Machine Learning

9. Neural networks and deep learning

Yannick Le Cacheux

CentraleSupélec - Université Paris Saclay

September 2024

Outline

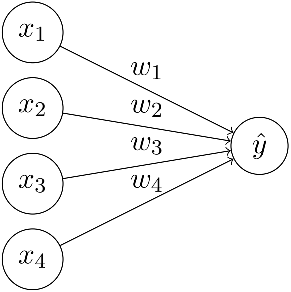
1. Artificial neural networks

2. Back-propagation

Logistic regression

- In logistic regression, for input vector $\mathbf{x} \in \mathbb{R}^D$, output probability $\hat{y}$ is estimated with

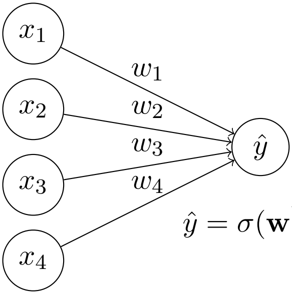
$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + \cdots + w_D x_D) = \sigma(\mathbf{w}^\top \mathbf{x})$$



$$\mathbf{x} \in \mathbb{R}^D \qquad \hat{y} = \sigma(\mathbf{w}^\top \mathbf{x}) \in [0, 1]$$

Artificial neuron

- This resembles a single neuron: the inputs w_d represent the *dendrites*, and the output $\hat{y}$ the *axon*.
- If enough "stimulation" arrives from the dendrites (*i.e.* $\mathbf{w}^\top \mathbf{x} \gg 0$), the neuron fires: $\hat{y} \approx 1$
- Otherwise, if $\mathbf{w}^\top \mathbf{x} \ll 0$, nothing happens: $\hat{y} \approx 0$



$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x})$$

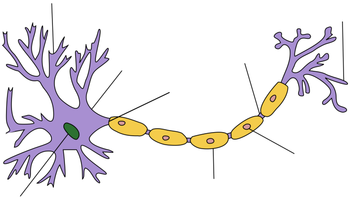
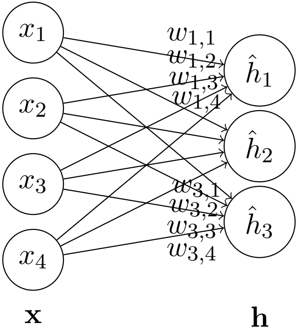


Figure: Representation of a human neuron, from Wikipedia

Multi-output logistic regression

- We can do multiple logistic regression at the same time to have H outputs $\mathbf{h} = (\hat{h}_1, \dots, \hat{h}_H)^\top$



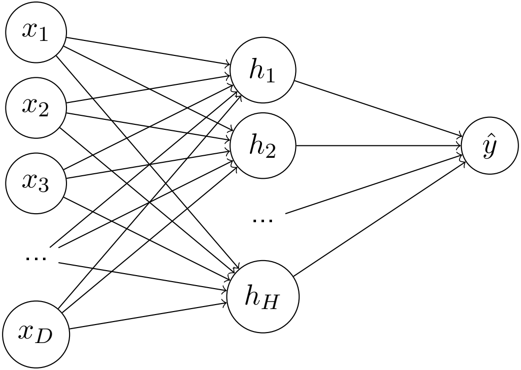
$$
\begin{aligned}
\hat{h}_1 &= \sigma(w_{1,1}x_1 + \cdots + w_{1,D}x_D) \\
&= \sigma(\mathbf{w}_1^\top \mathbf{x}) \\
\hat{h}_2 &= \sigma(\mathbf{w}_2^\top \mathbf{x}) \\
\dots \\
\hat{h}_H &= \sigma(\mathbf{w}_H^\top \mathbf{x})
\end{aligned}
$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x})$$

- We now have multiple neurons

Artificial neural network

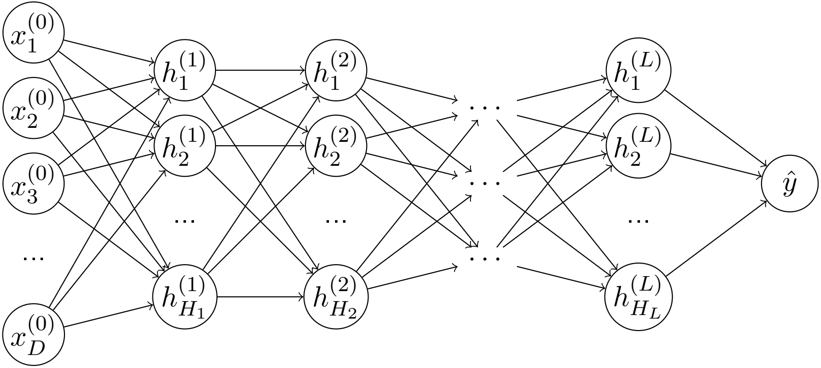
- In the brain, the outputs of neurons are the inputs to other neurons
- This is equivalent to using $\mathbf{h}$ as the features to other model(s)



