Deep Learning

Lecture 2: Neural networks

Gilles Louppe
g.louppe@uliege.be

LIÈGE université

Outline

Goal: explain and motivate the basic constructs of neural networks.

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

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 which such units.

Perceptron

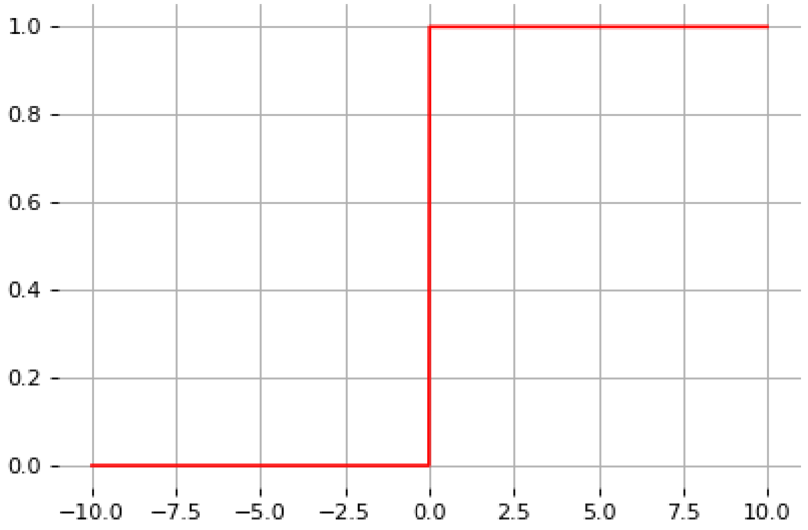
The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

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

- This model was originally motivated by biology, with w_i being synaptic weights and x_i and f firing rates.
- This is a cartoonesque biological model.

Let us define the activation function:

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



Therefore, the perceptron classification rule can be rewritten as

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b).$$

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 Σ (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

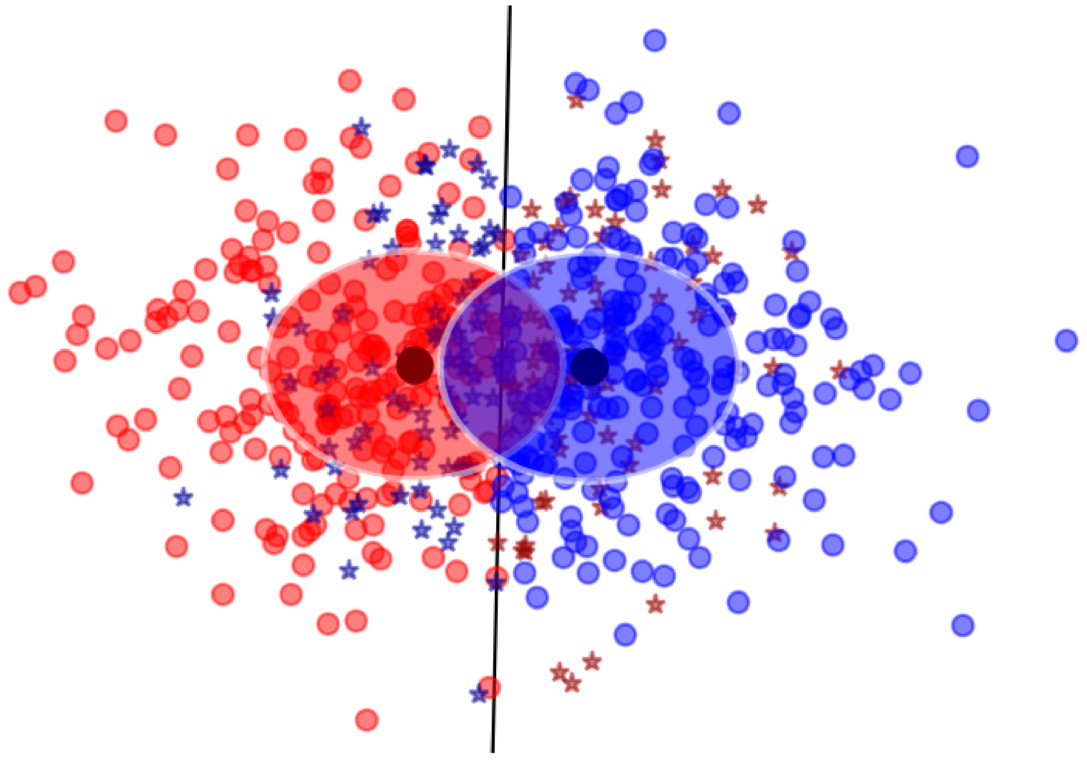
$$\sigma(x) = \frac{1}{1+\exp(-x)},$$

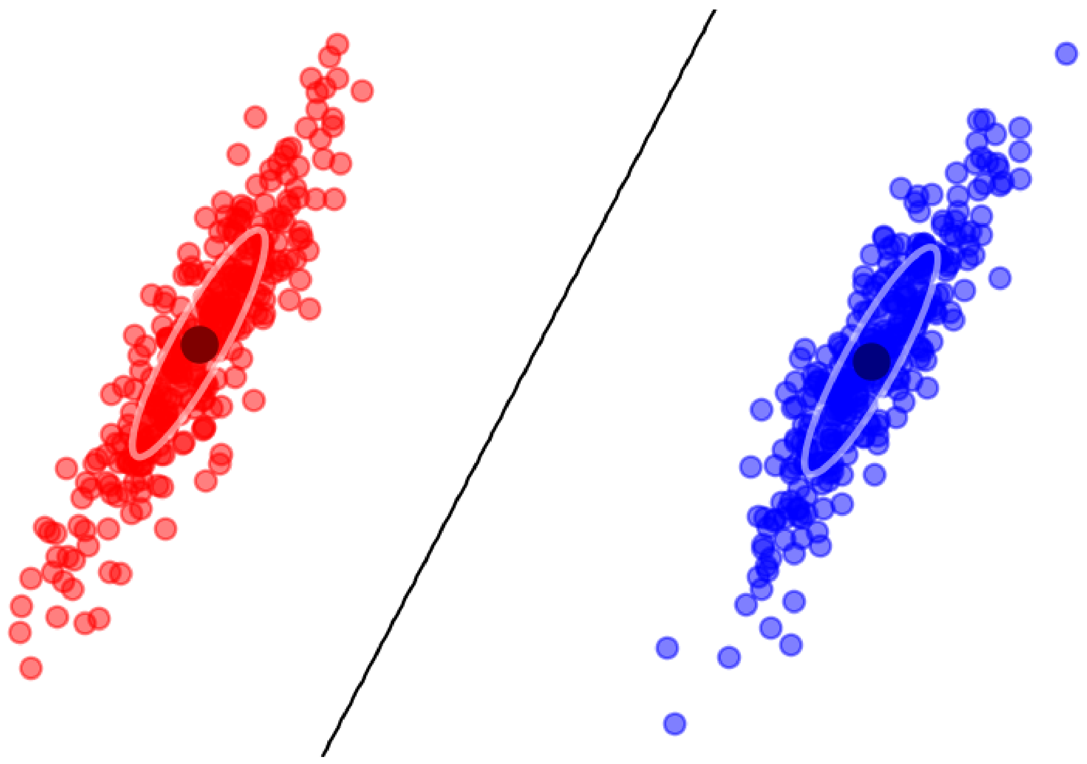
we get

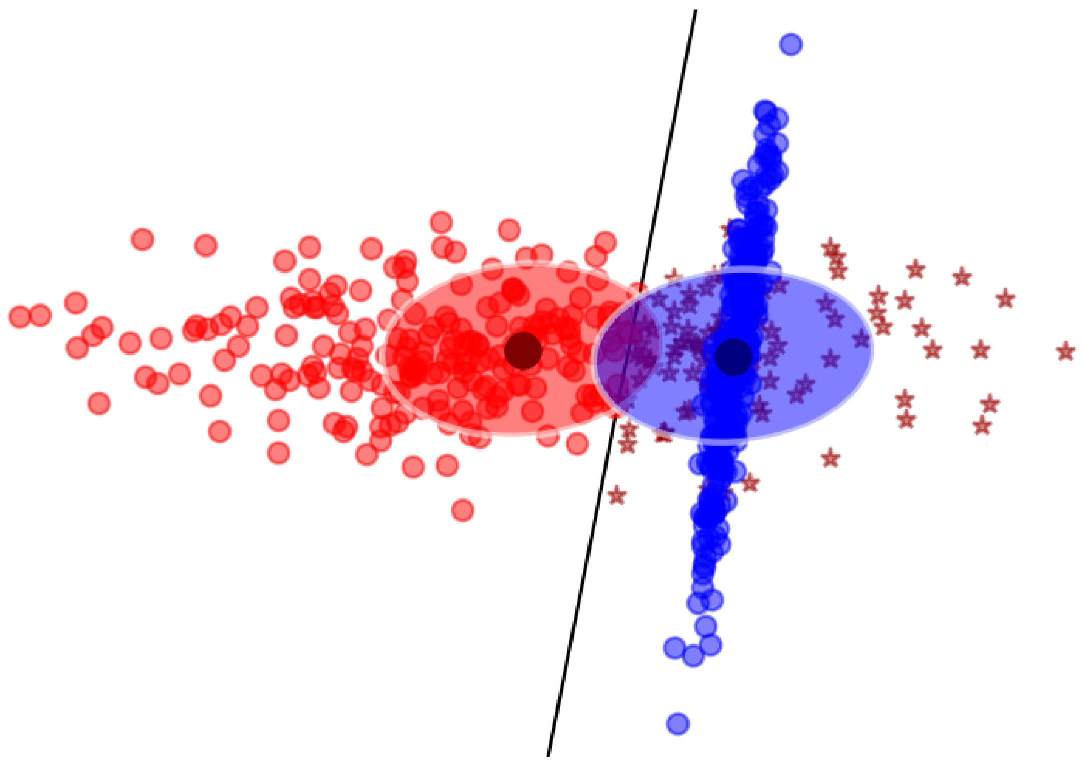
$$P(Y=1|\mathbf{x}) = \sigma\left(\log\frac{P(\mathbf{x}|Y=1)}{P(\mathbf{x}|Y=0)} + \log\frac{P(Y=1)}{P(Y=0)}\right).$$

Therefore,

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



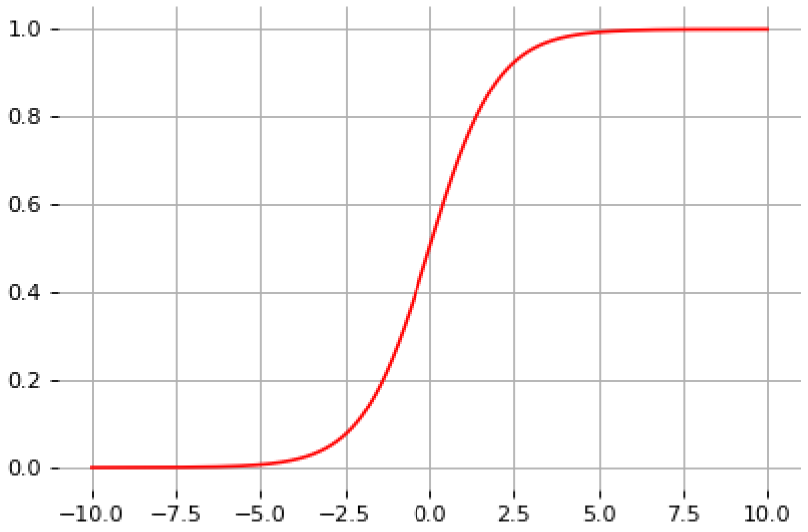




Note that the sigmoid function

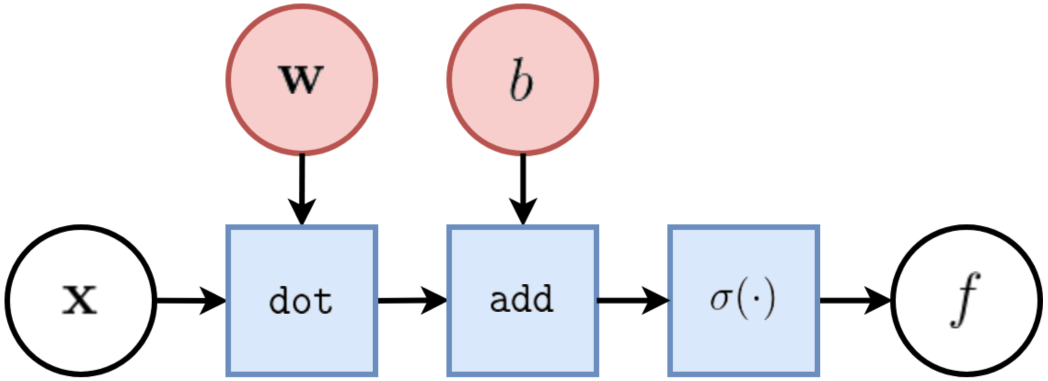
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

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.

