

# 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

# Cooking recipe

- Get data (loads of them).
- Get good hardware.
- Define the neural network architecture as a composition of differentiable functions.
  - Stick to non-saturating activation function to avoid vanishing gradients.
  - Prefer deep over shallow architectures.
- Optimize with (variants of) stochastic gradient descent.
  - Evaluate gradients with automatic differentiation.

# 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:

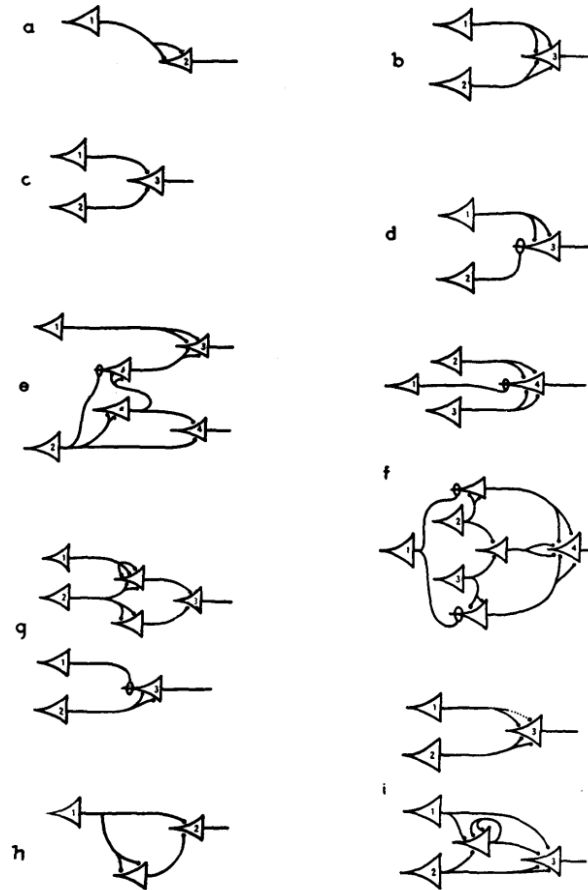
$$f(\mathbf{x}) = 1_{\{\sum_i w_i x_i + b \geq 0\}}$$

This unit can implement:

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

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.

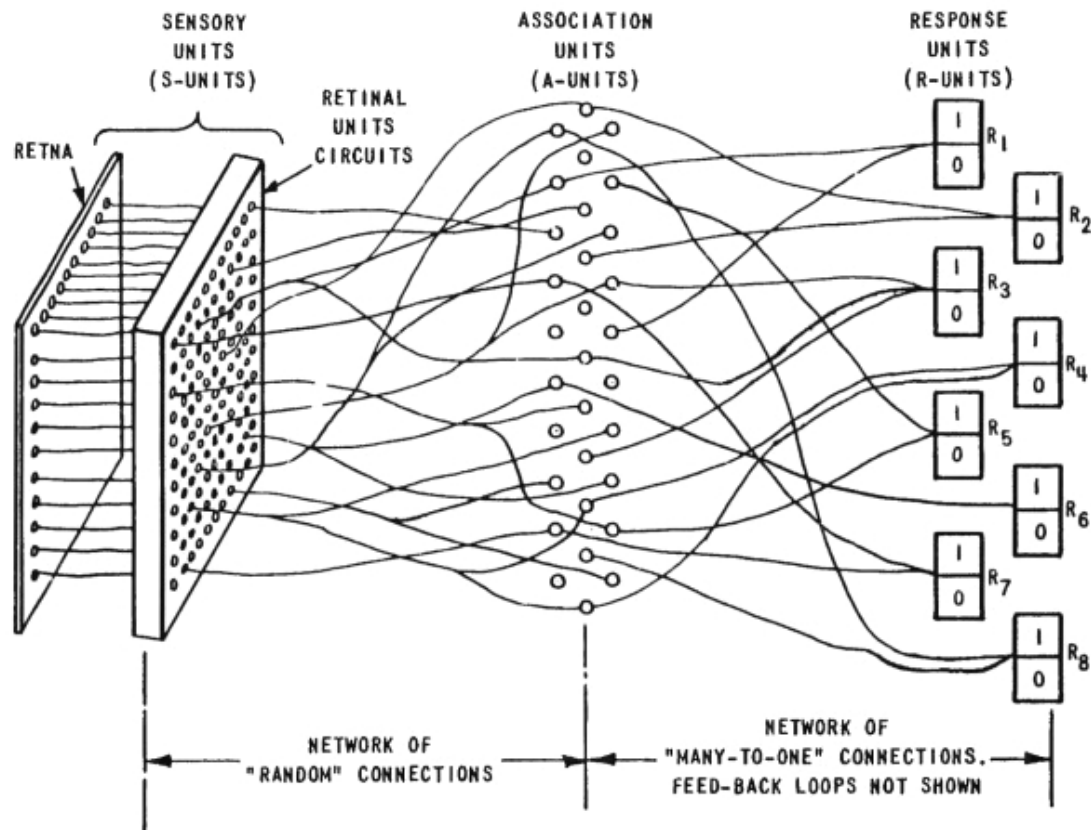
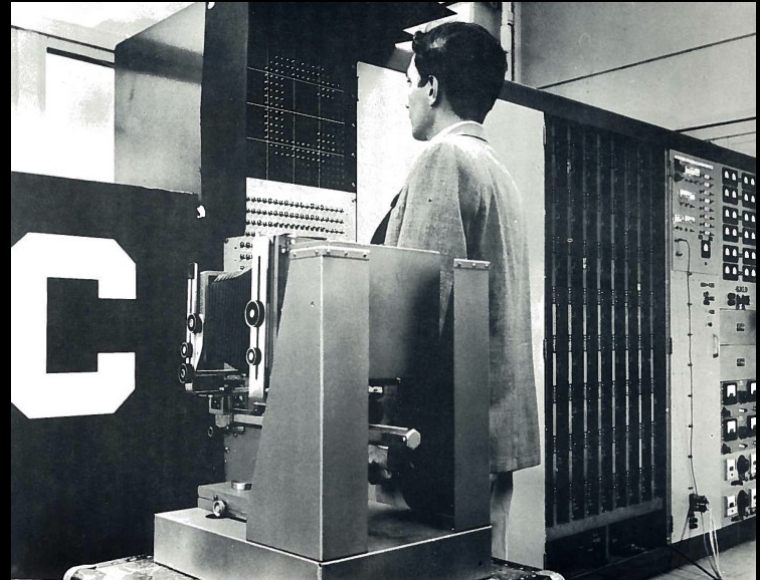
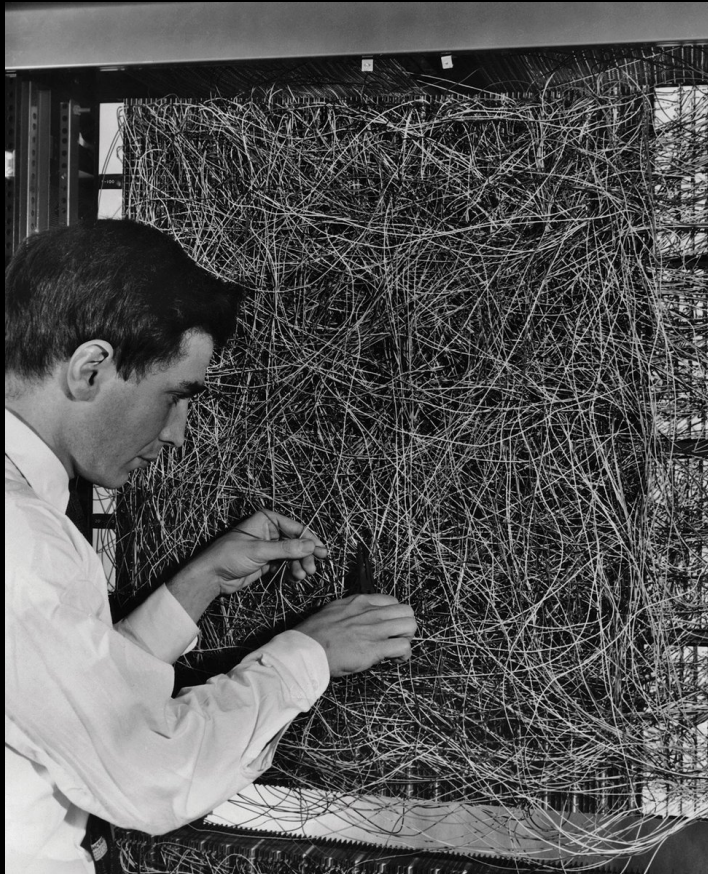


Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON





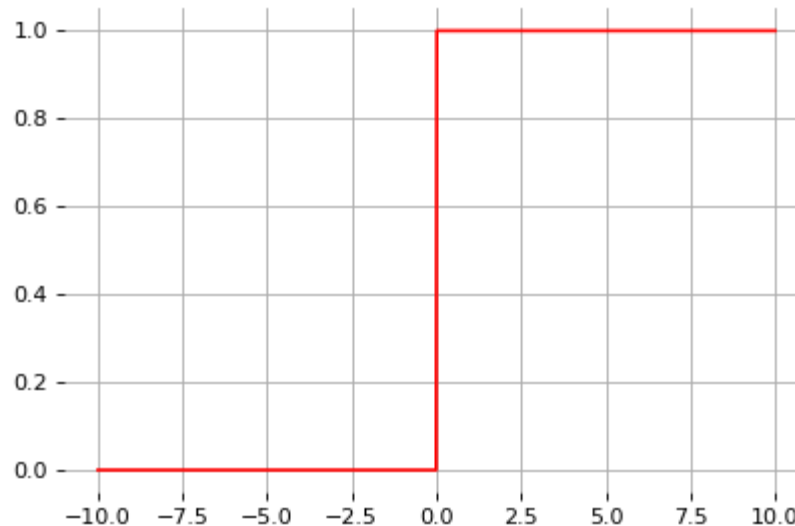
The Mark I Perceptron (Frank Rosenblatt).



The Perceptron

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

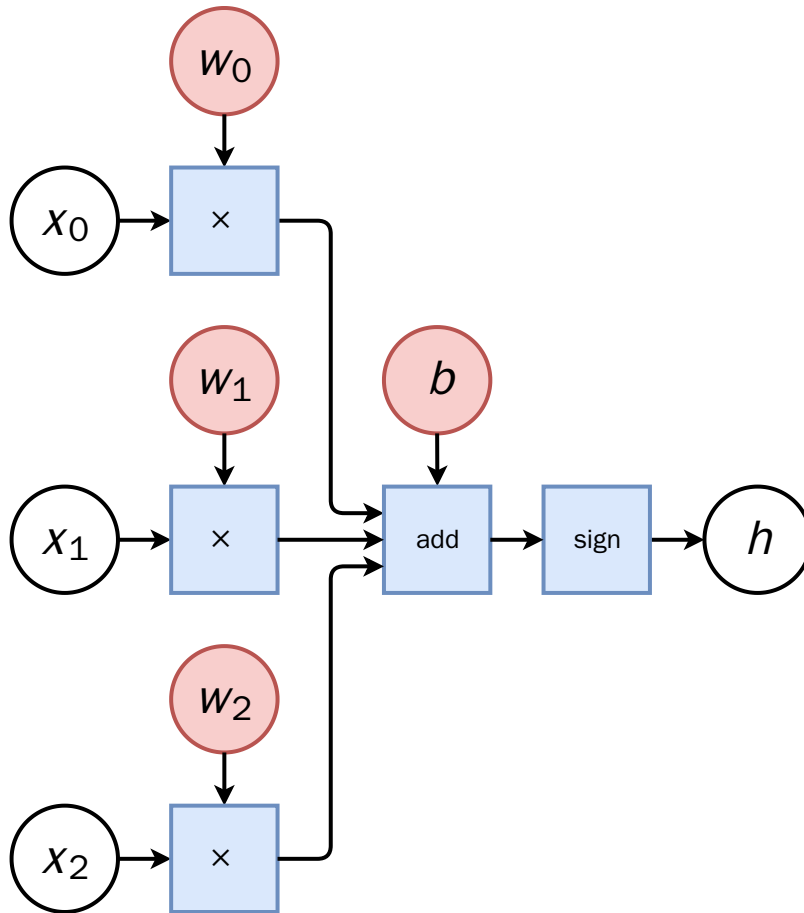
$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



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

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

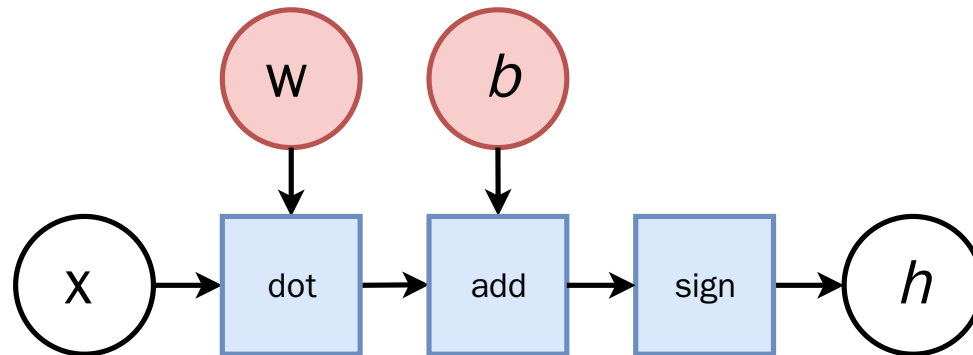
can be represented as a **computational graph** where

