INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Machine Learning – 4A
Neural Networks

Team: A. Bit-Monnot, E. Chanthery, A. Dorise, M-J. Huguet, P. Leleux, M. Siala

Université Fédérale Toulouse Midi-Pyrénées

Section 1

Multi-layer perceptron (MLP)

Limitation of the perceptron

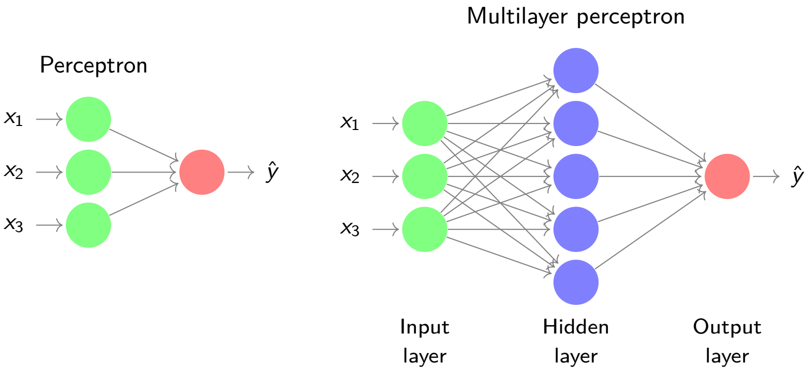
So far, we limited ourselves to functions of the form :

$$h_w(x) = w \cdot x \text{ or}$$
$$h'_w(x) = Logistic(w \cdot x)$$

Main limitations :

- each feature contributes to the output **independently of the others**.
- our hypothesis space is **linear**.

From one to multi-layers perceptron
Feedforward networks



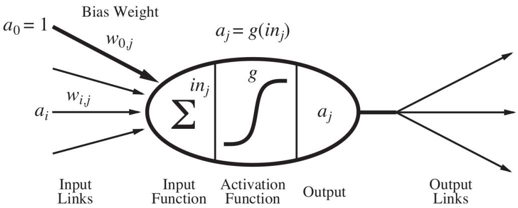
N.B. : some NN can feed their output (or intermediate results) back into their inputs $\Longrightarrow$ Recurrent Neural Networks (RNN).

Neural unit

A neural unit j has :

- an output a_j
- a weight $w_{i,j}$ for each unit i it takes as input
- an non-linear **activation function** g_j

$$a_j = g_j(\sum_i w_{i,j} \times a_i)$$



Common activation functions

- **logistic** or **sigmoid** function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- the **rectified linear unit** function (**ReLu**)

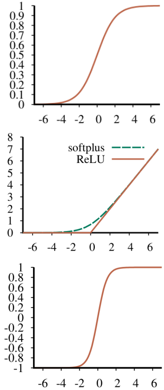
$$ReLU(z) = max(0, z)$$

- the **softplus** function

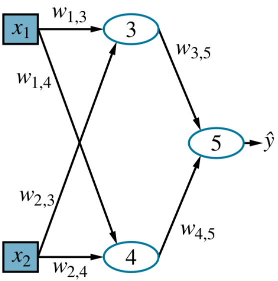
$$softplus(z) = log(1 + e^{z})$$

- the **tanh** function

$$tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



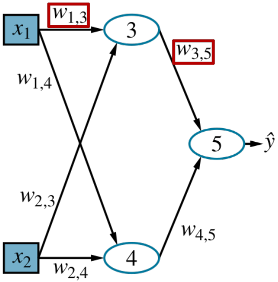
Neural network : training



$$\hat{y} = h_w(x) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

where $a_3 = g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)$ and $a_4 = g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2)$

Training network with $L2$-loss
Updating weights



$$\hat{y} = h_w(x) = g_5(in_5) \text{ with } in_5 = w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4$$
$$a_3 = g_3(in_3) \text{ with } in_3 = w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4$$
$$Loss(w) = \frac{1}{2}(y - \hat{y})^2$$

We remind that the weights are updated as :

$$w \leftarrow w - \alpha \times \vec{\nabla} Loss(w)$$

How to compute the gradients ? **Use the chain rule recursively !**

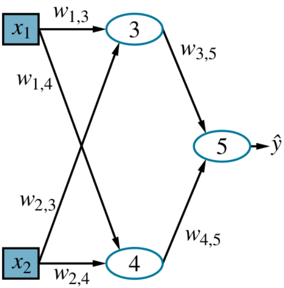
Output layer :

$$\frac{\partial Loss(w)}{\partial w_{3,5}} = \frac{\partial Loss(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial in_5} \cdot \frac{\partial in_5}{\partial w_{3,5}} = -(y - \hat{y}).g_5'(in_5).a_3 = \Delta_5.a_3$$

