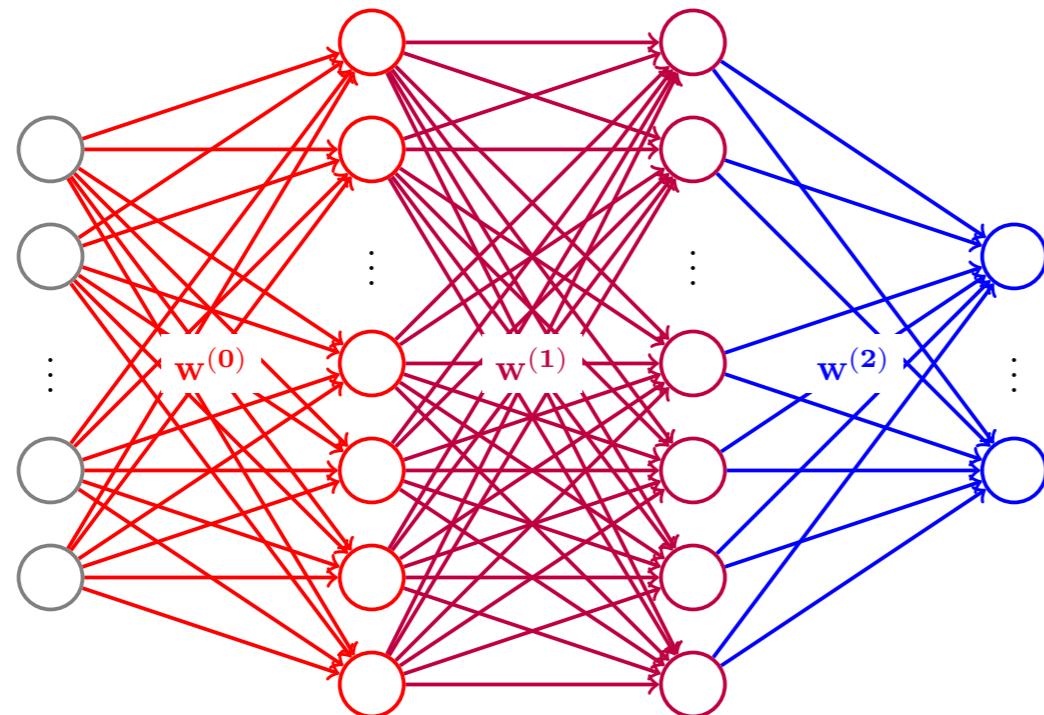
$$c = f(a, b)$$

$$e = g(c)$$

$$\frac{\partial e}{\partial b} = \underbrace{\left. \frac{\partial e}{\partial c} \right|_{c=c_0}}_{g'(c_0)} \cdot \left. \frac{\partial c}{\partial b} \right|_{b=b_0}$$

# Optimization

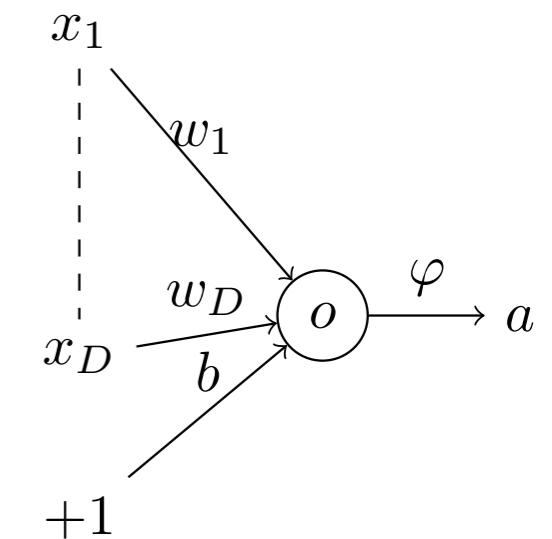
## Neural networks and back-propagation



$$\frac{\partial \mathcal{L}}{\partial w^{(2)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial w^{(2)}}$$

$$\frac{\partial \mathcal{L}}{\partial w^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial w^{(1)}}$$