- 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

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

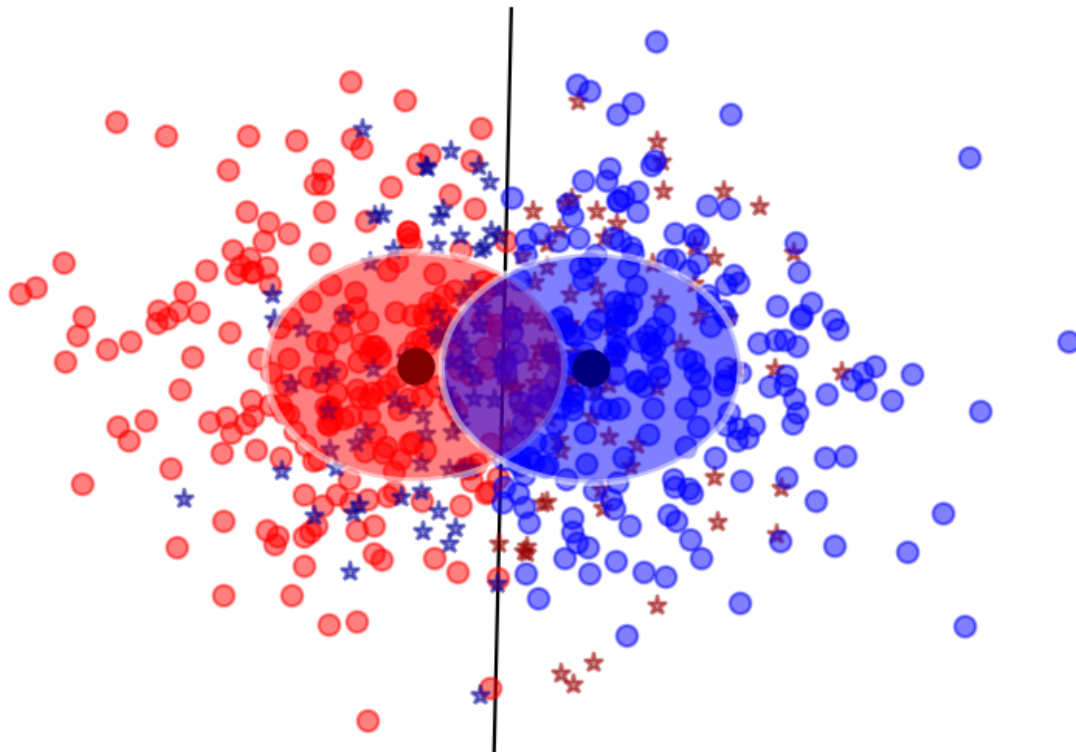
we get

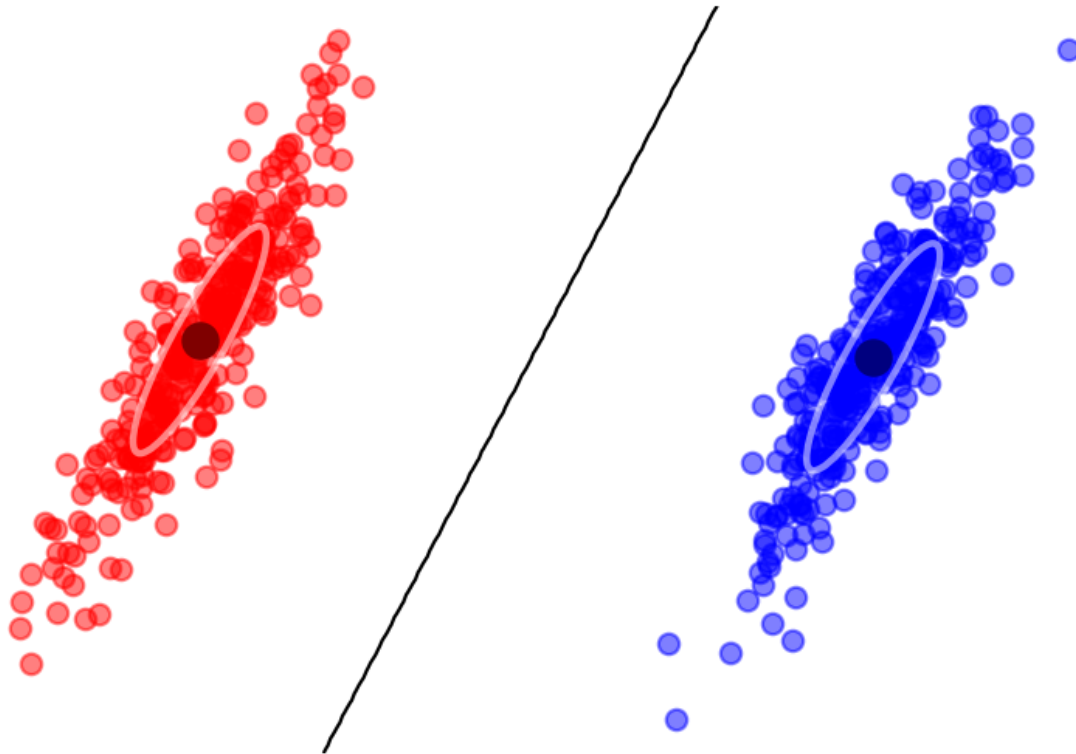
$$P(Y = 1|\mathbf{x}) = \sigma \left( \log \frac{P(\mathbf{x}|Y = 1)}{P(\mathbf{x}|Y = 0)} + \log \frac{P(Y = 1)}{P(Y = 0)} \right).$$

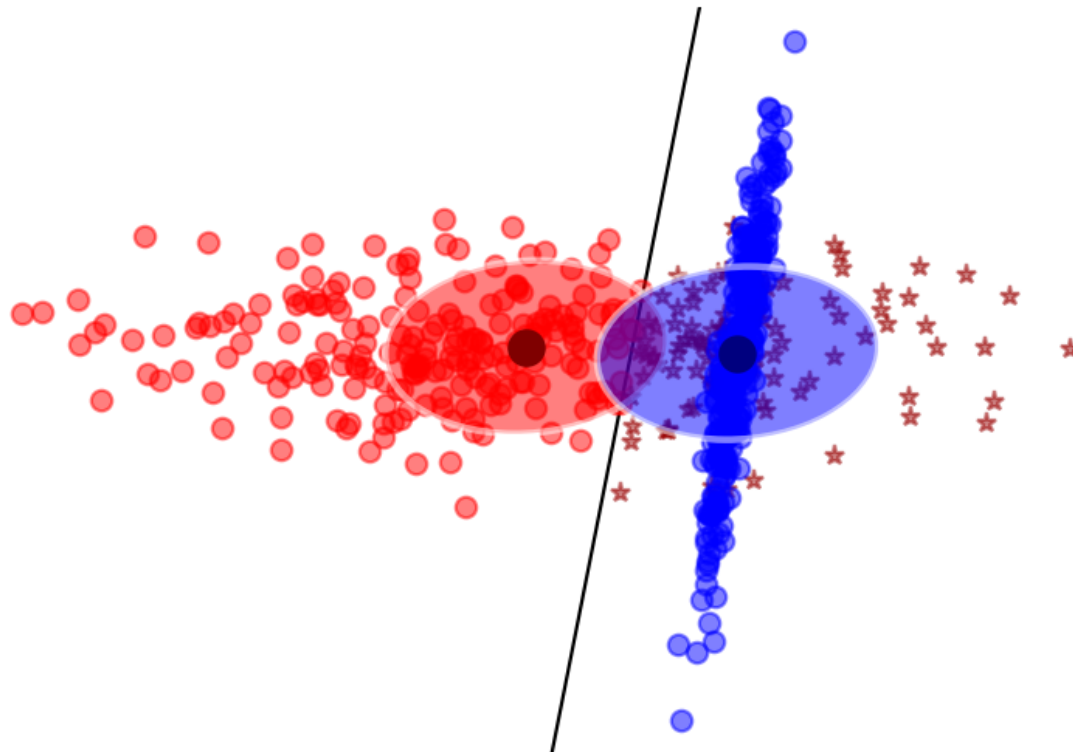


Therefore,

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



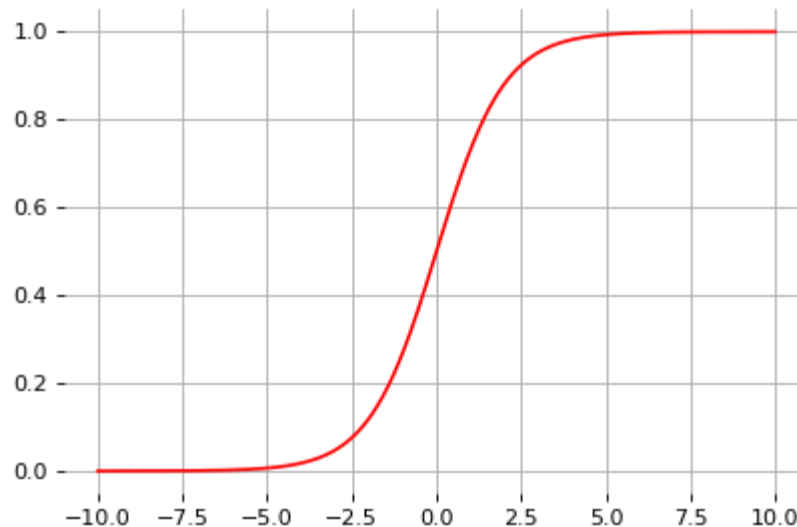




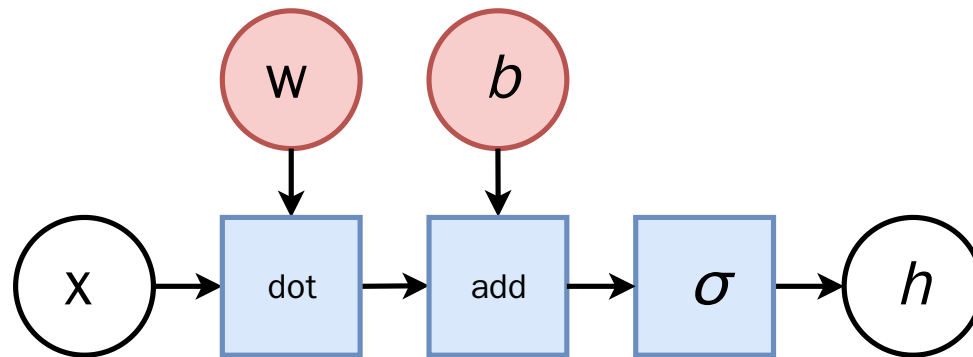
Note that the **sigmoid** function

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

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 **lego brick** 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**

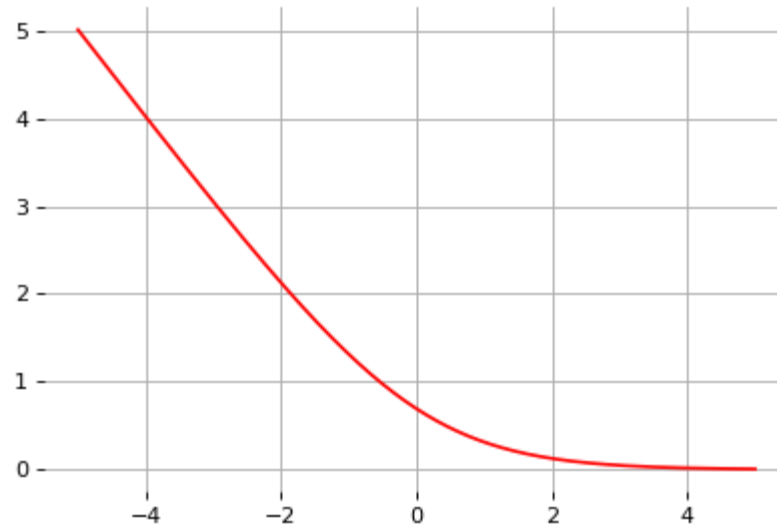
$$H(p, q) = \mathbb{E}_p[-\log q]$$

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**.

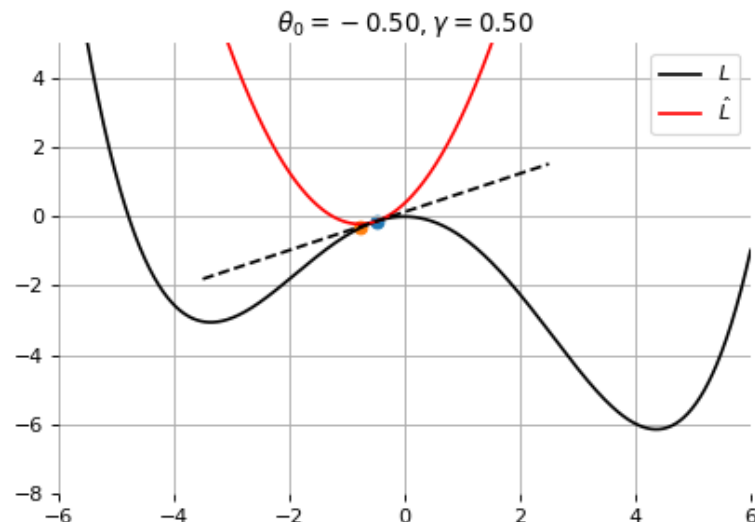
# Gradient descent

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

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



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

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

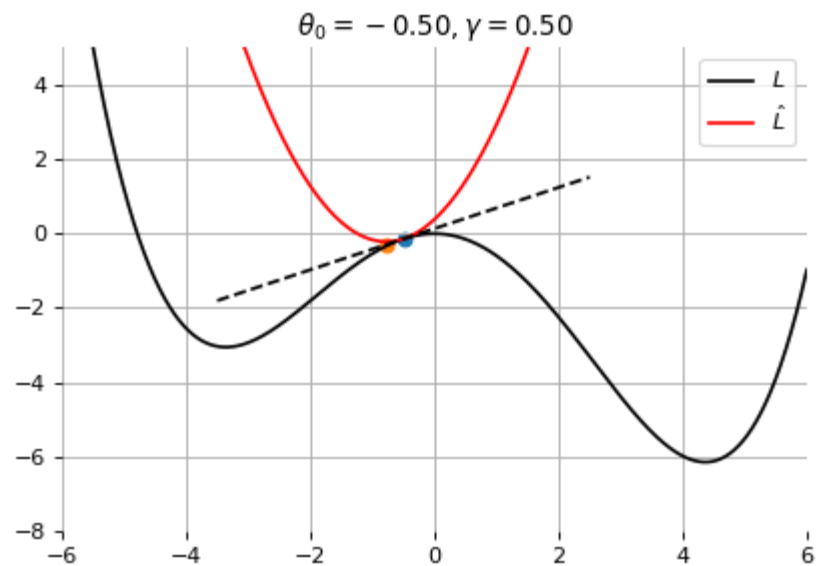
which results in the best improvement for the step  $\epsilon = -\gamma \nabla_{\theta} \mathcal{L}(\theta_0)$ .