Hidden layer :

$$\frac{\partial Loss(w)}{\partial w_{1,3}} = \frac{\partial Loss(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial in_5} \cdot \frac{\partial in_5}{\partial a_3} \cdot \frac{\partial a_3}{\partial in_3} \cdot \frac{\partial in_3}{\partial w_{1,3}} = -(y - \hat{y}).g_5'(in_5).w_{3,5}.g_3'(in_3).x_1 = \Delta_3.x_1$$

Error associated to a particular node



In the output node (here 5), we say that the modified error is :

$$\Delta_5 = -(y - \hat{y}) \times g_5'(in_5)$$

The error contributed by a link $j \to k$ is :

$$g_j'(in_j) \times w_{j,k} \times \Delta_k$$

The error of a hidden unit j is the sum of its contribution to the errors in the next layer :

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

Backpropagation

We can now define what's needed for a single iteration of gradient descent :

function $\textsc{Backprop-Iter}(E, \text{Network})$
  for each example $(x, y) \in E$ **do**
    for each node i in the input layer **do**
      $a_i \leftarrow x_i$
    for $\ell = 2$ to N **do**
      for each node j in layer ℓ **do**
        $in_j \leftarrow \sum_i w_{i,j} \times a_i$
        $a_j \leftarrow g_j(in_j)$

...
for each node j in the output layer **do**
  $\Delta_j \leftarrow g'(in_j) \times (y_j - a_j)$
for $\ell = N - 1$ to 1 **do**
  for each node i in layer ℓ **do**
    $\Delta_i \leftarrow g_i'(in_i) \sum_j w_{i,j} \Delta_j$
for each weight $w_{i,j}$ in the network **do**
  $w_{i,j} \leftarrow w_{i,j} - \alpha \times a_i \times \Delta_j$

Gradient descent

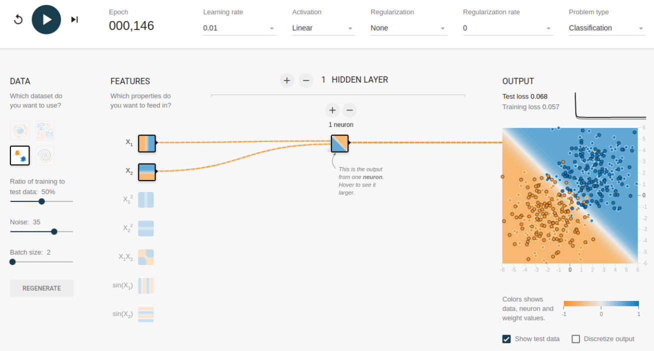
Network $\leftarrow$ neural network with initial weights
while not converged **do**
    BACKPROP-ITER(E, Network)

Remaining questions :

- how to choose the network structure ?
- how to initial the weights ? (critical in deep learning)

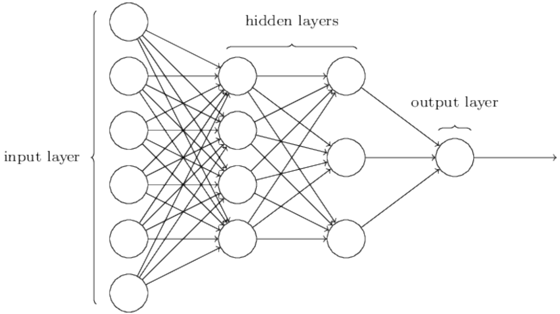
Tensorflow Playground

https://playground.tensorflow.org/



Network structure : MLP

A **multi-layer perceptron** is a network with **fully connected** hidden layers : each unit is connect to all unit of the previous layer.



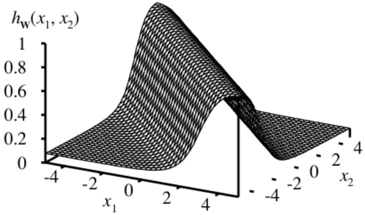
MLP : Universal approximators

*The **Universal Approximation Theorem** states that a neural network with 1 hidden layer can approximate any **continuous** function for inputs within a specific range.*

Caveats :

- The hidden layer might be arbitrary large
- If the function jumps around or has large gaps, we won't be able to approximate it.

