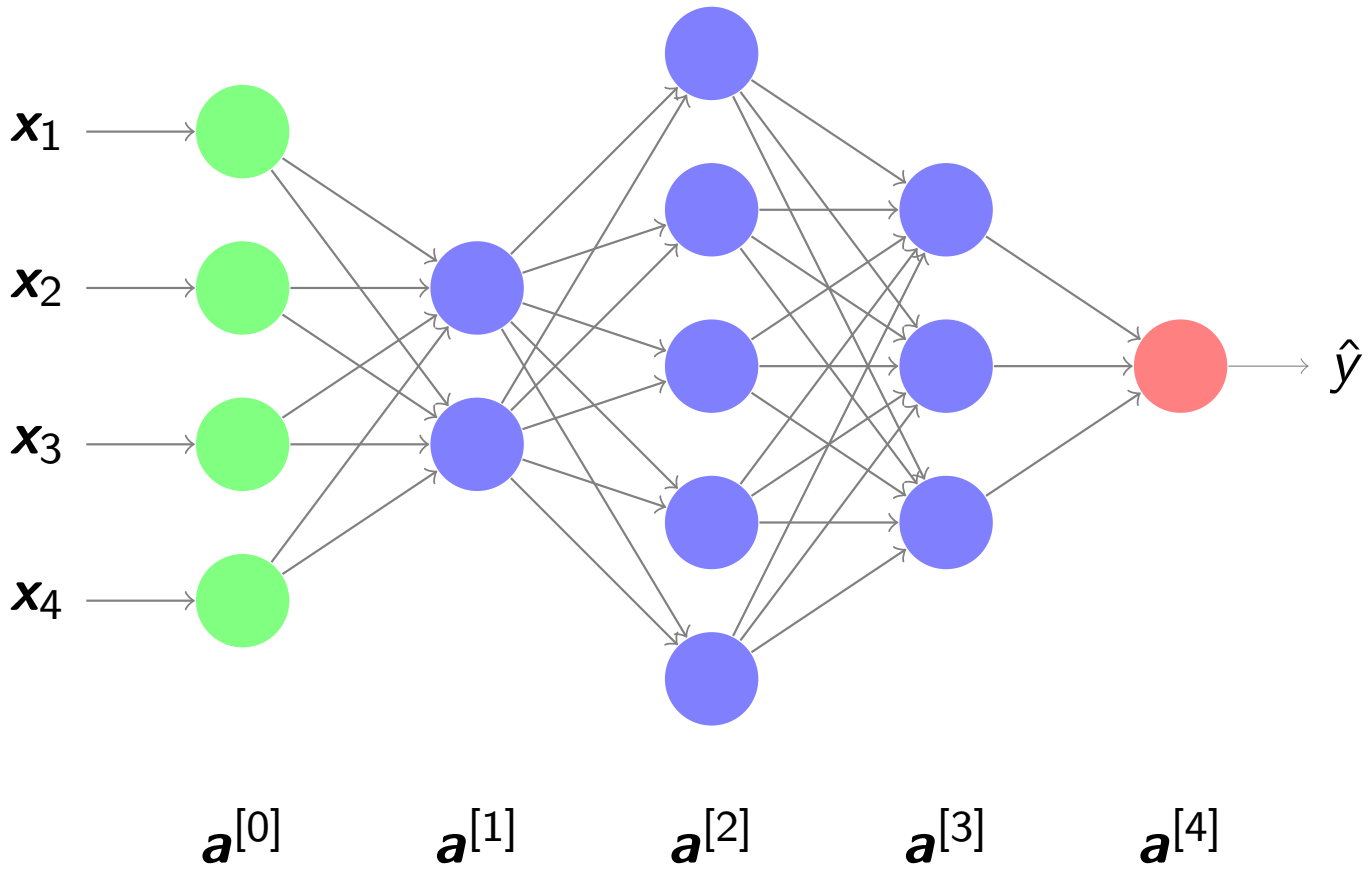
Neural networks

A. Carlier, J-Y. Tourneret

2024

Outline

Neural network: multi-layer perceptron



A multi-layer perceptron can be decomposed into one input layer, one output layer, and several hidden intermediate layers.
The network **depth** here is 4: 3 hidden layers plus 1 output layer.

Notations

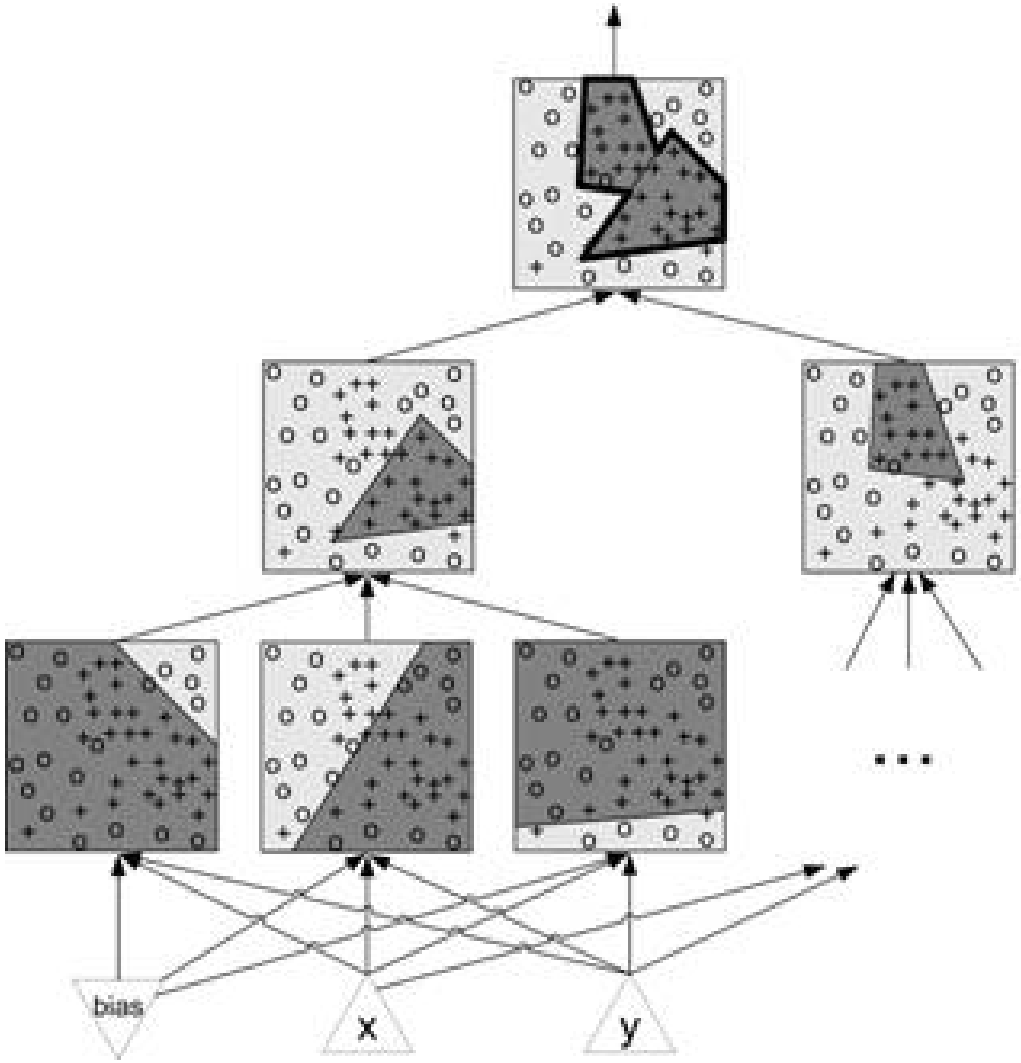
Notations are a little heavy:

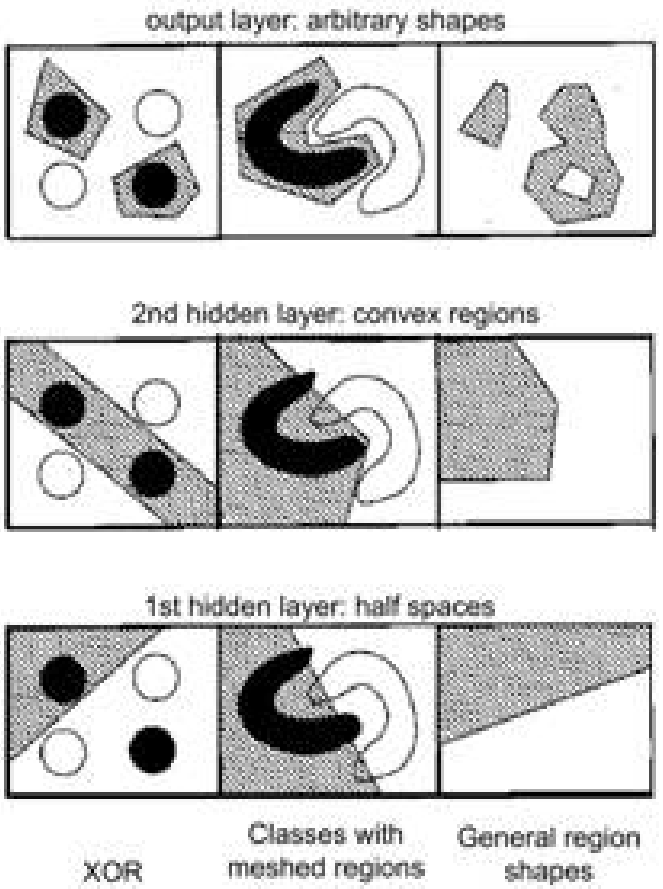
- "()" denotes the index of a data sample in the training set ($\boldsymbol{x}^{(i)}$)
- "$\{\}$" denotes the current iteration in gradient descent ($\boldsymbol{w}^{\{k\}}$)
- "[]" denotes the layer index ($\boldsymbol{a}^{[c]}$)

For example $\boldsymbol{a}_j^{(i)[c]\{k\}}$ denotes the activation of the j-th neuron of layer c, computed from the i-th data sample, during the k-th iteration of gradient descent.

In what follows we will try to simplify the notations whenever possible...

Multi-layer perceptron: Interpretation





Increasing the number of layers and the number of neurons in a multi-layer perceptron increases the network's capacity, and thus its separating ability.

Universal Approximation Theorem

Universal Approximation Theorem (Cybenko 1989)

Any function f, continuous, from $[0,1]^m$ to $\mathbb{R}$, can be approximated by a multi-layer perceptron with a single hidden layer (and sigmoid activation) provided that it has a large enough number of neurons (units).

N.B. The theorem has been proven with `reLU` function too.

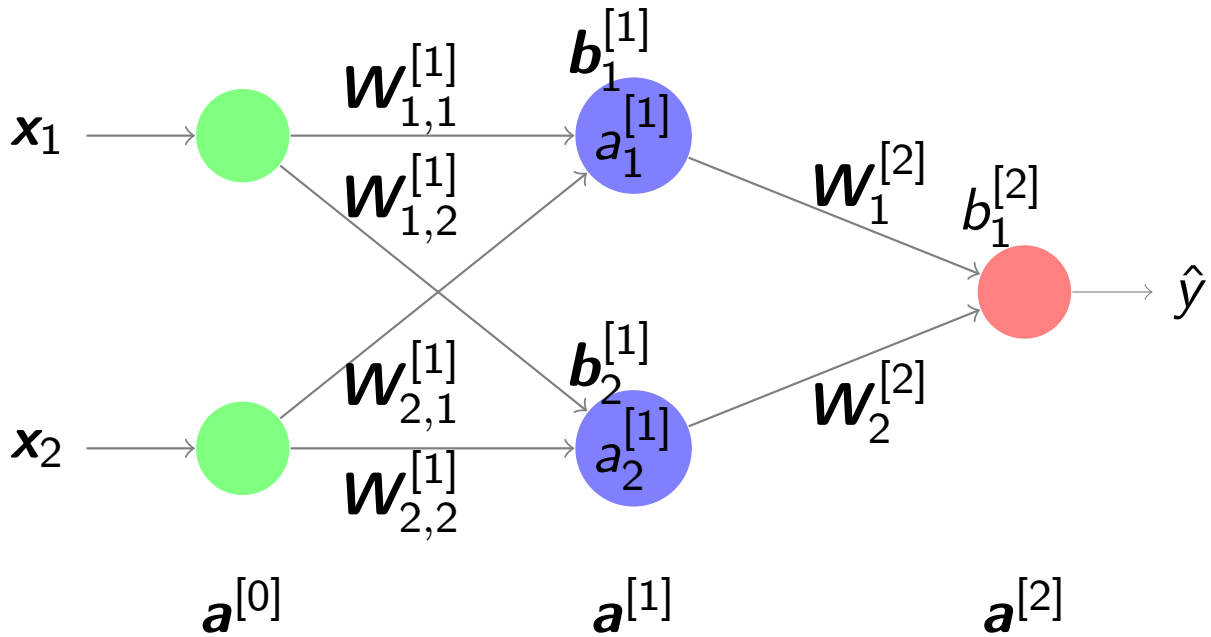
This theorem **does not say how** to choose the best network architecture and how to estimate the network parameters efficiently!