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

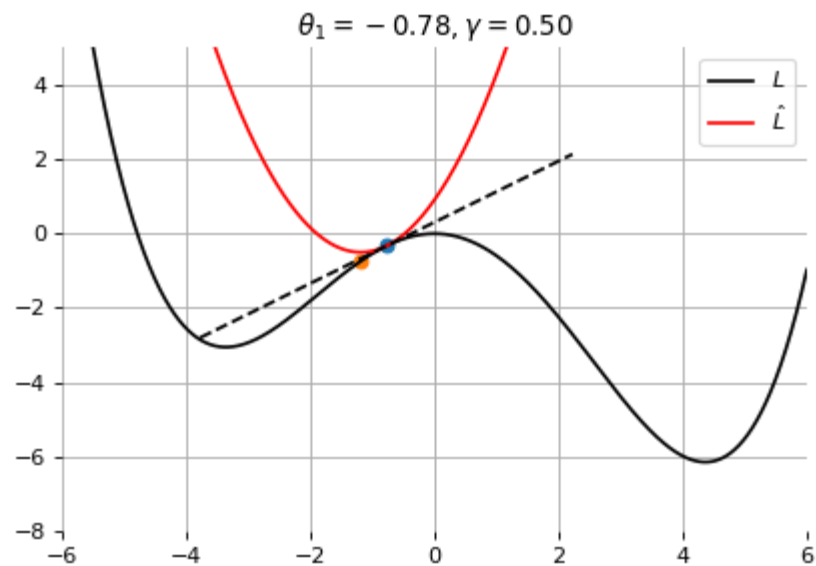
$$\theta_{t+1} = \theta_t - \gamma \nabla_{\theta} \mathcal{L}(\theta_t),$$

where

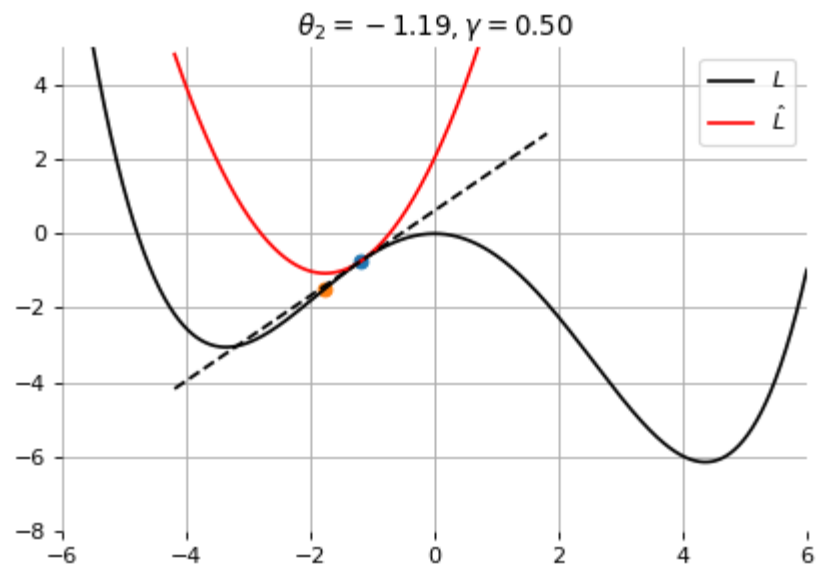
- $\theta_0$  are the initial parameters of the model;
- $\gamma$  is the **learning rate**;
- both are critical for the convergence of the update rule.



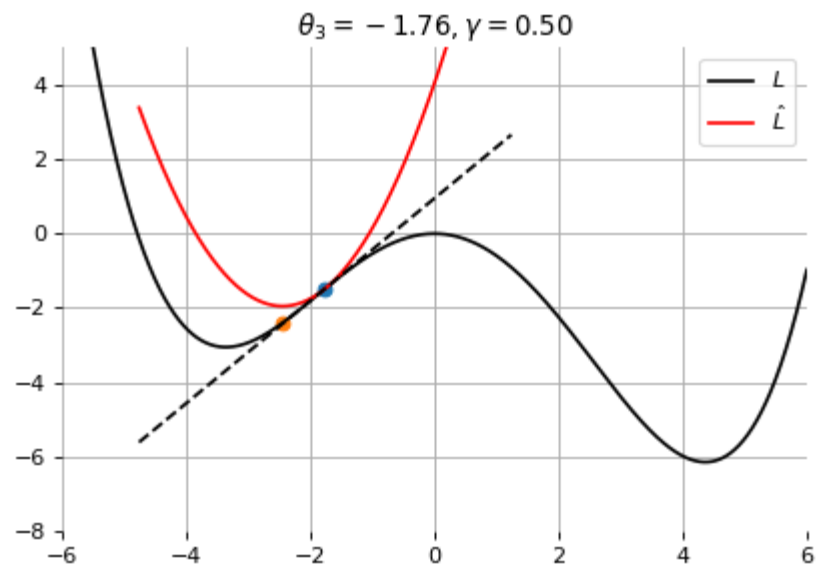
Example 1: Convergence to a local minima



Example 1: Convergence to a local minima

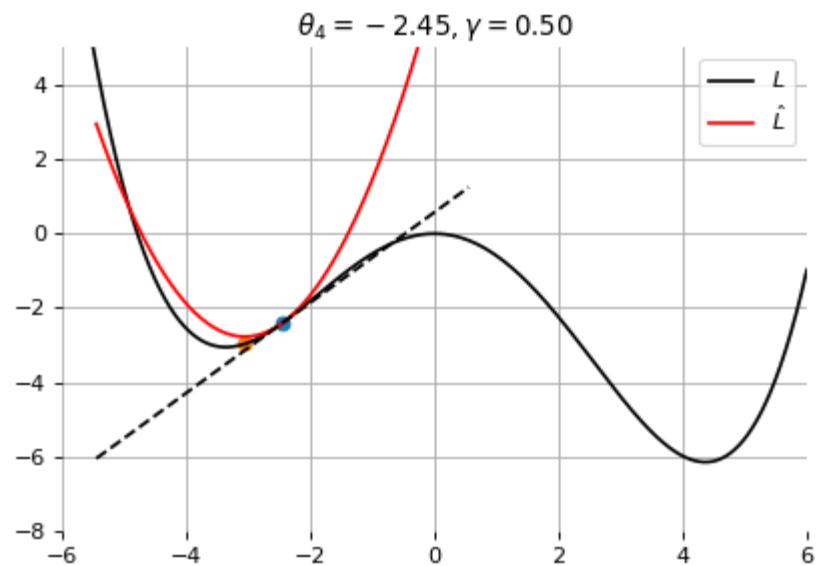


Example 1: Convergence to a local minima

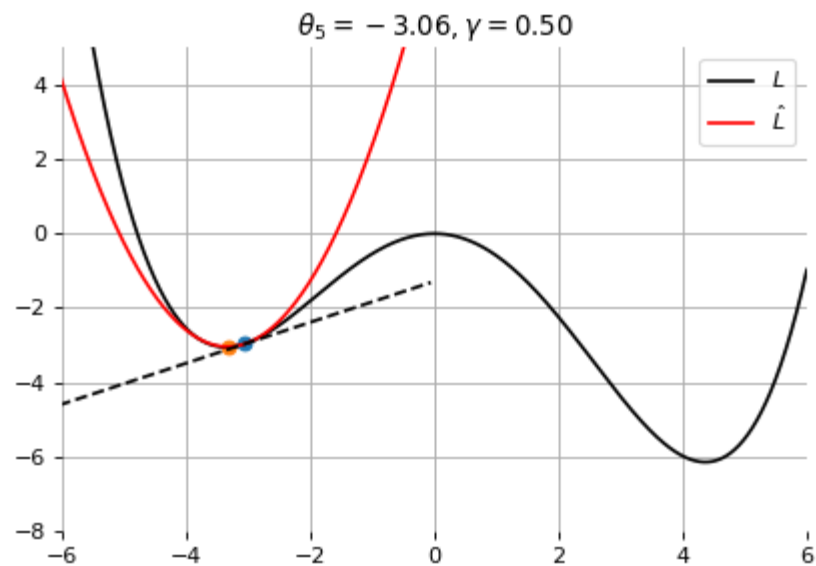


Example 1: Convergence to a local minima

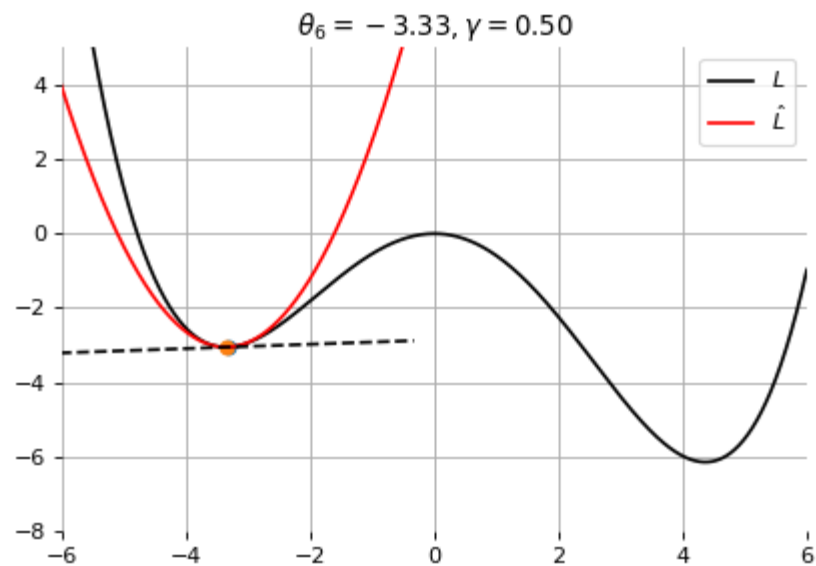




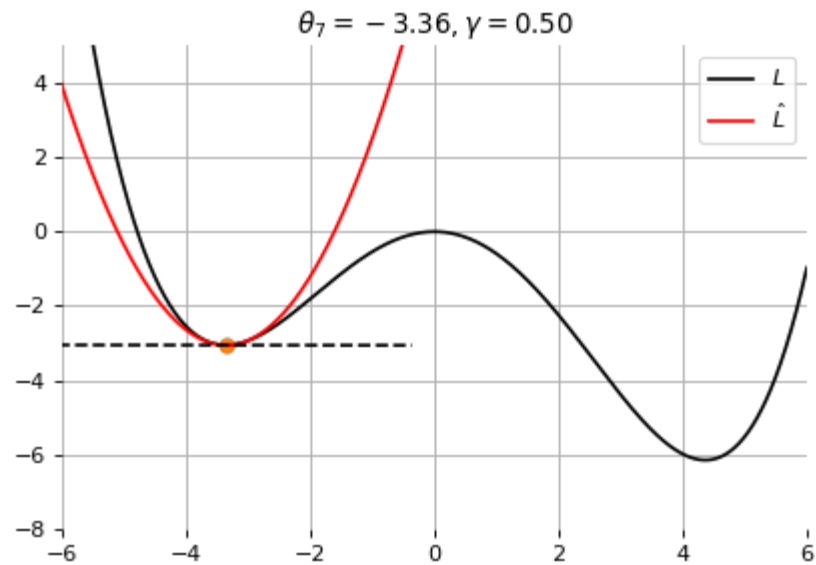
Example 1: Convergence to a local minima



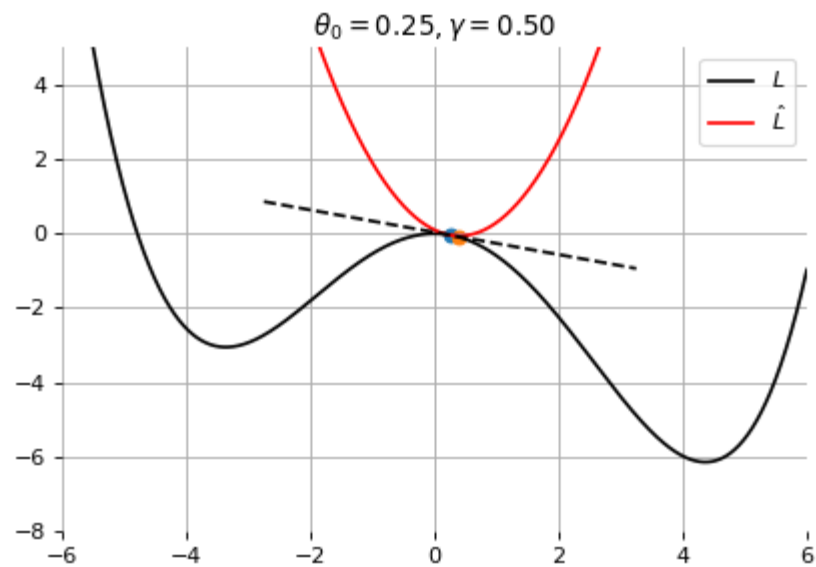
Example 1: Convergence to a local minima