Behind the universal approximation theorem



Combining two sigmoids produces a ridge, combining two ridges produce a bump

Beyond the MLP

There are many possible settings for a MLP :

- depth
- width
- connectivy (full/local)
- activation function (sigmoid/relu/tanh)

And there are even more network topologies beyond the MLP

To this day, choosing the right topology remains a difficult process based on experience and trial and error. [1]

1. This process is sometime called the "graduate student descent".

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Section 2

Learning Algorithms

Université Fédérale Toulouse Midi-Pyrénées

Gradient descent

Network $\leftarrow$ neural network with initial weights
while not converged **do**
    $\textsc{Backprop-Iter}(E, \text{Network})$

Problems :

- slow
- overfits
- requires the derivatives

Stochastic gradient descent

Problem :

- gradient computation is costly
  and increases with
  - number of weight
  - number of examples

$$O(|w| \times |E|)$$

Stochastic gradient descent

Problem :

- gradient computation is costly and increases with
  - number of weight
  - number of examples

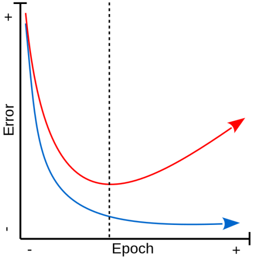
$$O(|w| \times |E|)$$

Solution : select a small subset of example on which to propagate the error

Network $\leftarrow$ neural network with initial weights
while not converged **do**
  $MiniBatch \leftarrow sample(E, k)$
  $\textsc{Backprop-Iter}(MiniBatch,$ Network$)$

This is called **stochastic gradient descent (SGD)** or **mini-batch gradient descent**.

Stopping criterion

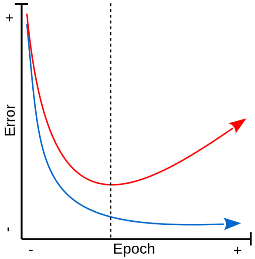


Error on training set (blue) and test set (red)

Problem :

- training tend to overfit the data

Stopping criterion

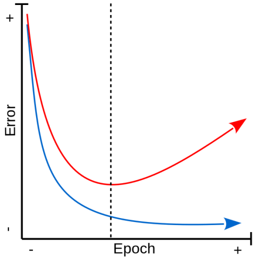


Error on training set (blue) and test set (red)

Problem :

- training tend to overfit the data
- we cannot touch the test data

Stopping criterion



Error on training set (blue) and test set (red)

Problem :

- training tend to overfit the data
- we cannot touch the test data

Solution :

- in the training algorithm, reserve a small portion of the test data for internal **validation**
- do not use it for training
- stop when performance decreases on the validation set

The problem of differentiation

What's f'(z)

Symbolic differentiation (manual or computed) is not always possible/tractable as it can lead to very large computation graphs

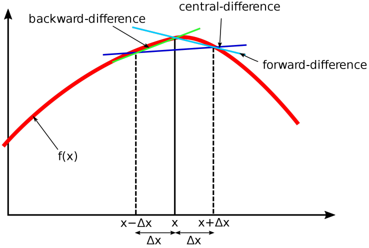
The problem of differentiation

What's f'(z)

Symbolic differentiation (manual or computed) is not always possible/tractable as it can lead to very large computation graphs

However :

- we do **not** need to know f'
- we could compute $f'(z)$ on demand for the current z



Finite differences

Automatic differentiation

In practice, finite difference is too costly as it requires repeated evaluations for all parameters

Machine learning libraries (and in optimization tools in general) use **automatic differentiation (AD)**

Reverse mode AD computes, for a function f and a scalar z :

$$(f(z), f'(z))$$

with low overhead.

Key in enabling neural networks to be trained with complex and arbitrary functions. [2]

2. but beyond the scope of this course.