$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}$$



$$\frac{\partial a^{(l)}}{\partial o^{(l)}} = \varphi'(o^{(l)})$$

$$\frac{\partial o^{(l)}}{\partial a^{(l-1)}} = w^{(l-1)}$$

# Neural networks and back-propagation: Losses



$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}$$

- Requirement
  - $\mathcal{L}$  should be differentiable wrt. to the net's output
- Standard losses
  - Mean Squared Error (MSE) for regression
$$\mathcal{L}(x_i, y_i; \theta) = (m(x_i; \theta) - y_i)^2$$
  - Cross-entropy for classification
$$\mathcal{L}(x_i, y_i; \theta) = -\log P_\theta(y = y_i | x_i)$$

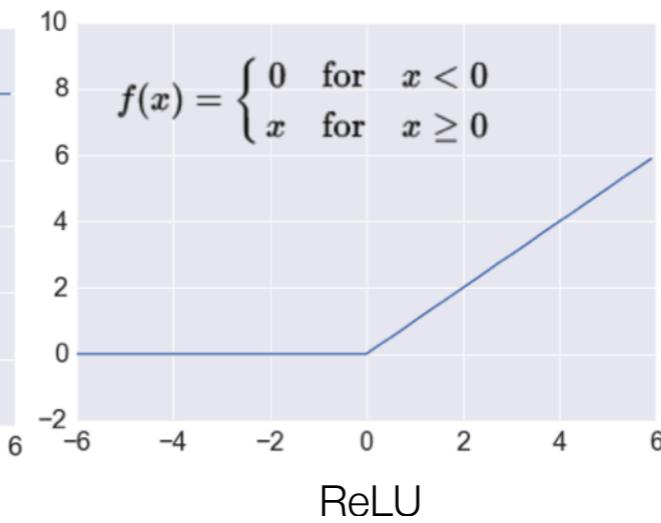
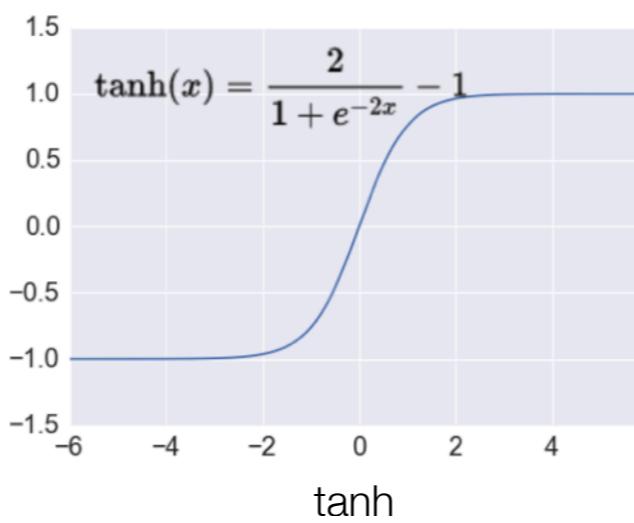
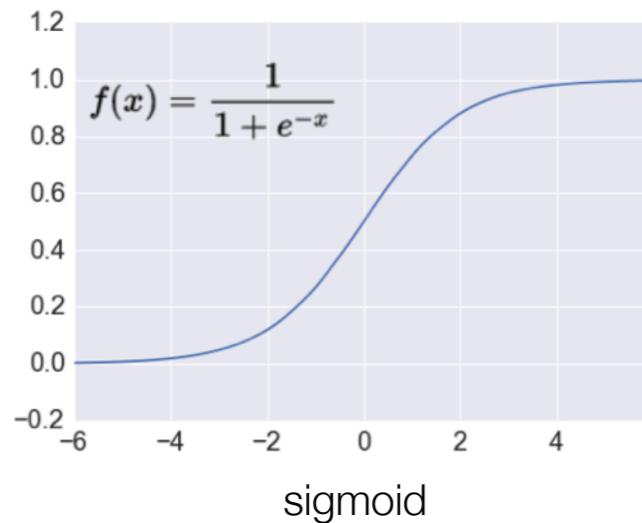
# Neural networks and back-propagation: Activation functions



$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}$$

$$\frac{\partial a^{(l)}}{\partial o^{(l)}} = \varphi'(o^{(l)})$$

- Important features
  - $\varphi$  should be differentiable almost everywhere
  - Non-linearities
  - Some linear regime
- Examples