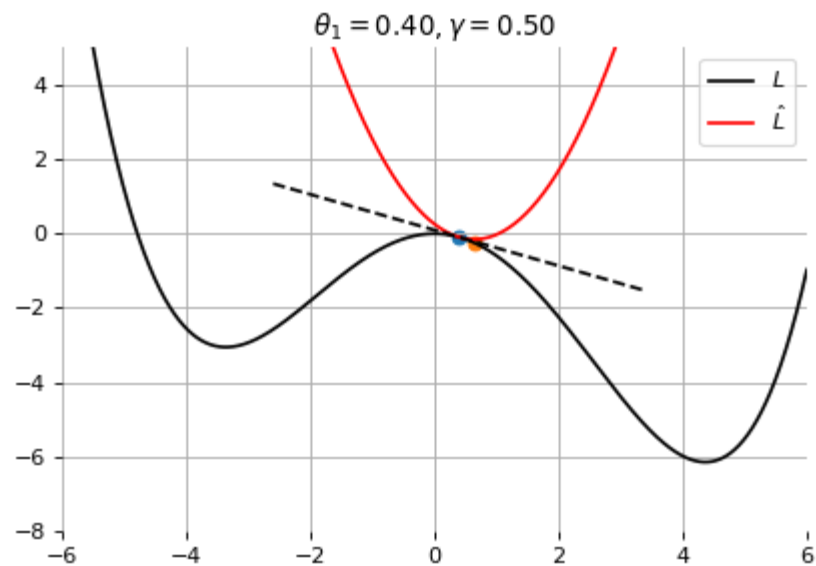
Example 1: Convergence to a local minima



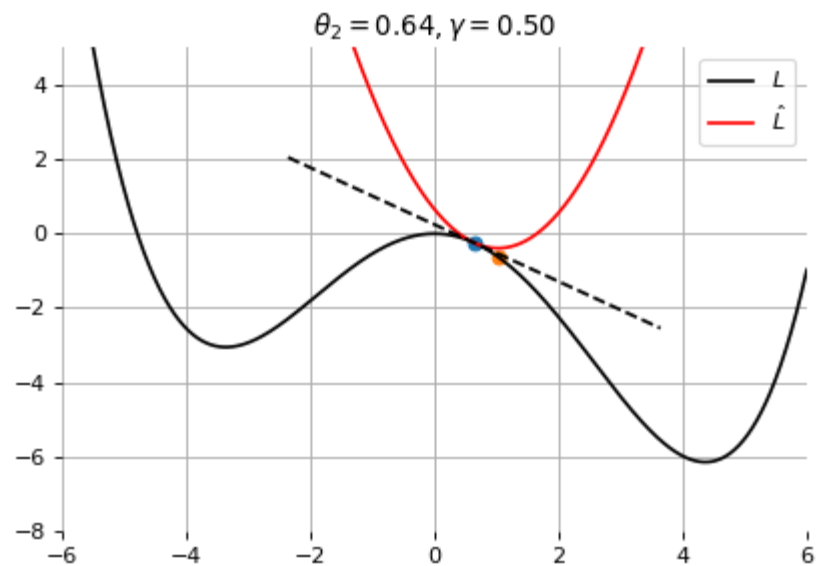
Example 1: Convergence to a local minima



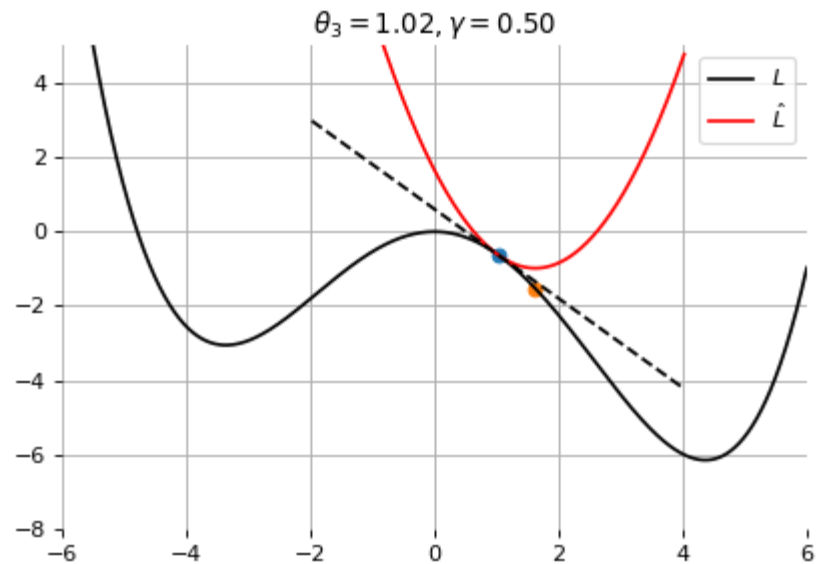
Example 2: Convergence to the global minima



Example 2: Convergence to the global minima

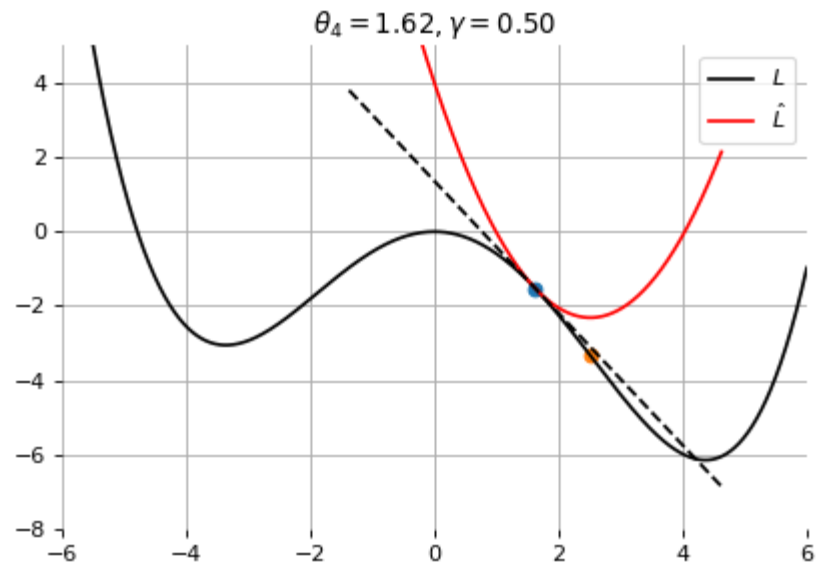


Example 2: Convergence to the global minima

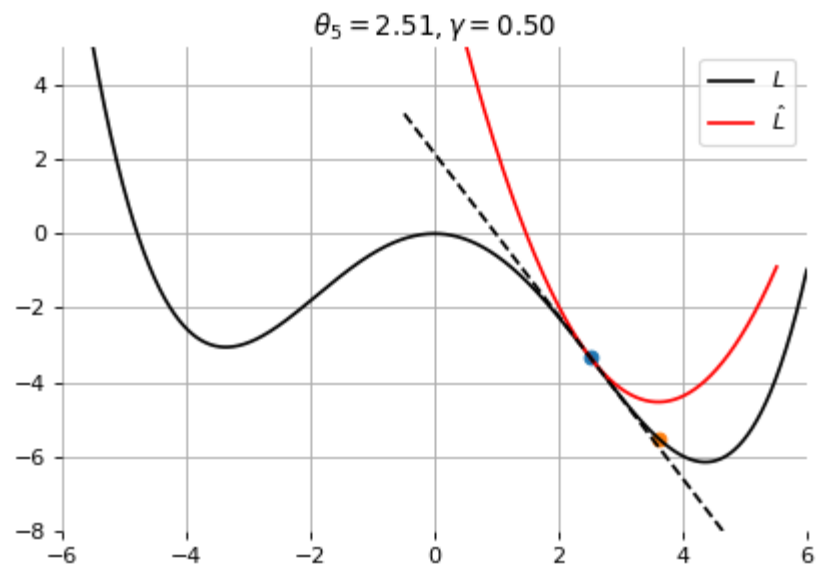


Example 2: Convergence to the global minima

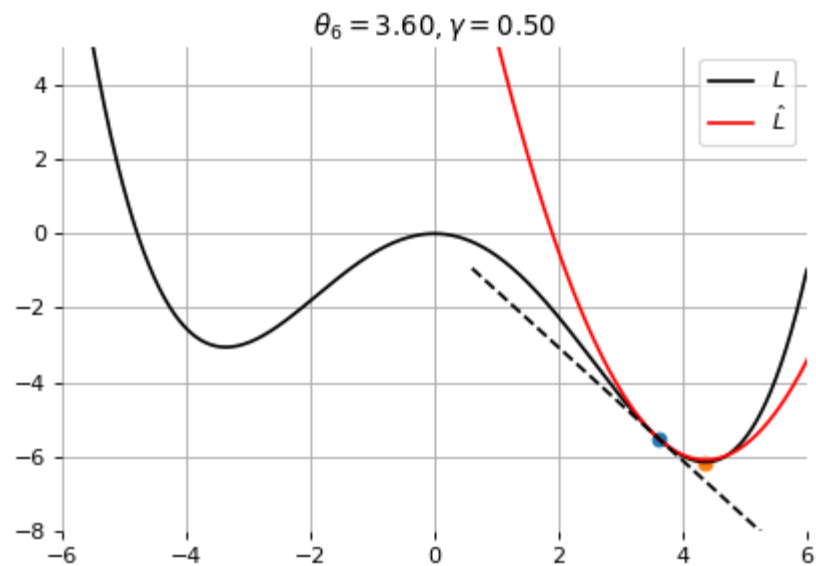




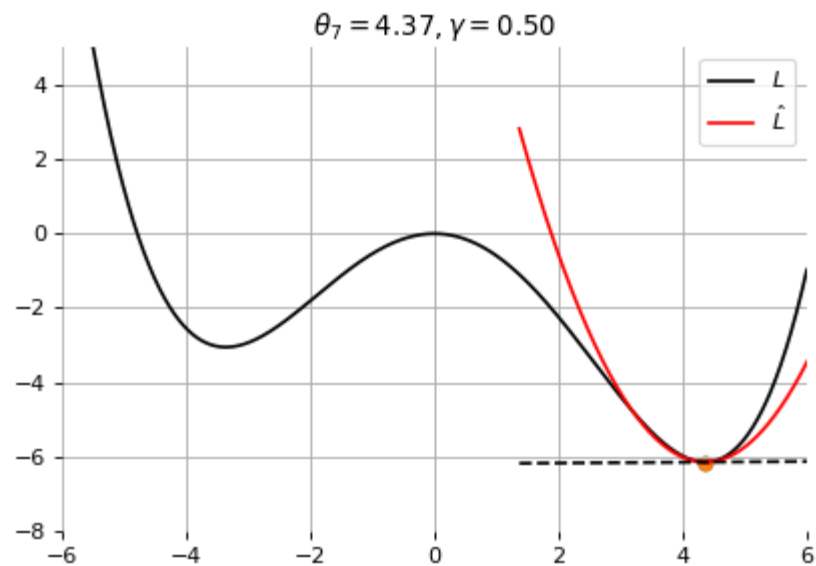
Example 2: Convergence to the global minima



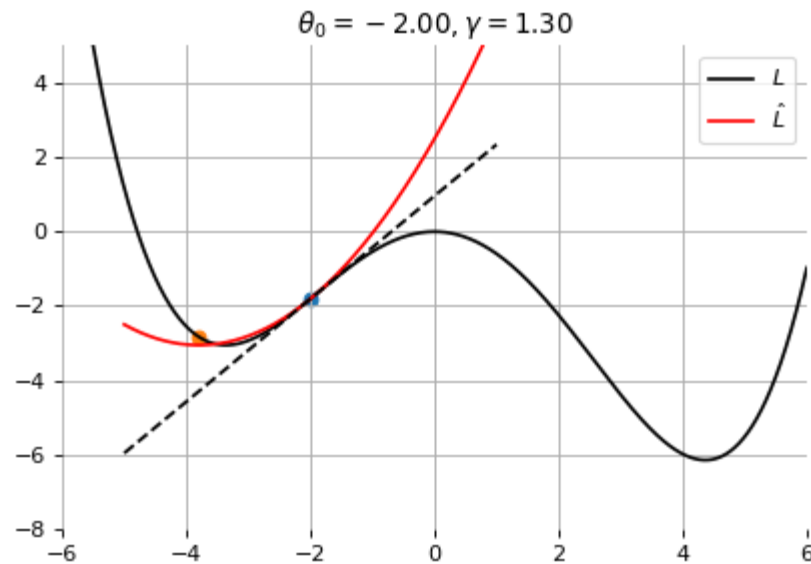
Example 2: Convergence to the global minima



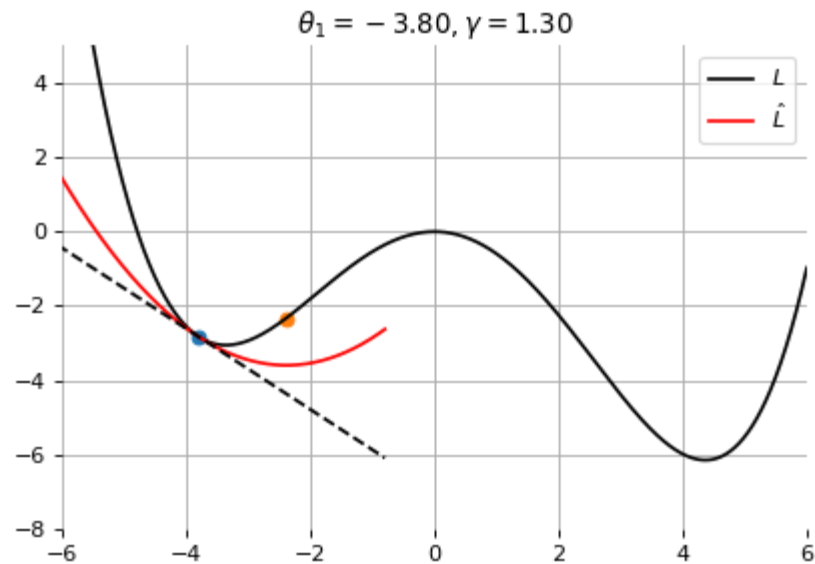
Example 2: Convergence to the global minima