In terms of tensor operations, the computational graph of f can be represented as:



where

- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations, which themselves produce intermediate output values (not represented).

This unit is the core component all neural networks!

Logistic regression

Same model

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

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,

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

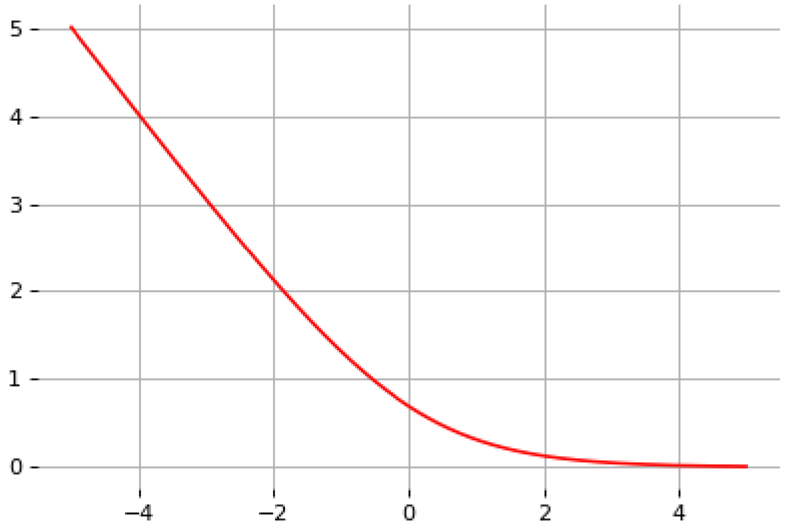
This loss is an instance of the cross-entropy

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

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

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



- 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 θ (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 θ_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

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

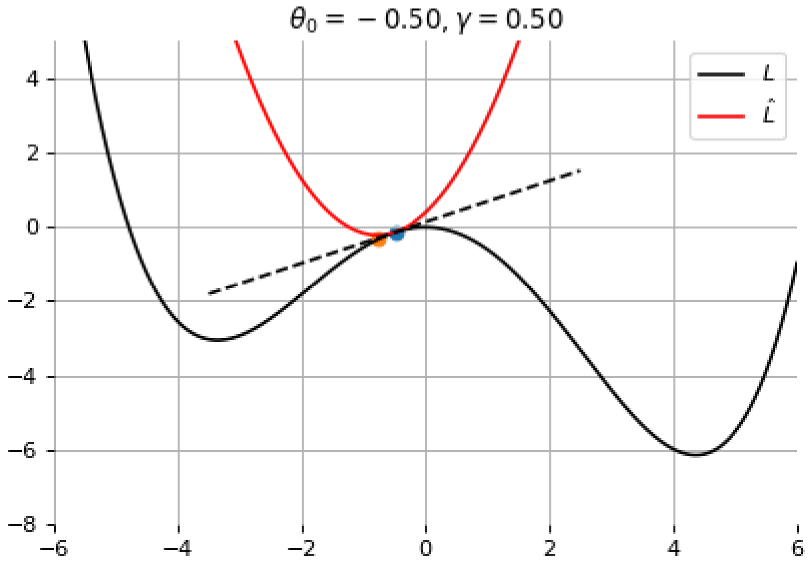
which results in the best improvement for the step $\epsilon = -\gamma \nabla_\theta \mathcal{L}(\theta_0)$.

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

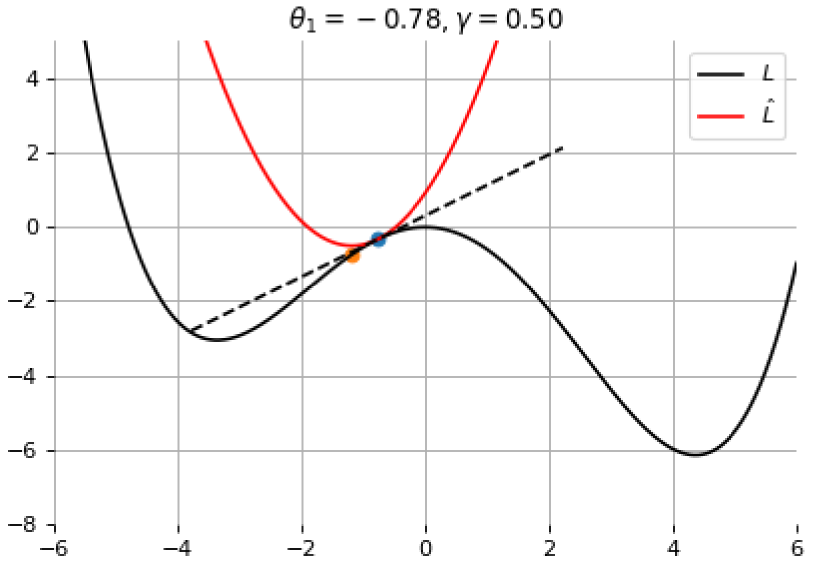
$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(\theta_t)$$

Notes:

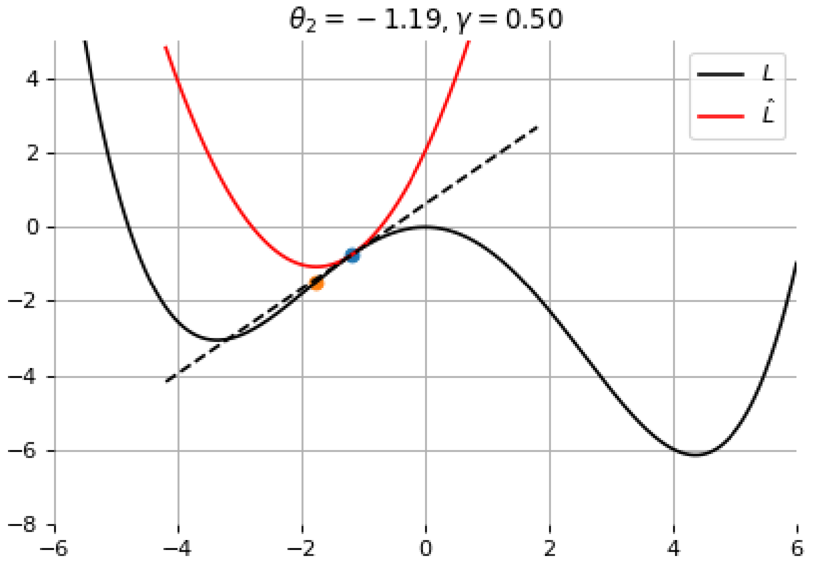
- θ_0 are the initial parameters of the model;
- γ is the learning rate;
- both are critical for the convergence of the update rule.



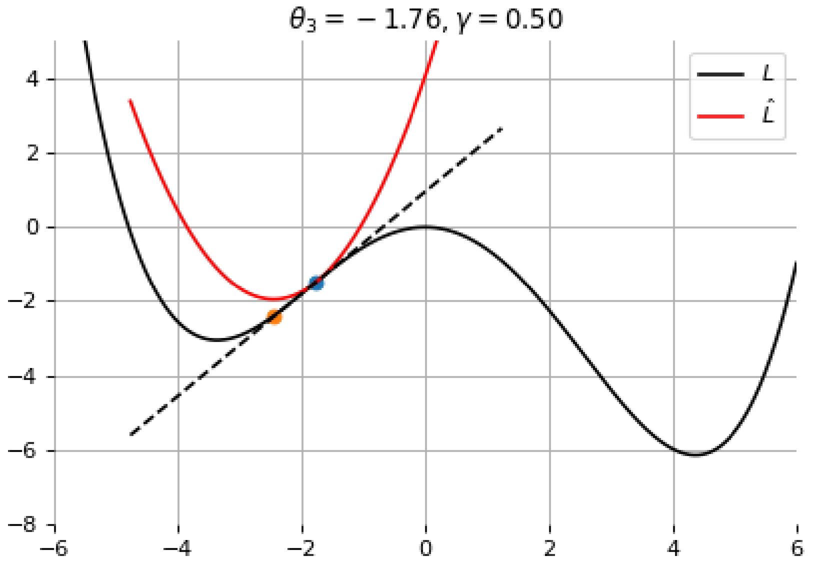
Example 1: Convergence to a local minima



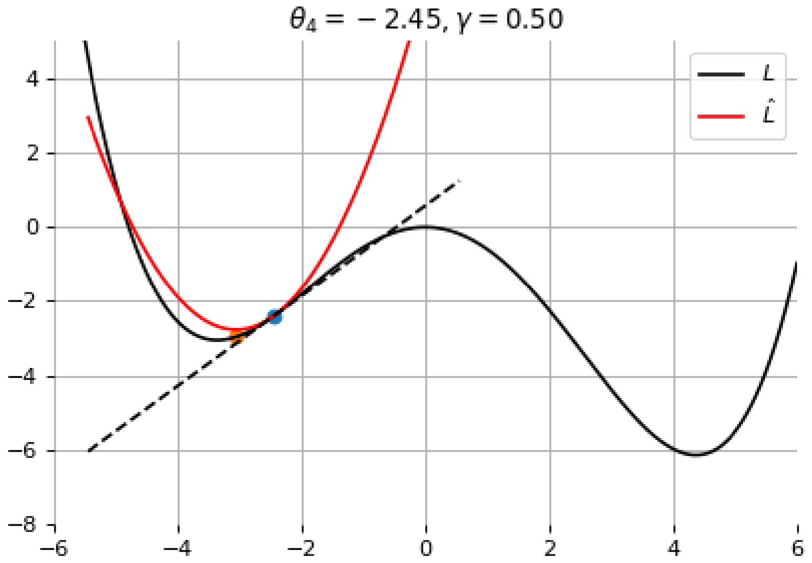
Example 1: Convergence to a local minima



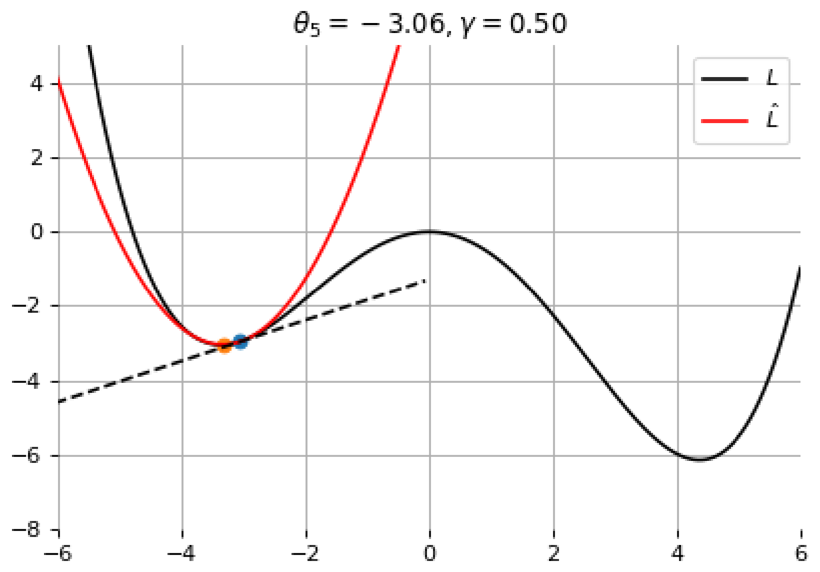
Example 1: Convergence to a local minima



