

# Deep Learning

Lecture 2: Neural networks

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



# Today

Explain and motivate the basic constructs of neural networks.

- From linear discriminant analysis to logistic regression
- Stochastic gradient descent
- From logistic regression to the multi-layer perceptron
- Vanishing gradients and rectified networks
- Universal approximation theorem (teaser)

# **Neural networks**

# Threshold Logic Unit

The Threshold Logic Unit (McCulloch and Pitts, 1943) was the first mathematical model for a **neuron**. Assuming Boolean inputs and outputs, it is defined as:

$$f(\mathbf{x}) = \mathbf{1}_{\{\sum_i w_i x_i + b \geq 0\}}$$

This unit can implement:

- $\text{or}(a, b) = \mathbf{1}_{\{a+b-0.5 \geq 0\}}$
- $\text{and}(a, b) = \mathbf{1}_{\{a+b-1.5 \geq 0\}}$
- $\text{not}(a) = \mathbf{1}_{\{-a+0.5 \geq 0\}}$

Therefore, any Boolean function can be built with such units.

*A Logical Calculus of Ideas Immanent in Nervous Activity*

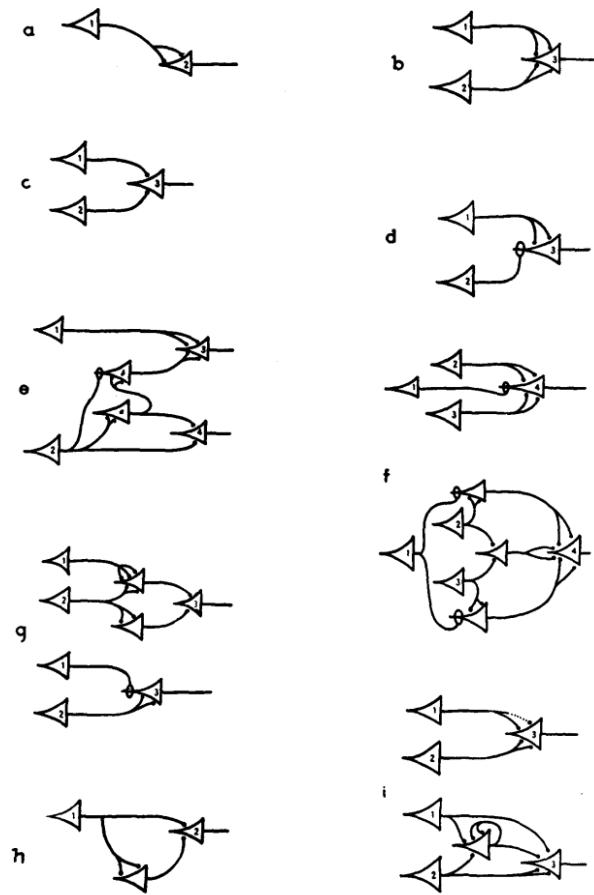


FIGURE 1

# Perceptron

The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

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

This model was originally motivated by biology, with  $w_i$  being synaptic weights and  $x_i$  and  $f$  firing rates.

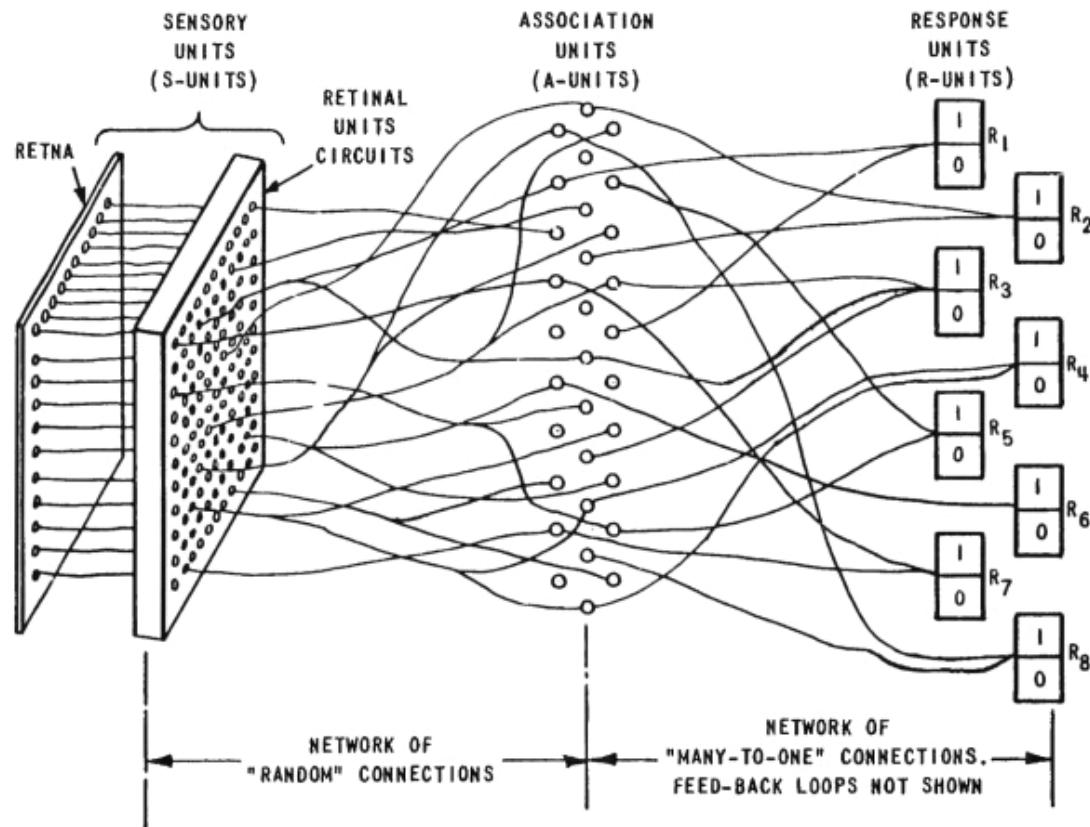
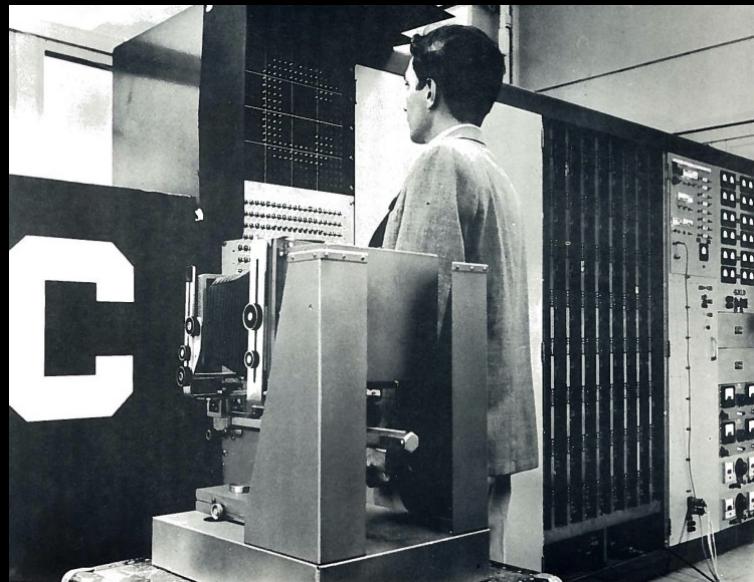
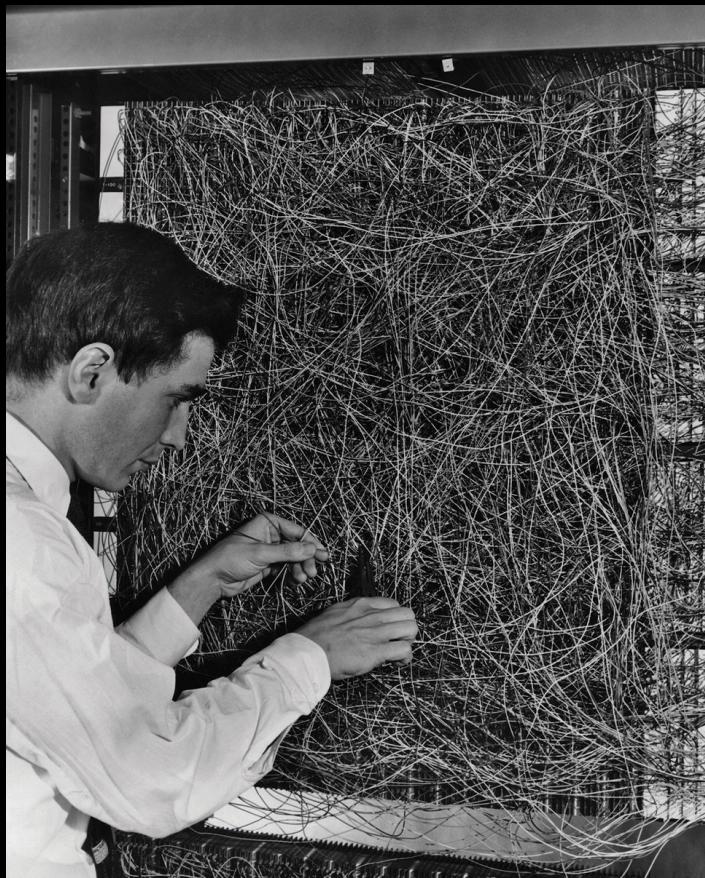


Figure I ORGANIZATION OF THE MARK I PERCEPTRON



The Mark I Perceptron (Frank Rosenblatt).



Perceptron Research from the 50's & 60's, clip



Watch later



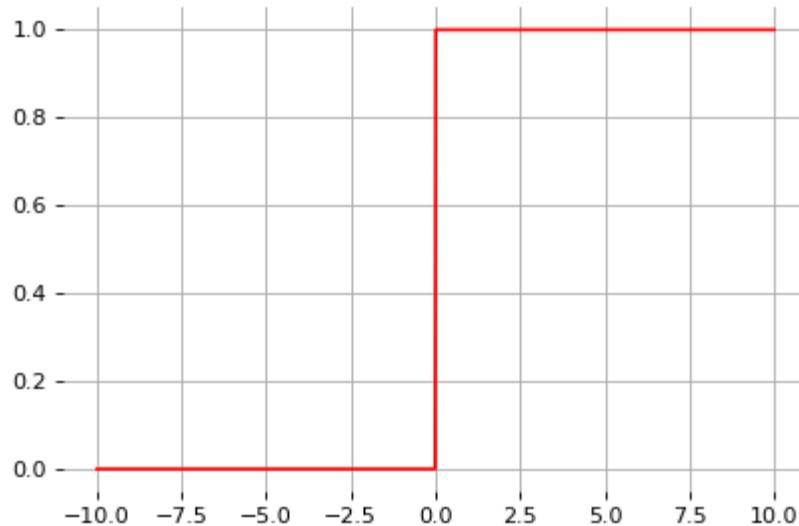
Share



The Perceptron

Let us define the (non-linear) **activation** function:

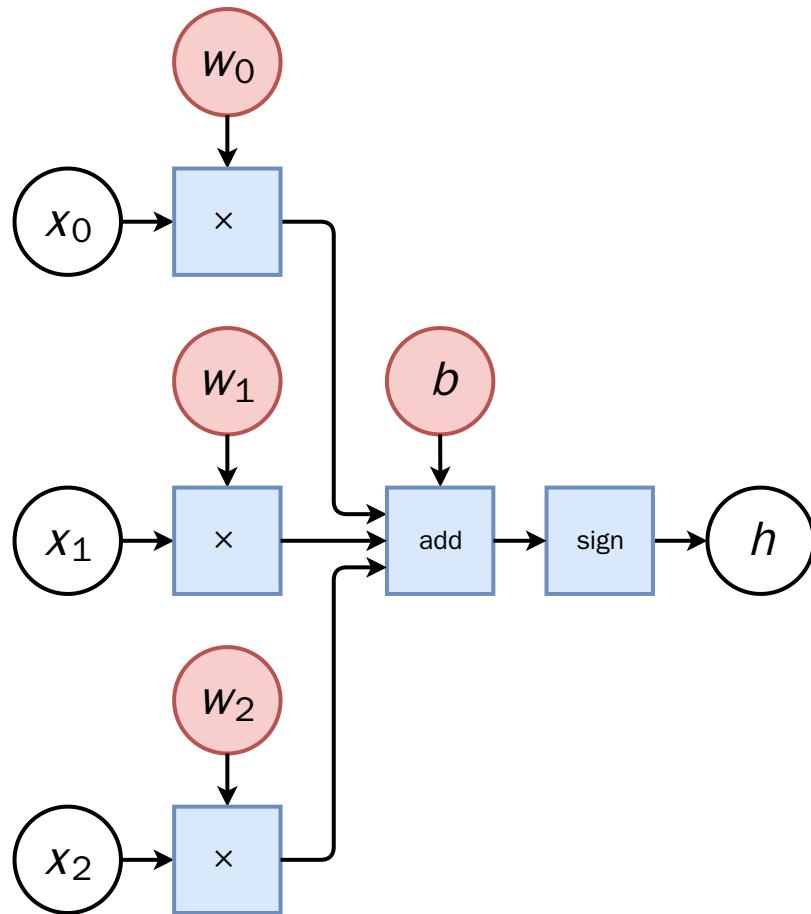
$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



The perceptron classification rule can be rewritten as

$$f(\mathbf{x}) = \text{sign}\left(\sum_i w_i x_i + b\right).$$

## Computational graphs



The computation of

$$f(\mathbf{x}) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

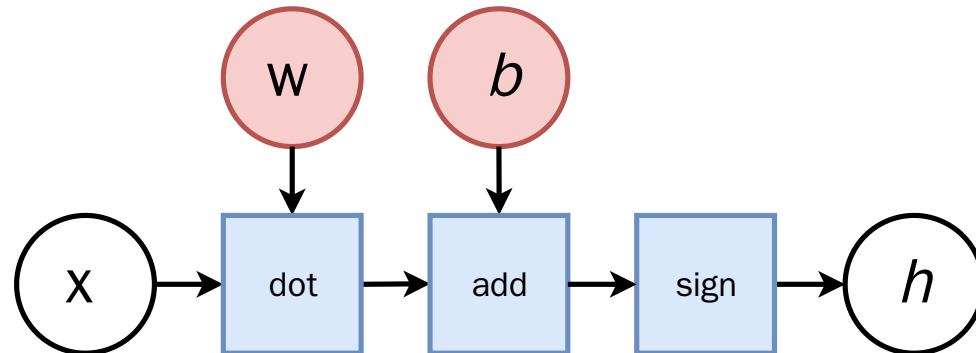
can be represented as a **computational graph** where

- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations.