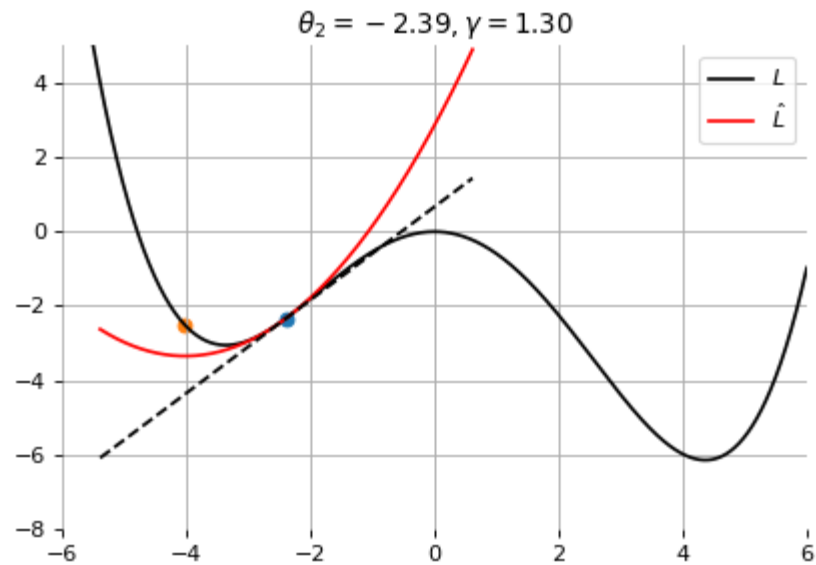
Example 2: Convergence to the global minima



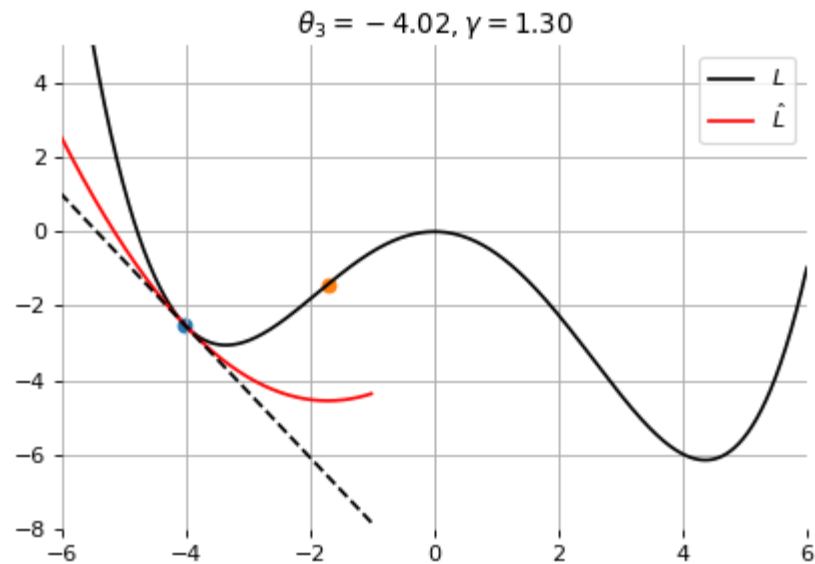
Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate

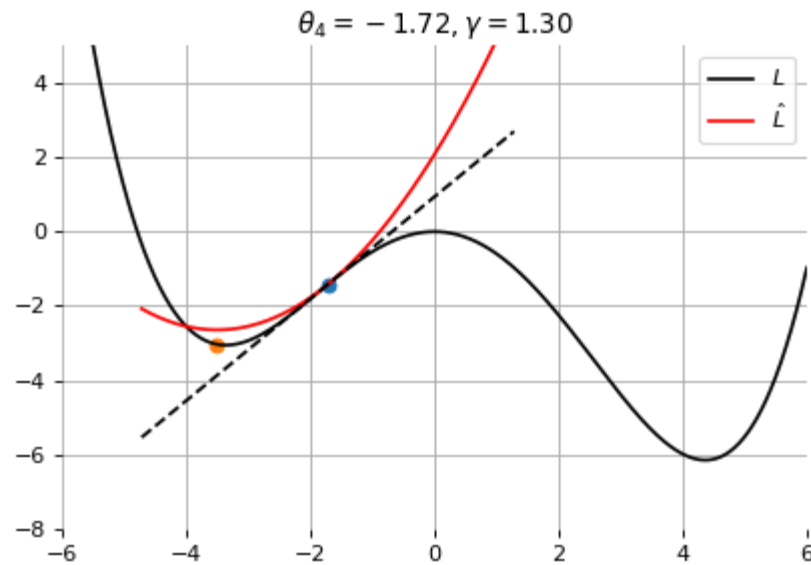


Example 3: Divergence due to a too large learning rate

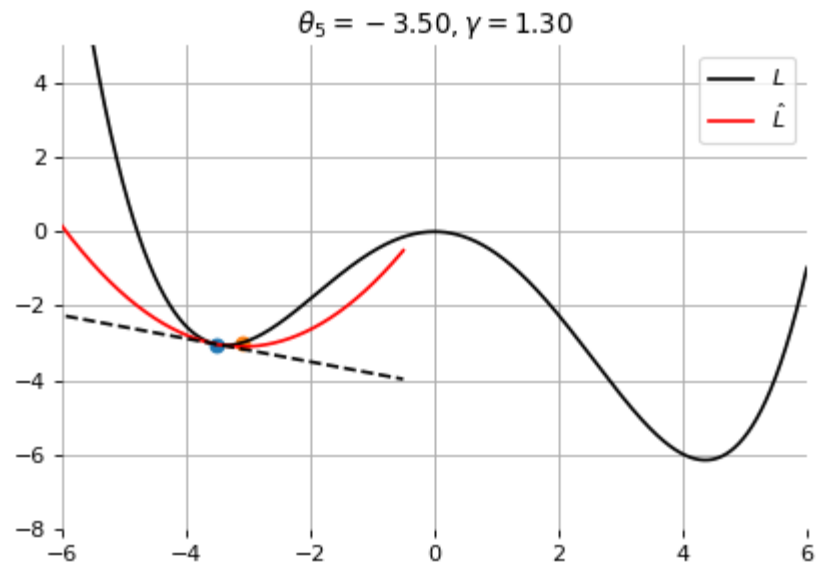


Example 3: Divergence due to a too large learning rate





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:

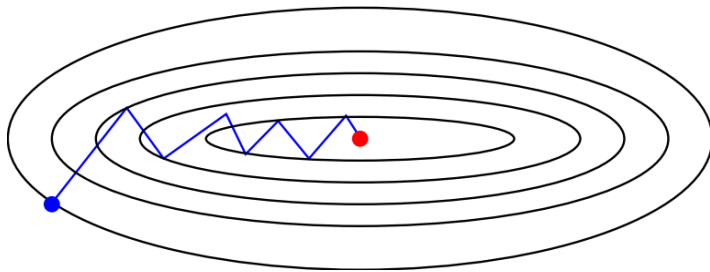
$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- 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.

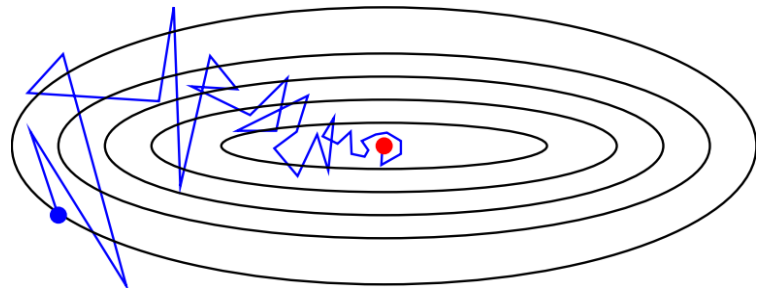
Instead, **stochastic** gradient descent uses as update rule:

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

- 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).

- Interestingly, if training examples  $\mathbf{x}_i, y_i \sim P_{X,Y}$  are received and used in an online fashion, then SGD directly minimizes the **expected** risk.

When decomposing the excess error in terms of approximation, estimation and optimization errors, stochastic algorithms yield the best generalization performance (in terms of **expected** risk) despite being the worst optimization algorithms (in terms of **empirical risk**) (Bottou, 2011).

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

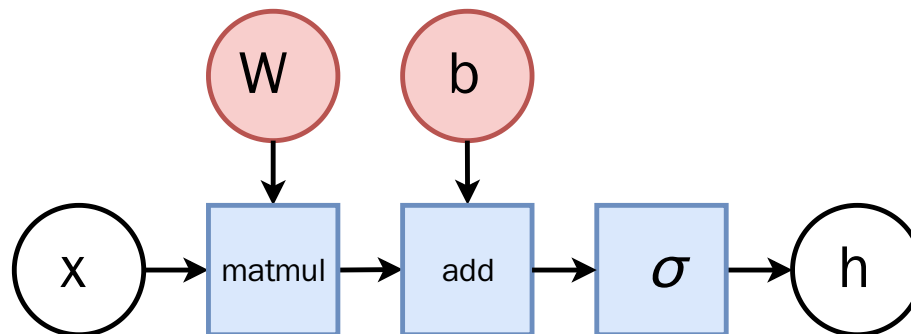
# Layers

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.





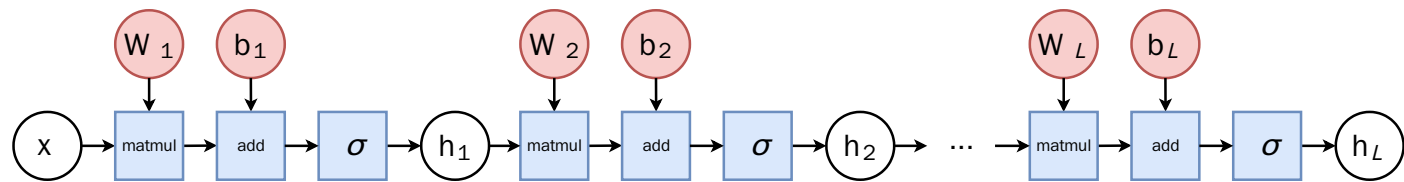
# Multi-layer perceptron

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

$$\begin{aligned}\mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\ &\dots \\ \mathbf{h}_L &= \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L) \\ f(\mathbf{x}; \theta) &= \hat{y} = \mathbf{h}_L\end{aligned}$$

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.



## Classification

- 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 (normalized) vector  $\mathbf{h}_L \in [0, 1]^C$  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

The last activation  $\sigma$  can be skipped to produce unbounded output values  $h_L \in \mathbb{R}$ .

# Automatic differentiation

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

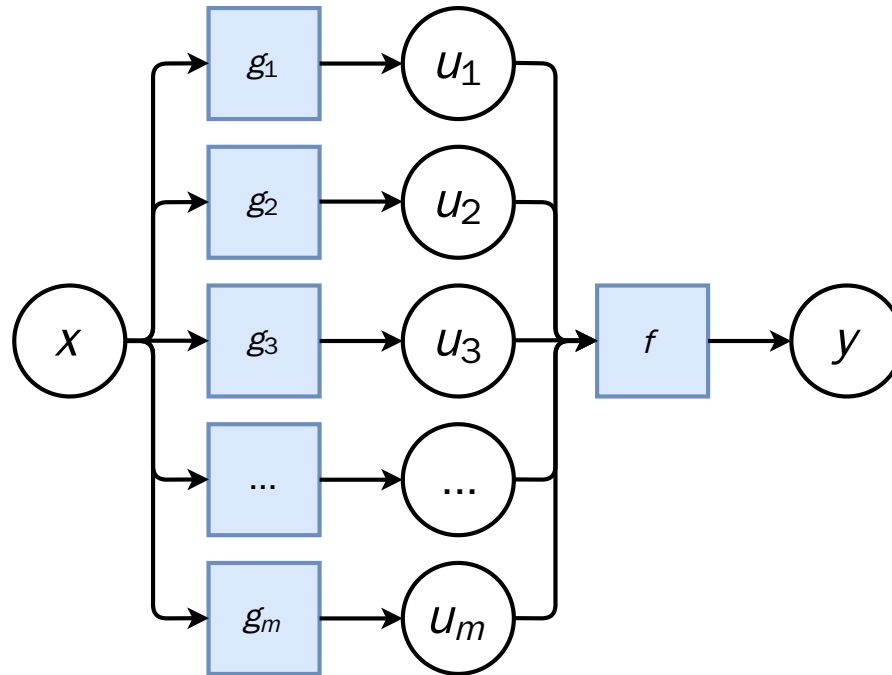
Therefore, we require the evaluation of the (total) derivatives

$$\frac{d\ell}{d\mathbf{W}_k}, \frac{d\ell}{d\mathbf{b}_k}$$

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**.