Multi-layer perceptron training

After the initialization phase, the training algorithm (based on gradient descent) is composed of 4 steps that are repeated until convergence:

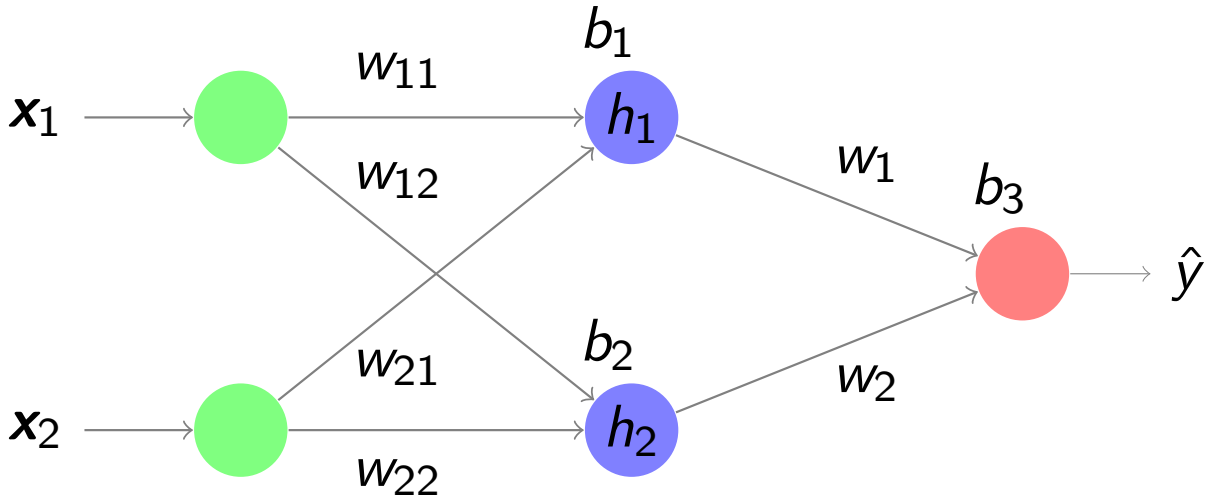
1. **Forward pass** from the input to the output layer;
2. **Objective function** computation at the end of the *forward* pass;
3. **Objective function gradient** computation to update parameters from output and hidden layers;
4. **Parameter update** thanks to previously computed gradients.

Illustration of a multi-layer perceptron training



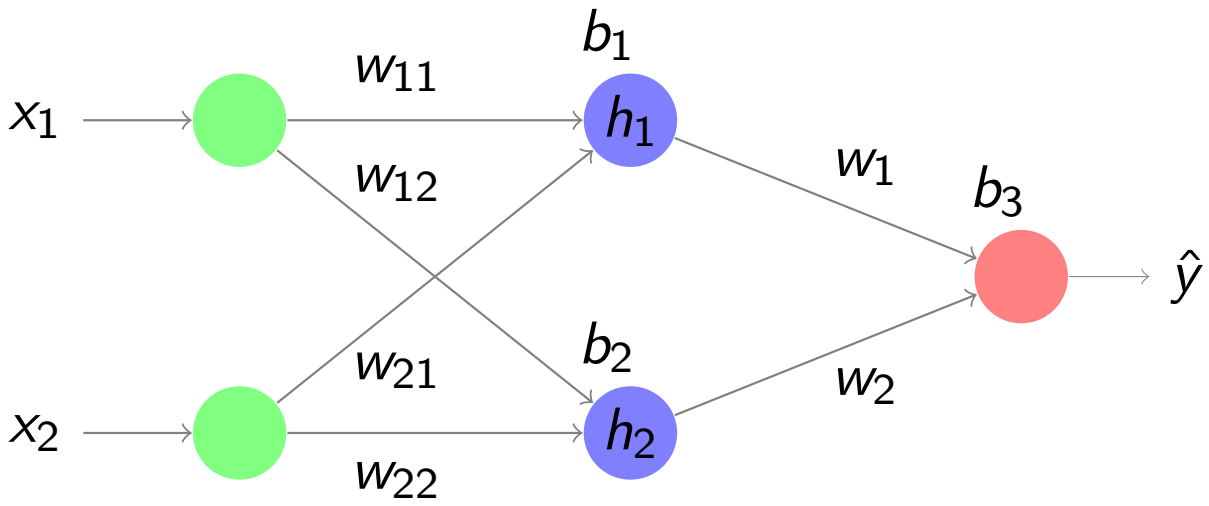
Let us simplify the notations by removing the layer index

Illustration of a multi-layer perceptron training



$$\begin{cases} \hat{y} = \sigma(z) \text{ where } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) \text{ where } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) \text{ where } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

Multi-layer perceptron training



$$\begin{cases} \hat{y} = \sigma(z) \text{ where } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) \text{ where } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) \text{ where } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

1) **Forward pass** from the input to the output layer:

$$\hat{y} = \sigma\left(w_1 f(w_{11} x_1 + w_{21} x_2 + b_1) + w_2 f(w_{12} x_1 + w_{22} x_2 + b_2) - b_3\right)$$