In terms of **tensor operations**,  $f$  can be rewritten as

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

for which the corresponding computational graph of  $f$  is:



# Linear discriminant analysis

Consider training data  $(\mathbf{x}, y) \sim P(X, Y)$ , with

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

Assume class populations are Gaussian, with same covariance matrix  $\Sigma$  (homoscedasticity):

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

Using the Bayes' rule, we have:

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

Using the Bayes' rule, we have:

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

It follows that with

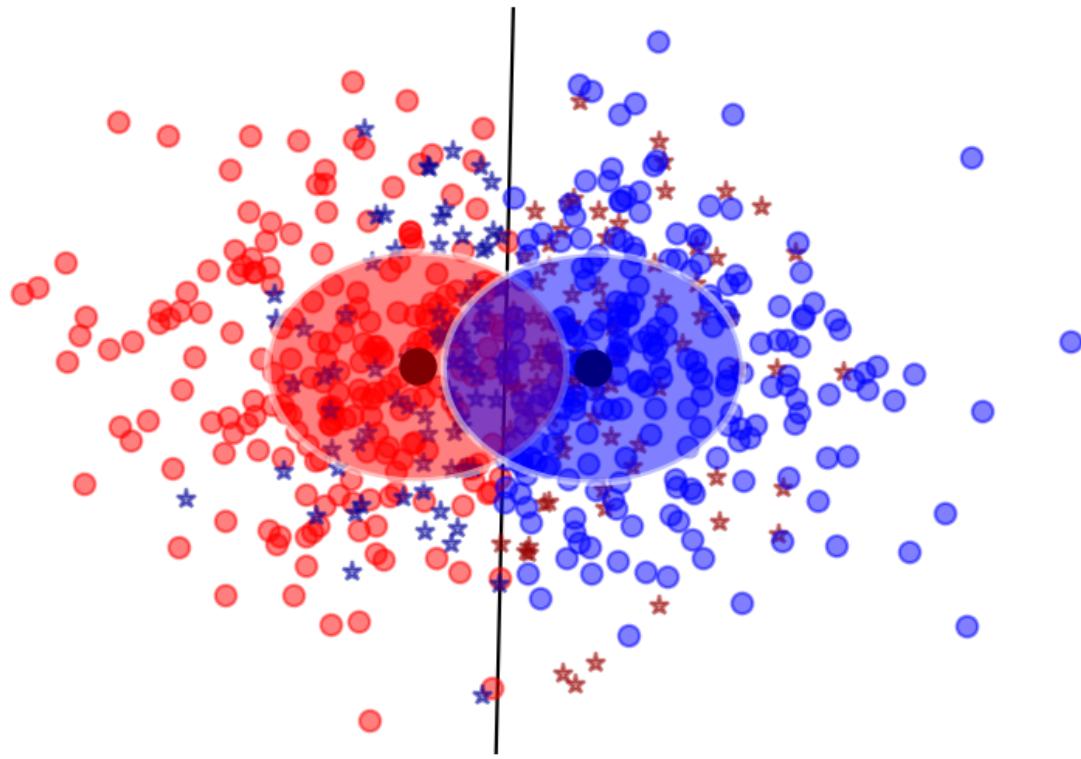
$$\sigma(x) = \frac{1}{1 + \exp(-x)},$$

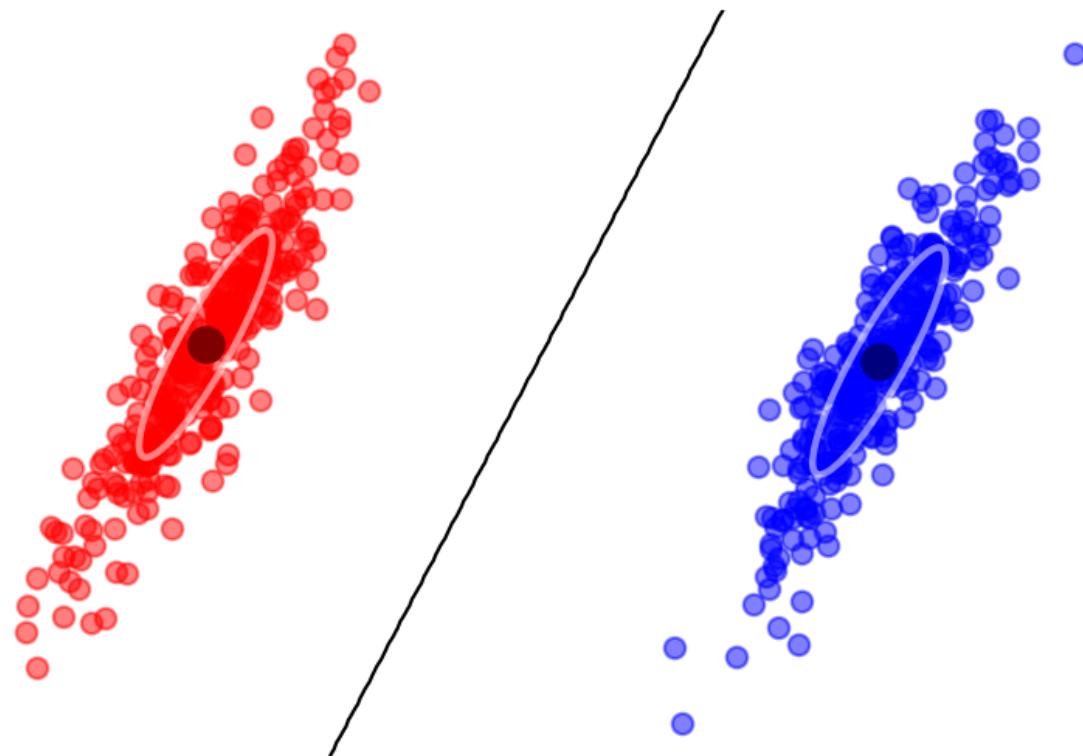
we get

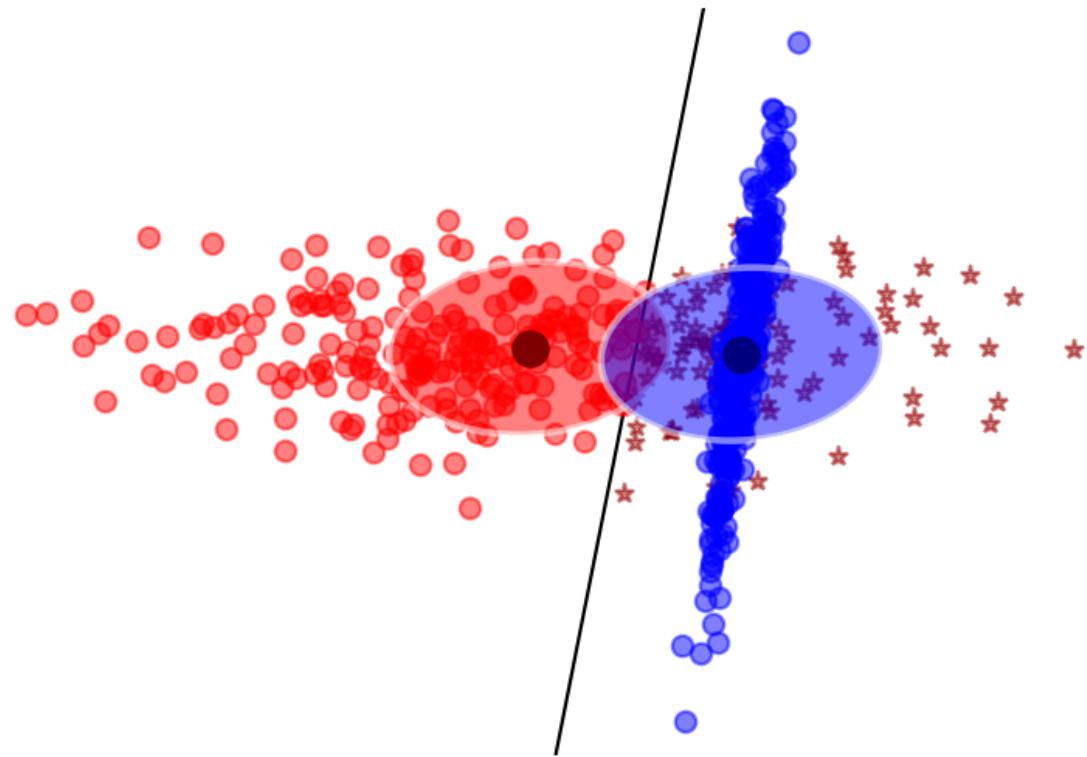
$$P(Y = 1|\mathbf{x}) = \sigma \left( \log \frac{P(\mathbf{x}|Y = 1)}{P(\mathbf{x}|Y = 0)} + \log \frac{P(Y = 1)}{P(Y = 0)} \right).$$

Therefore,

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



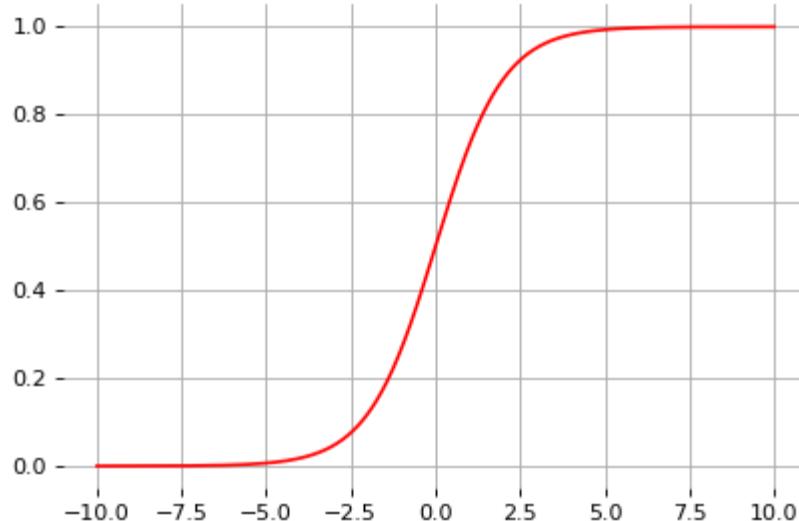




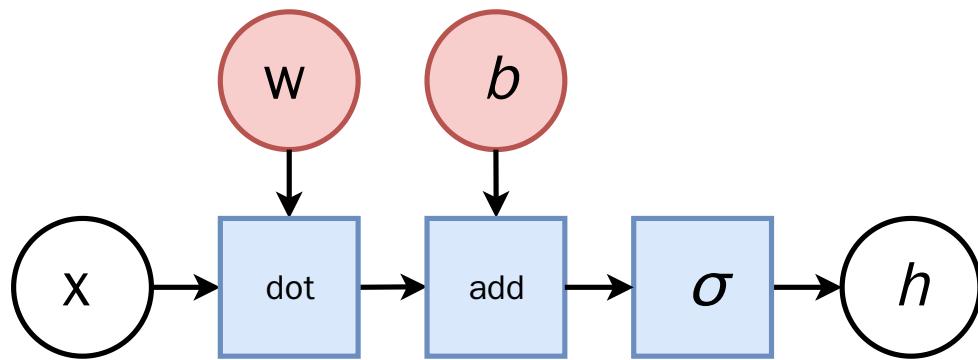
Note that the **sigmoid** function

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

looks like a soft heavyside:



Therefore, the overall model  $f(\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{w}^T \mathbf{x} + b)$  is very similar to the perceptron.



This unit is the main **primitive** of all neural networks!

# Logistic regression

Same model

$$P(Y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

as for linear discriminant analysis.

But,

- ignore model assumptions (Gaussian class populations, homoscedasticity);
- instead, find  $\mathbf{w}, b$  that maximizes the likelihood of the data.

We have,

$$\begin{aligned} & \arg \max_{\mathbf{w}, b} P(\mathbf{d} | \mathbf{w}, b) \\ &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} P(Y = y_i | \mathbf{x}_i, \mathbf{w}, b) \\ &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} \sigma(\mathbf{w}^T \mathbf{x}_i + b)^{y_i} (1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))^{1-y_i} \\ &= \arg \min_{\mathbf{w}, b} \underbrace{\sum_{\mathbf{x}_i, y_i \in \mathbf{d}} -y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i + b) - (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))}_{\mathcal{L}(\mathbf{w}, b) = \sum_i \ell(y_i, \hat{y}(\mathbf{x}_i; \mathbf{w}, b))} \end{aligned}$$

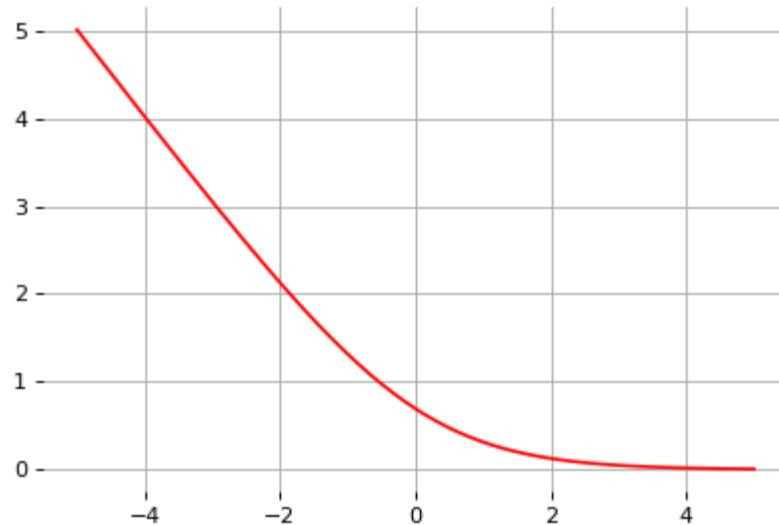
This loss is an instance of the **cross-entropy**

$$H(p, q) = \mathbb{E}_p[-\log q]$$

for  $p = Y | \mathbf{x}_i$  and  $q = \hat{Y} | \mathbf{x}_i$ .

When  $Y$  takes values in  $\{-1, 1\}$ , a similar derivation yields the **logistic loss**

$$\mathcal{L}(\mathbf{w}, b) = - \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \log \sigma(y_i(\mathbf{w}^T \mathbf{x}_i + b)).$$



- In general, the cross-entropy and the logistic losses do not admit a minimizer that can be expressed analytically in closed form.
- However, a minimizer can be found numerically, using a general minimization technique such as **gradient descent**.

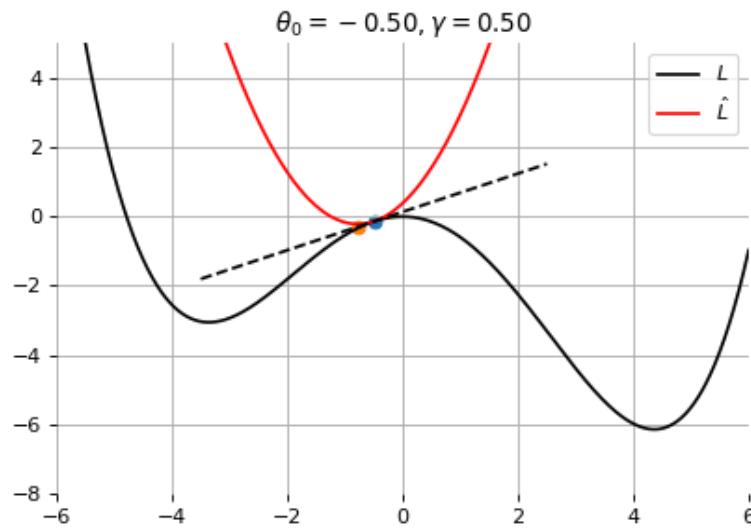
# Gradient descent

Let  $\mathcal{L}(\theta)$  denote a loss function defined over model parameters  $\theta$  (e.g.,  $\mathbf{w}$  and  $b$ ).

To minimize  $\mathcal{L}(\theta)$ , **gradient descent** uses local linear information to iteratively move towards a (local) minimum.