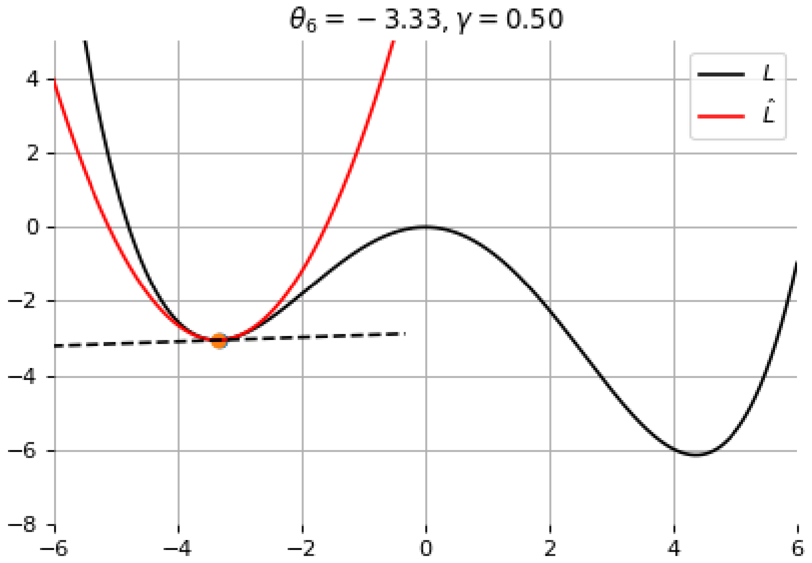
Example 1: Convergence to a local minima



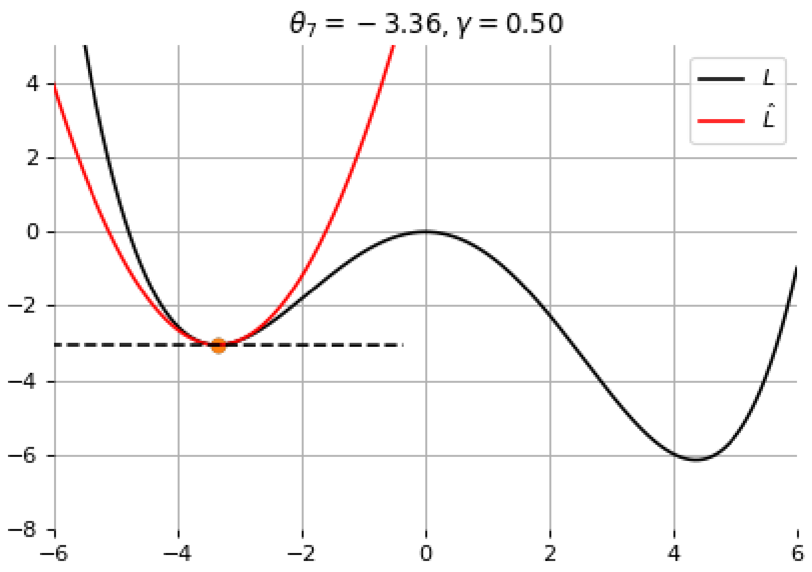
Example 1: Convergence to a local minima



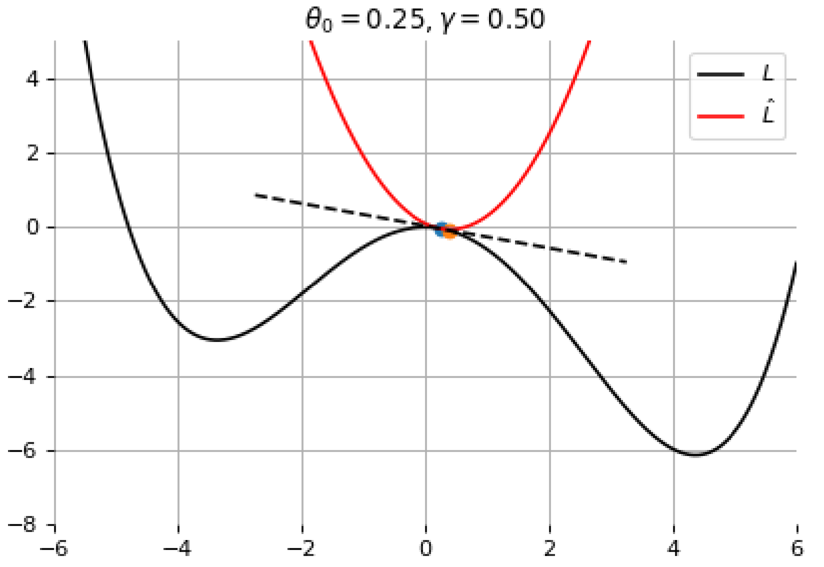
Example 1: Convergence to a local minima



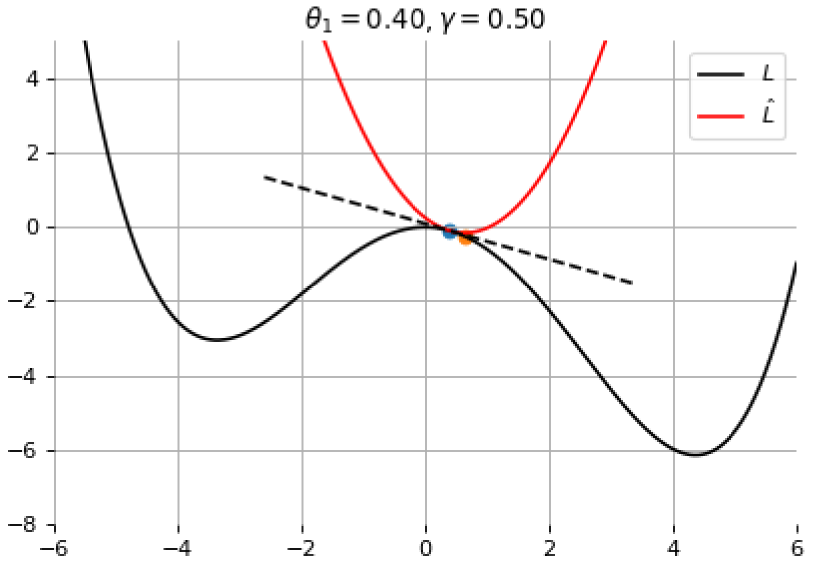
Example 1: Convergence to a local minima



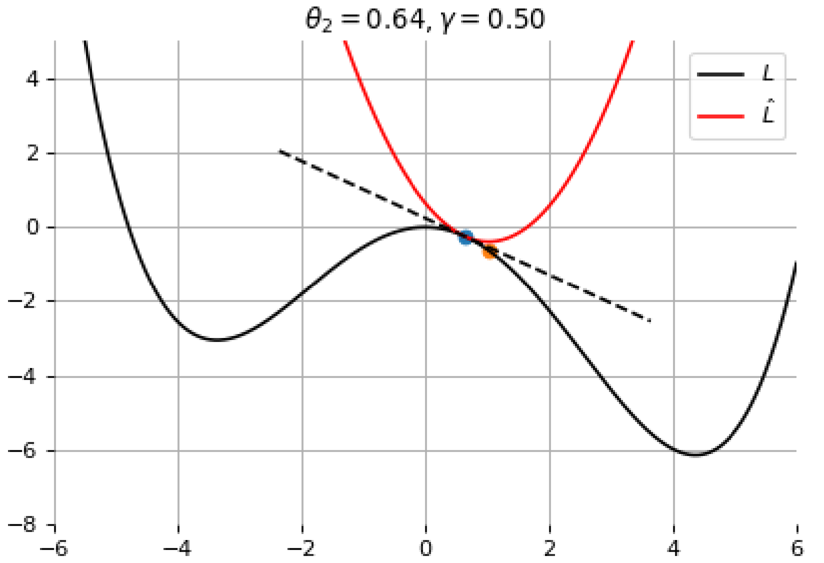
Example 1: Convergence to a local minima



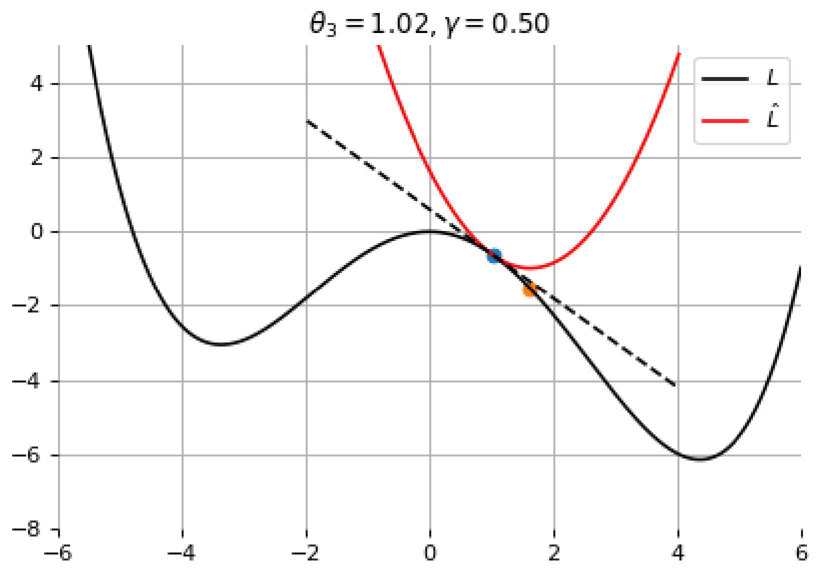
Example 2: Convergence to the global minima



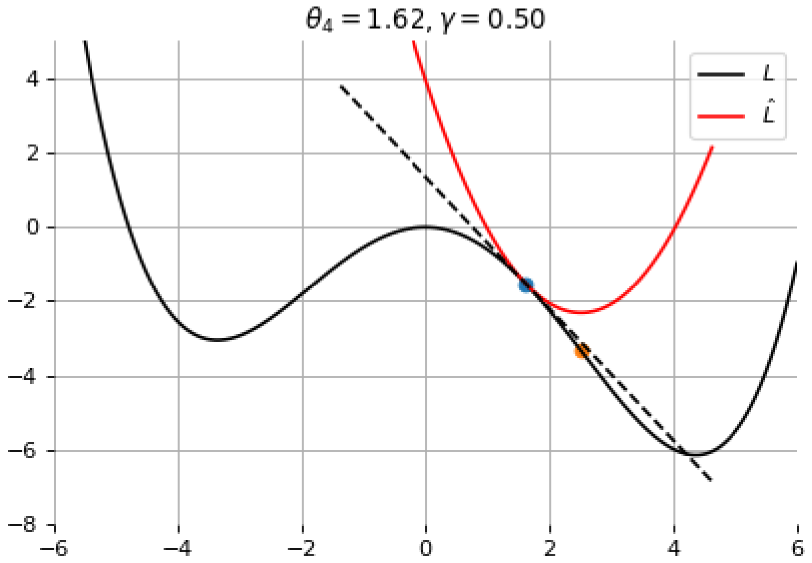
Example 2: Convergence to the global minima



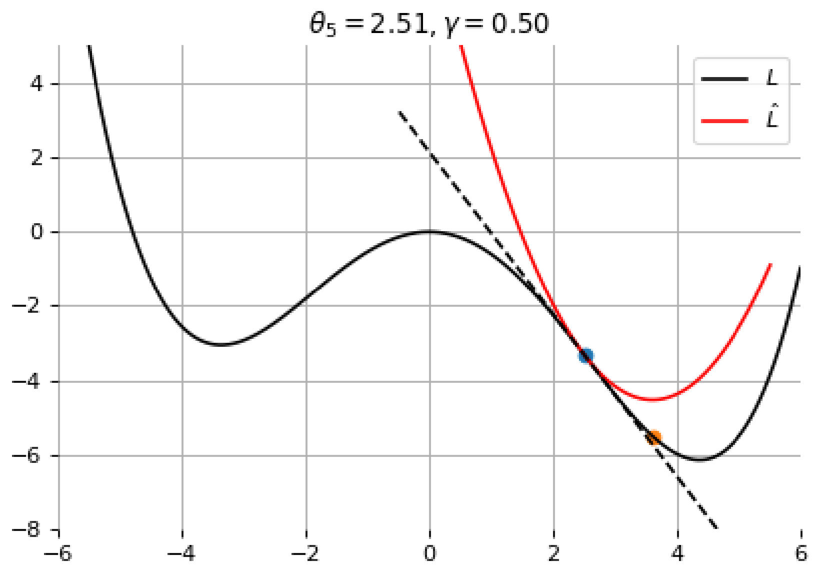
Example 2: Convergence to the global minima