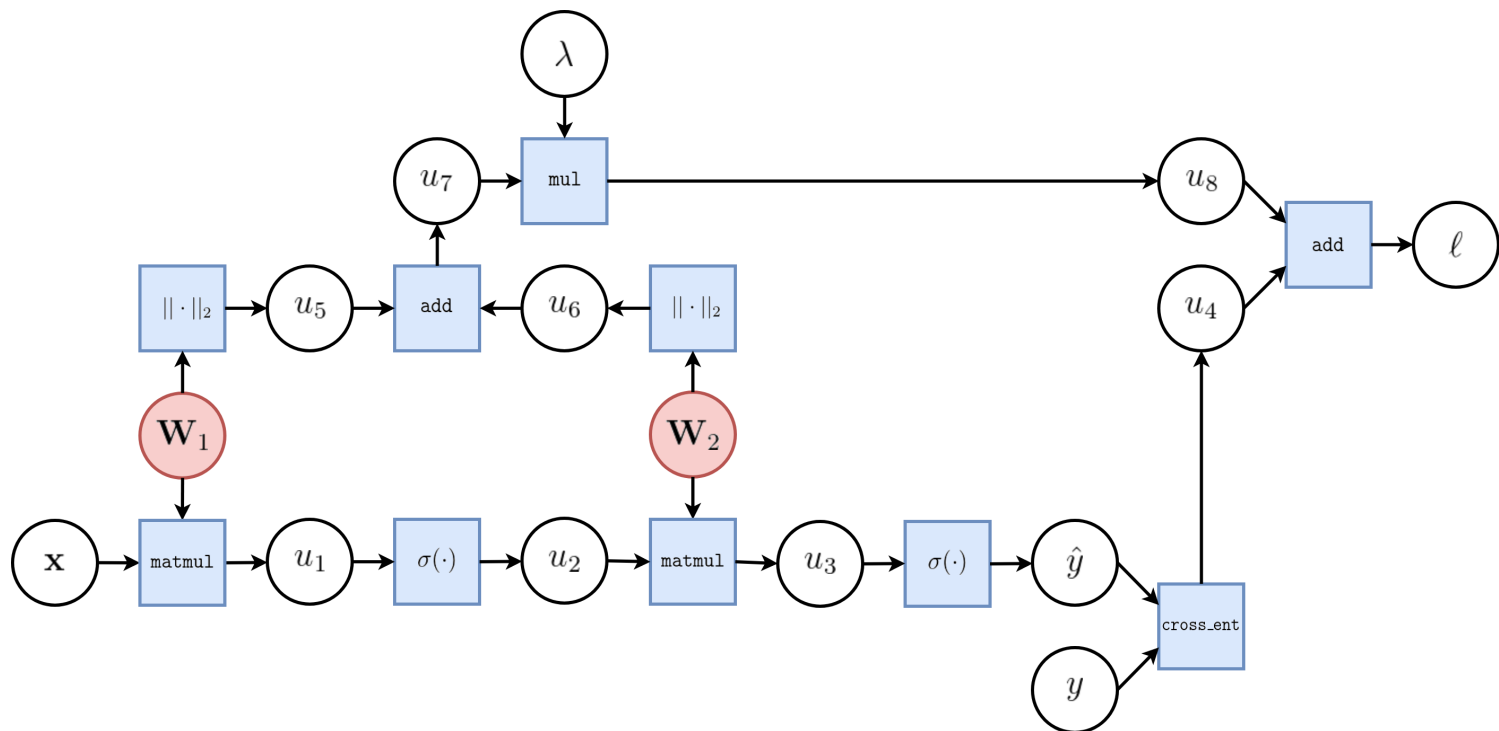


Let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma(\mathbf{W}_2^T \sigma(\mathbf{W}_1^T \mathbf{x}))$$

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

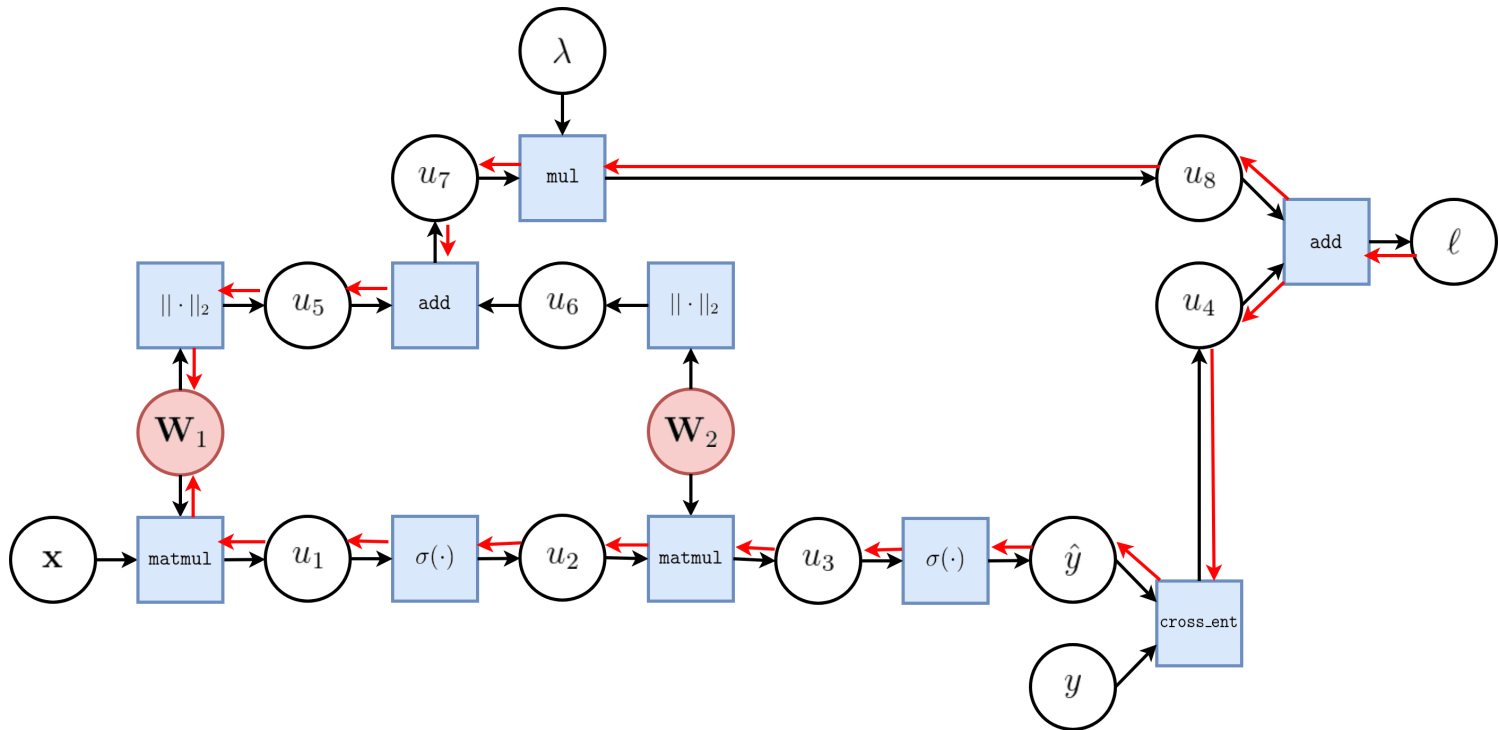
for  $\mathbf{x} \in \mathbb{R}^p, y \in \mathbb{R}, \mathbf{W}_1 \in \mathbb{R}^{p \times q}$  and  $\mathbf{W}_2 \in \mathbb{R}^q$ .



The total derivative  $\frac{d\ell}{d\mathbf{W}_1}$  can be computed **backward**, by walking through all paths from  $\ell$  to  $\mathbf{W}_1$  in the computational graph and accumulating the terms:

$$\frac{d\ell}{d\mathbf{W}_1} = \frac{\partial \ell}{\partial u_8} \frac{du_8}{d\mathbf{W}_1} + \frac{\partial \ell}{\partial u_4} \frac{du_4}{d\mathbf{W}_1}$$

$$\frac{du_8}{d\mathbf{W}_1} = \dots$$

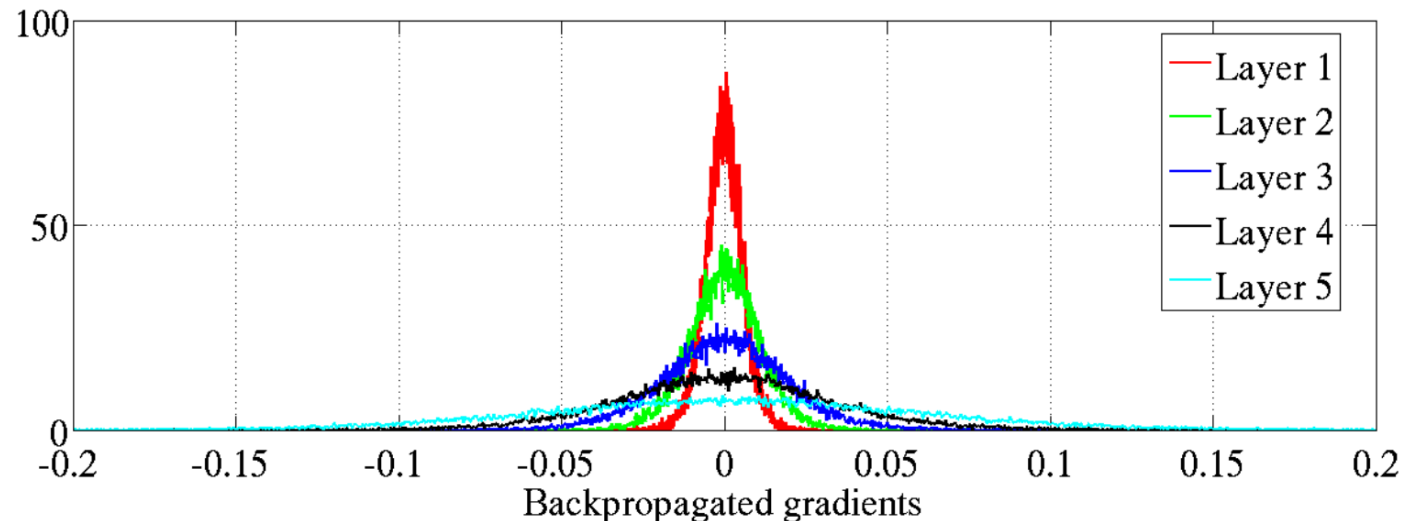


- This algorithm is also known as **backpropagation**.
- An equivalent procedure can be defined to evaluate the derivatives in **forward mode**, from inputs to outputs.
- Since differentiation is a linear operator, automatic differentiation can be implemented efficiently in terms of tensor operations.

# 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  $x, w_1, w_2, w_3 \in \mathbb{R}$ , such that

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

Under the hood, this would be evaluated as

$$u_1 = w_1 x$$

$$u_2 = \sigma(u_1)$$

$$u_3 = w_2 u_2$$

$$u_4 = \sigma(u_3)$$

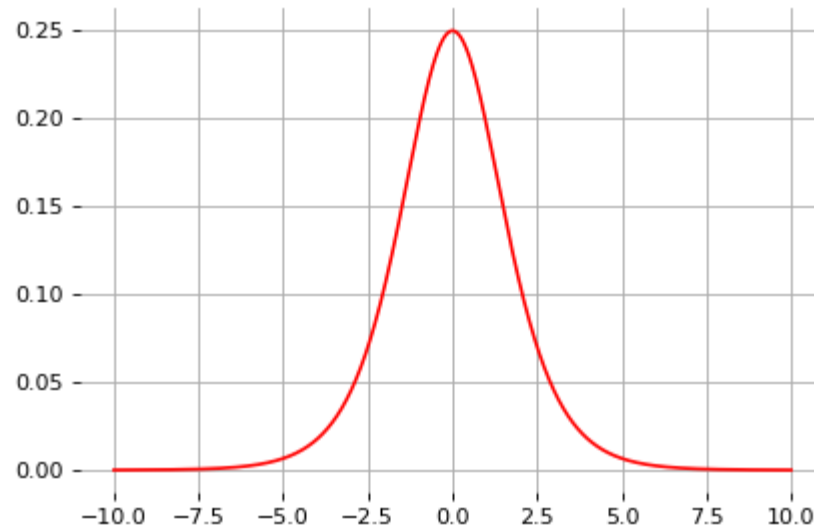
$$u_5 = w_3 u_4$$

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

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.

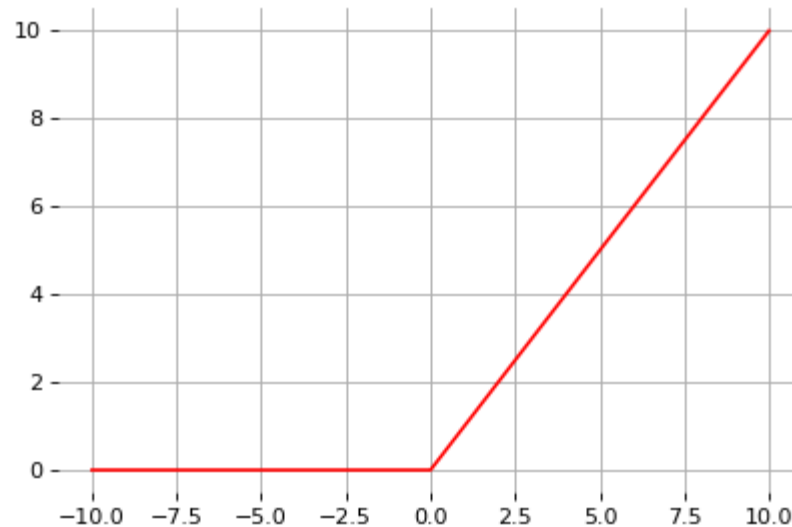
Hence the vanishing gradient problem.