Multi-layer perceptron training

2) **Objective function** computation at the end of the *forward* pass

$$J = \frac{1}{n} \sum_{i=1}^{n} \text{loss}(\hat{y}^{(i)}, y^{(i)})$$

where the loss can be the binary cross-entropy or the mean squared error.
3) **Objective function gradient** computation to correct output layer parameters:
Using the *chain rule* ($\frac{\partial f(y)}{\partial x} = \frac{\partial f(y)}{\partial y} \frac{\partial y}{\partial x}$), we get:

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j},$$

$$\frac{\partial J}{\partial b_3} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial b_3},$$

for $j = \{1, 2\}$

Multi-layer perceptron training

4) **Objective function gradient** computation to correct hidden layers parameters:

$$\frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_{j'}} \frac{\partial h_{j'}}{\partial z_{j'}} \frac{\partial z_{j'}}{\partial w_{jj'}},$$
$$\frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial b_j},$$

for $j, j' = \{1, 2\}$.

5) Parameter **update** of synaptic weights in the output layer:

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}$$

and the hidden layer:

$$w_{jj'} \leftarrow w_{jj'} - \alpha \frac{\partial J}{\partial w_{jj'}}.$$

(biases are updated the same way)

Gradient backpropagation

In order to compute gradients with respect to synaptic weights:

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j},$$

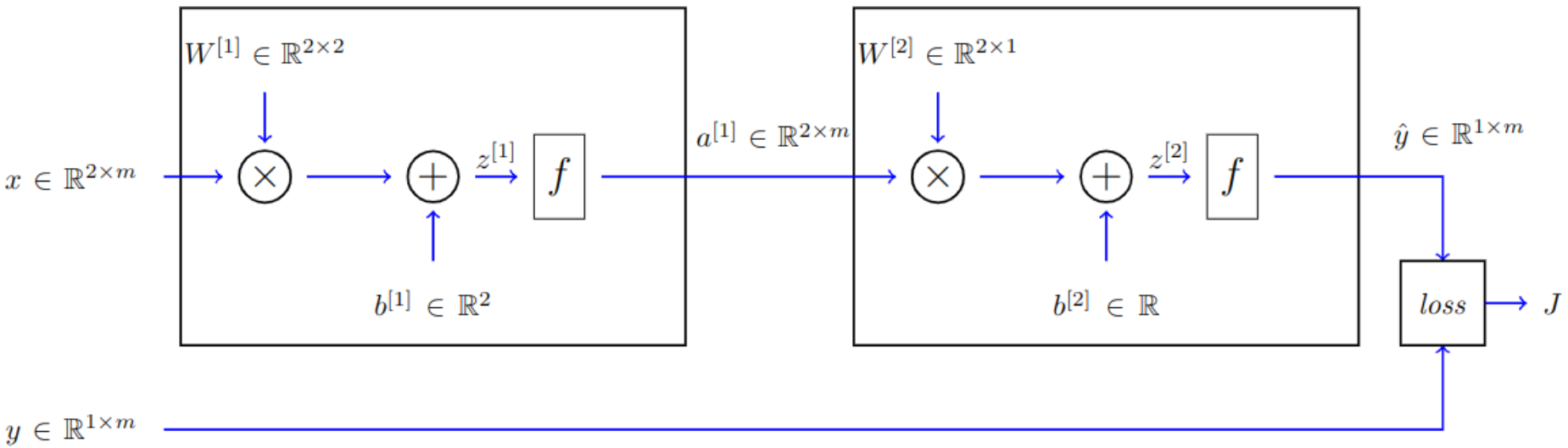
and

$$\frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_{j'}} \frac{\partial h_{j'}}{\partial z_{j'}} \frac{\partial z_{j'}}{\partial w_{jj'}},$$

It is interesting to compute first $\frac{\partial J}{\partial w_j}$, and reuse the intermediate computations to then compute $\frac{\partial J}{\partial w_{jj'}}$.
This is the efficient algorithm of **gradient backpropagation** which is applied from the last to the first layer of the network.

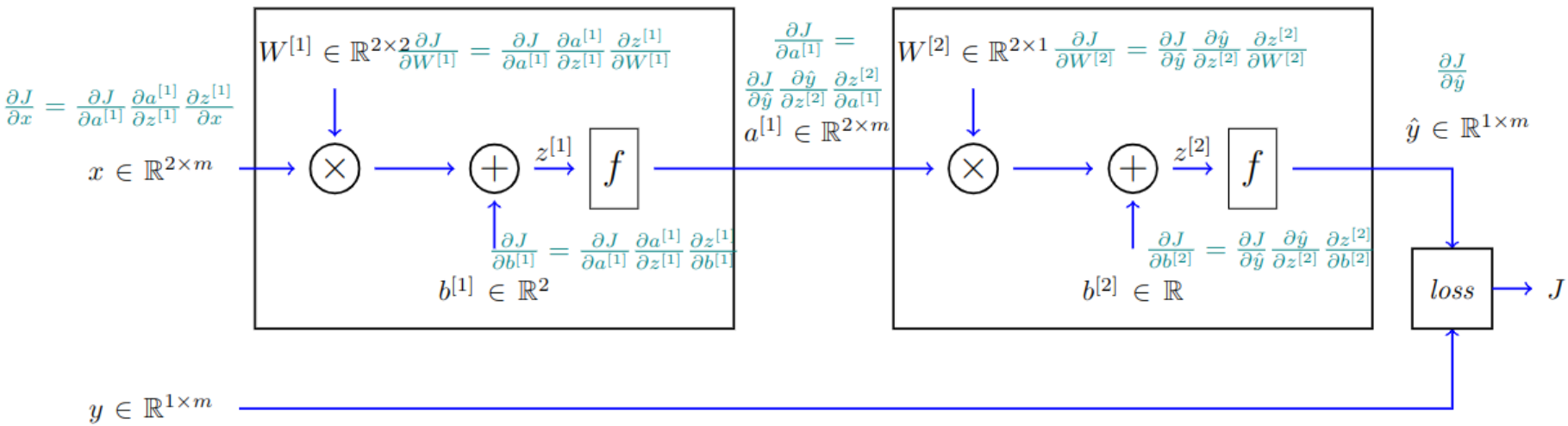
Gradient backpropagation

Another visualization:



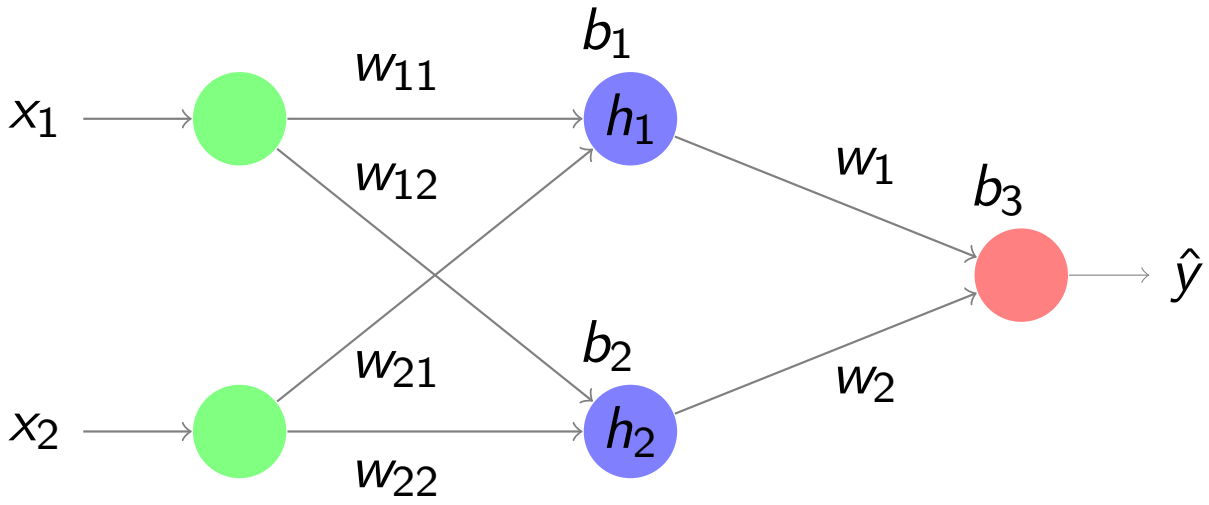
Gradient backpropagation

Another visualization:



Outline

Back to gradient computation



$$\begin{cases} \hat{y} = \sigma(z) \text{ where } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) \text{ where } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) \text{ where } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

Back to gradient computation

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

We can compute:

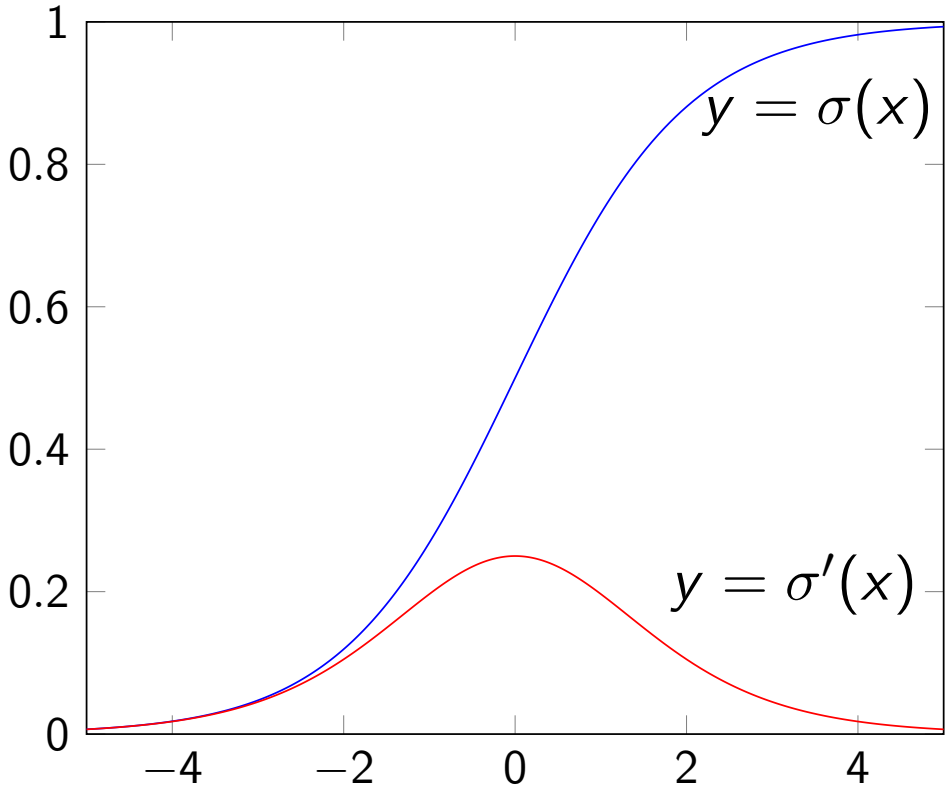
$$\frac{\partial z}{\partial h_j} = w_j$$

$$\frac{\partial h_j}{\partial z_j} = f'(z_j)$$

where f' is the activation function derivative.