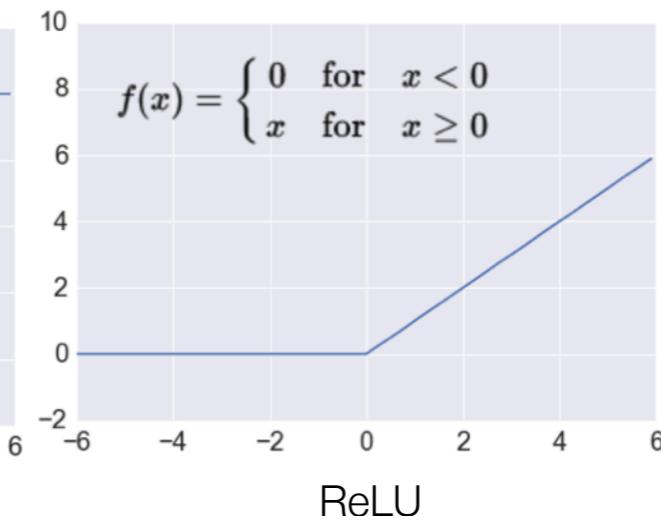
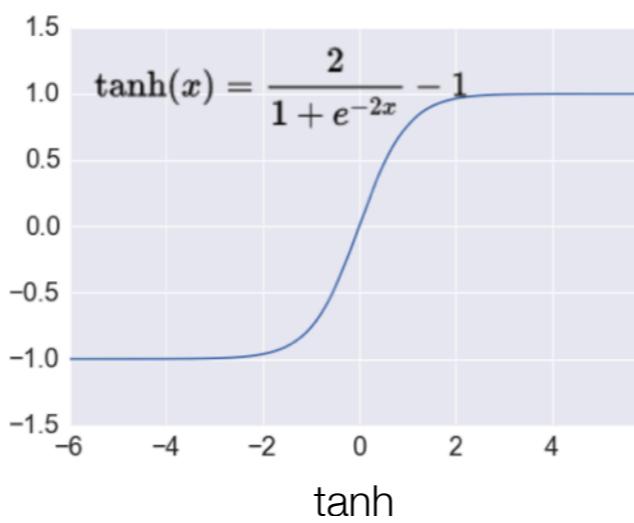
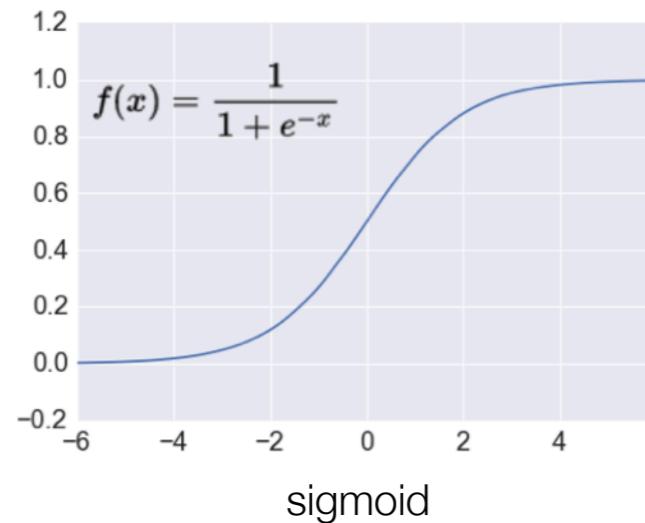


# Neural networks and back-propagation: Activation functions: the reign of ReLU



$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}$$
$$\frac{\partial a^{(l)}}{\partial o^{(l)}} = \varphi'(o^{(l)})$$