For  $\theta_0 \in \mathbb{R}^d$ , a first-order approximation around  $\theta_0$  can be defined as

$$\hat{\mathcal{L}}(\epsilon; \theta_0) = \mathcal{L}(\theta_0) + \epsilon^T \nabla_{\theta} \mathcal{L}(\theta_0) + \frac{1}{2\gamma} \|\epsilon\|^2.$$



A minimizer of the approximation  $\hat{\mathcal{L}}(\epsilon; \theta_0)$  is given for

$$\begin{aligned}\nabla_\epsilon \hat{\mathcal{L}}(\epsilon; \theta_0) &= 0 \\ &= \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{\gamma} \epsilon,\end{aligned}$$

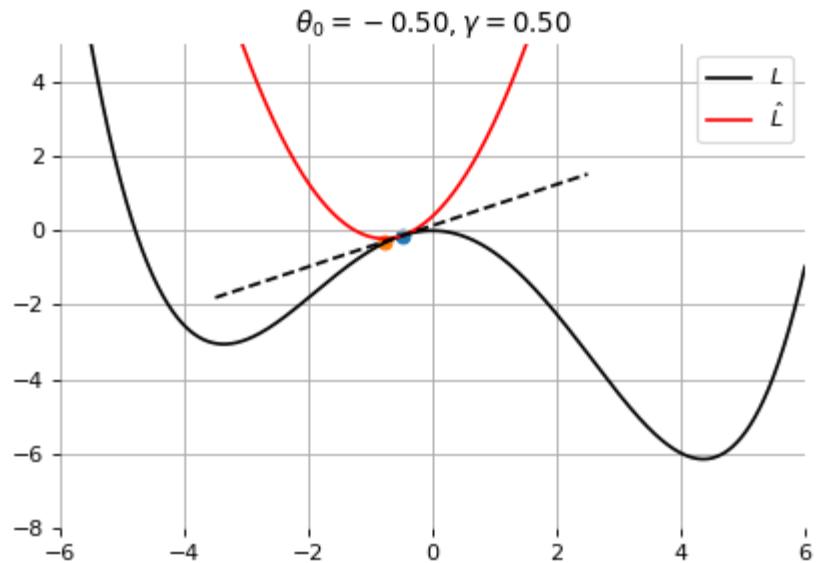
which results in the best improvement for the step  $\epsilon = -\gamma \nabla_\theta \mathcal{L}(\theta_0)$ .

Therefore, model parameters can be updated iteratively using the update rule

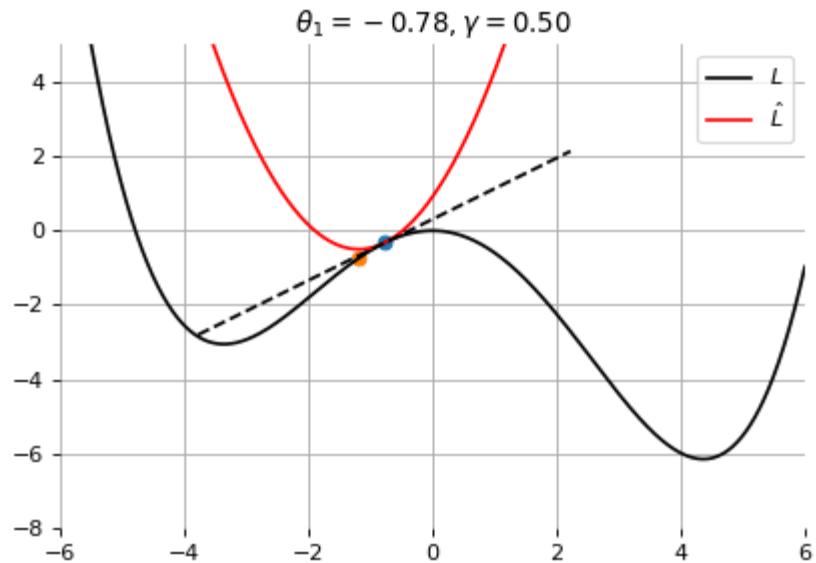
$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(\theta_t),$$

where

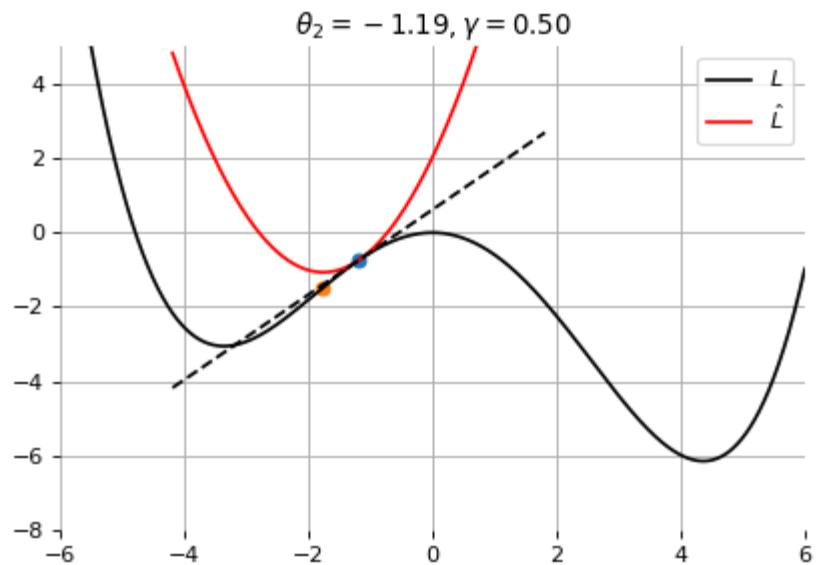
- $\theta_0$  are the initial parameters of the model;
- $\gamma$  is the **learning rate**;
- both are critical for the convergence of the update rule.



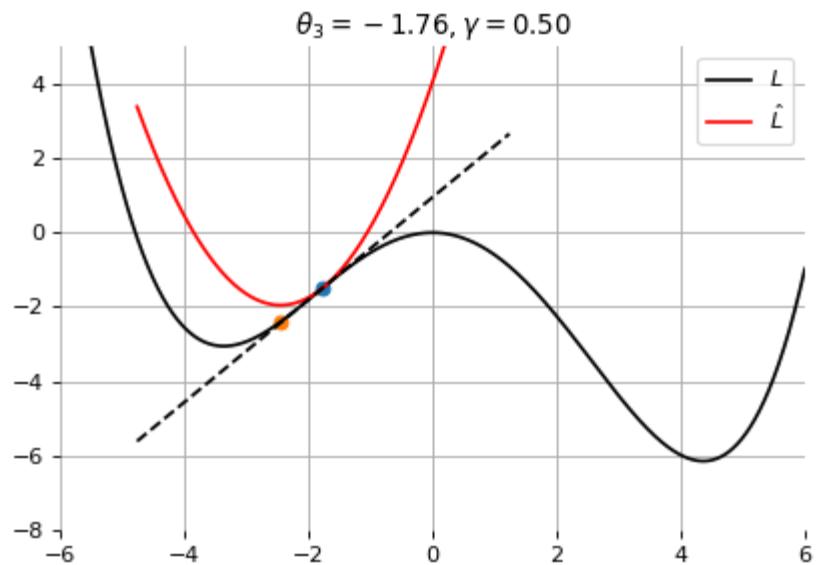
Example 1: Convergence to a local minima



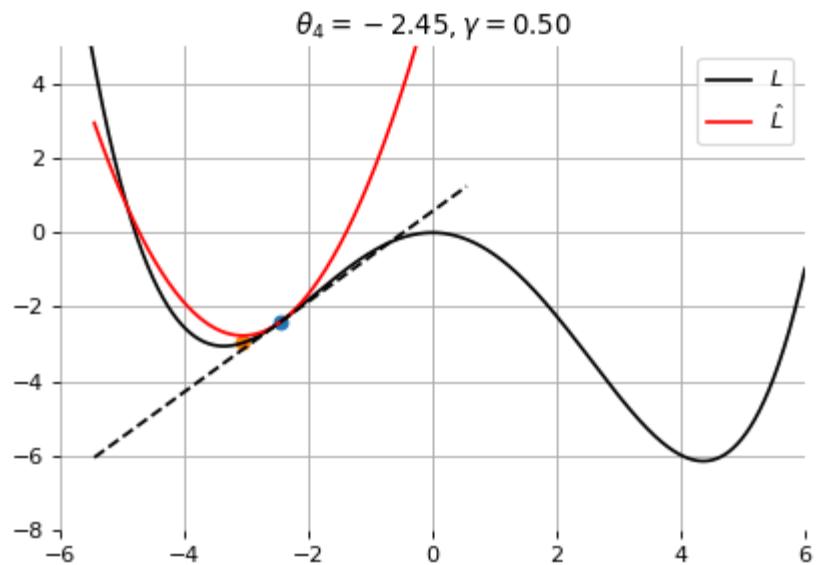
Example 1: Convergence to a local minima



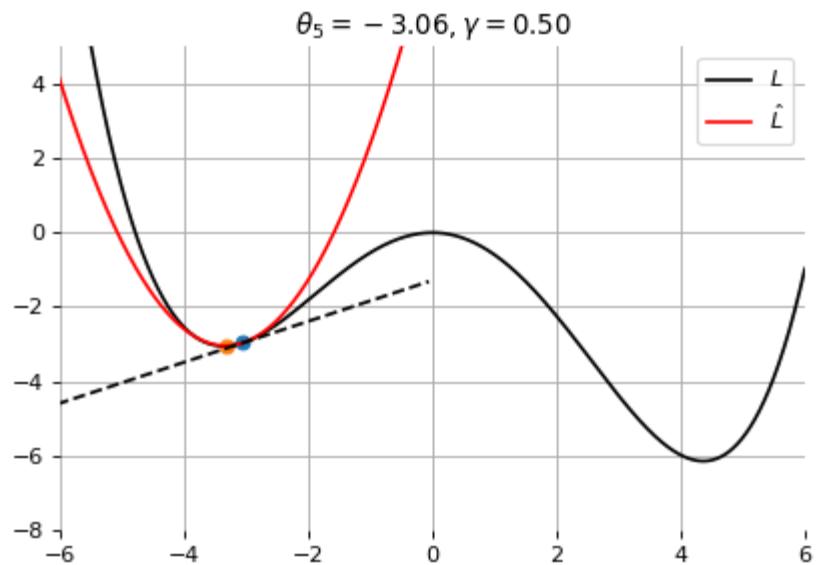
Example 1: Convergence to a local minima



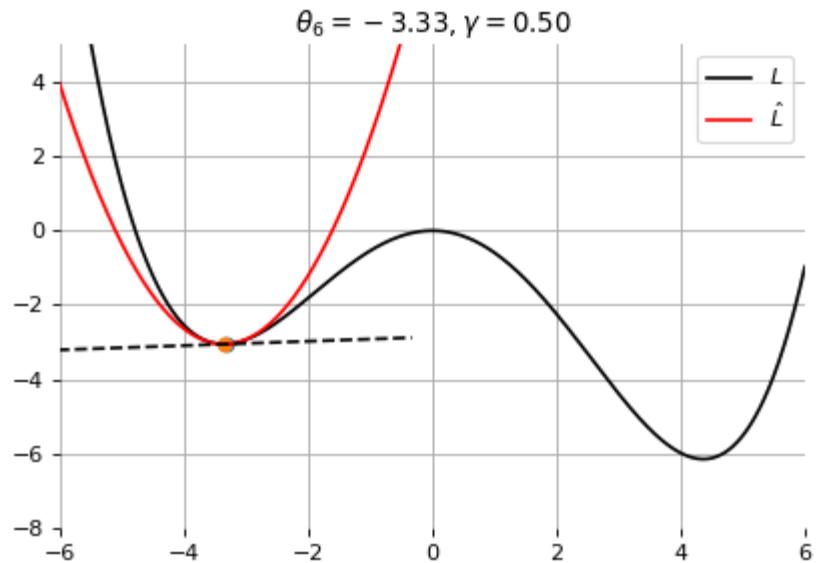
Example 1: Convergence to a local minima



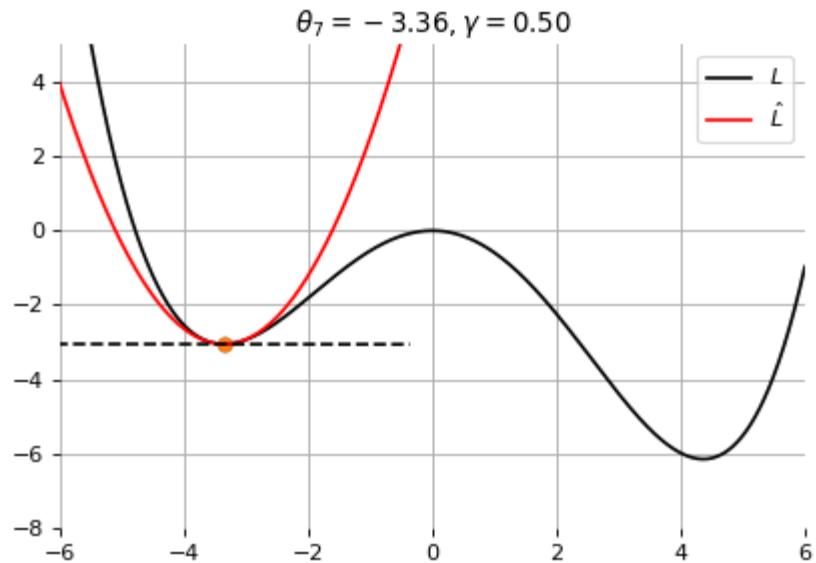
Example 1: Convergence to a local minima



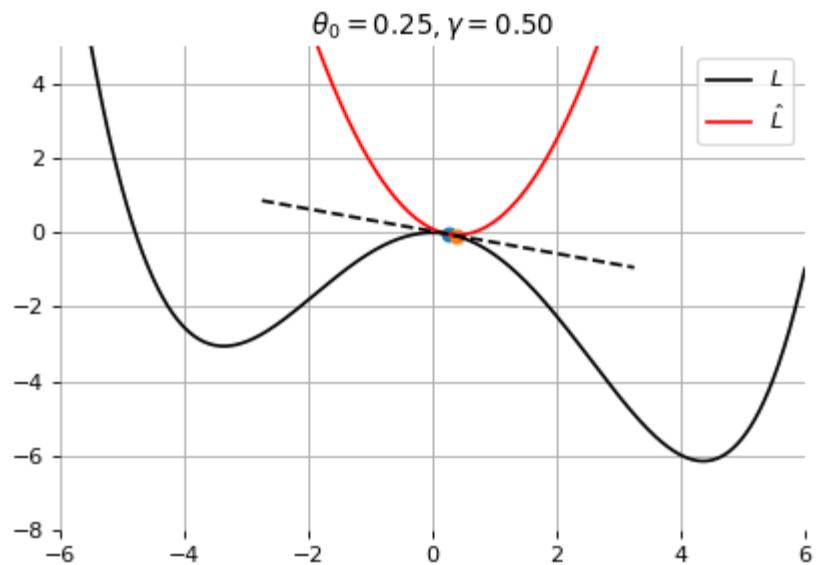
Example 1: Convergence to a local minima



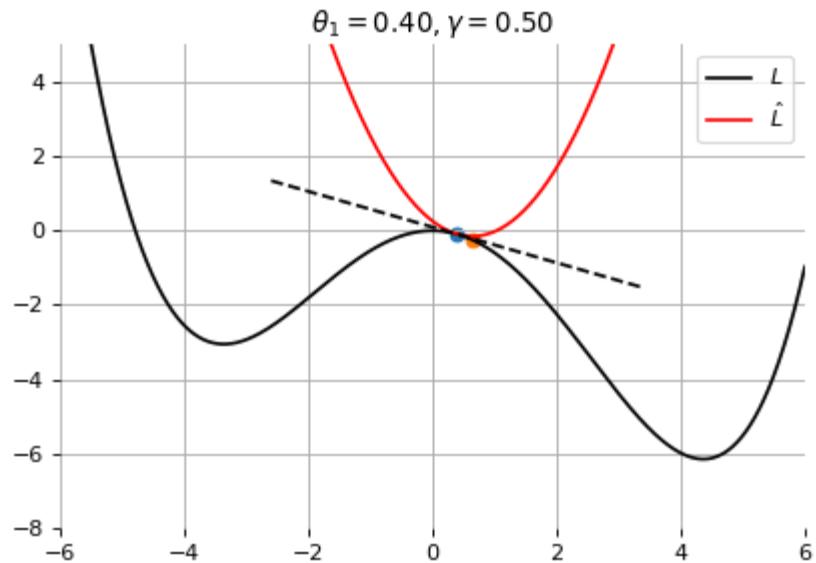
Example 1: Convergence to a local minima