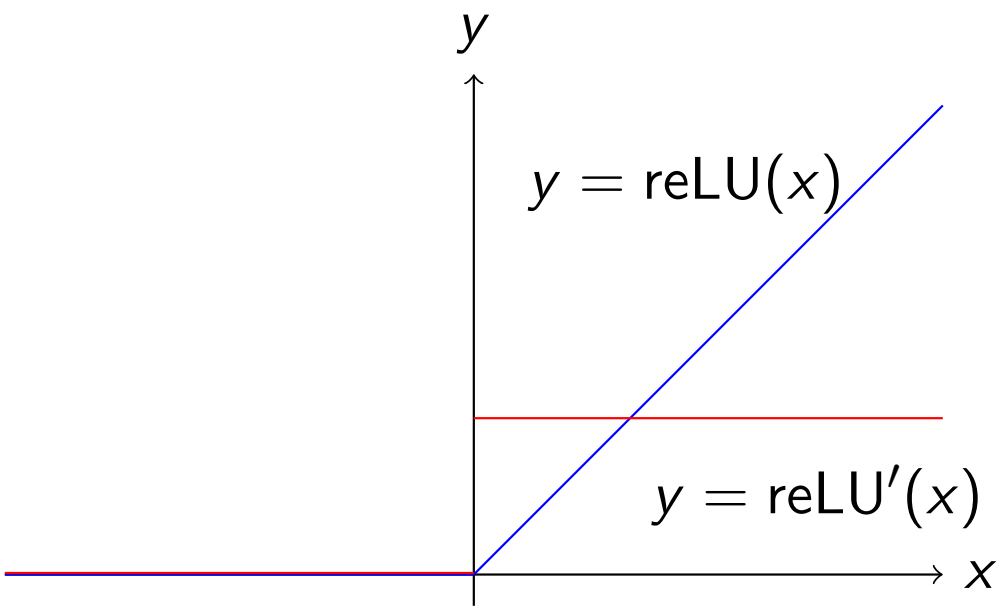
$$\frac{\partial z_j}{\partial w_{ij}} = x_i$$

Zoom on the sigmoid function



The derivative of the sigmoid function has values in $[0, \frac{1}{4}]$, which reduces the amplitude of the gradients propagated through the layers of the neural network. This is known as the **vanishing gradient** problem.

The *rectified Linear Unit* function



$$\text{reLU}(x) = \begin{cases} 0 & \text{if} \quad x < 0 \\ x & \text{else} \end{cases} \tag{1}$$

A gradient that is always 1 in the activated domain eliminates the vanishing gradient problem and greatly improves convergence.

Outline

Gradient Descent

Algorithm: Gradient descent $(\mathcal{D}, \alpha)$
Initialize $\boldsymbol{\beta}^{\{0\}} \leftarrow 0$, $k \leftarrow 0$
WHILE no convergence **DO**
  FOR j from 1 to p **DO**
    $$\boldsymbol{\beta}_j^{\{k+1\}} \leftarrow \boldsymbol{\beta}_j^{\{k\}} - \alpha \frac{\partial J(\boldsymbol{\beta}^{\{k\}})}{\partial \boldsymbol{\beta}_j}$$
  END FOR
  $k \leftarrow k + 1$
END WHILE

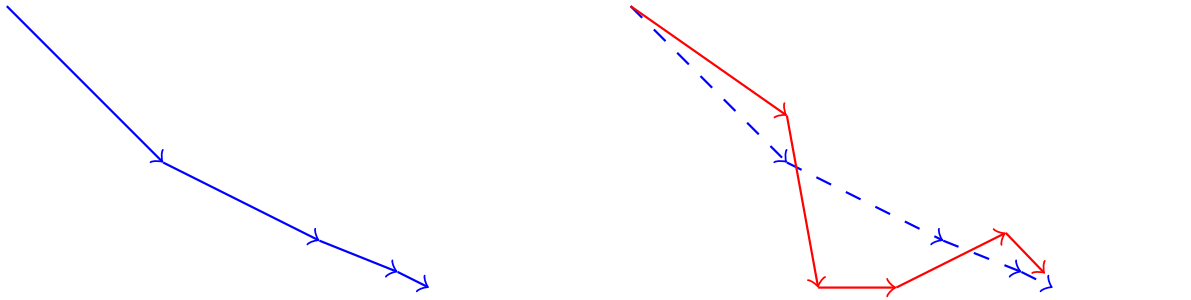
Gradient descent requires computing $J(\boldsymbol{\beta}^{\{k\}})$ at every step,

$$J(\boldsymbol{\beta}^{\{k\}}) = \frac{1}{n} \sum_{i=1}^{n} \text{loss}(\hat{y}^{(i)}, y^{(i)})$$

i.e. it requires to perform n (number of training samples) network predictions for each step, which is not convenient.

Stochastic gradient descent

The idea of stochastic gradient descent is to approximate $J(\beta^{\{k\}})$ using a small set of samples, called a *mini-batch*.



Gradient descent Stochastic gradient descent

Note: a mini-batch that is too small can generate too much noise on the gradient estimation and prevent convergence.

In practice, we partition the training set into mini-batches, and call an **epoch** the set of iterations required to predict the entire training set.

Stochastic Gradient Descent

Algorithm: Gradient descent ($\mathcal{D}, \alpha$, nb_epochs, batch_size)
Initialize $\boldsymbol{\beta}^{\{0\}} \leftarrow 0$, $k \leftarrow 0$
FOR e from 1 to nb_epochs **DO**
  FOR b from 1 to $\lfloor \frac{n}{\text{batch\_size}} \rfloor$ **DO**
    Compute $J(\boldsymbol{\beta}^{\{k\}}) = \frac{1}{\text{batch\_size}} \sum_{i=1}^{\text{batch\_size}} \text{loss}(\hat{y}^{(i)}, y^{(i)})$
    FOR j from 1 to p **DO**

$$\boldsymbol{\beta}_j^{\{k+1\}} \leftarrow \boldsymbol{\beta}_j^{\{k\}} - \alpha \frac{\partial J(\boldsymbol{\beta}^{\{k\}})}{\partial \boldsymbol{\beta}_j}$$

    END FOR
    $k \leftarrow k + 1$
  END FOR
END WHILE

Momentum

To avoid problems related to noisy gradients, we add an inertia term (*momentum*)