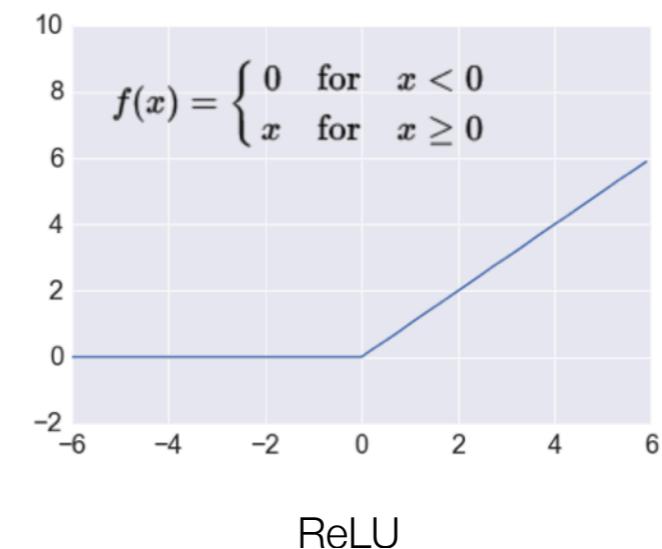
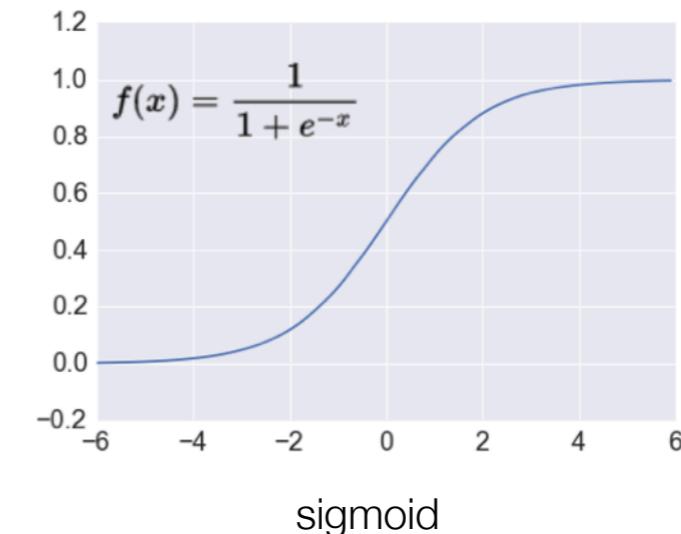
- ReLU has become the default choice over time
- 2 main reasons:
  - cheap to compute (both ReLU and its derivative)
  - vanishing gradients phenomenon (more on that later)



# Neural networks and back-propagation: activation functions: the case of the output layer



- Output activation functions drive the possible output values:
  - identity ("linear" in keras): any real value
  - ReLU: any positive value
  - sigmoid: any value in  $[0, 1]$
  - softmax:  
 $>0$  and sums to 1  
(across output neurons)