- 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):

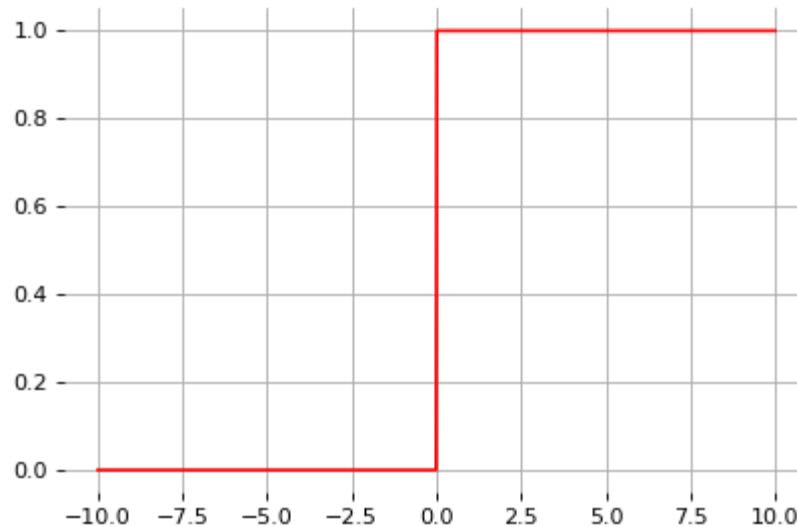
$$\text{ReLU}(x) = \max(0, x)$$





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

**Theorem.** (Cybenko 1989; Hornik et al, 1991) Let  $\sigma(\cdot)$  be a bounded, non-constant continuous function. Let  $I_p$  denote the  $p$ -dimensional hypercube, and  $C(I_p)$  denote the space of continuous functions on  $I_p$ . Given any  $f \in C(I_p)$  and  $\epsilon > 0$ , there exists  $q > 0$  and  $v_i, w_i, b_i, i = 1, \dots, q$  such that

$$F(x) = \sum_{i \leq q} v_i \sigma(w_i^T x + b_i)$$

satisfies

$$\sup_{x \in I_p} |f(x) - F(x)| < \epsilon.$$

- It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth);
- It does not inform about good/bad architectures, nor how they relate to the optimization procedure.
- The universal approximation theorem generalizes to any non-polynomial (possibly unbounded) activation function, including the ReLU (Leshno, 1993).

**Theorem** (Barron, 1992) The mean integrated square error between the estimated network  $\hat{F}$  and the target function  $f$  is bounded by

$$O \left( \frac{C_f^2}{q} + \frac{qp}{N} \log N \right)$$

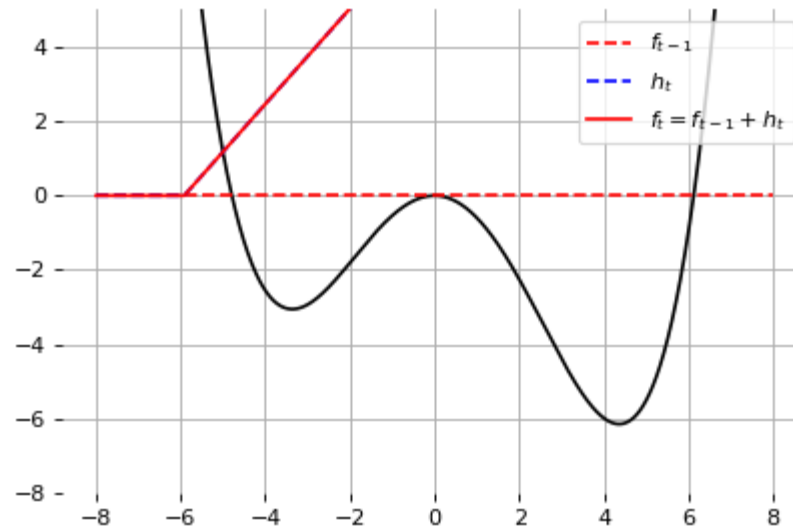
where  $N$  is the number of training points,  $q$  is the number of neurons,  $p$  is the input dimension, and  $C_f$  measures the global smoothness of  $f$ .

- Combines approximation and estimation errors.
- Provided enough data, it guarantees that adding more neurons will result in a better approximation.

Let us consider the 1-layer MLP

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

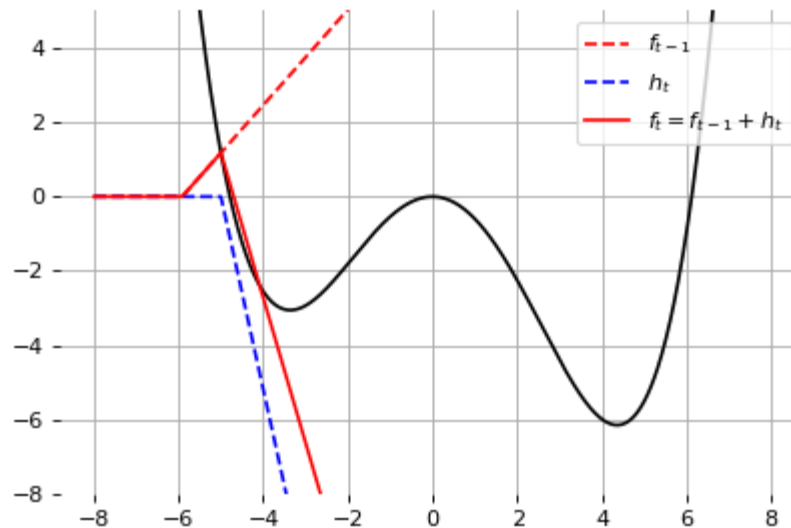
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

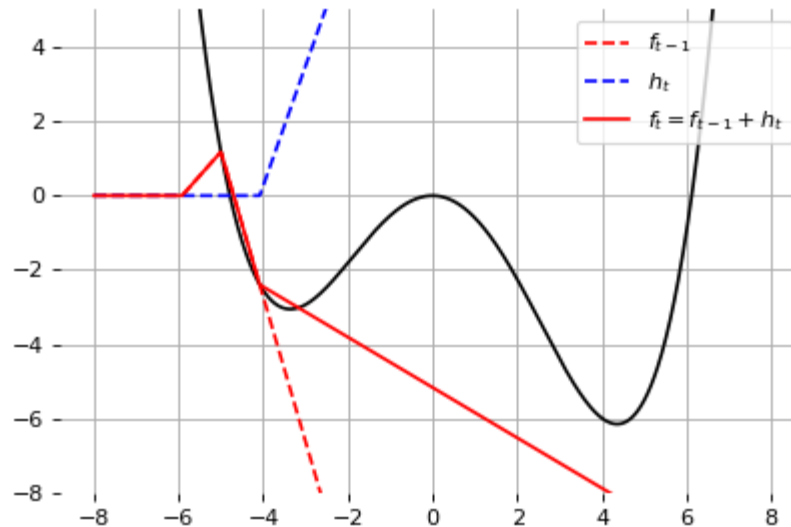
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

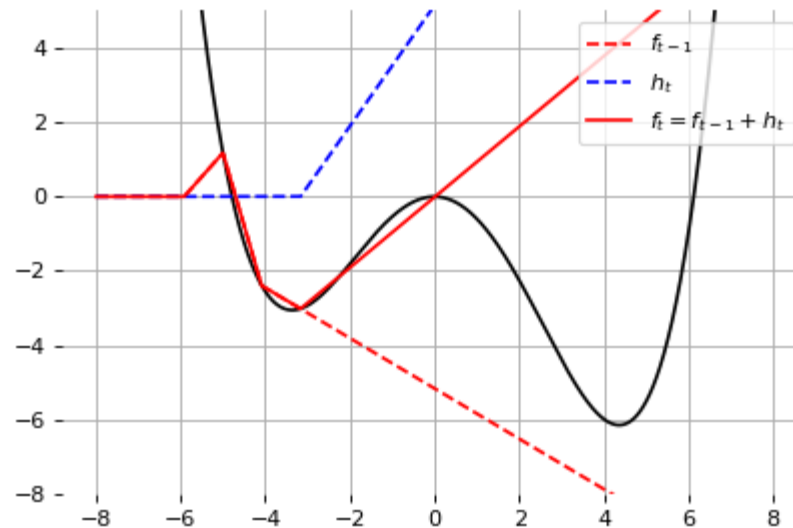
This model can approximate any smooth 1D function, provided enough hidden units.



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.

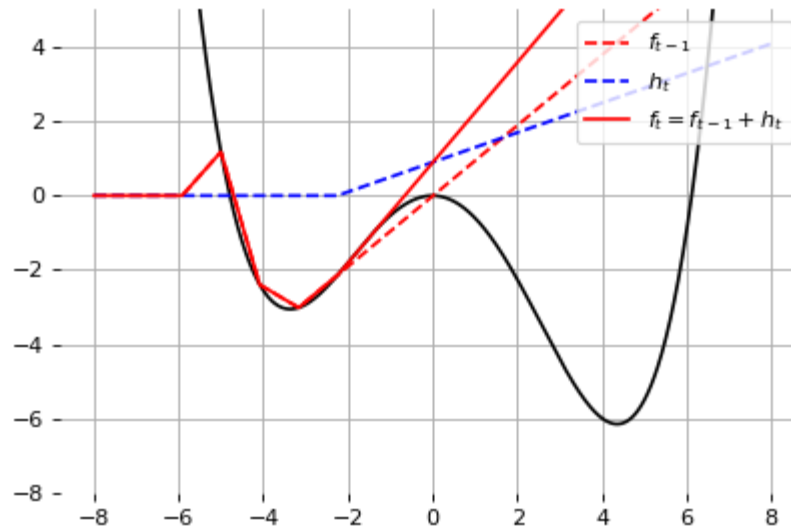




Let us consider the 1-layer MLP

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

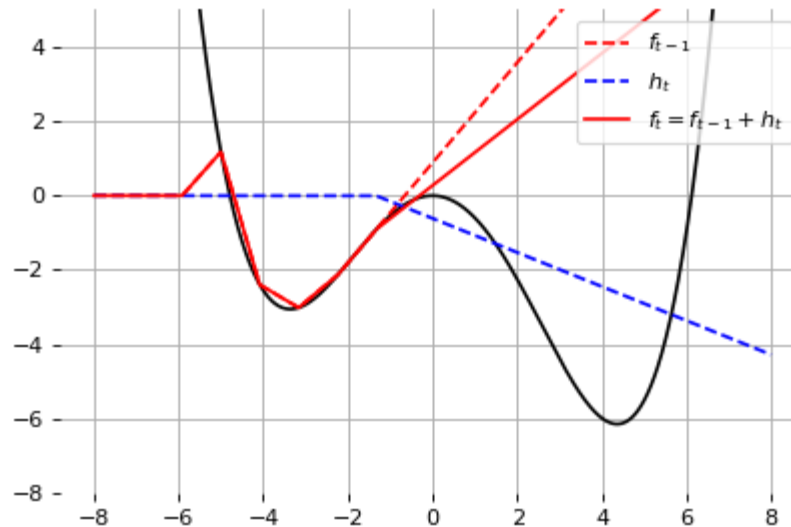
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

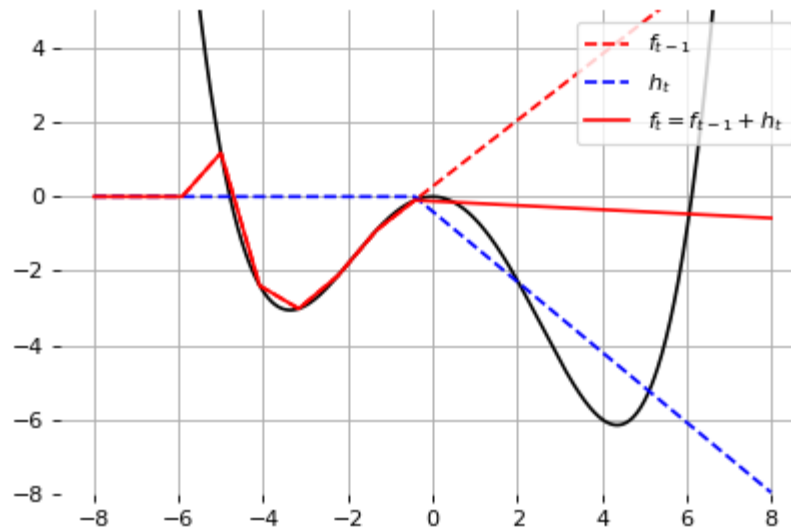
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

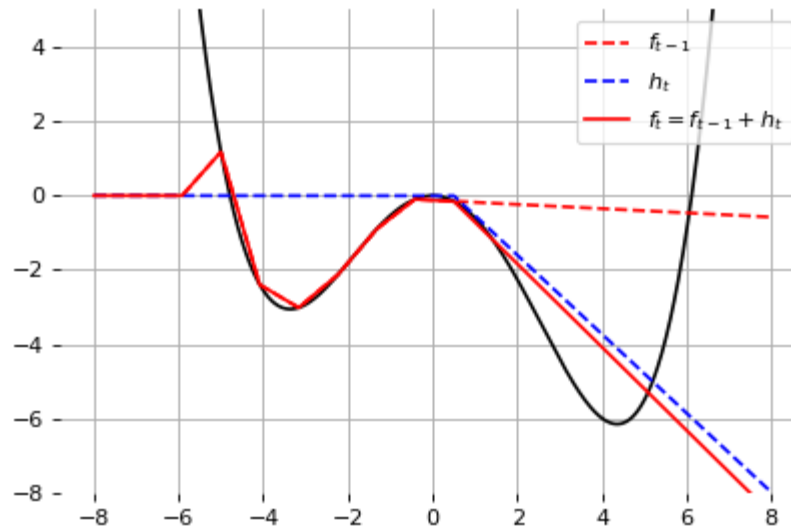
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

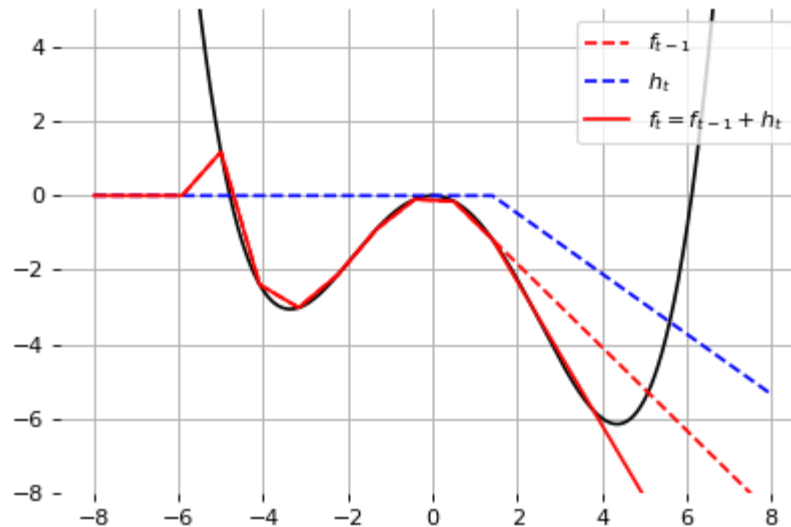
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