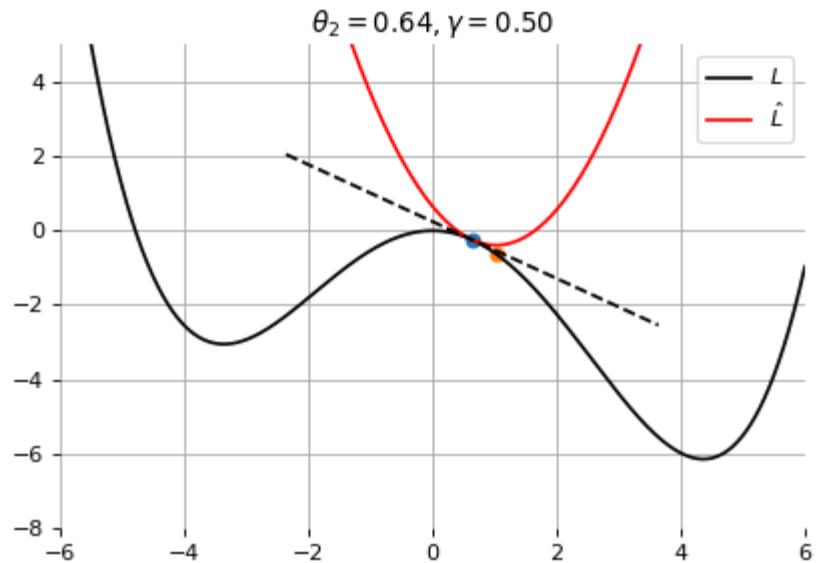
Example 1: Convergence to a local minima



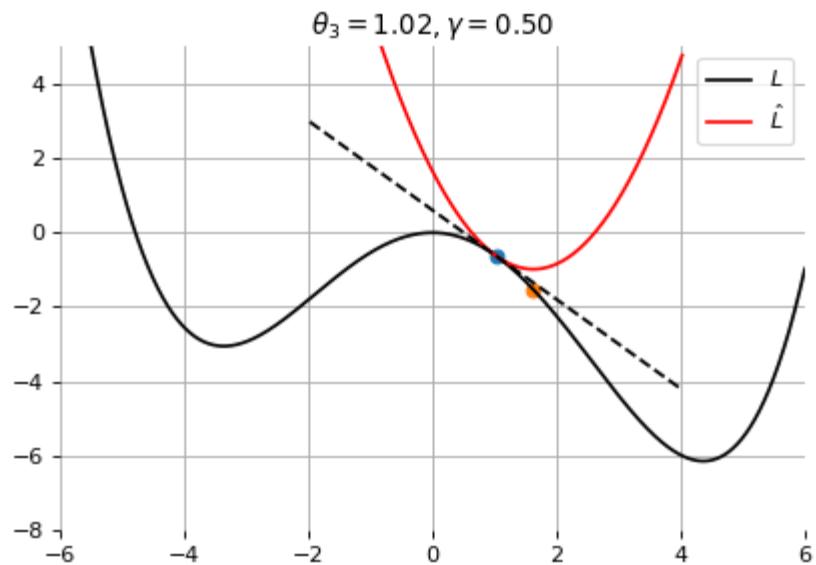
Example 2: Convergence to the global minima



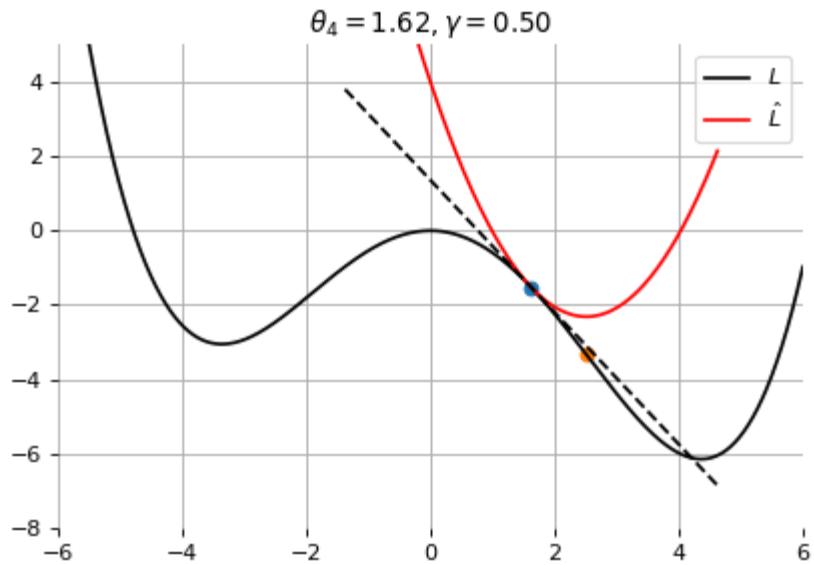
Example 2: Convergence to the global minima



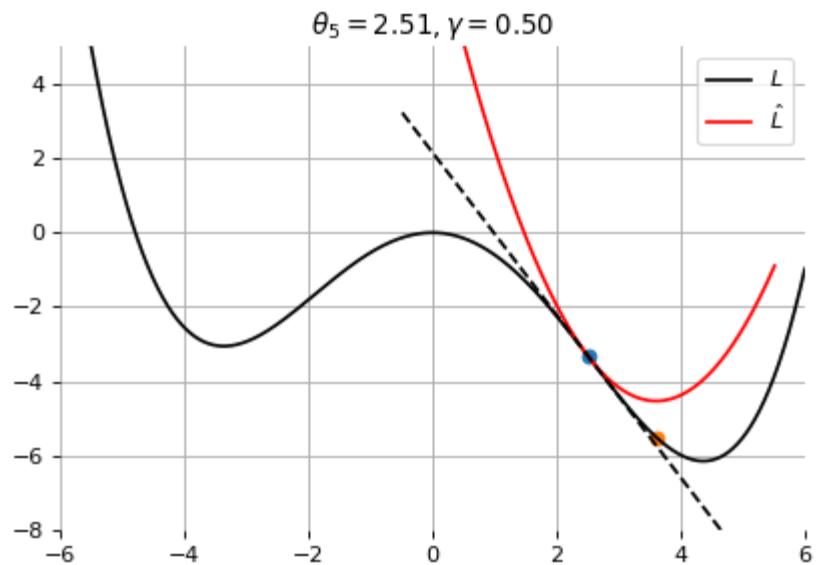
Example 2: Convergence to the global minima



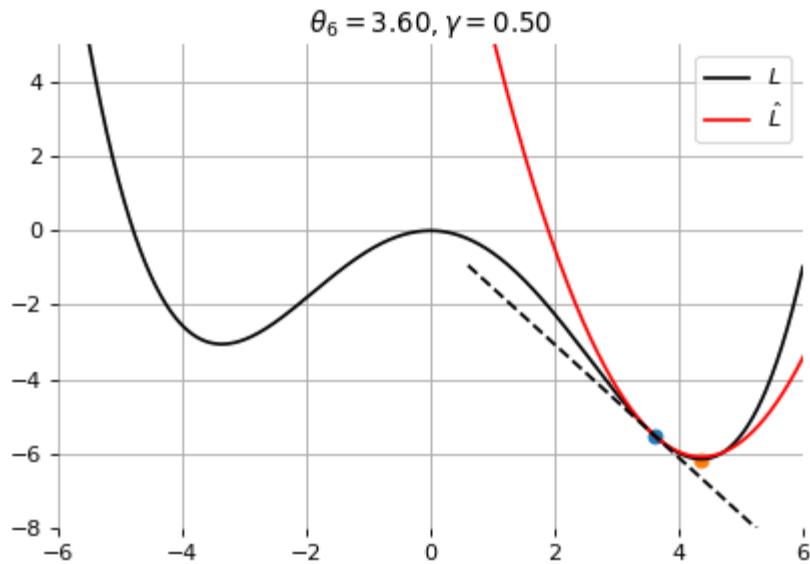
Example 2: Convergence to the global minima



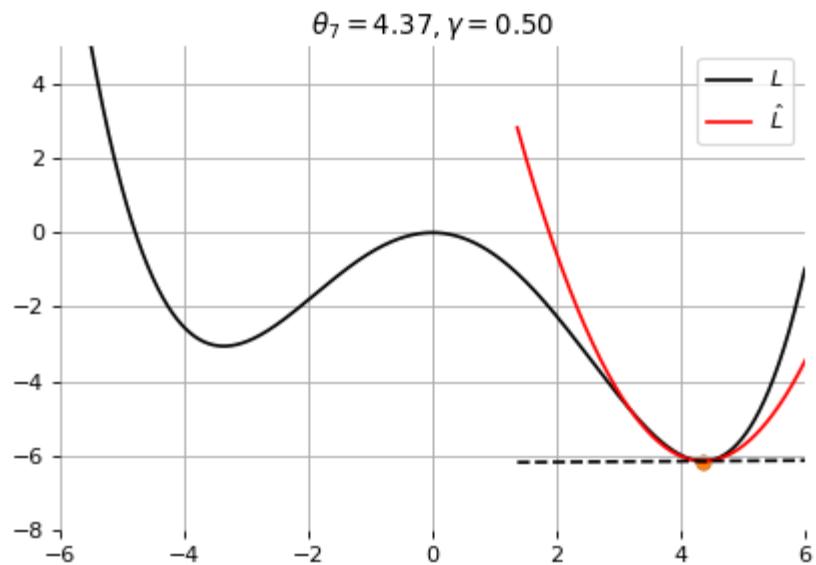
Example 2: Convergence to the global minima



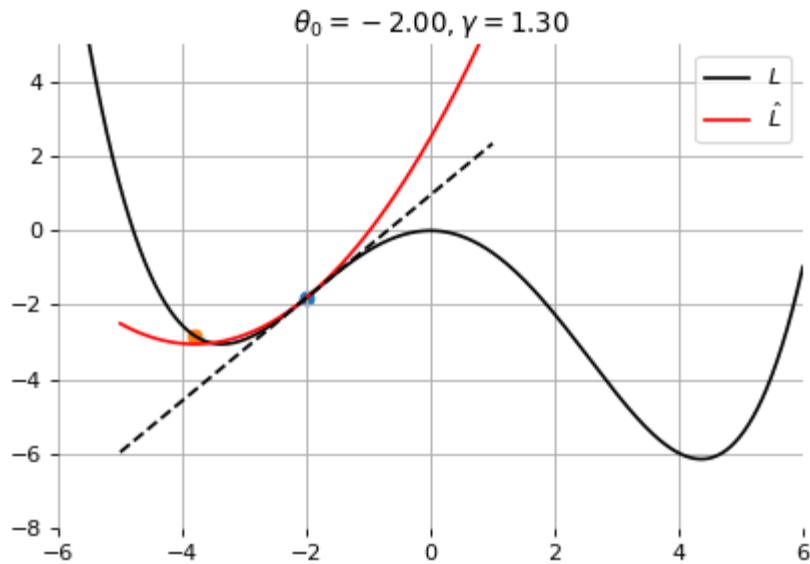
Example 2: Convergence to the global minima



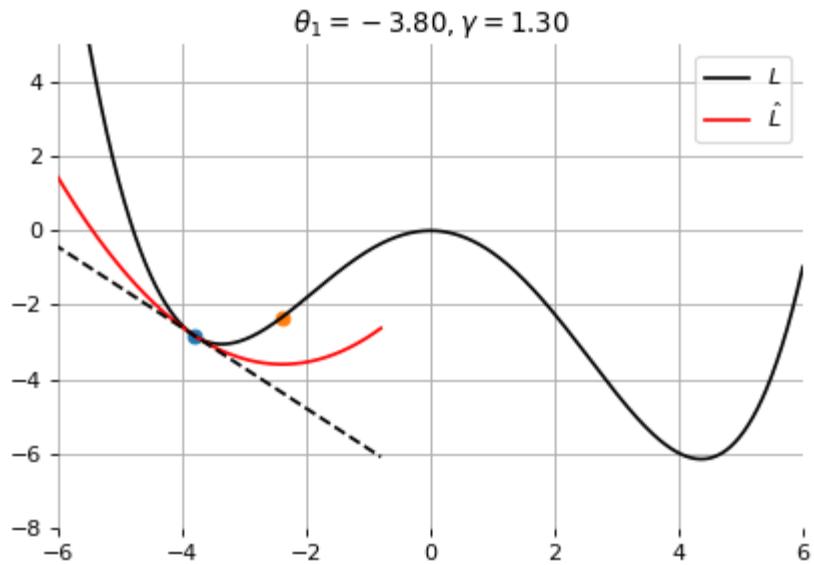
Example 2: Convergence to the global minima



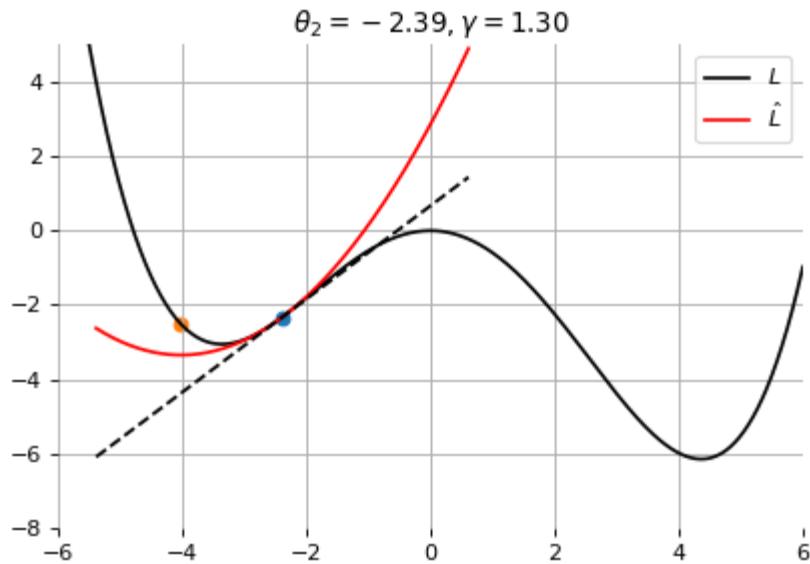
Example 2: Convergence to the global minima



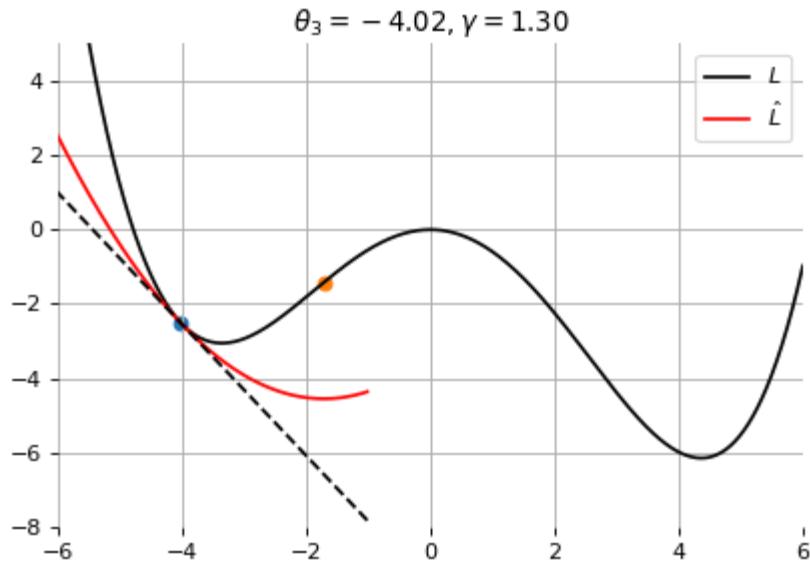
Example 3: Divergence due to a too large learning rate