$$\text{soft-max}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

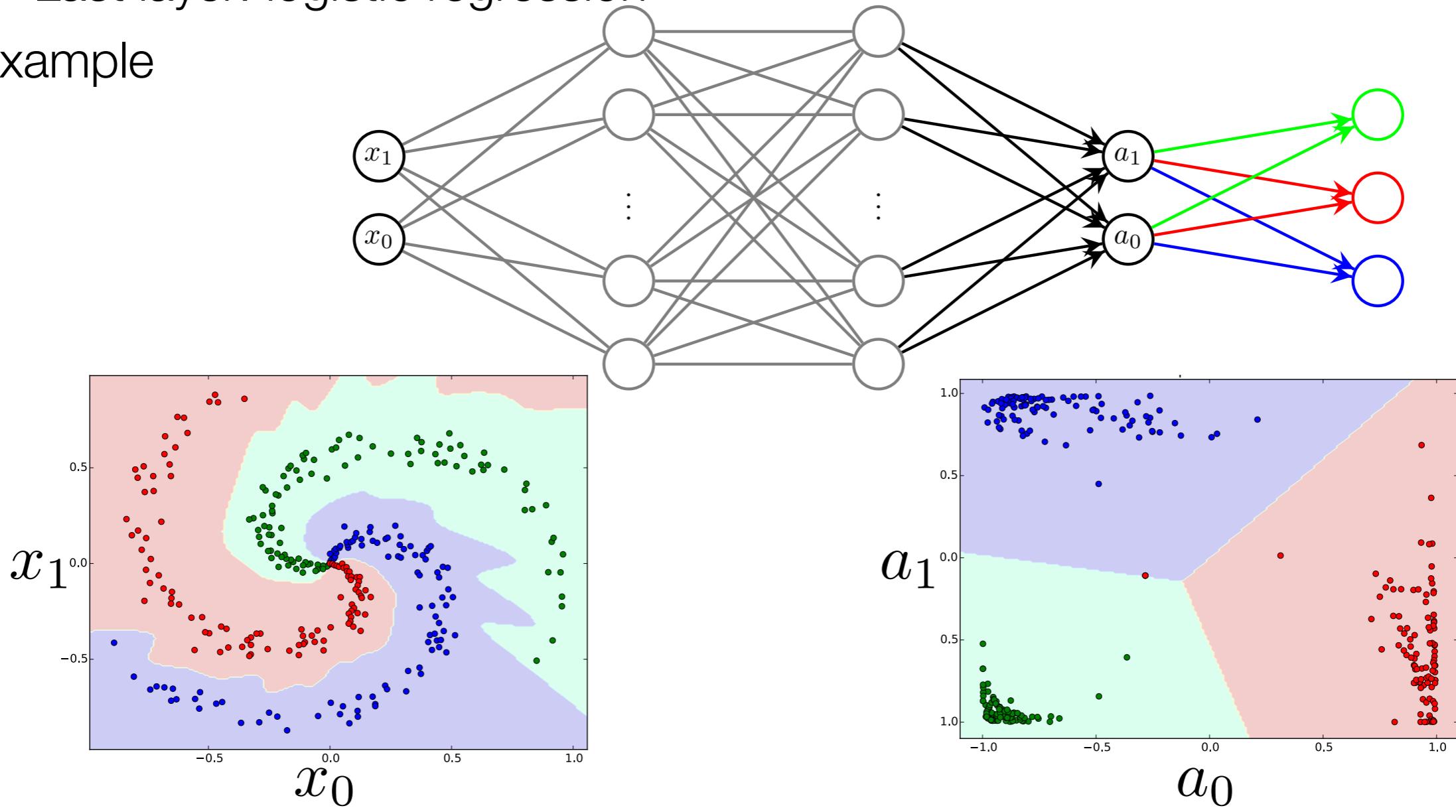
# Neural networks and back-propagation: link with keras implementation



- In keras, these considerations have practical impact:
  - Model structure:
    - Input layer dimension is the number of features in the dataset
    - Output layer has as many units as columns in y
  - Output layer activation:
    - Binary classification: "sigmoid"
    - Multiclass classification: "softmax"
  - Loss function:
    - Binary classification: "binary\_crossentropy"
    - Multiclass classification: "categorical\_crossentropy"
    - Regression: "mse"

# End-to-end learning

- Classification using MLP
  - Hidden layers: non-linear transformations
  - Last layer: logistic regression
- Example