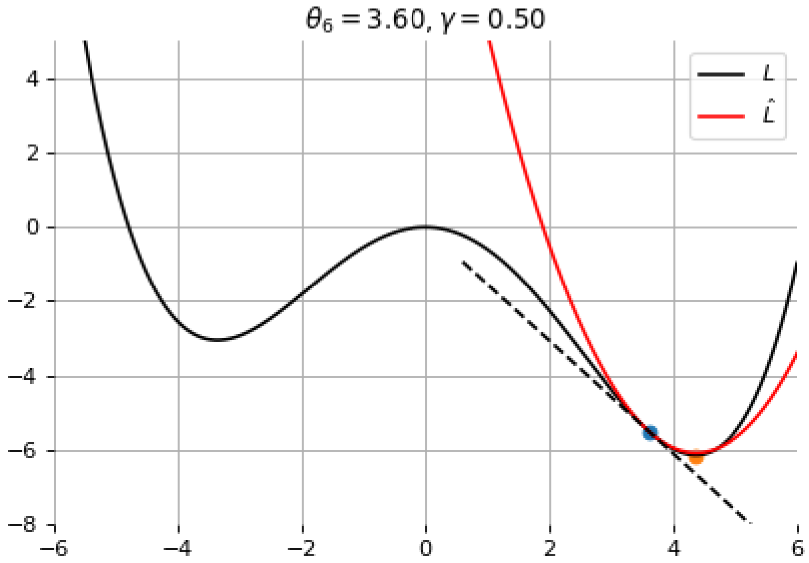
Example 2: Convergence to the global minima



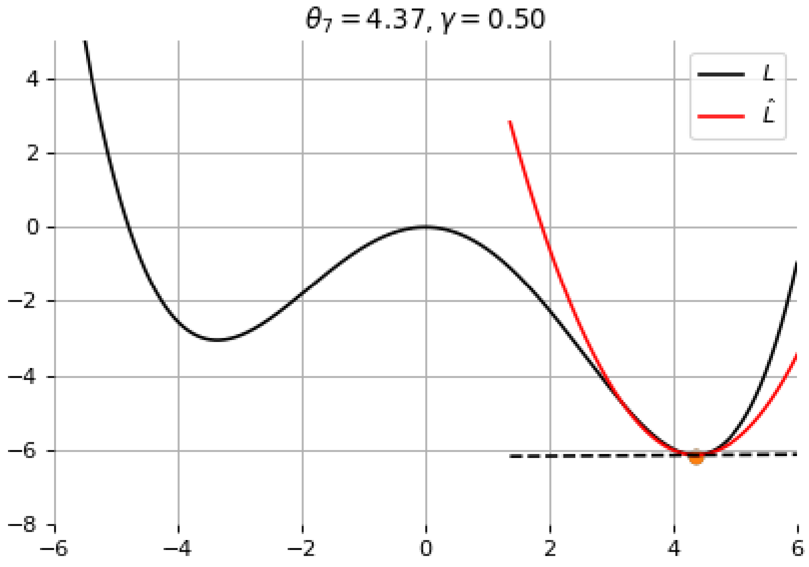
Example 2: Convergence to the global minima



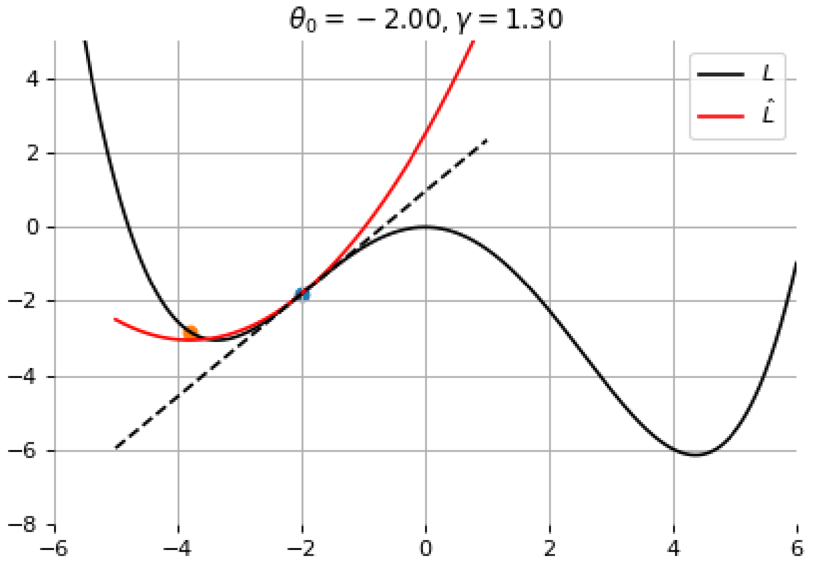
Example 2: Convergence to the global minima



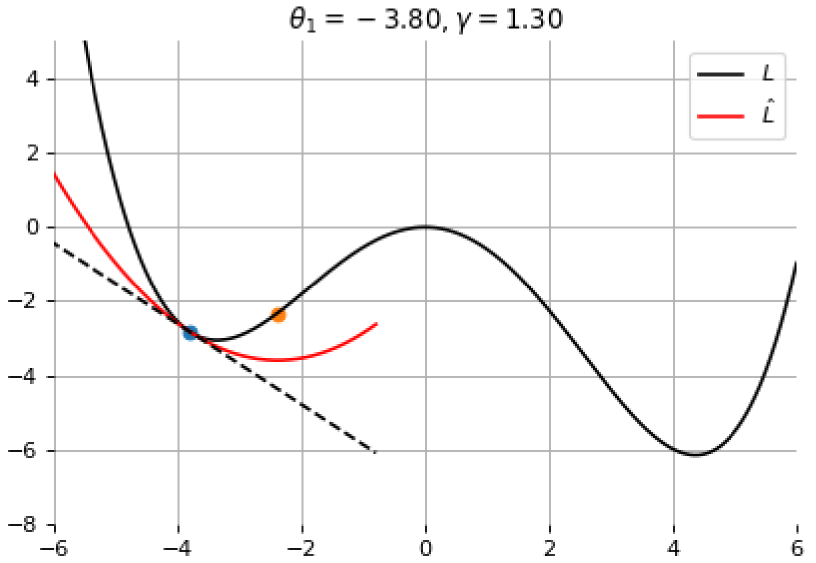
Example 2: Convergence to the global minima



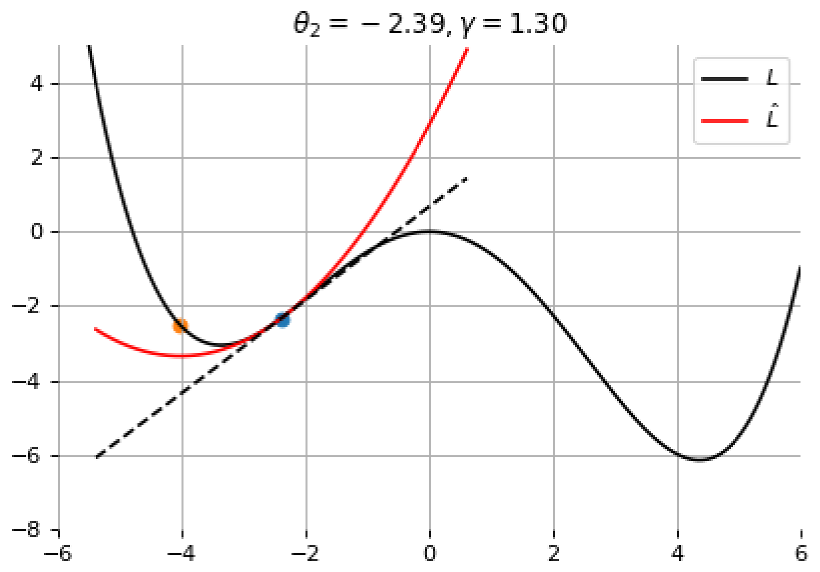
Example 2: Convergence to the global minima



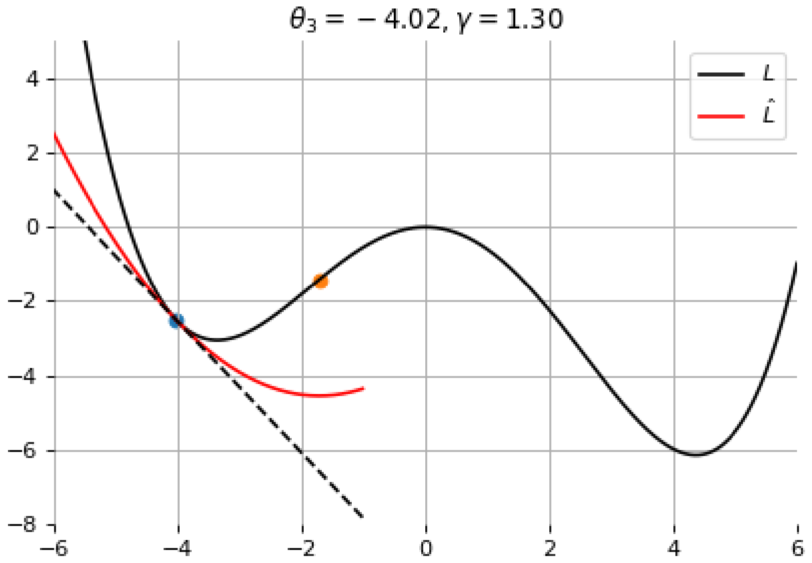
Example 3: Divergence due to a too large learning rate



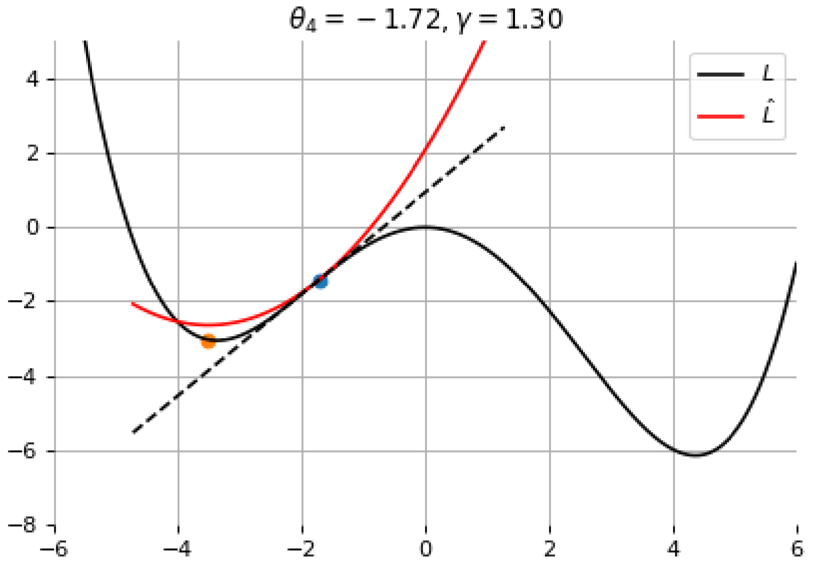
Example 3: Divergence due to a too large learning rate