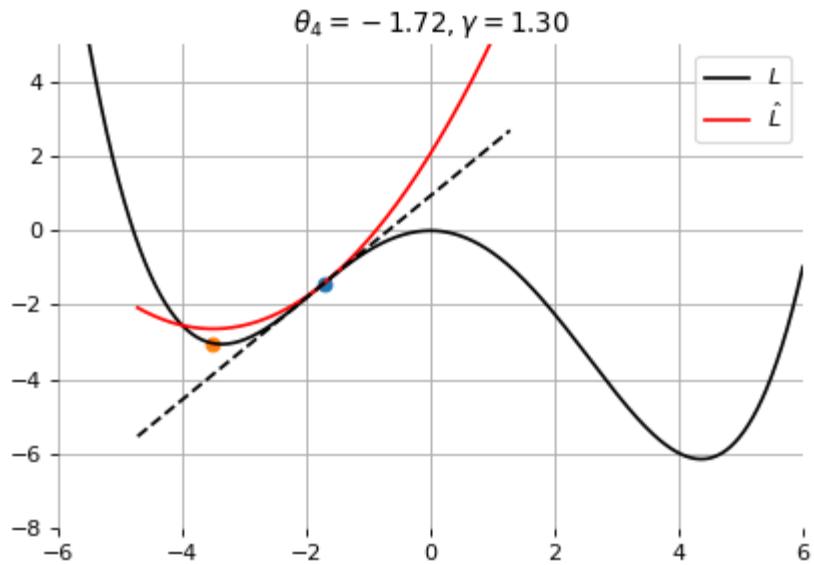
Example 3: Divergence due to a too large learning rate



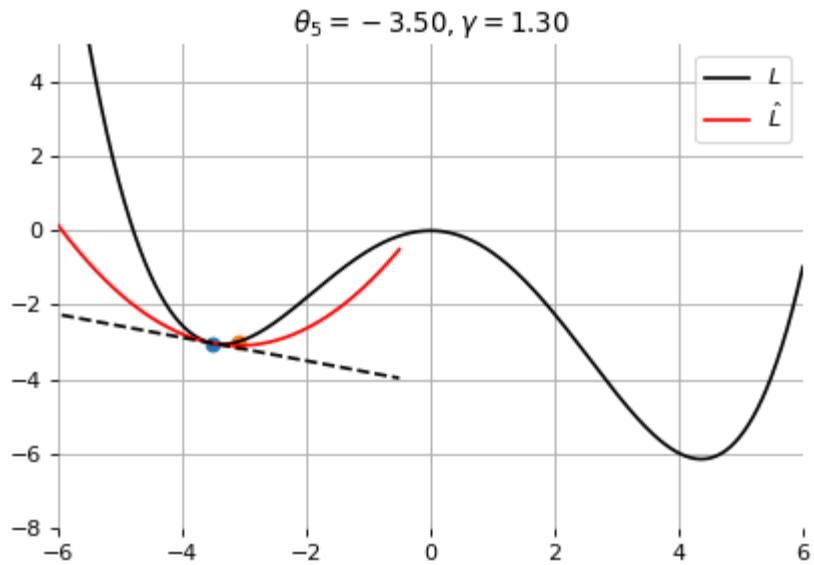
Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate

# Stochastic gradient descent

In the empirical risk minimization setup,  $\mathcal{L}(\theta)$  and its gradient decompose as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i; \theta))$$
$$\nabla \mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Therefore, in **batch** gradient descent the complexity of an update grows linearly with the size  $N$  of the dataset.

More importantly, since the empirical risk is already an approximation of the expected risk, it should not be necessary to carry out the minimization with great accuracy.

Instead, **stochastic** gradient descent uses as update rule:

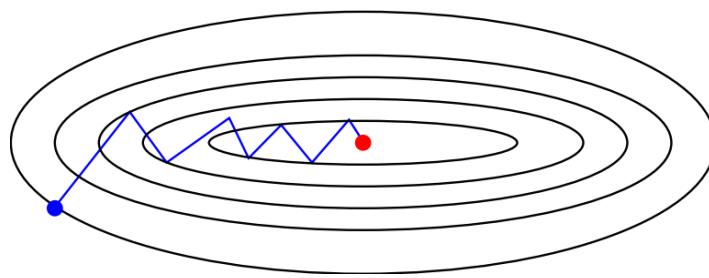
$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of  $N$ .
- The stochastic process  $\{\theta_t | t = 1, \dots\}$  depends on the examples  $i(t)$  picked randomly at each iteration.

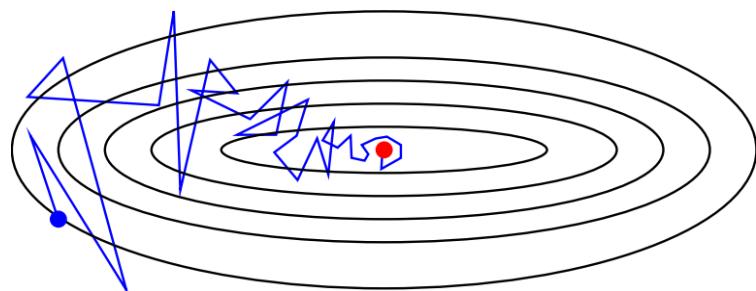
Instead, **stochastic** gradient descent uses as update rule:

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of  $N$ .
- The stochastic process  $\{\theta_t | t = 1, \dots\}$  depends on the examples  $i(t)$  picked randomly at each iteration.



*Batch gradient descent*



*Stochastic gradient descent*

Why is stochastic gradient descent still a good idea?

- Informally, averaging the update

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

over all choices  $i(t+1)$  restores batch gradient descent.

- Formally, if the gradient estimate is **unbiased**, e.g., if

$$\begin{aligned}\mathbb{E}_{i(t+1)} [\nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))] &= \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta_t)) \\ &= \nabla \mathcal{L}(\theta_t)\end{aligned}$$

then the formal convergence of SGD can be proved, under appropriate assumptions (see references).

- If training is limited to single pass over the data, then SGD directly minimizes the **expected** risk.

The excess error characterizes the expected risk discrepancy between the Bayes model and the approximate empirical risk minimizer. It can be decomposed as

$$\begin{aligned} & \mathbb{E} \left[ R(\tilde{f}_*^{\mathbf{d}}) - R(f_B) \right] \\ &= \mathbb{E} [R(f_*) - R(f_B)] + \mathbb{E} [R(f_*^{\mathbf{d}}) - R(f_*)] + \mathbb{E} \left[ R(\tilde{f}_*^{\mathbf{d}}) - R(f_*^{\mathbf{d}}) \right] \\ &= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \end{aligned}$$

where

- $\mathcal{E}_{\text{app}}$  is the approximation error due to the choice of an hypothesis space,
- $\mathcal{E}_{\text{est}}$  is the estimation error due to the empirical risk minimization principle,
- $\mathcal{E}_{\text{opt}}$  is the optimization error due to the approximate optimization algorithm.

A fundamental result due to Bottou and Bousquet (2011) states that stochastic optimization algorithms (e.g., SGD) yield the best generalization performance (in terms of excess error) despite being the worst optimization algorithms for minimizing the empirical risk.

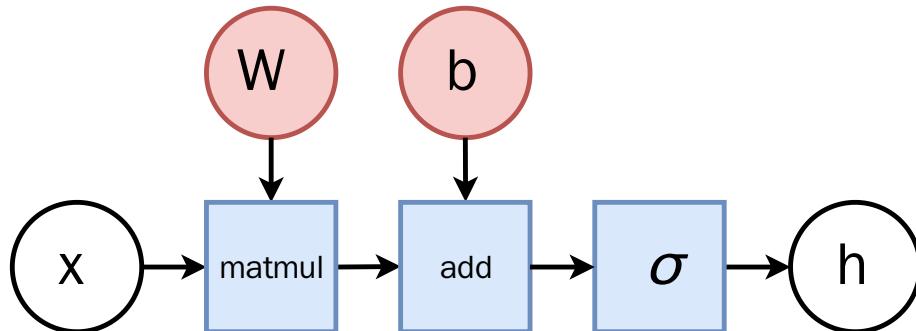
# Multi-layer perceptron

So far we considered the logistic unit  $h = \sigma(\mathbf{w}^T \mathbf{x} + b)$ , where  $h \in \mathbb{R}$ ,  $\mathbf{x} \in \mathbb{R}^p$ ,  $\mathbf{w} \in \mathbb{R}^p$  and  $b \in \mathbb{R}$ .

These units can be composed in parallel to form a **layer** with  $q$  outputs:

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

where  $\mathbf{h} \in \mathbb{R}^q$ ,  $\mathbf{x} \in \mathbb{R}^p$ ,  $\mathbf{W} \in \mathbb{R}^{p \times q}$ ,  $\mathbf{b} \in \mathbb{R}^q$  and where  $\sigma(\cdot)$  is upgraded to the element-wise sigmoid function.



Similarly, layers can be composed **in series**, such that:

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}_1 = \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1)$$

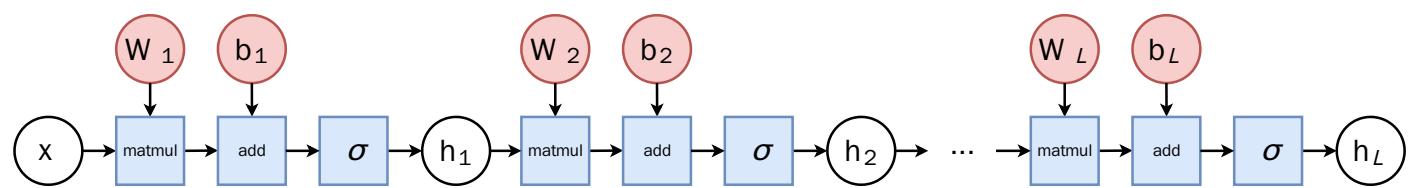
...

$$\mathbf{h}_L = \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L)$$

$$f(\mathbf{x}; \theta) = \hat{y} = \mathbf{h}_L$$

where  $\theta$  denotes the model parameters  $\{\mathbf{W}_k, \mathbf{b}_k, \dots | k = 1, \dots, L\}$ .

This model is the **multi-layer perceptron**, also known as the fully connected feedforward network.



## Output layer

- For binary classification, the width  $q$  of the last layer  $L$  is set to 1, which results in a single output  $h_L \in [0, 1]$  that models the probability  $P(Y = 1 | \mathbf{x})$ .
- For multi-class classification, the sigmoid action  $\sigma$  in the last layer can be generalized to produce a vector  $\mathbf{h}_L \in \Delta^C$  of probability estimates  $P(Y = i | \mathbf{x})$ .

This activation is the **Softmax** function, where its  $i$ -th output is defined as

$$\text{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)},$$

for  $i = 1, \dots, C$ .

# Regression

For regression problems, one usually starts with the assumption that

$$P(y|\mathbf{x}) = \mathcal{N}(y; \mu = f(\mathbf{x}; \theta), \sigma^2 = 1),$$

where  $f$  is parameterized with a neural network which last layer does not contain any final activation.

We have,

$$\begin{aligned} & \arg \max_{\theta} P(\mathbf{d}|\theta) \\ &= \arg \max_{\theta} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} P(Y = y_i | \mathbf{x}_i, \theta) \\ &= \arg \min_{\theta} - \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \log P(Y = y_i | \mathbf{x}_i, \theta) \\ &= \arg \min_{\theta} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \frac{1}{2} (y_i - f(\mathbf{x}; \theta))^2 + \text{constant} \\ &= \arg \min_{\theta} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} (y_i - f(\mathbf{x}; \theta))^2, \end{aligned}$$

which recovers the common **squared error** loss.

# Automatic differentiation

To minimize  $\mathcal{L}(\theta)$  with stochastic gradient descent, we need the gradient  $\nabla_{\theta}\ell(\theta_t)$ .

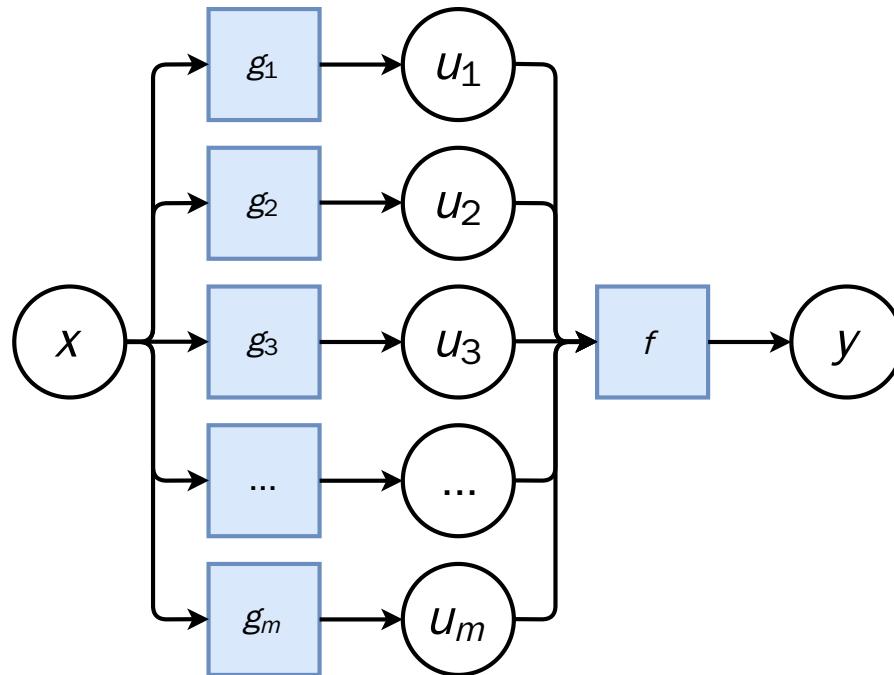
Therefore, we require the evaluation of the (total) derivatives

$$\frac{d\ell}{d\mathbf{W}_k} \text{ and } \frac{d\ell}{d\mathbf{b}_k}$$

of the loss  $\ell$  with respect to all model parameters  $\mathbf{W}_k, \mathbf{b}_k$ , for  $k = 1, \dots, L$ .

These derivatives can be evaluated automatically from the computational graph of  $\ell$  using automatic differentiation.

## Chain rule



Let us consider a 1-dimensional output composition  $f \circ g$ , such that

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

$$\mathbf{u} = g(x) = (g_1(x), \dots, g_m(x)).$$

The **chain rule** states that  $(f \circ g)' = (f' \circ g)g'$ .

For the total derivative, the chain rule generalizes to

$$\frac{dy}{dx} = \sum_{k=1}^m \frac{\partial y}{\partial u_k} \underbrace{\frac{du_k}{dx}}_{\text{recursive case}}$$

## Reverse automatic differentiation

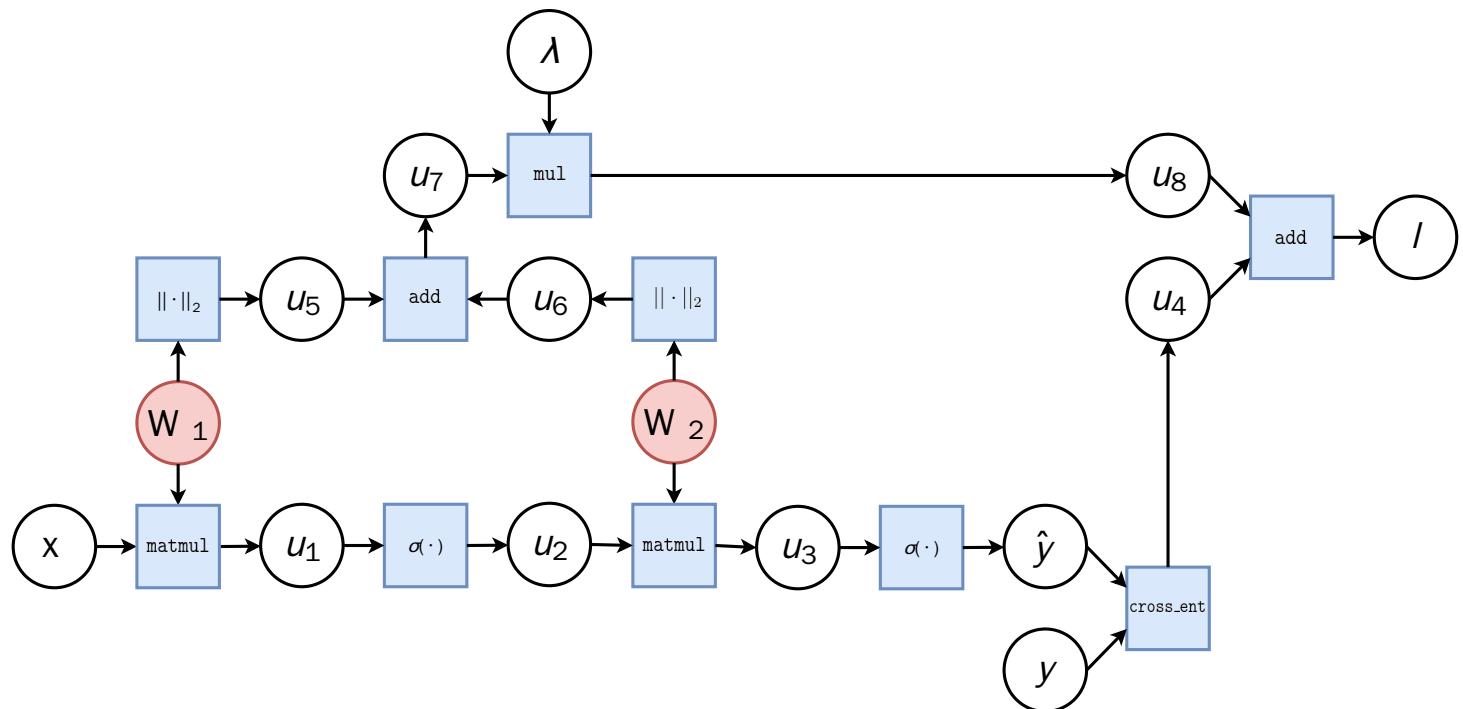
- Since a neural network is a **composition of differentiable functions**, the total derivatives of the loss can be evaluated backward, by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called reverse **automatic differentiation**.