$\mathbf{x} \in \mathbb{R}^D$ $\quad \mathbf{h} = \sigma(\mathbf{W}^{(1)}\mathbf{x}) \in \mathbb{R}^H$ $\quad \hat{y} = \sigma(\mathbf{W}^{(2)}\mathbf{h}) \in \mathbb{R}$

Deep learning

- We can stack even more layers, to obtain a *deep* neural network with L hidden layers $\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(L)}$



$\mathbf{x}^{(0)} \in \mathbb{R}^D$ $\quad \mathbf{h}^{(1)} \in \mathbb{R}^{H_1} = \sigma(\mathbf{W}^{(1)}\mathbf{x})$ $\quad \mathbf{h}^{(2)} \in \mathbb{R}^{H_2} = \sigma(\mathbf{W}^{(2)}\mathbf{h}^{(1)})$ $\quad \mathbf{h}^{(L)} \in \mathbb{R}^{H_L} = \sigma(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)})$ $\quad \hat{y} \in [0,1]$ $\sigma(\mathbf{W}^{(L+1)}\mathbf{h}^{(L)})$

- In general $\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}) \in \mathbb{R}^{H_l}$

Training a deep neural net

- How do we find the appropriate values for all P parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L+1)}\}_{\text{flattened}} \quad (\in \mathbb{R}^P)$ of the network?
- Given training dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$ and a loss function $J_{\mathcal{D}}(\boldsymbol{\theta})$, for example

$$J_{\mathcal{D}}(\boldsymbol{\theta}) = -\sum_n \left(y_n \log(\hat{y}_n) + (1 - y_n) \log(1 - \hat{y}_n)\right)$$

- We can apply gradient descent: at time step t,

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \frac{\partial J_{\mathcal{D}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_{t-1}}$$

Computing the gradients

- Gradient for parameter θ_i can be numerically estimated with finite differences method: apply a small change ϵ in direction $\mathbf{e}_i$ of θ_i, and measure the impact on the loss:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{J(\boldsymbol{\theta} + \epsilon \mathbf{e}_i) - J(\boldsymbol{\theta})}{\epsilon}$$

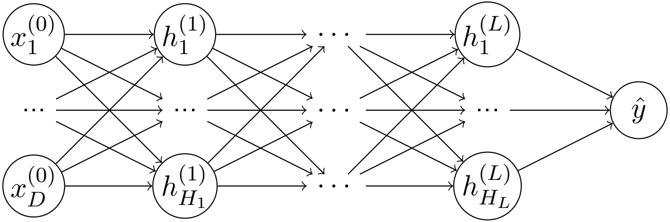
- This can be very computationaly intensive: to obtain the gradient of a single parameter θ_i, we need to apply a small change ϵ and propagate the impact forward in the whole network until we can measure the impact on the loss J. Then repeat these operations on every single parameter.
  - And we may many have layers, many parameters and many training samples.
- This is not efficient at scale.

Back-propagation

- Main idea: use dynamic progamming and the chain rule of derivatives to avoid redundant computations in gradient estimation.

Neural network for regression

- For regression, the idea is exactly the same: we compute intermediate features as hidden layers $\mathbf{h}^{(l)}$ and find suitable parameters with gradient descent, with two changes:
  - We remove the output sigmoid σ so that $\hat{y} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)}$ can have values in $\mathbb{R}$
  - We use a suitable loss function J (*e.g.* an MSE) to train the network



$\mathbf{x}^{(0)} \in \mathbb{R}^D$ $\quad$ $\mathbf{h}^{(1)} \in \mathbb{R}^{H_1} = \sigma(\mathbf{W}^{(1)}\mathbf{x})$ $\quad$ $\mathbf{h}^{(l)} \in \mathbb{R}^{H_l} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)})$ $\quad$ $\hat{y} \in \mathbb{R} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)}$

- Should we keep the intermediate sigmoids $\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)})$?

Activation function

- In practice, we often use *rectified linear units* (ReLU) as *activation* functions instead of the sigmoid σ for intermediate layers[1]:

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

- Is it necessary to use an activation function?

[1] This is to avoid a problem called *vanishing gradients*; out of scope for now.

Activation function

- In practice, we often use *rectified linear units* (ReLU) as *activation* functions instead of the sigmoid σ for intermediate layers[1]:

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

- Is it necessary to use an activation function?