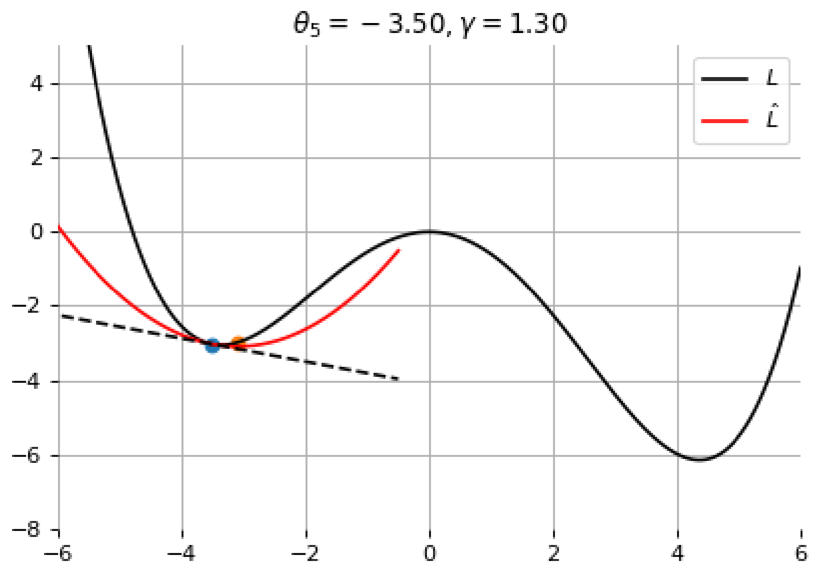
Example 3: Divergence due to a too large learning rate



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:

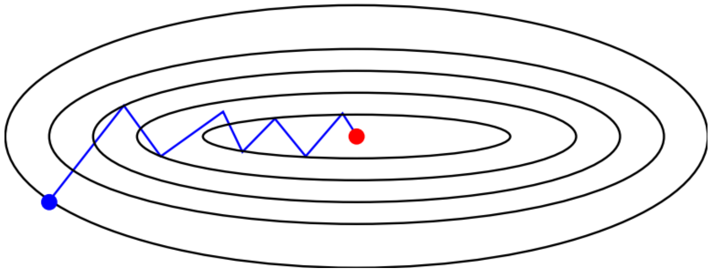
$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

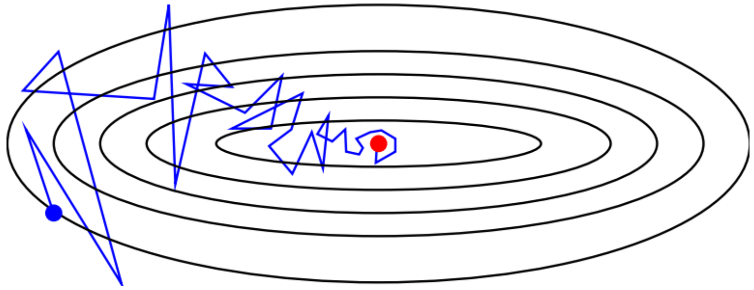
- Iteration complexity is independent of N.
- The stochastic process $\{\theta_t | t = 1, ...\}$ depends on the examples $i(t)$ picked randomly at each iteration.

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 $\{\theta_t | t = 1, ...\}$ 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).

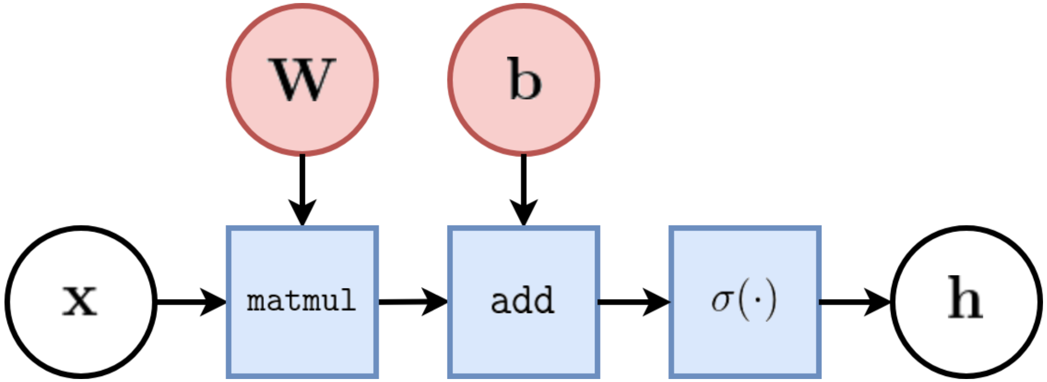
Layers

So far we considered the logistic unit $h = \sigma\left(\mathbf{w}^T \mathbf{x} + b\right)$, 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 $\mathbf{h} \in \mathbb{R}^q$, $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{W} \in \mathbb{R}^{p \times q}$, $b \in \mathbb{R}^q$ and where $\sigma(\cdot)$ is upgraded to the element-wise sigmoid function.



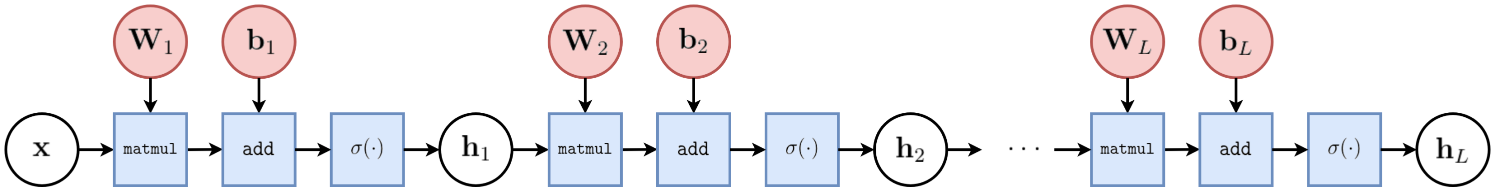
Multi-layer perceptron

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

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

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

- This model is the multi-layer perceptron, also known as the fully connected feedforward network.
- Optionally, the last activation σ can be skipped to produce unbounded output values $\hat{y} \in \mathbb{R}$.



$\mathbf{x}$
$\mathbf{W}_1$
$\mathbf{b}_1$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_1$
$\mathbf{W}_2$
$\mathbf{b}_2$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_2$
$\cdots$
$\mathbf{W}_L$
$\mathbf{b}_L$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_L$

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

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

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

These derivatives can be evaluated automatically from the computational graph of ℓ using automatic differentiation.

Automatic differentiation

Consider a 1-dimensional output composition $f \circ g$, such that

$$
\begin{aligned}
y &= f(\mathbf{u}) \\
\mathbf{u} &= g(x) = (g_1(x), ..., g_m(x)).
\end{aligned}
$$

The chain rule of total derivatives states that

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \underbrace{\frac{\mathrm{d}u_k}{\mathrm{d}x}}_{\text{recursive case}}$$

- Since a neural network is a composition of differentiable functions, the total derivatives of the loss can be evaluated by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called (reverse) automatic differentiation (AD).
- AD is not numerical differentiation, nor symbolic differentiation.

As a guiding example, let us consider a simplified 2-layer MLP and the following loss function:

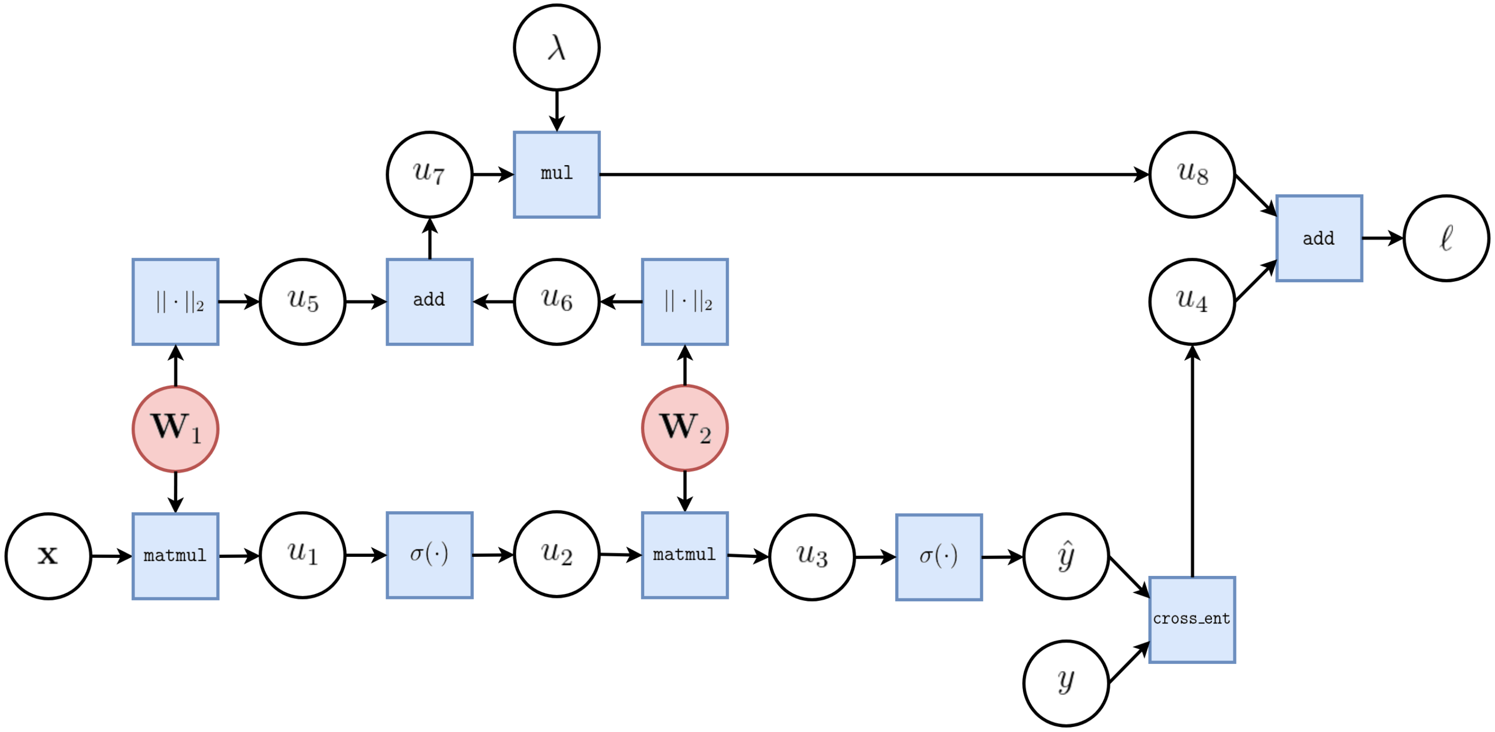
$$\begin{aligned} f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) &= \sigma\left(\mathbf{W}_2^T \sigma\left(\mathbf{W}_1^T \mathbf{x}\right)\right) \\ \ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) &= \text{cross\_ent}(y, \hat{y}) + \lambda\left(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2\right) \end{aligned}$$

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

As a guiding example, let us consider a simplified 2-layer MLP and the following loss function:

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

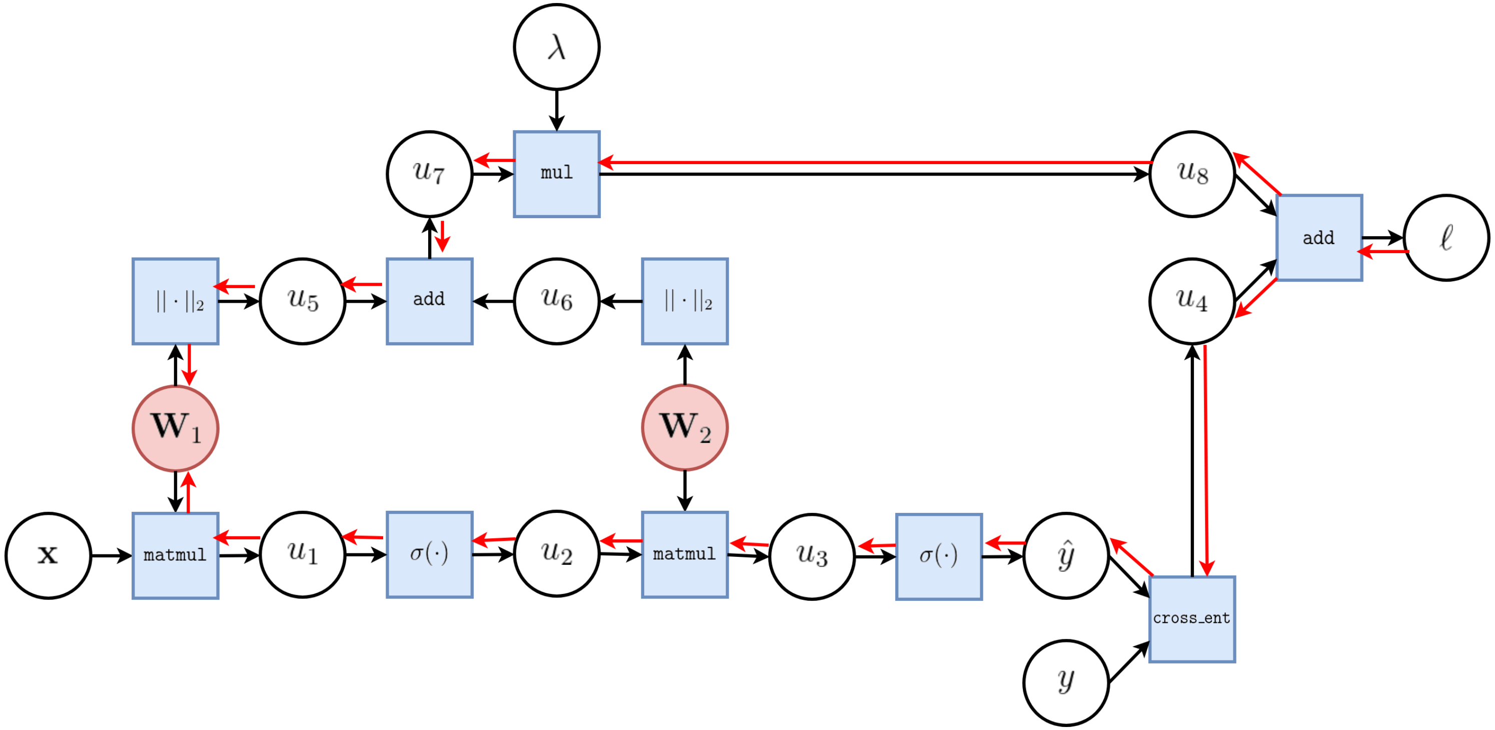
for $\mathbf{x} \in \mathbb{R}^p$, $y \in \mathbb{R}$, $\mathbf{W}_1 \in \mathbb{R}^{p\times q}$ and $\mathbf{W}_2 \in \mathbb{R}^q$.



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

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

$$\frac{\mathrm{d}u_8}{\mathrm{d}\mathbf{W}_1} = \ldots$$

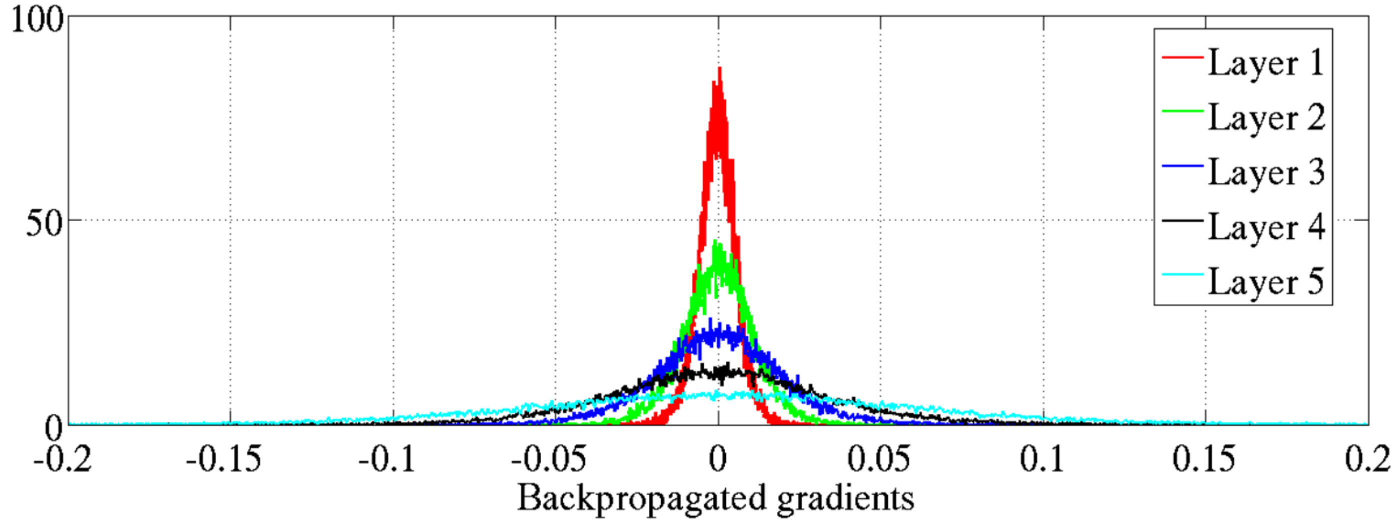


- This algorithm is known as reverse-mode automatic differentiation, also called backpropagation.
- An equivalent procedure can be defined to evaluate the derivatives in forward mode, from inputs to outputs.
- Automatic differentiation generalizes to N inputs and M outputs.
    - if $N \gg M$, reverse-mode automatic differentiation is computationally more efficient.
    - otherwise, if $M \gg N$, forward automatic differentiation is better.
- Since differentiation is a linear operator, AD can be implemented efficiently in terms of matrix 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.

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

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

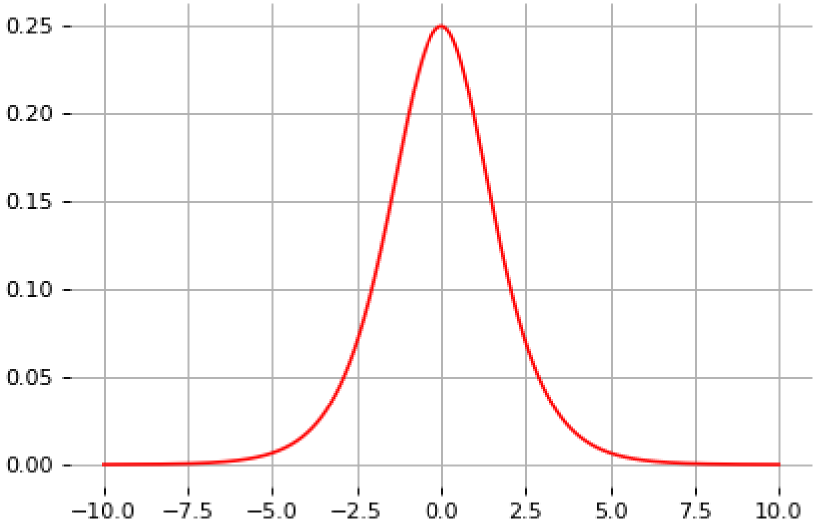
Under the hood, this would be evaluated as

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

and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

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

The derivative of the sigmoid activation function σ is:



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

Notice that $0 \leq \frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) \leq \frac{1}{4}$ 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,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ exponentially shrinks to zero as the number of layers in the network increases.

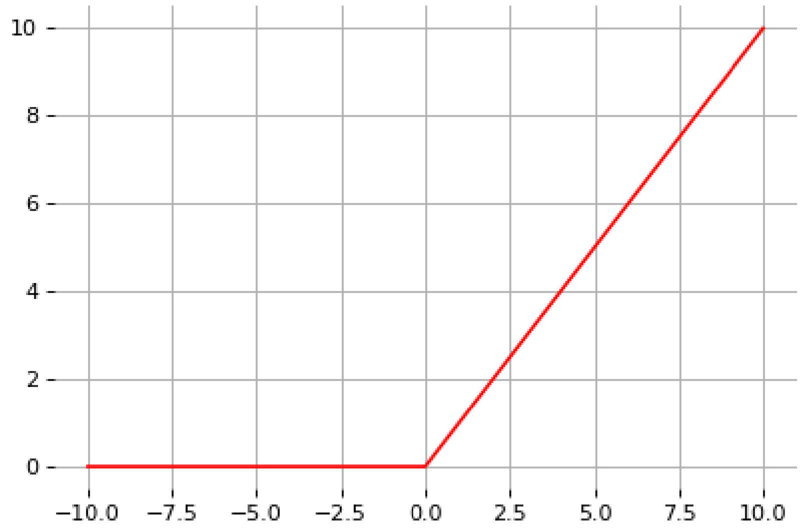
Hence the vanishing gradient problem.

- 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

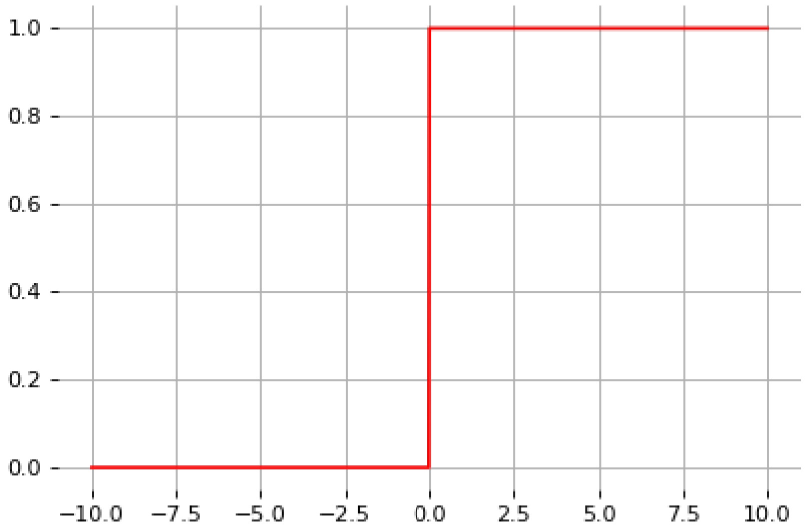
Instead of the sigmoid activation function, modern neural networks are for most based on rectified linear units (ReLU) (Glorot et al, 2011):

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



Note that the derivative of the ReLU function is

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



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

Therefore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{=1} x$$

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 $v_i, w_i, b_i, i = 1, ..., q$ 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

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

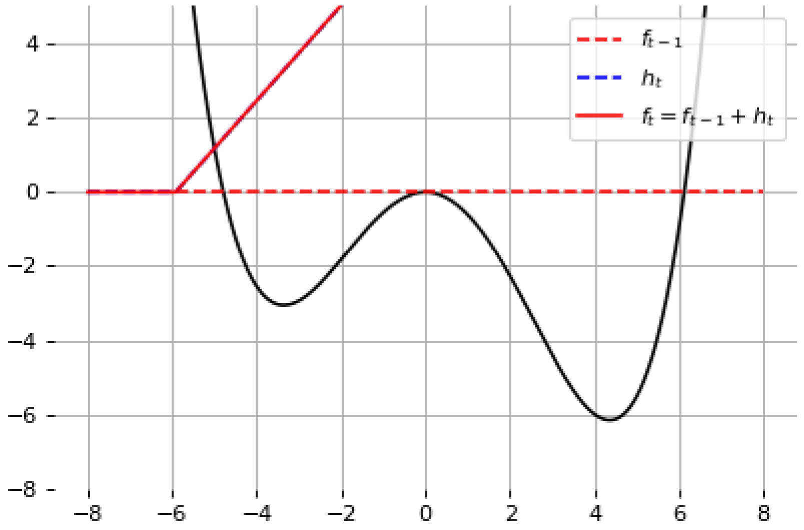
where N is the number of training points, q is the number of neurons, p is the input dimension, and C_f measures the global smoothness of f.