# Optimization

## Over-parametrization in deep learning

---

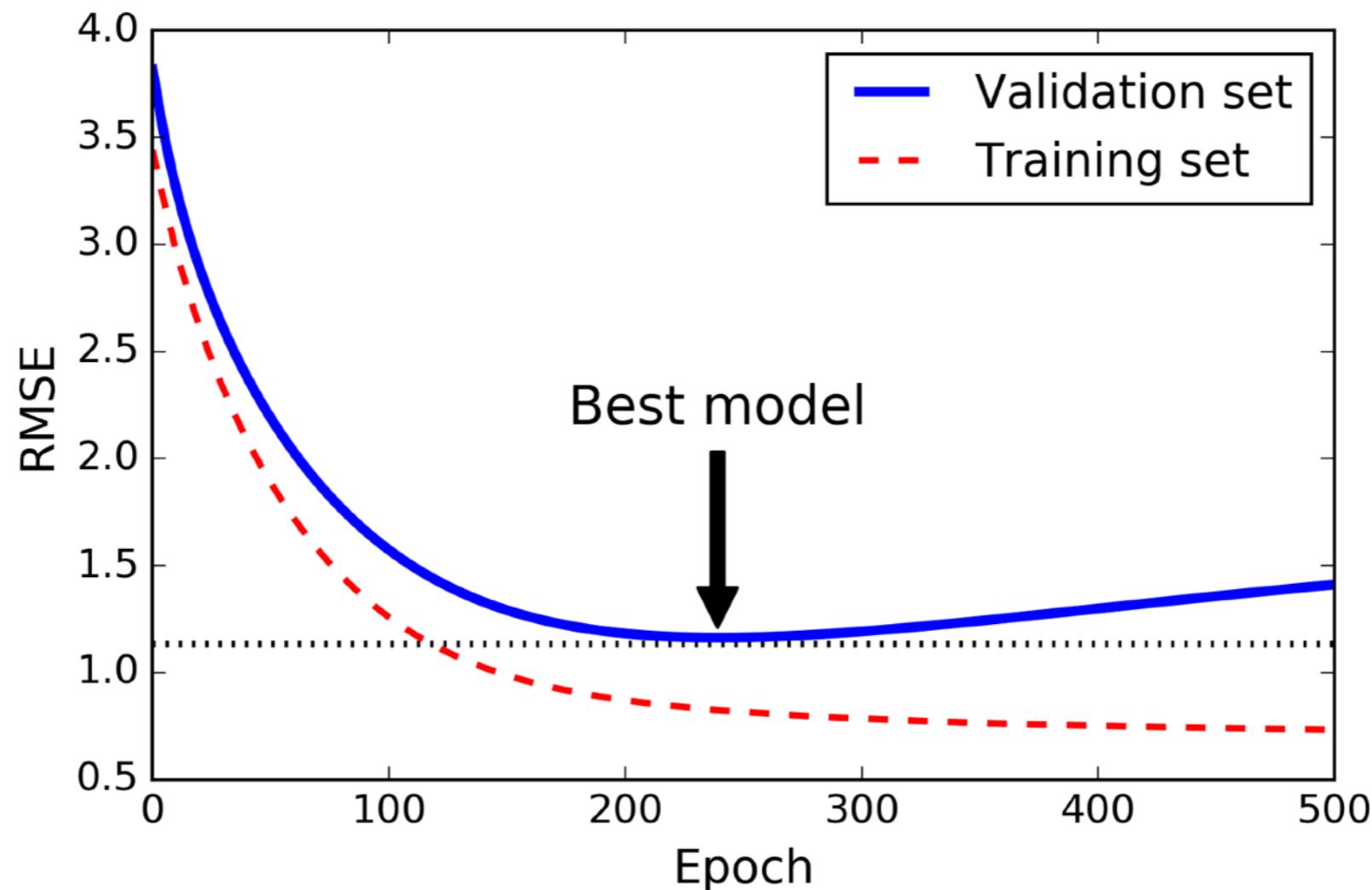
- Optimization (SGD) to minimize a loss function
  - Larger & deeper nets improve (training) performance
  - Risks over-fitting

$$\arg \min_{\theta} \sum_{(x_i, y_i) \in \mathcal{D}_t} \mathcal{L}(x_i, y_i; \theta) \neq \arg \min_{\theta} \mathbb{E}_{x, y \sim \mathcal{D}} \mathcal{L}(x, y; \theta)$$