Let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma(\mathbf{W}_2^T \sigma(\mathbf{W}_1^T \mathbf{x}))$$
$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \text{cross\_ent}(y, \hat{y}) + \lambda(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2)$$

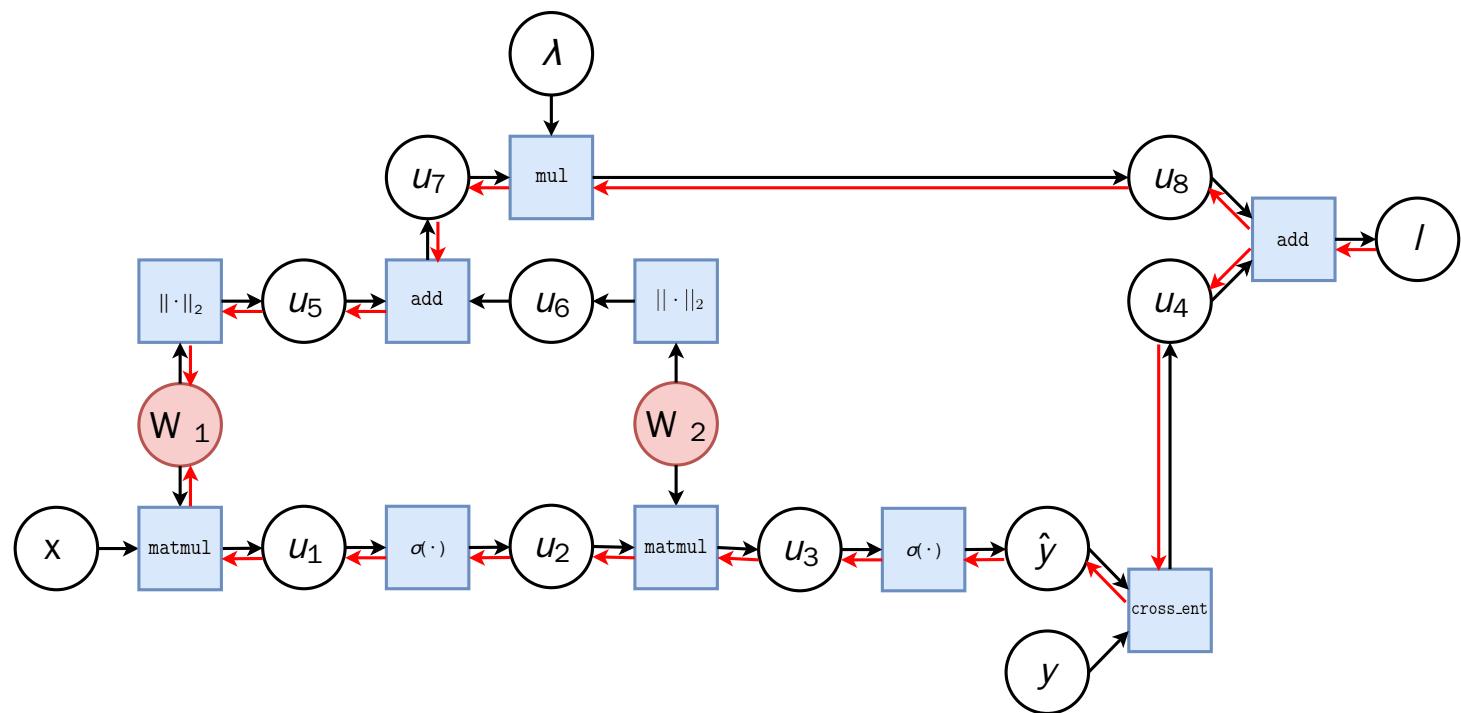
for  $\mathbf{x} \in \mathbb{R}^p, y \in \mathbb{R}, \mathbf{W}_1 \in \mathbb{R}^{p \times q}$  and  $\mathbf{W}_2 \in \mathbb{R}^q$ .

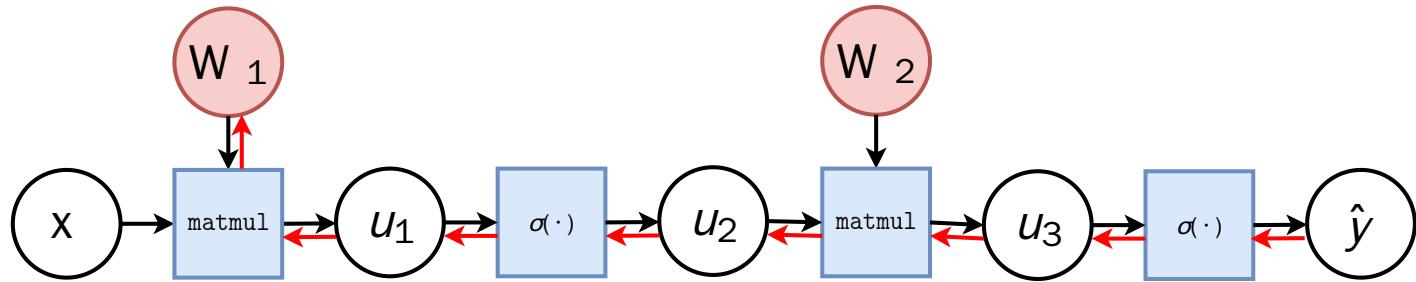
In the **forward pass**, intermediate values are all computed from inputs to outputs, which results in the annotated computational graph below:



The total derivative can be computed through a **backward pass**, by walking through all paths from outputs to parameters in the computational graph and accumulating the terms. For example, for  $\frac{d\ell}{d\mathbf{W}_1}$  we have:

$$\begin{aligned}\frac{d\ell}{d\mathbf{W}_1} &= \frac{\partial\ell}{\partial u_8} \frac{du_8}{d\mathbf{W}_1} + \frac{\partial\ell}{\partial u_4} \frac{du_4}{d\mathbf{W}_1} \\ \frac{du_8}{d\mathbf{W}_1} &= \dots\end{aligned}$$





Let us zoom in on the computation of the network output  $\hat{y}$  and of its derivative with respect to  $\mathbf{W}_1$ .

- **Forward pass:** values  $u_1, u_2, u_3$  and  $\hat{y}$  are computed by traversing the graph from inputs to outputs given  $\mathbf{x}, \mathbf{W}_1$  and  $\mathbf{W}_2$ .
- **Backward pass:** by the chain rule we have

$$\begin{aligned}
\frac{d\hat{y}}{d\mathbf{W}_1} &= \frac{\partial \hat{y}}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial \mathbf{W}_1} \\
&= \frac{\partial \sigma(u_3)}{\partial u_3} \frac{\partial \mathbf{W}_2^T u_2}{\partial u_2} \frac{\partial \sigma(u_1)}{\partial u_1} \frac{\partial \mathbf{W}_1^T \mathbf{x}}{\partial \mathbf{W}_1}
\end{aligned}$$

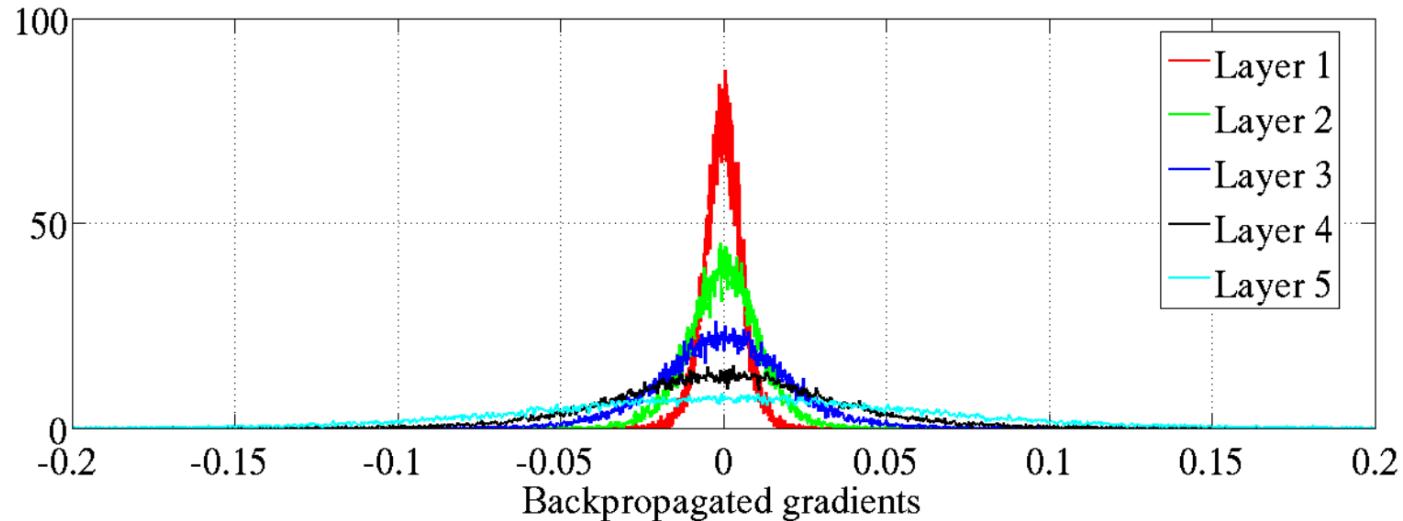
Note how evaluating the partial derivatives requires the intermediate values computed forward.

- This algorithm is also known as **backpropagation**.
- An equivalent procedure can be defined to evaluate the derivatives in **forward mode**, from inputs to outputs.
- Since differentiation is a linear operator, automatic differentiation can be implemented efficiently in terms of tensor operations.

# Vanishing gradients

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the **vanishing gradient** problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



*Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).*

*Gradients for layers far from the output vanish to zero.*

Let us consider a simplified 3-layer MLP, with  $\mathbf{x}, w_1, w_2, w_3 \in \mathbb{R}$ , such that

$$f(\mathbf{x}; w_1, w_2, w_3) = \sigma(w_3 \sigma(w_2 \sigma(w_1 \mathbf{x}))).$$

Under the hood, this would be evaluated as

$$u_1 = w_1 \mathbf{x}$$

$$u_2 = \sigma(u_1)$$

$$u_3 = w_2 u_2$$

$$u_4 = \sigma(u_3)$$

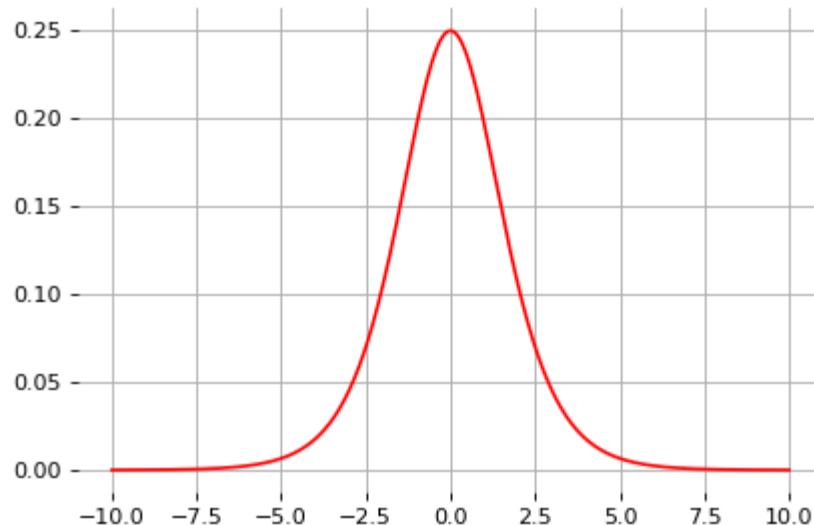
$$u_5 = w_3 u_4$$

$$\hat{y} = \sigma(u_5)$$

and its derivative  $\frac{d\hat{y}}{dw_1}$  as

$$\begin{aligned}\frac{d\hat{y}}{dw_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x\end{aligned}$$

The derivative of the sigmoid activation function  $\sigma$  is:



$$\frac{d\sigma}{dx}(x) = \sigma(x)(1 - \sigma(x))$$

Notice that  $0 \leq \frac{d\sigma}{dx}(x) \leq \frac{1}{4}$  for all  $x$ .

Assume that weights  $w_1, w_2, w_3$  are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability  $-1 \leq w_i \leq 1$ .

Then,

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient  $\frac{d\hat{y}}{dw_1}$  **exponentially** shrinks to zero as the number of layers in the network increases.

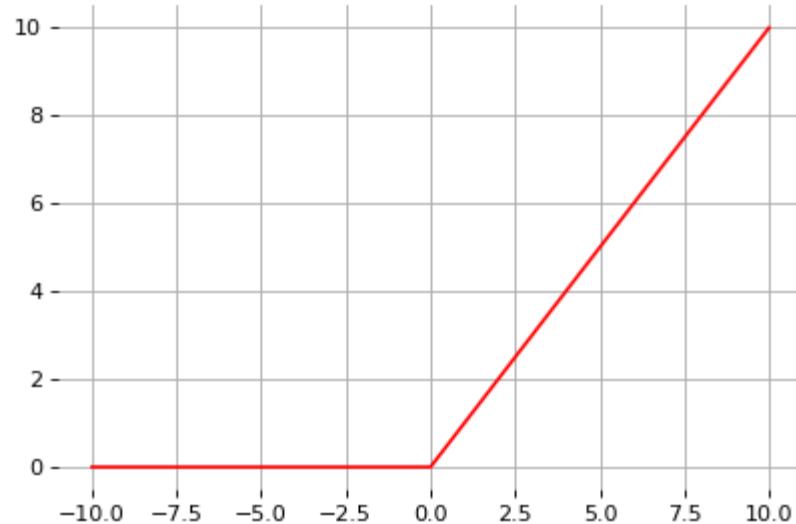
Hence the vanishing gradient problem.

- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

# Rectified linear units

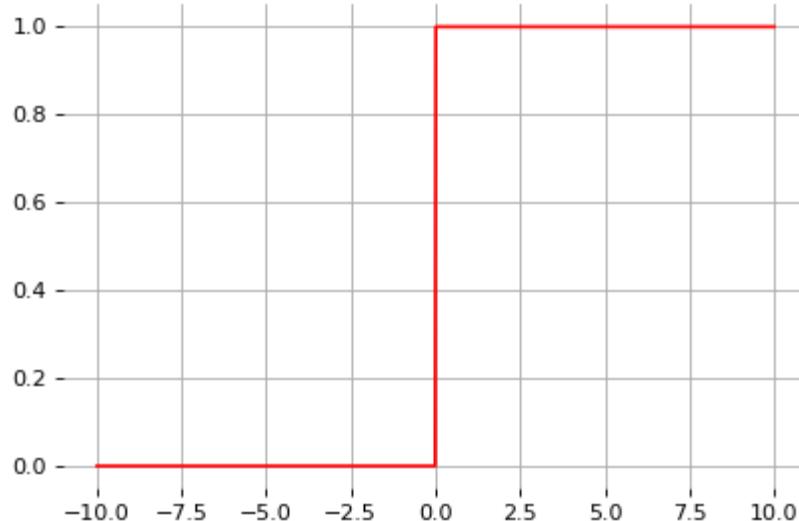
Instead of the sigmoid activation function, modern neural networks are for most based on **rectified linear units** (ReLU) (Glorot et al, 2011):

$$\text{ReLU}(x) = \max(0, x)$$



Note that the derivative of the ReLU function is

$$\frac{d}{dx} \text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



For  $x = 0$ , the derivative is undefined. In practice, it is set to zero.

Therefore,

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{=1} x$$

This **solves** the vanishing gradient problem, even for deep networks! (provided proper initialization)

Note that:

- The ReLU unit dies when its input is negative, which might block gradient descent.
- This is actually a useful property to induce **sparsity**.
- This issue can also be solved using **leaky** ReLUs, defined as

$$\text{LeakyReLU}(x) = \max(\alpha x, x)$$

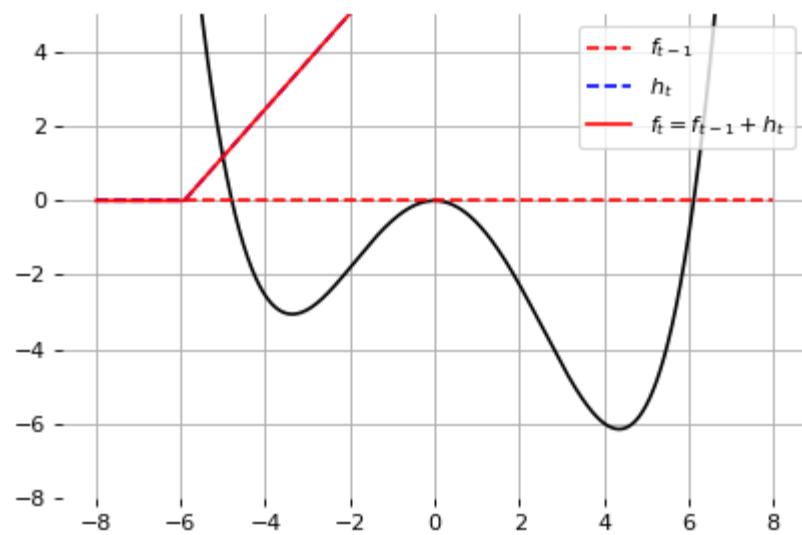
for a small  $\alpha \in \mathbb{R}^+$  (e.g.,  $\alpha = 0.1$ ).

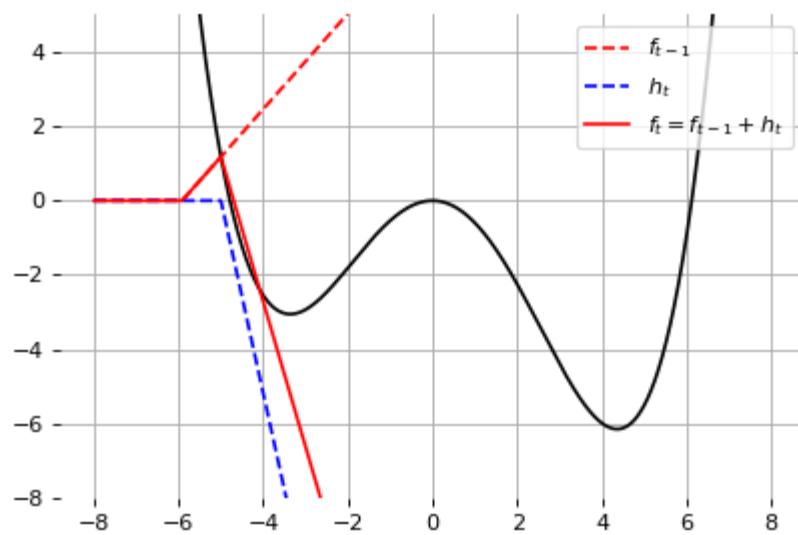
# Universal approximation (teaser)

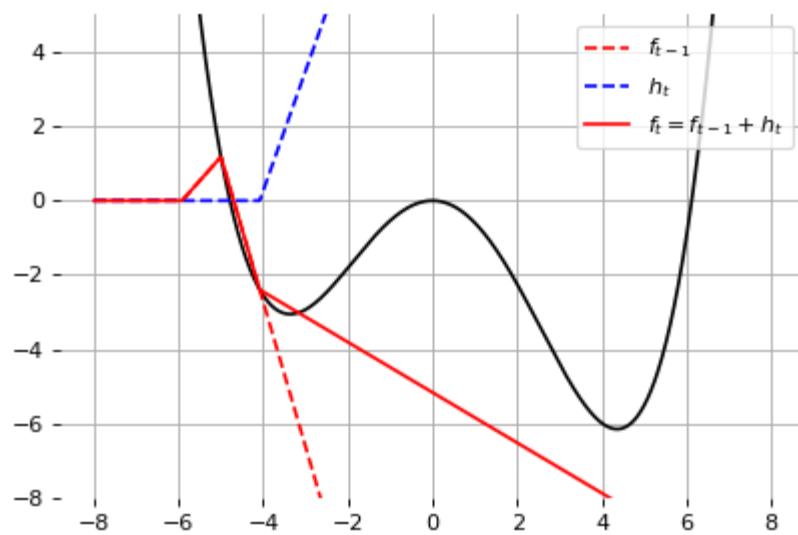
Let us consider the 1-layer MLP

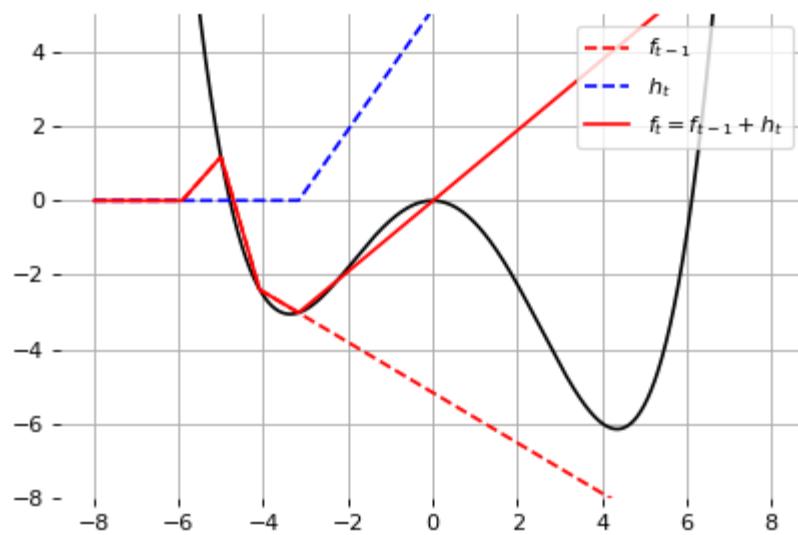
$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

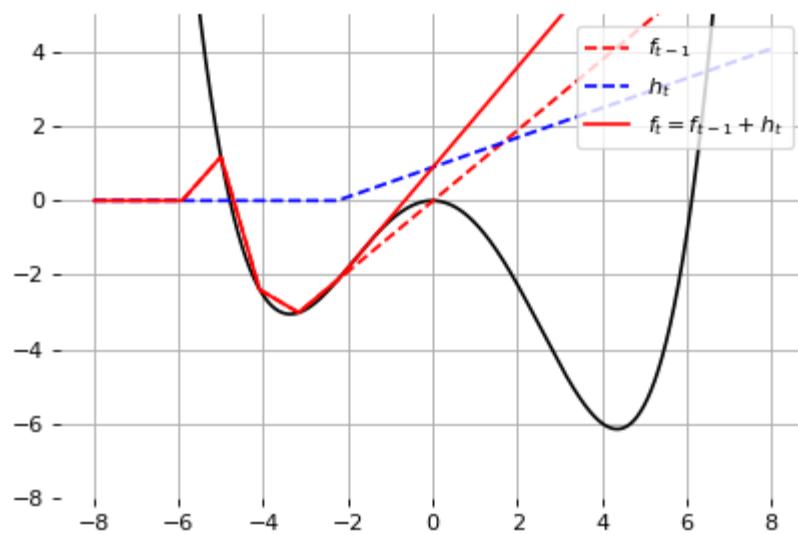
This model can approximate **any** smooth 1D function, provided enough hidden units.