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

Consider the 1-layer MLP

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

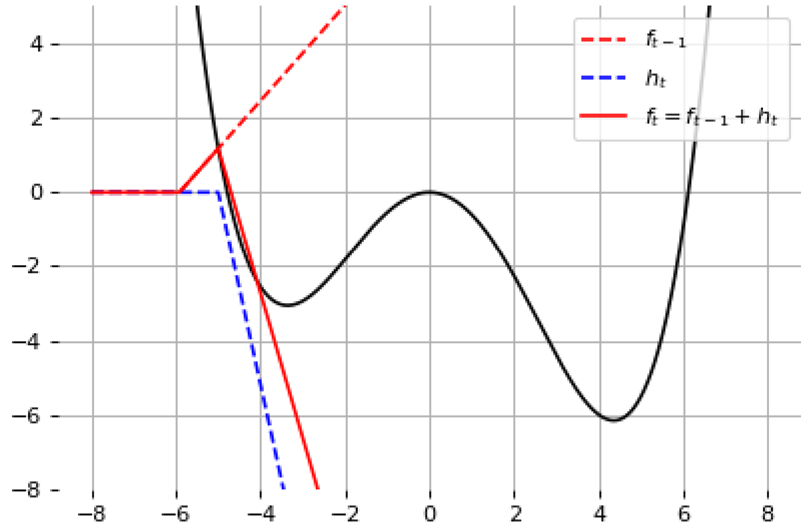
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

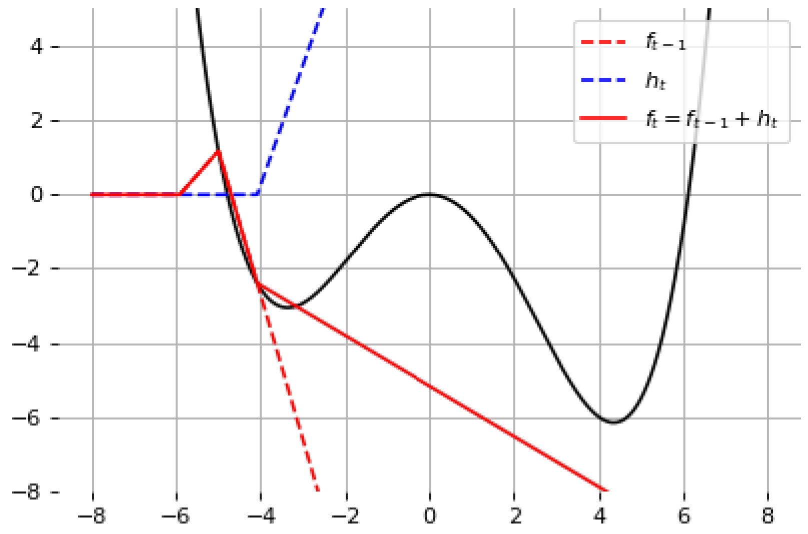
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

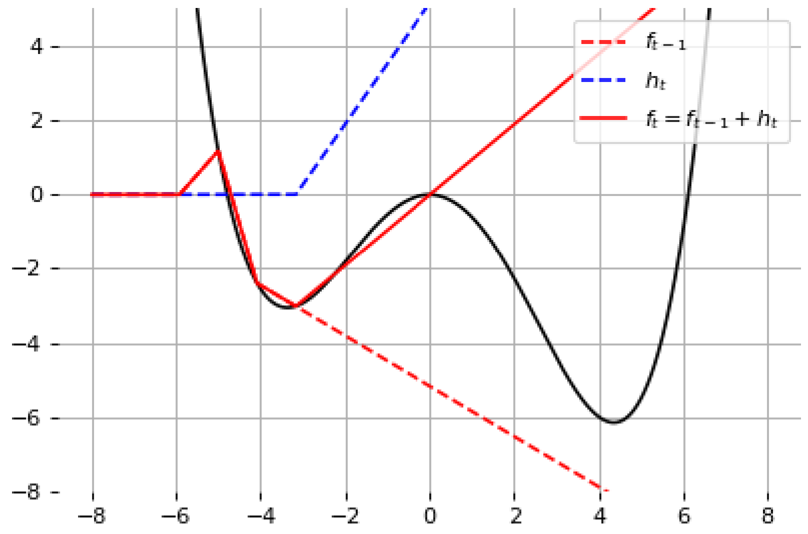
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

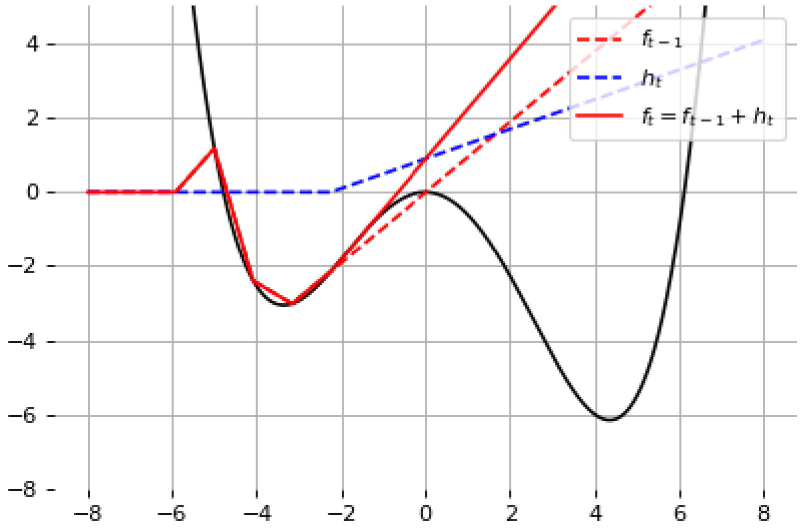
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

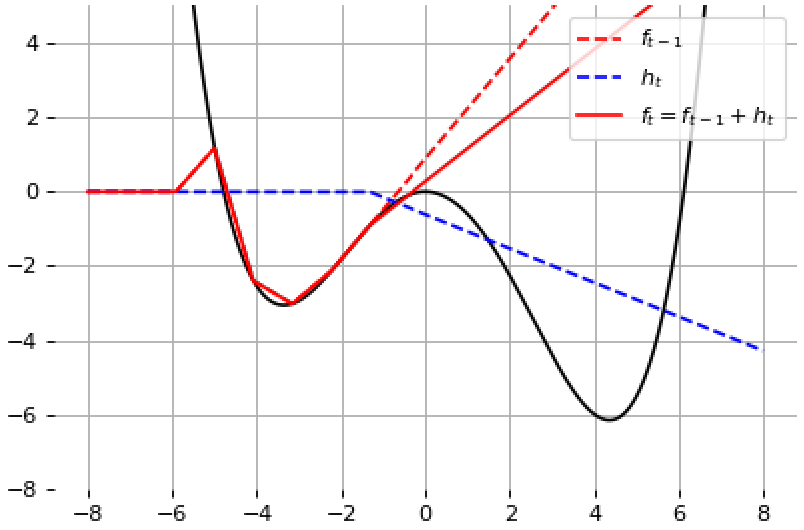
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

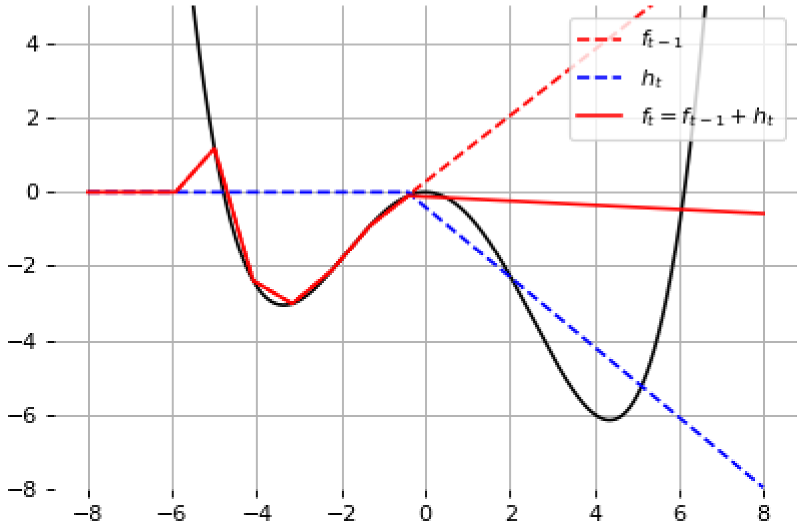
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

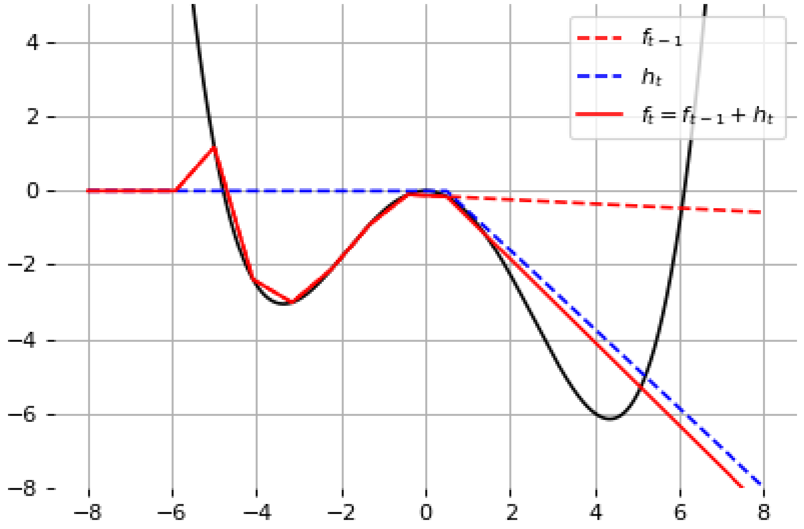
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

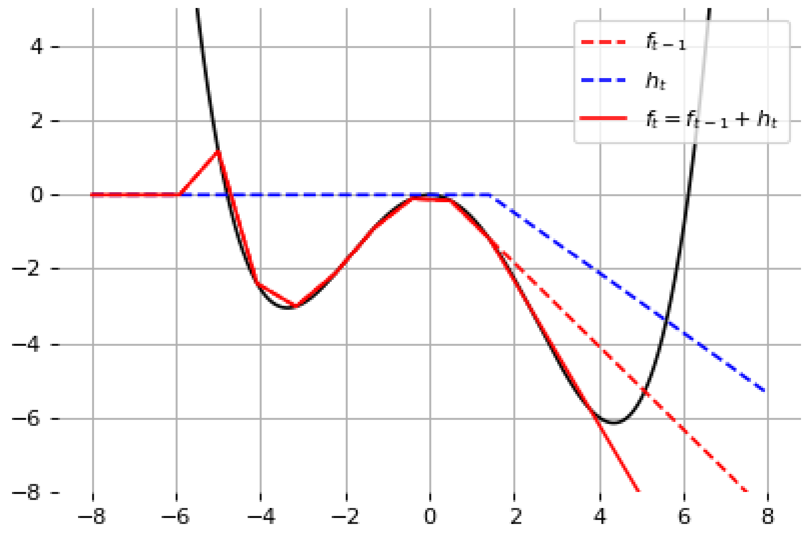
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

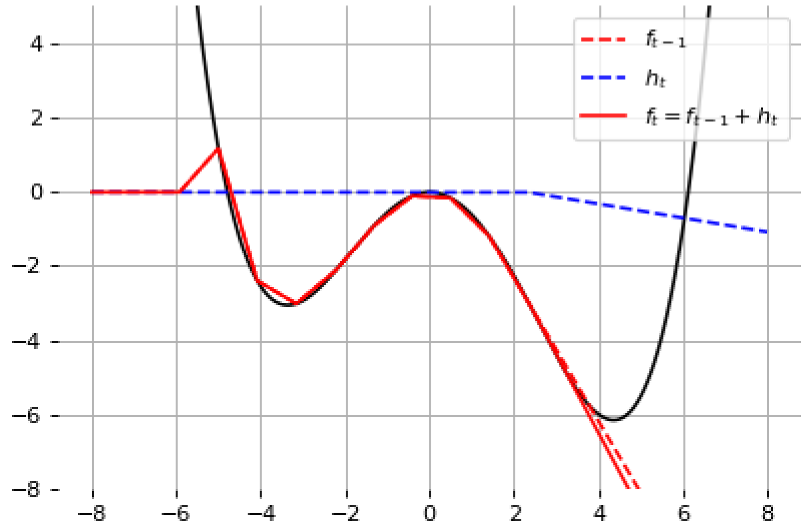
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