- Yes: a linear combination of linear combinations is a linear combination.
- Without non-linear activation functions, our network is equivalent to a network with no hidden layer: we would be doing linear regression(with unnecessary complications)

Without activation functions:

$$\begin{aligned} \hat{y} &= \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} \text{ and } \mathbf{h}^{(L)} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} \\ \text{so } \hat{y} &= \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} \\ &= \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)}\mathbf{h}^{(L-2)} \\ &= \dots \\ &= \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\dots\mathbf{W}^{(1)}\mathbf{x} \\ &= \mathbf{W}_{\text{eq}}\mathbf{x} \end{aligned}$$

[1] This is to avoid a problem called *vanishing gradients*; out of scope for now.

Deep learning is a whole field

There are many additional things that we would need to cover:

- Different architectures: number of hidden layers, dimensions of the hidden layers...
- But also different ways to organize and connect neurons: convolutional neural networks (CNN), recurrent neural networks, transformers...
- Different ways to efficiently train a network: stochastic gradient descent, momentum, batch norm...
- Different ways to limit overfitting: early stopping, weight penalty, drop-out...
- ...

But all of this is out of scope for now.

[1] *Cf.* I. Goodfellow et al., *Deep learning* for a good introduction, and S. Prince, *Understanding deep learning* for coverage of more recent topics (transformers, diffusion models etc)

Outline

1 Artificial neural networks

2 Back-propagation

Chain rule of derivatives

- Let us consider values $x, y, z \in \mathbb{R}$, and functions $f, g : \mathbb{R} \to \mathbb{R}$:

$$x \xrightarrow{f} y \xrightarrow{g} z$$

$$x \qquad y = f(x) \qquad \begin{aligned} z &= g(y) \\ &= g(f(x)) \end{aligned}$$

- Chain rule of derivatives[2]:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

- (French high-school version):

$$\underbrace{g(f(x))'}_{\frac{\partial z}{\partial x}} = \underbrace{f'(x)}_{\frac{\partial y}{\partial x}} \cdot \underbrace{g'(f(x))}_{\frac{\partial z}{\partial y}}$$

[2] Assuming derivatives exist etc etc

Chain rule of derivatives

- Given $\mathbf{x} \in \mathbb{R}^L, \mathbf{y} \in \mathbb{R}^M, \mathbf{z} \in \mathbb{R}^N$ and $\mathbf{f} : \mathbb{R}^L \to \mathbb{R}^M$, $\mathbf{g} : \mathbb{R}^M \to \mathbb{R}^N$:

$$\mathbf{x} \xrightarrow{\mathbf{f}} \mathbf{y} \xrightarrow{\mathbf{g}} \mathbf{z}$$

- Chain rule of derivatives, vector version:

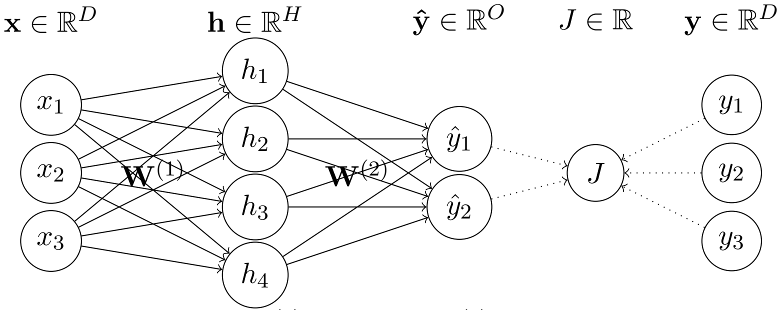
$$\underbrace{\frac{\partial \mathbf{z}}{\partial \mathbf{x}}}_{\in \mathbb{R}^{N \times L}} = \underbrace{\frac{\partial \mathbf{z}}{\partial \mathbf{y}}}_{\in \mathbb{R}^{N \times M}} \cdot \underbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}_{\in \mathbb{R}^{M \times L}}$$

where " $\cdot$ " is now the matrix product[3]

[3] or tensor product in an even more general case

Application to deep learning

- Let us consider the following neural network trained with loss J
  - *How many outputs O does it have? Hidden layers L? Parameters P?...*



$= \sigma(\mathbf{W}^{(1)}\mathbf{x}) \quad = \sigma(\mathbf{W}^{(2)}\mathbf{h}) \quad = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$

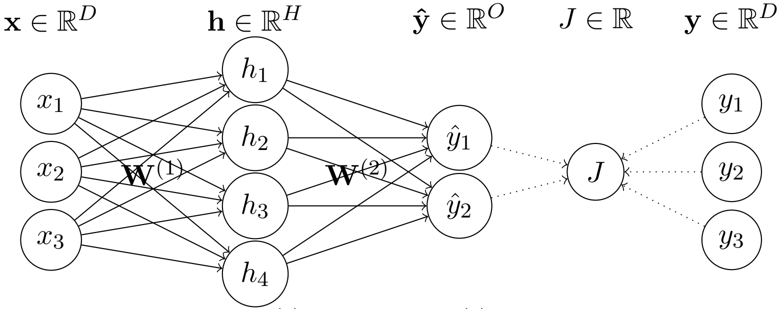
We want the gradient $\frac{\partial J}{\partial \boldsymbol{\theta}}$ of all the P parameters θ_p in $\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$

- So we want $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$

[3] σ can be any (non linear) activation function

Application to deep learning

- Let us consider the following neural network trained with loss J
  - $P = H \times D + O \times H$. Here $L = 1$, $D = 4$, $H = 3$, $O = 2$.