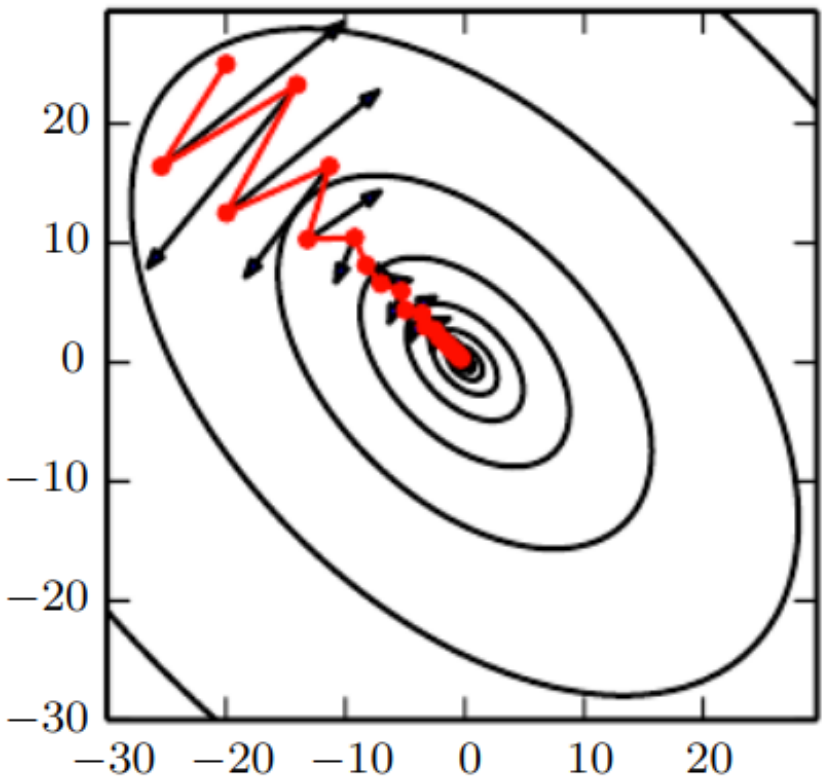


Image from [Goodfellow et al. 2015] Deep Learning

Momentum

In practice, we adapt the gradient descent algorithm, replacing the update of the parameters

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \frac{\partial J}{\partial \boldsymbol{\beta}}$$

with 2 steps:

$$\boldsymbol{v} = \eta \boldsymbol{v} - \alpha \frac{\partial J}{\partial \boldsymbol{\beta}}$$

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \boldsymbol{v}$$

where $\boldsymbol{v}$ (for velocity) denotes the direction in which the parameters will be modified. $\boldsymbol{v}$ takes the previous gradients into account via the parameter η ($0 < \eta < 1$), which quantifies the relative importance of the previous gradients compared to the current gradient

Advanced optimizers to improve SGD

In high dimensional parameter spaces, the topology of the objective function makes gradient descent sometimes inefficient. We can improve the latter by using adapted optimizers.

The **AdaGrad** optimizer introduces a form of learning rate adaptation by accumulating the squares of previous gradients.

1. Gradient computation: $\boldsymbol{g} = \frac{\partial J}{\partial \boldsymbol{\beta}}$
2. Gradient norm accumulation: $r = r + ||\boldsymbol{g}||_2$
3. Parameter update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \frac{\alpha}{\sqrt{r}}\boldsymbol{g}$

Advanced optimizers to improve SGD

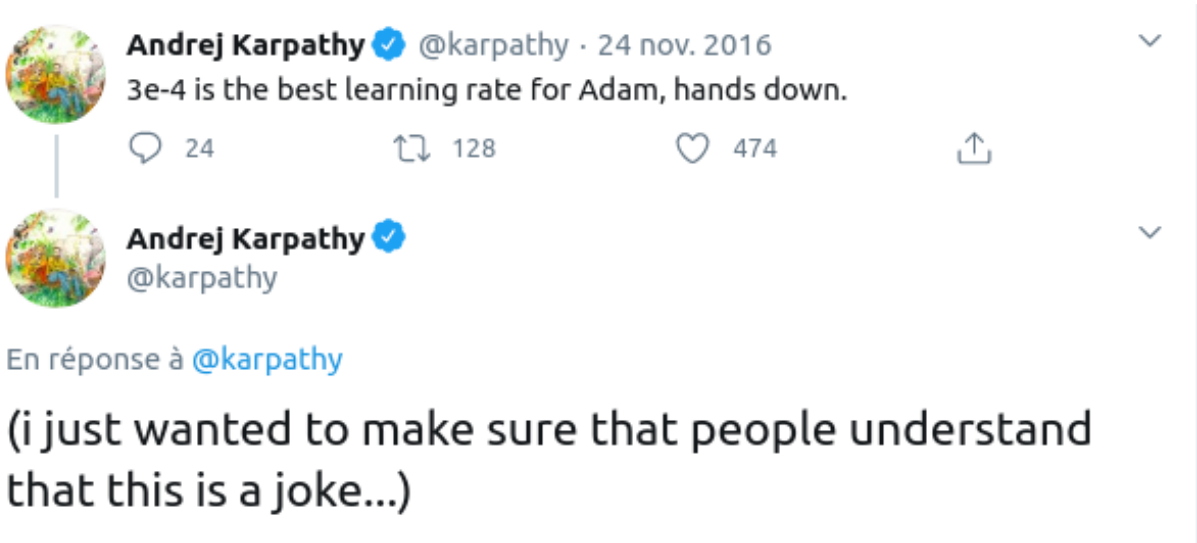
The **RMSProp** optimizer is almost identical to AdaGrad, but the impact of the oldest gradients is altered by a multiplicative coefficient ρ less than 1 (*weight decay*), which improves the behavior of the algorithm in the case of elongated bowls.

1. Gradient computation: $\boldsymbol{g} = \frac{\partial J}{\partial \boldsymbol{\beta}}$
2. Gradient accumulation: $r = \rho r + (1-\rho)||\boldsymbol{g}||_2$
3. Parameter update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \frac{\alpha}{\sqrt{r}}\boldsymbol{g}$

Finally, the **Adam** optimized is similaire to RMSProp but also adapts momentum.