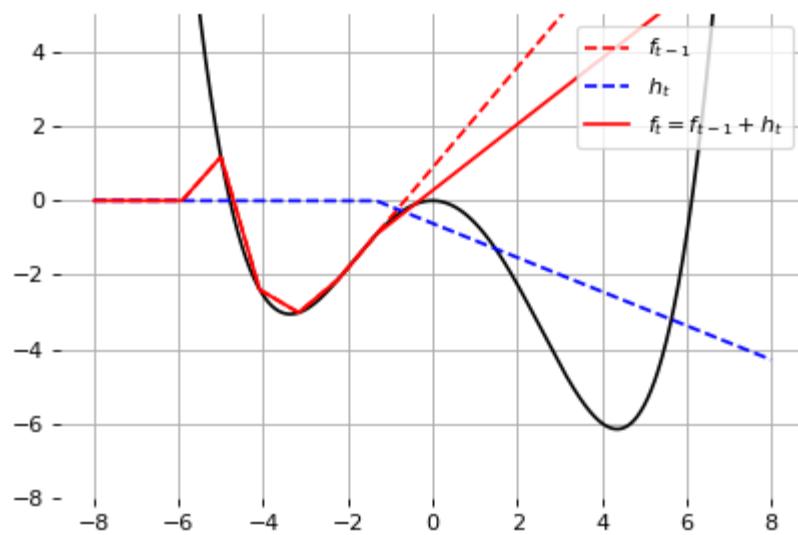


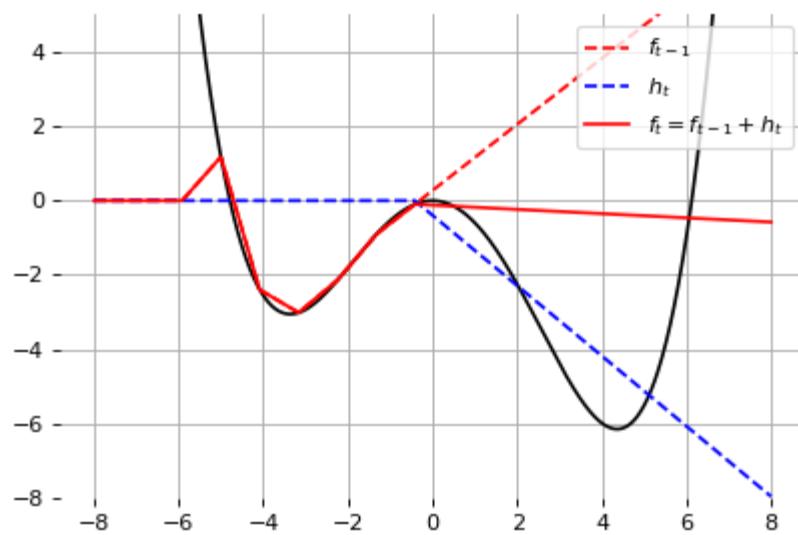


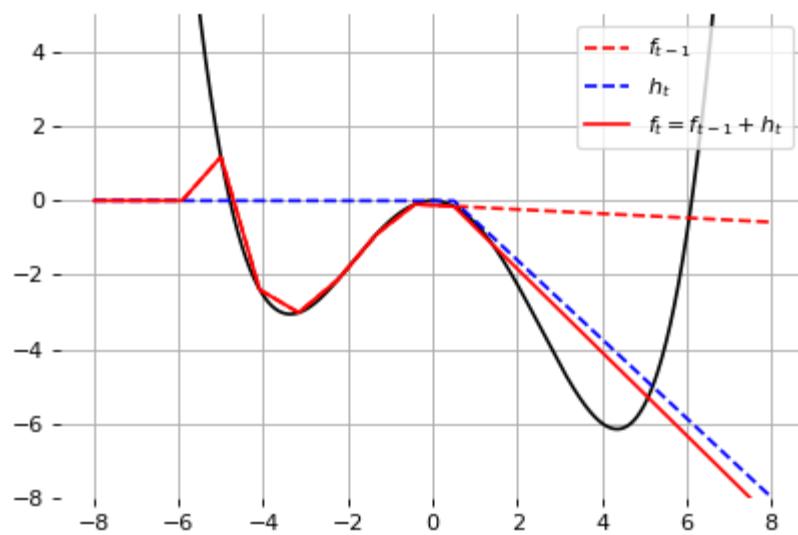


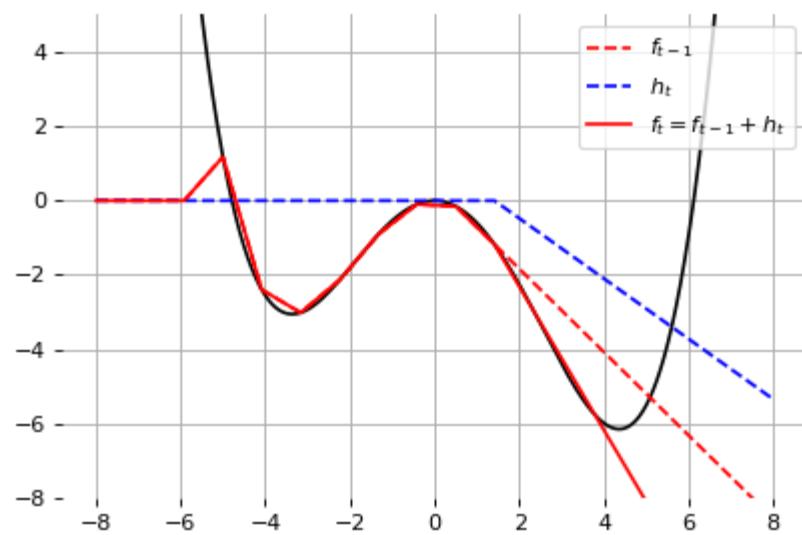


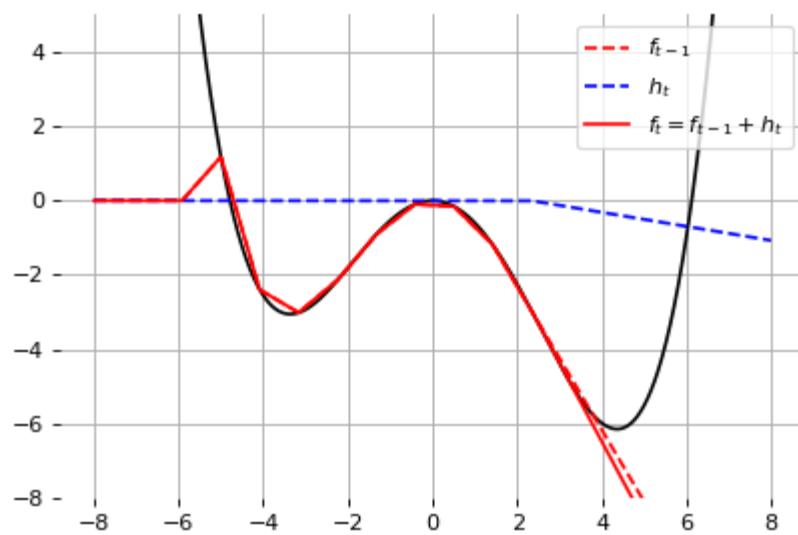


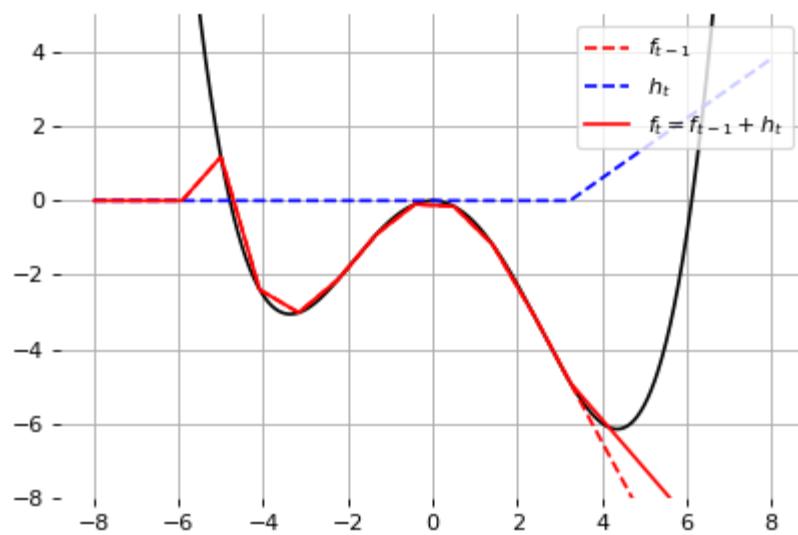


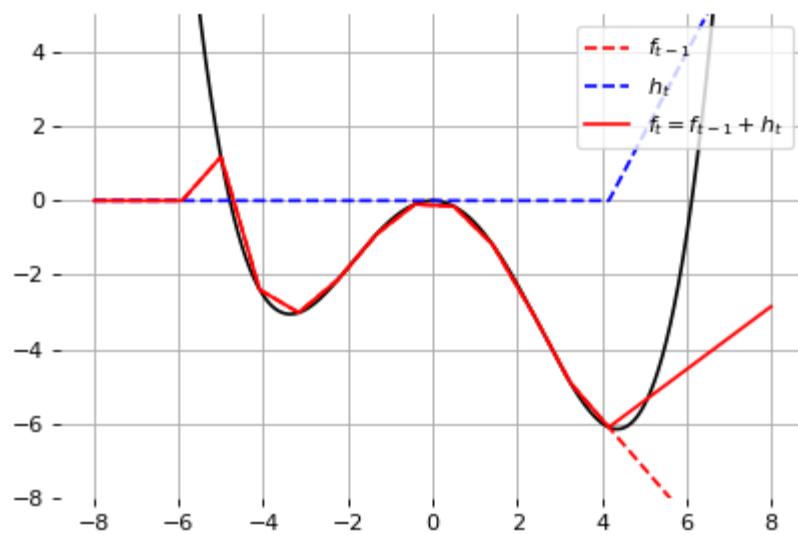


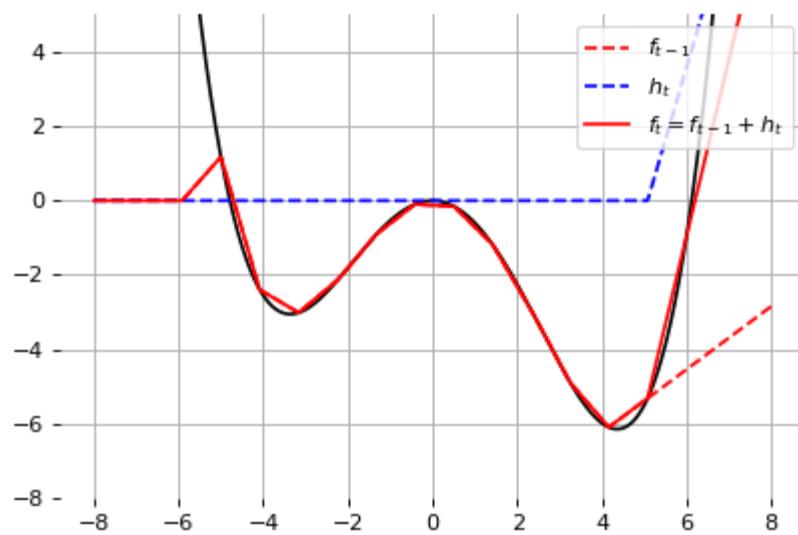






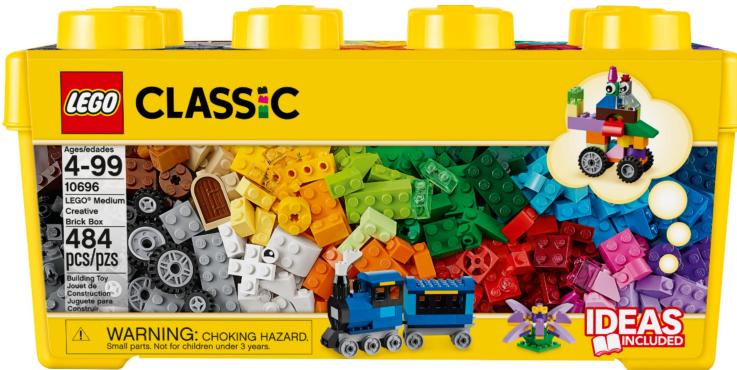






(demo)

# LEGO® Deep Learning

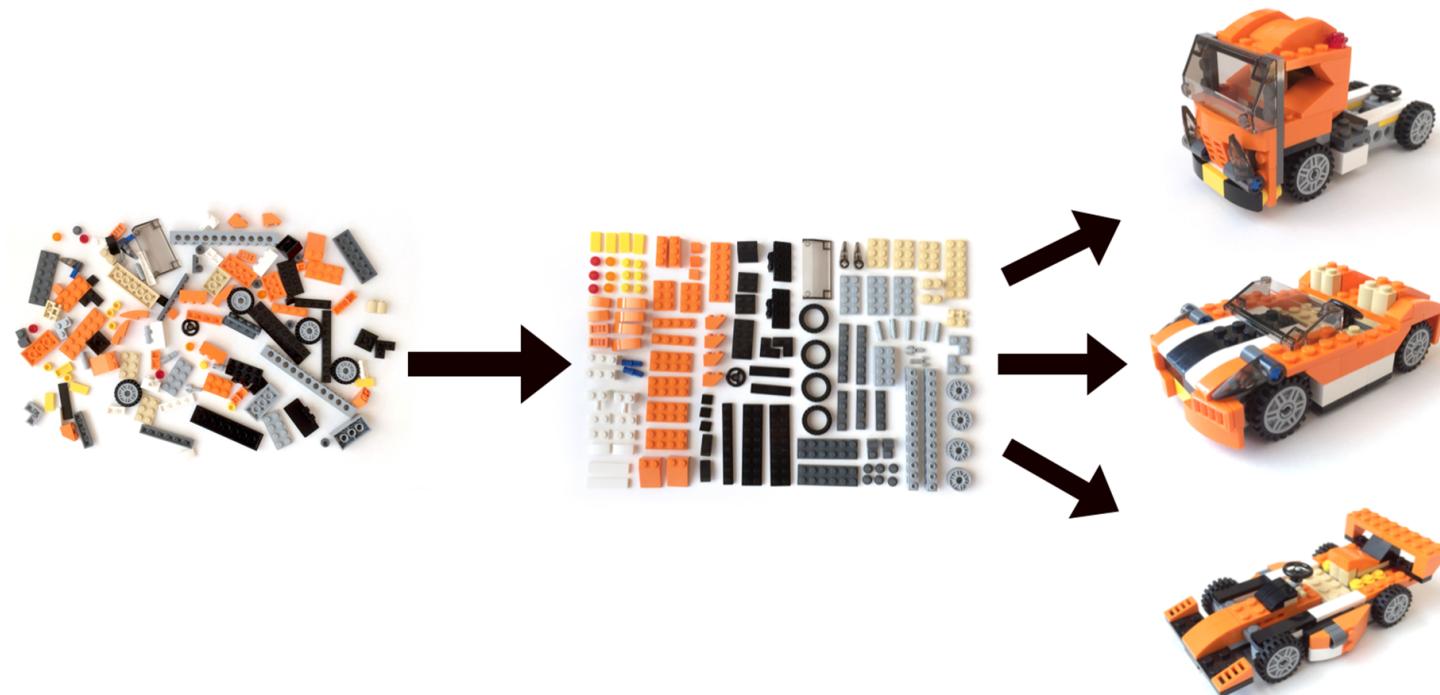




*People are now building a new kind of software by **assembling networks of parameterized functional blocks** and by **training them from examples using some form of gradient-based optimization**.*

Yann LeCun, 2018.

## DL as an architectural language



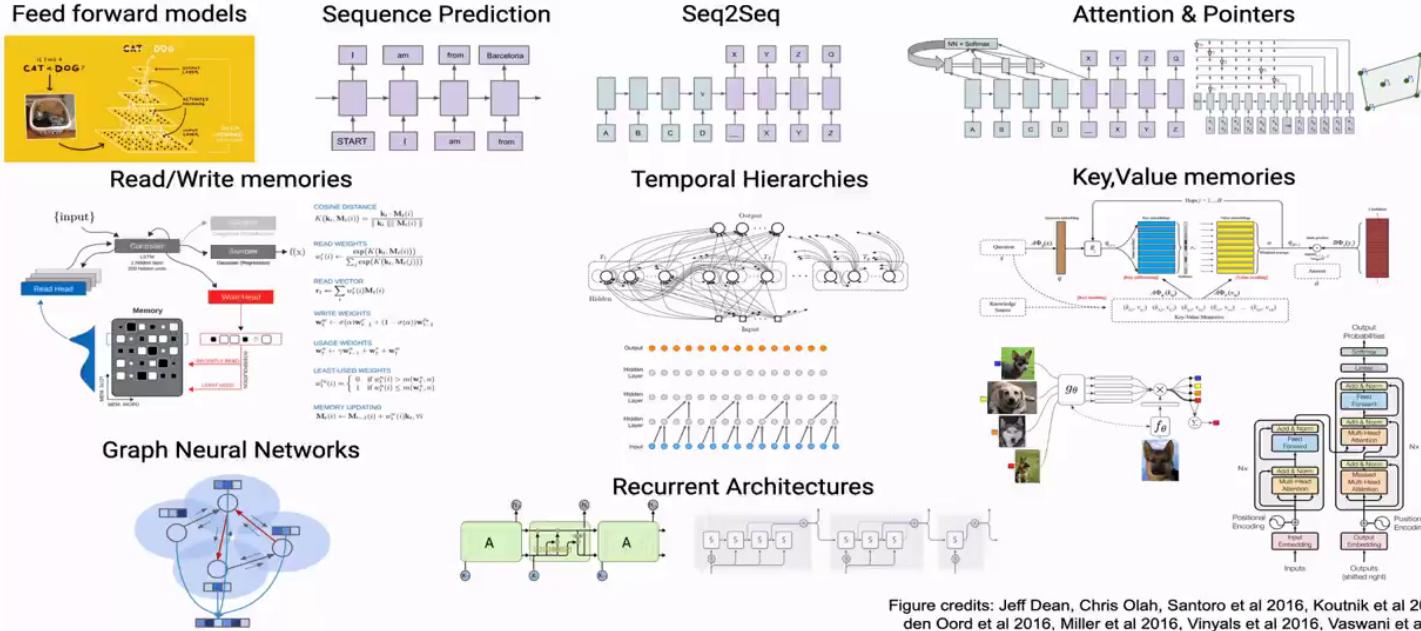
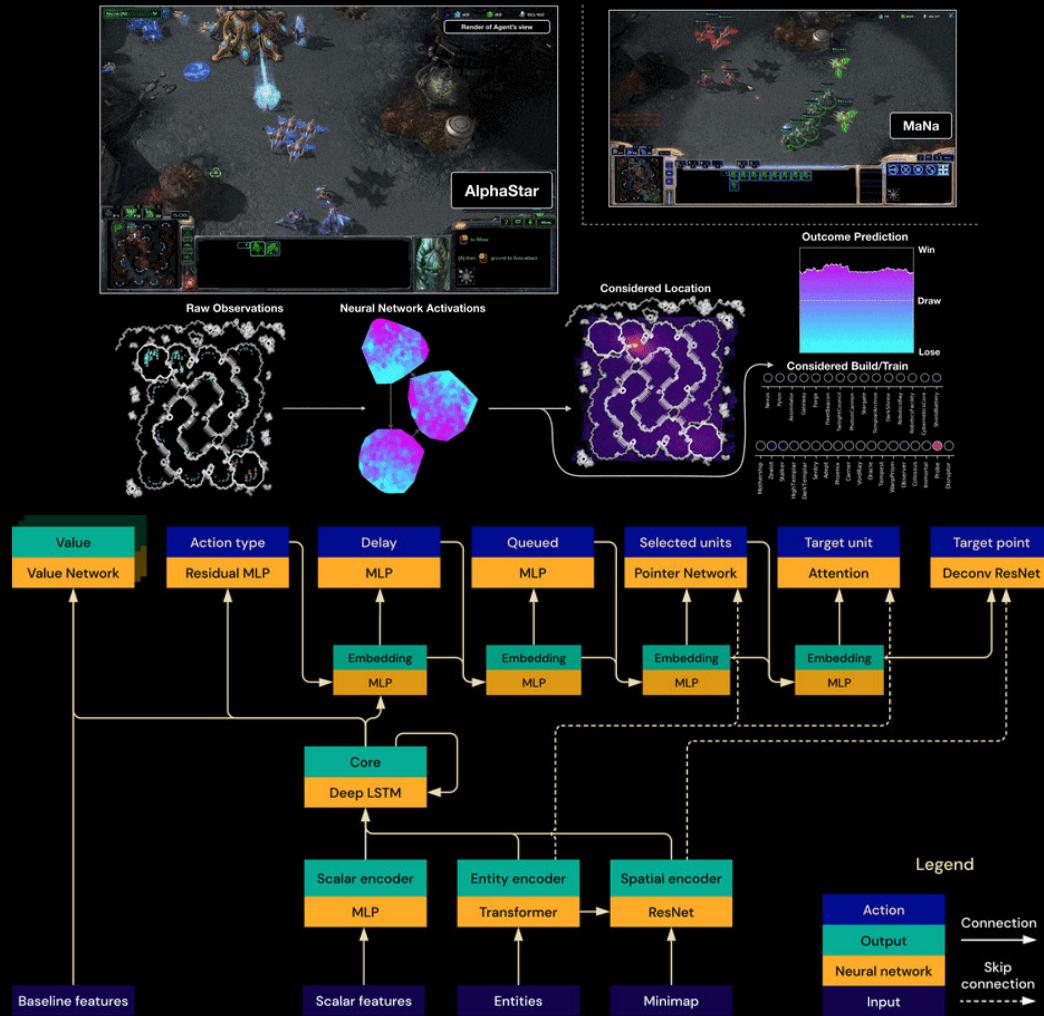


Figure credits: Jeff Dean, Chris Olah, Santoro et al 2016, Koutnik et al 2014, van den Oord et al 2016, Miller et al 2016, Vinyals et al 2016, Vaswani et al 2017

## The toolbox

# LEGO® Creator Expert



The end.

# References

- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), 386.
- Bottou, L., & Bousquet, O. (2008). The tradeoffs of large scale learning. In *Advances in neural information processing systems* (pp. 161-168).
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088), 533.