$= \sigma(\mathbf{W}^{(1)}\mathbf{x}) \quad = \sigma(\mathbf{W}^{(2)}\mathbf{h}) \quad = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$

We want the gradient $\frac{\partial J}{\partial \boldsymbol{\theta}}$ of all the P parameters θ_p in $\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$

- So we want $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$

[3] σ can be any (non linear) activation function

Gradient of $\mathbf{W}^{(2)}$

- $\mathbf{W}^{(2)}$: this is a matrix of size $O \times H$, we can "flatten" it to a vector $\mathbf{w}^{(2)}$ of size $S_2 = O \cdot H$. Then

$$\frac{\partial J}{\partial \mathbf{w}^{(2)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}^{(2)}}$$

- We can estimate $\dfrac{\partial J}{\partial \hat{\mathbf{y}}}$ and $\dfrac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}^{(2)}}$ with the finite differences method: apply small pertubations on $\mathbf{w}^{(2)}$ and measure the impact on $\hat{\mathbf{y}}$, apply small pertubations on $\hat{\mathbf{y}}$ on measure the impact on J.

- Note: as the other gradients, $\frac{\partial J}{\partial \hat{\mathbf{y}}}$ depends on the choice of the loss function J. For the MSE $J = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$ we have $\frac{\partial J}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y})$. For intermediate gradients that rely on matrix multiplication, we can similarly find more efficient computations than finite differences method. But in general, we may not always have analytical solutions.

Gradient of $\mathbf{W}^{(1)}$

- We similarly flatten $\mathbf{W}^{(1)} \in \mathbb{R}^{H \times D}$ to $\mathbf{w}^{(1)} \in \mathbb{R}^{H \cdot D}$, and then

$$\frac{\partial J}{\partial \mathbf{w}^{(1)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \cdot \underbrace{\left(\quad \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}} \quad \right)} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{w}^{(1)}}$$

$$= \frac{\partial J}{\partial \hat{\mathbf{y}}} \cdot \left(\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}^{(2)}} \cdot \frac{\partial \mathbf{w}^{(2)}}{\partial \mathbf{h}} \right) \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{w}^{(1)}}$$

- We have already computed the first 2 factors $\frac{\partial J}{\partial \hat{\mathbf{y}}}$ and $\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}^{(2)}}$ when computing the gradient of $\mathbf{W}^{(2)}$
- All that is left to do is compute the last 2 factors $\frac{\partial \mathbf{w}^{(2)}}{\partial \mathbf{h}}$ and $\frac{\partial \mathbf{h}}{\partial \mathbf{w}^{(1)}}$, again with a numerical estimation
- This is efficient: we do not need to propagate forward from $\mathbf{w}^{(1)}$ along the whole network until we reach J

Gradient of $\mathbf{W}^{(l)}$

- Generalization: with L hidden layers $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(L)}$ with associated weights $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}$ and $\mathbf{W}^{(L+1)}$ (output weights)
- We first compute $\dfrac{\partial J}{\partial \mathbf{w}^{(L+1)}}$, then $\dfrac{\partial J}{\partial \mathbf{w}^{(L)}}$, $\dfrac{\partial J}{\partial \mathbf{w}^{(L-1)}}$ etc, using

$$\frac{\partial J}{\partial \mathbf{w}^{(l)}} = \underbrace{\frac{\partial J}{\partial \mathbf{w}^{(l+1)}}}_{\text{already computed}} \cdot \underbrace{\frac{\partial \mathbf{w}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{w}^{(l)}}}_{\text{left to estimate}}$$

- We *propagate* the gradients backwards, hence the name *back-propagation*

Back-propagation: recap

Back-propagation

Back-propagation is a method that lets us efficiently estimate the gradients of all parameters of a (deep) neural network, by propagating already computed gradients backwards in the network to avoid redundant computations.

- We can then find good parameters $\hat{\boldsymbol{\theta}}$ by applying a simple gradient descent:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \frac{\partial J_{\mathcal{D}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_{t-1}}$$

or more complex gradient-based methods (Adam, etc)