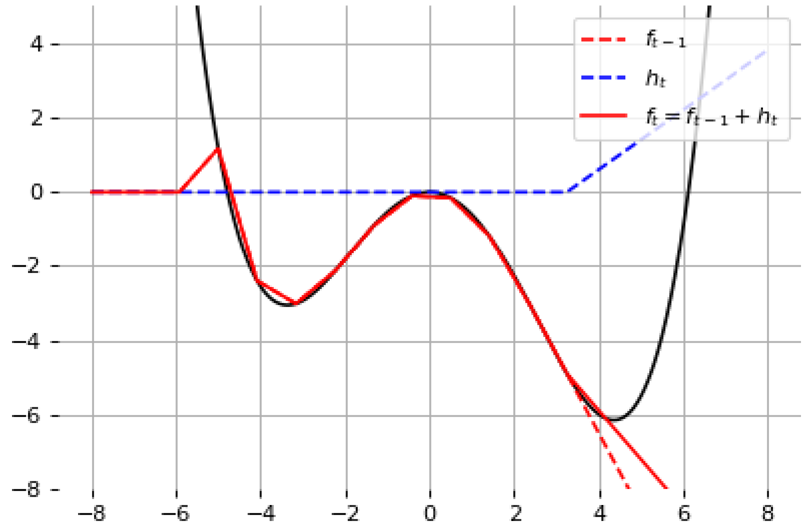
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

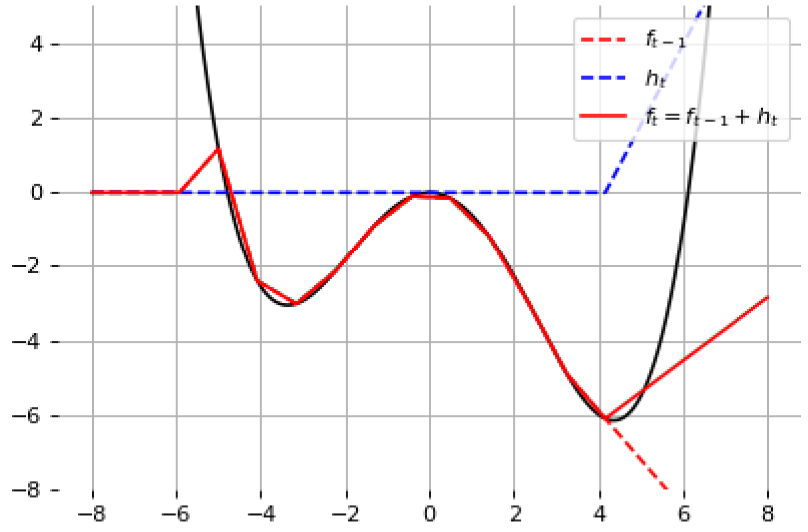
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

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

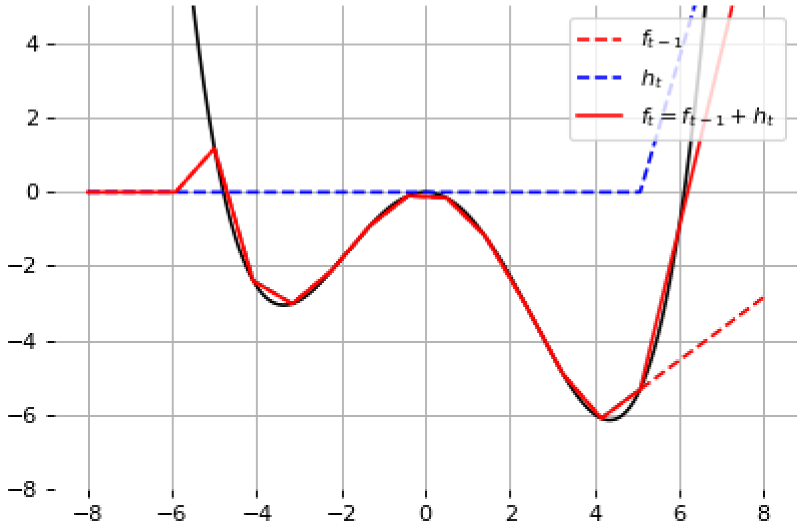
This model can approximate any smooth 1D function, provided enough hidden units.



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.



(Bayesian) Infinite networks

What if $q \to \infty$?

Consider the 1-layer MLP with a hidden layer of size q and a bounded activation function σ:

$$f(x) = b + \sum_{j=1}^{q} v_j h_j(x)$$

$$h_j(x) = \sigma\left(a_j + \sum_{i=1}^{p} u_{i,j} x_i\right)$$

Assume Gaussian priors $v_j \sim \mathcal{N}(0, \sigma_v^2), b \sim \mathcal{N}(0, \sigma_b^2), u_{i,j} \sim \mathcal{N}(0, \sigma_u^2)$ and $a_j \sim \mathcal{N}(0, \sigma_a^2)$.

For a fixed value $x^{(1)}$, let us consider the prior distribution of $f(x^{(1)})$ implied by the prior distributions for the weights and biases.

We have

$$\mathbb{E}[v_j h_j(x^{(1)})] = \mathbb{E}[v_j]\mathbb{E}[h_j(x^{(1)})] = 0,$$

since v_j and $h_j(x^{(1)})$ are statistically independent and v_j has zero mean by hypothesis.

The variance of the contribution of each hidden unit h_j is

$$\begin{aligned}\mathbb{V}[v_j h_j(x^{(1)})] &= \mathbb{E}[(v_j h_j(x^{(1)}))^2] - \mathbb{E}[v_j h_j(x^{(1)})]^2 \\ &= \mathbb{E}[v_j^2]\mathbb{E}[h_j(x^{(1)})^2] \\ &= \sigma_v^2 \mathbb{E}[h_j(x^{(1)})^2],\end{aligned}$$

which must be finite since h_j is bounded by its activation function.

We define $V(x^{(1)}) = \mathbb{E}[h_j(x^{(1)})^2]$, and is the same for all j.

By the Central Limit Theorem, as $q \to \infty$, the total contribution of the hidden units, $\sum_{j=1}^{q} v_j h_j(x)$, to the value of $f(x^{(1)})$ becomes a Gaussian with variance $q\sigma_v^2 V(x^{(1)})$.

The bias b is also Gaussian, of variance σ_b^2, so for large q, the prior distribution $f(x^{(1)})$ is a Gaussian of variance $\sigma_b^2 + q\sigma_v^2 V(x^{(1)})$.

Accordingly, for $\sigma_v = \omega_v q^{-\frac{1}{2}}$, for some fixed ω_v, the prior $f(x^{(1)})$ converges to a Gaussian of mean zero and variance $\sigma_b^2 + \omega_v^2 \sigma_v^2 V(x^{(1)})$ as $q \to \infty$.

For two or more fixed values $x^{(1)}, x^{(2)}, \ldots$, a similar argument shows that, as $q \to \infty$, the joint distribution of the outputs converges to a multivariate Gaussian with means of zero and covariances of

$$\begin{aligned} \mathbb{E}[f(x^{(1)})f(x^{(2)})] &= \sigma_b^2 + \sum_{j=1}^{q} \sigma_v^2 \mathbb{E}[h_j(x^{(1)})h_j(x^{(2)})] \\ &= \sigma_b^2 + \omega_v^2 C(x^{(1)}, x^{(2)}) \end{aligned}$$

where $C(x^{(1)}, x^{(2)}) = \mathbb{E}[h_j(x^{(1)})h_j(x^{(2)})]$ and is the same for all j.

This result states that for any set of fixed points $x^{(1)}, x^{(2)}, \ldots$, the joint distribution of $f(x^{(1)}), f(x^{(2)}), \ldots$ is a multivariate Gaussian.

In other words, the infinitely wide 1-layer MLP converges towards a Gaussian process.

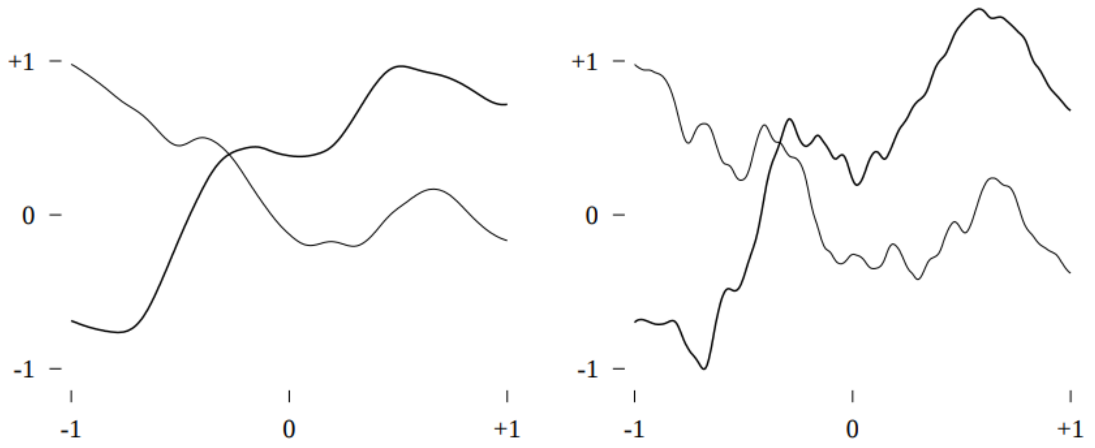


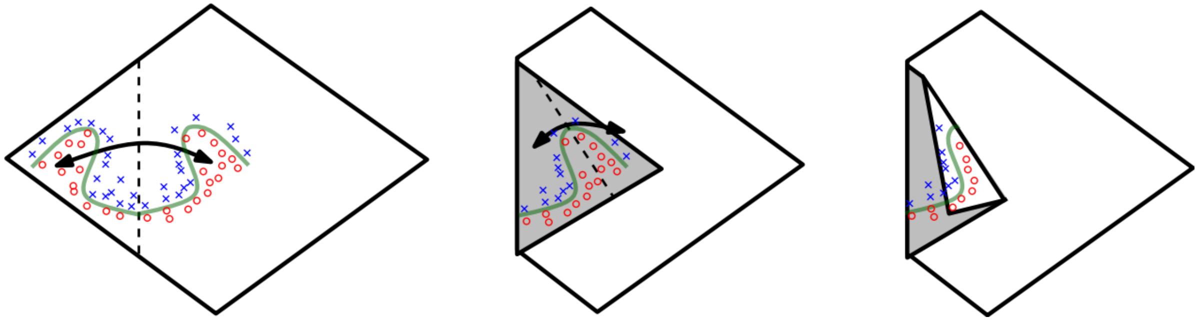
Figure 2.2: Functions drawn from Gaussian priors for a network with 10 000 tanh hidden units. Two functions drawn from a prior with $\sigma_u = 5$ are shown on the left, two from a prior with $\sigma_u = 20$ on the right. In both cases, $\sigma_a/\sigma_u = 1$ and $\sigma_b = \omega_v = 1$. The functions with different σ_u were generated using the same random number seed, the same as that used to generate the functions in the lower-right of Figure 2.1. This allows a direct evaluation of the effect of changing σ_u. (Use of a step function is equivalent to letting σ_u go to infinity, while keeping σ_a/σ_u fixed.)

(Neal, 1995)

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 $\Omega((\frac{q}{p})^{(L-1)p} q^p)$ 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.



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.

The end.

References

Materials from the first part of the lecture are inspired from the excellent Deep Learning Course by Francois Fleuret (EPFL, 2018).

- Lecture 3a: Linear classifiers, perceptron
- Lecture 3b: Multi-layer perceptron

Further references:

- Introduction to ML and Stochastic optimization (Gower, 2017)
- Why are deep neural networks hard to train? (Nielsen, 2017)
- Automatic differentiation in machine learning: a survey (Baydin, 2015)