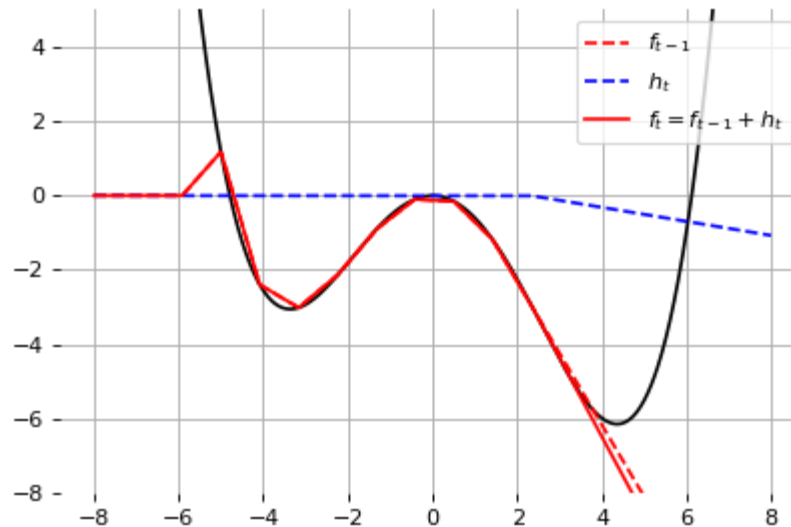
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

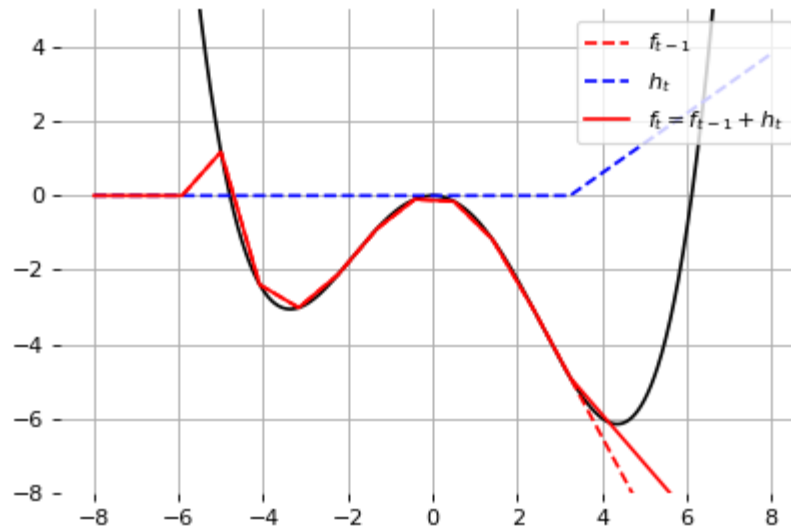
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

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

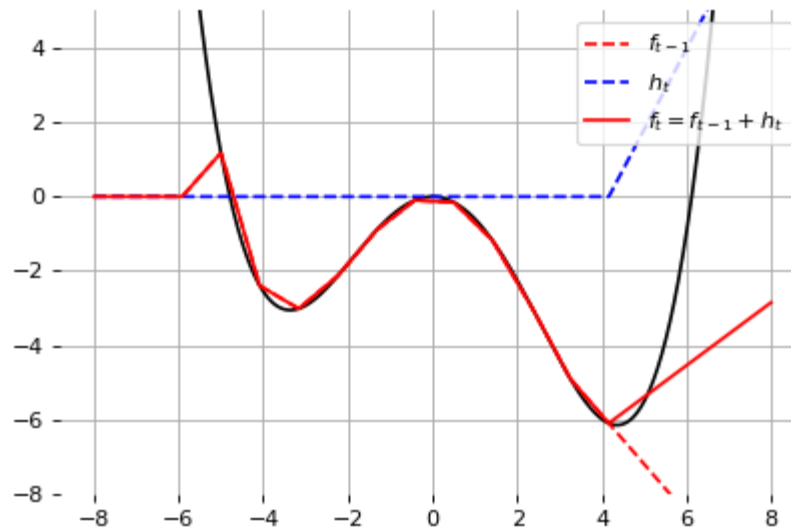
This model can approximate any smooth 1D function, provided enough hidden units.



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.

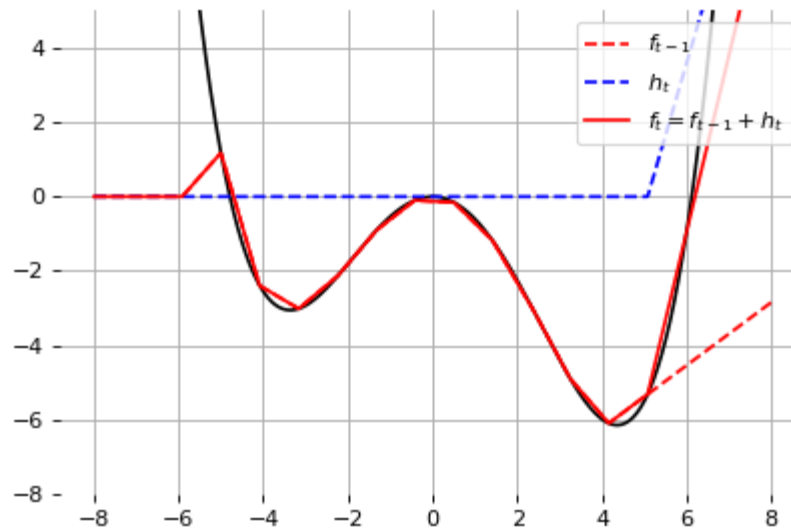




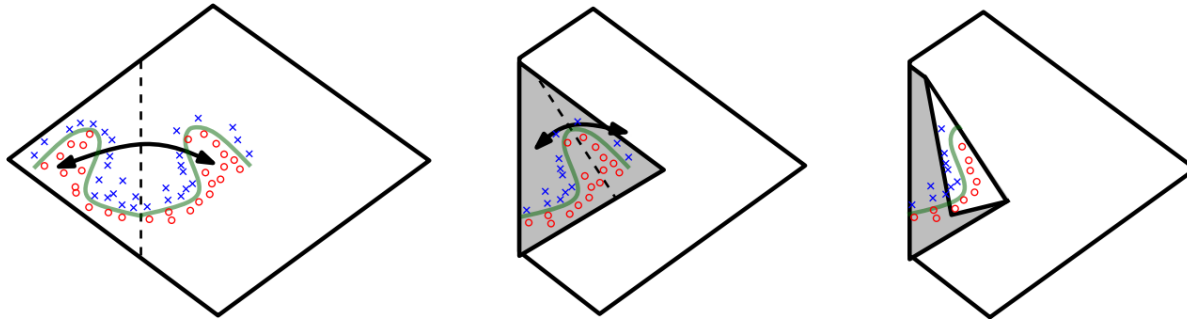
Let us consider the 1-layer MLP

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

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



# Effect of depth



**Theorem** (Montúfar et al, 2014) A rectifier neural network with  $p$  input units and  $L$  hidden layers of width  $q \geq p$  can compute functions that have  $\Omega\left(\left(\frac{q}{p}\right)^{(L-1)p} q^p\right)$  linear regions.

- That is, the number of linear regions of deep models grows **exponentially** in  $L$  and polynomially in  $q$ .
- Even for small values of  $L$  and  $q$ , deep rectifier models are able to produce substantially more linear regions than shallow rectifier models.

# Deep learning

Recent advances and model architectures in deep learning are built on a natural generalization of a neural network: **a graph of tensor operators**, taking advantage of

- the chain rule
- stochastic gradient descent
- convolutions
- parallel operations on GPUs.

This does not differ much from networks from the 90s, as covered in Today's lecture.

This generalization allows to **compose** and design complex networks of operators, possibly dynamically, dealing with images, sound, text, sequences, etc. and to train them **end-to-end**.

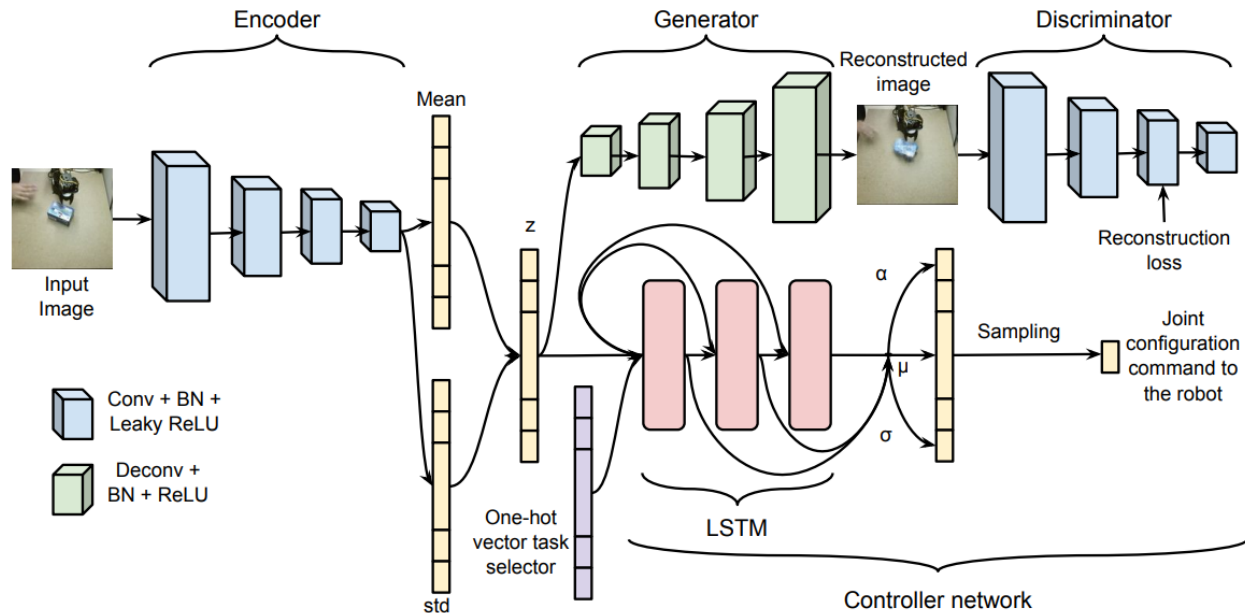


Fig. 2: Our proposed architecture for multi-task robot manipulation learning. The neural network consists of a controller network that outputs joint commands based on a multi-modal autoregressive estimator and a VAE-GAN autoencoder that reconstructs the input image. The encoder is shared between the VAE-GAN autoencoder and the controller network and extracts some shared features that will be used for two tasks (reconstruction and controlling the robot).

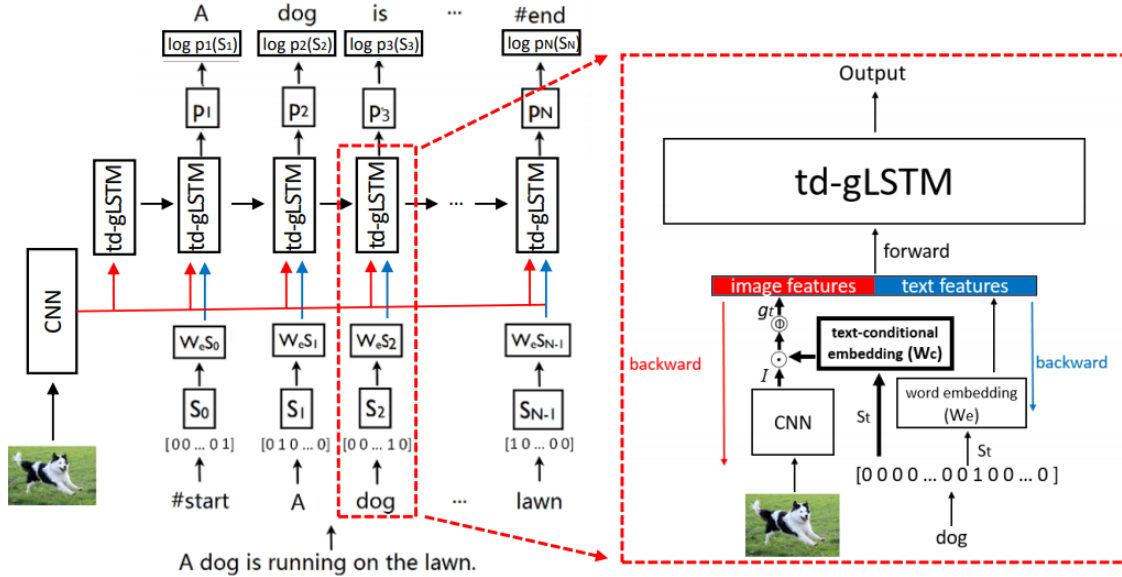


Figure 3: Text-conditional semantic attention. The text-conditional attention part are highlighted in **bold**.  $S_t$  indicates the one-hot vector representation of the  $t^{th}$  word in the sentence.  $W_e$  is word embedding matrix,  $W_c$  is text-conditional embedding matrix,  $I$  is image feature and  $g_t$  is time-dependent guidance. See text for more details.



# 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.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4), 303-314.
- Montufar, G. F., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *Advances in neural information processing systems* (pp. 2924-2932).