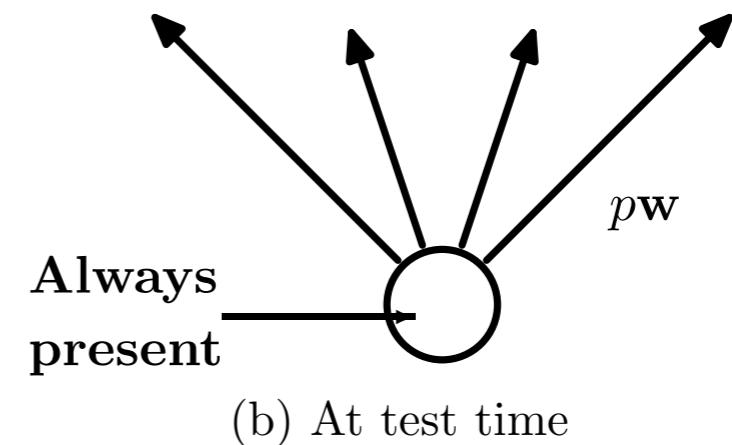
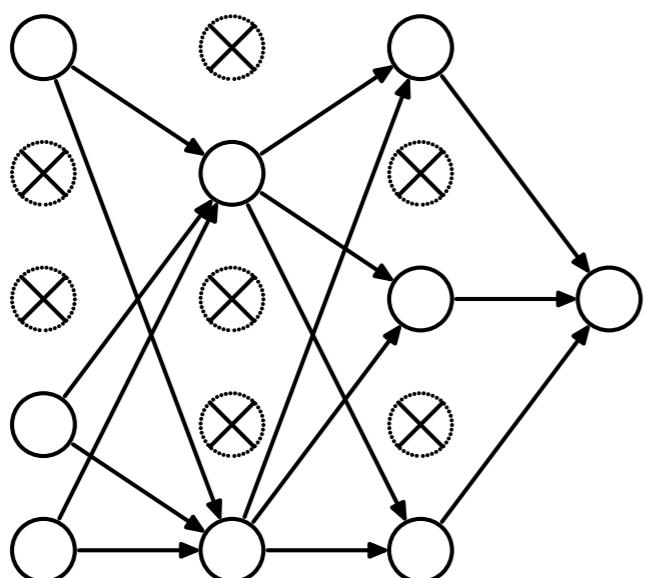
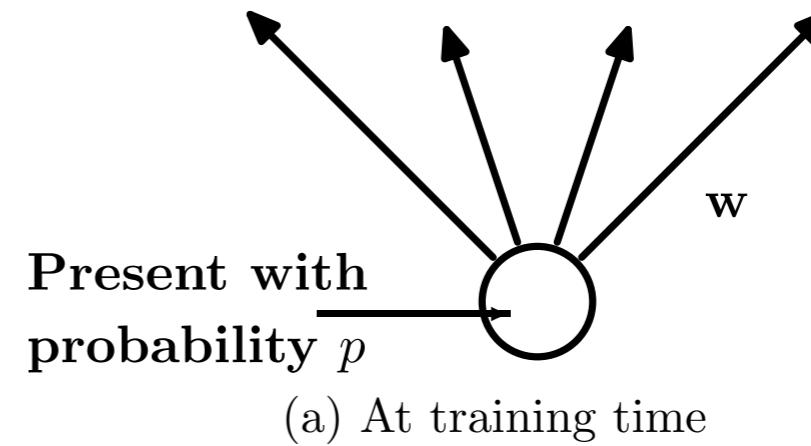
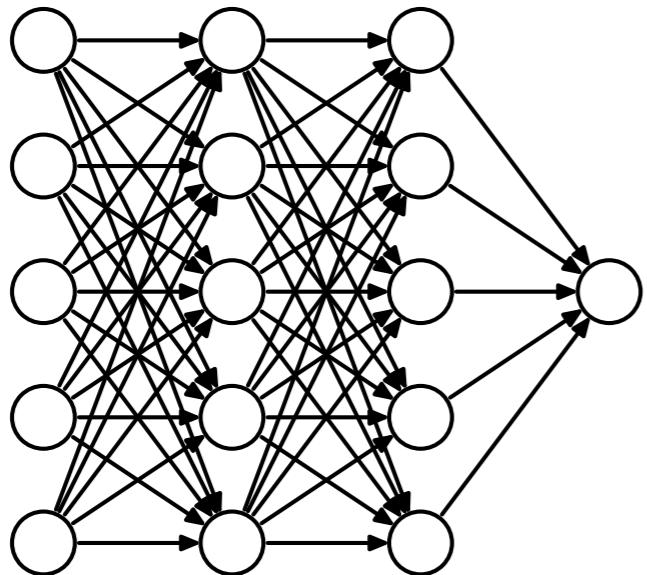
- Regularization tricks
  - L2 penalty on weights (cf. Ridge regression)
  - Early stopping (cf. Gradient boosting)
  - Dropout (relates to Random Forests)

# Optimization

## Regularization: Early Stopping



# Optimization Regularization: Dropout



Images from [Srivastava et al., 2014]

# Conclusion

---

- Early stage: 1943 - 1969
  - learning with stochastic gradient descent
- Back in the game: 1985 - 1995
  - NN are universal approximators
- A *de facto* standard in computer vision: 2009 - ?
  - deep nets can leverage on big data + high perf. computers