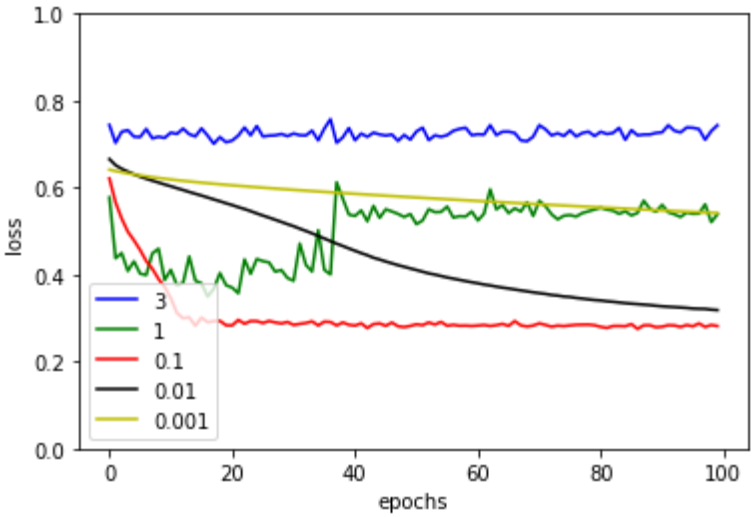
Choosing an optimizer in practice

Adam is often a good choice to start (with the "magic" learning rate):

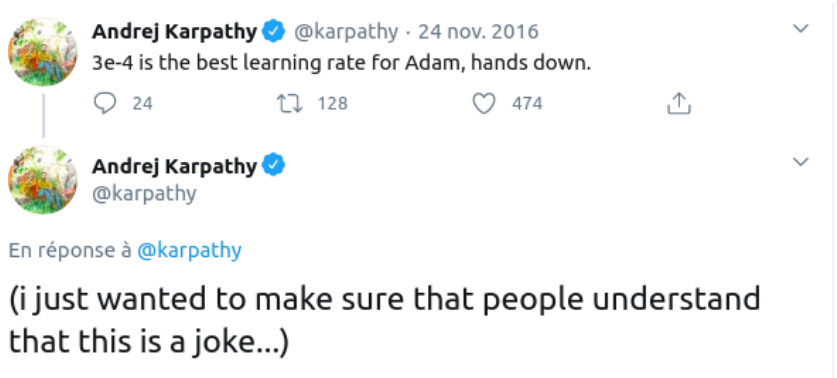


In practice, the best results in the state-of-the-art are obtained with a simple stochastic gradient descent, and a programmed update of the learning rate (cyclic, cosine, etc.)

SGD vs. Adam on a simple example



SGD



Adam

(Evolution of the loss during training, for different learning rates, with SGD and Adam)

Outline

Early stopping

Early stopping is a regularization strategy which consists in observing the validation error and stopping the learning when this error starts to rise.

In practice, validation error is noisy, we have to wait a little (*patience* parameter) before stopping for good.

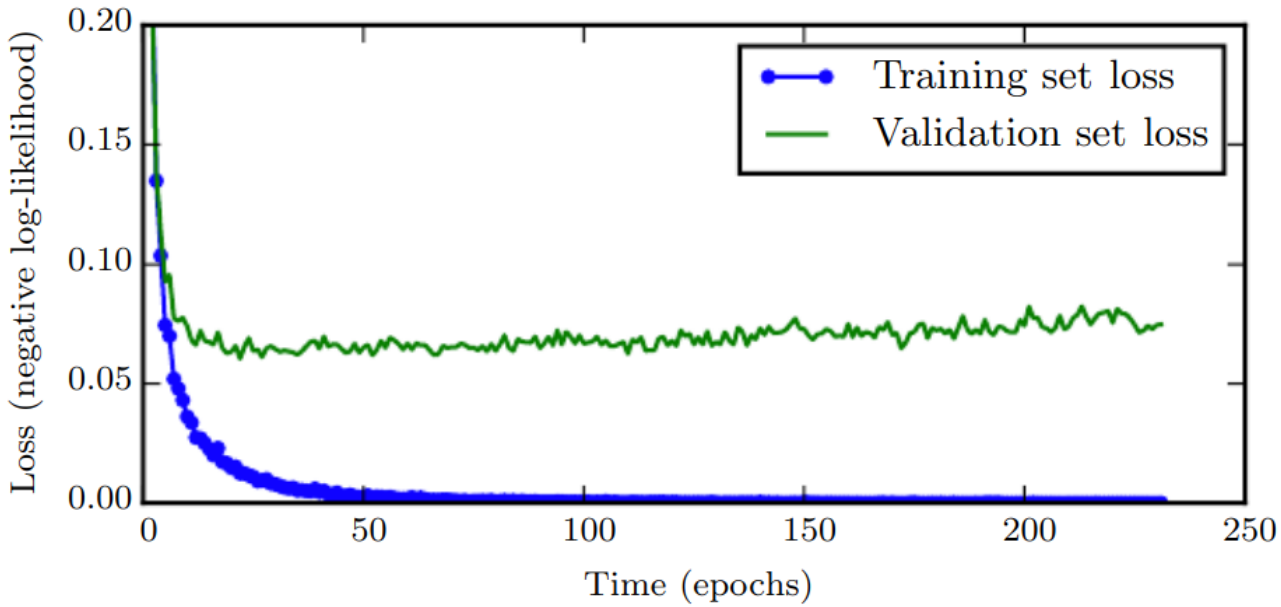


Image from [Goodfellow et al. 2015] Deep Learning

Weight decay

Add a constraint on the network parameters:

- $\mathcal{L}^2$ or **Ridge** regularization keeps the model coefficients as small as possible:

$$J(\theta) = \text{EmpiricalRisk}(\theta) + \lambda \frac{1}{2} \sum_{i=1}^{m} \theta_i^2$$

  where λ controls the desired regularization quality
- $\mathcal{L}^1$ or **Lasso** regularization: tends to completely eliminate the weights of the least important variables ($\Rightarrow$ produces a sparse model):

$$J(\theta) = \text{EmpiricalRisk}(\theta) + \lambda \sum_{i=1}^{m} |\theta_i|$$

[Krogh, Hertz 1992] A simple weight decay can improve generalization

Weight decay

- **Elastic net** regularization: compromise of Ridge and Lasso handled by an extra parameter $r \in [0, 1]$:

$$J(\theta) = \text{EmpiricalRisk}(\theta) + r\lambda\frac{1}{2}\sum_{i=1}^{m}|\theta_i| + \frac{1-r}{2}\lambda\sum_{i=1}^{m}\theta_i^2$$

`https://playground.tensorflow.org/`

Data augmentation

Using small datasets can lead to overfitting.

Image de `https://github.com/aleju/imgaug`