Automatic differentiation

In practice, finite difference is too costly as it requires repeated evaluations for all parameters

Machine learning libraries (and in optimization tools in general) use **automatic differentiation (AD)**

Reverse mode AD computes, for a function f and a scalar z :

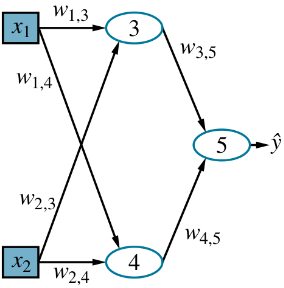
$$(f(z), f'(z))$$

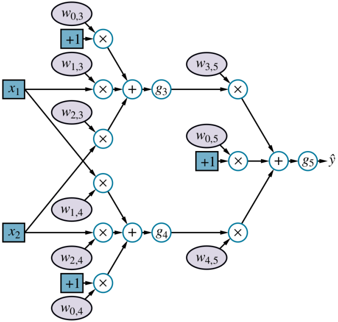
with low overhead.

Key in enabling neural networks to be trained with complex and arbitrary functions. [3]

3. but beyond the scope of this course.

Automatic differentiation : computation graph





Section 3

Convolutional neural networks

Convolutional neural networks

Is there a left turn in the following images ?

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Convolution Kernel

$$input = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}$$

$$kernel = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

$$f_w(x) = \sum_i w_i x_i$$

Kernel (manually defined)

$$kernel = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & 1 \\ -1 & 1 & -1 \end{bmatrix}$$

$$f_w(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}) = 3 \quad f_w(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}) = 2 \quad f_w(\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}) = 1 \quad f_w(\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}) = 2$$

- When $f_w(x) = 3$ our kernel is able to detect a "right turn" in a 3x3 image. [4]
- Our kernel is essentially a neural unit (perceptron).
- The weights could be learned

4. It could be combined with an activation function to get an answer between 0 and 1.

Scaling up to 4x4

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$TL = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad TR = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$BL = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad BR = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Convolutional layer

Key idea : apply the convolutional unit to each 3x3 sub-images.

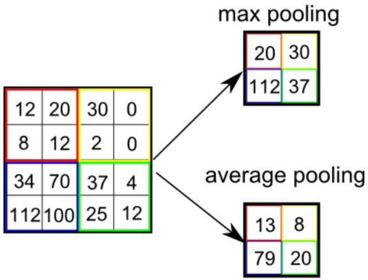
$$\begin{bmatrix} f_w(TL) & f_w(TR) \\ f_w(BL) & f_w(BR) \end{bmatrix} = \begin{bmatrix} 3 & -3 \\ -1 & -4 \end{bmatrix} = \begin{bmatrix} a_{17} & a_{18} \\ a_{19} & a_{20} \end{bmatrix}$$

Interpretation : there is a "right turn" in the top left corner, the rest is garbage.

Key insight :

- in this convolutional layer, we have 4 (2x2) output nodes
- each uses the **same** function, with the **same weights**
- the kernel is trained to detect a feature independently of its location in the source image

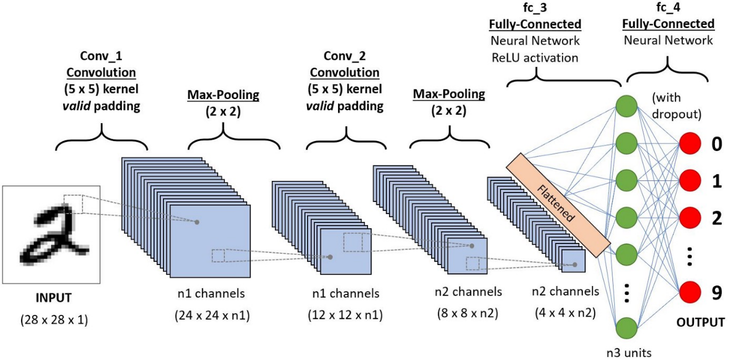
Pooling



2x2 pooling

- reduces dimensionality and variance
- suppresses the noise

A full network



Section 4

Conclusion

Subsection 1

History

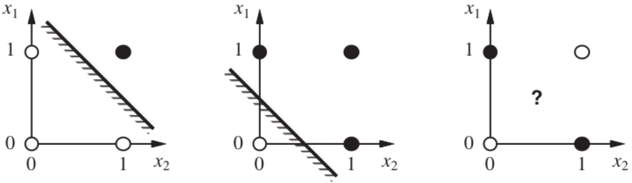
In the old days

- Least-square linear regression
  - Legendre (1805) and Gauss (1809)
  - Initially applied for the prediction of planetary movement

- 1958 : discovery of the **perceptron** and the associated **perceptron learning rule** by F. Rosenblatt

"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself an be conscious of its existence . . . Dr. Frank Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers"

The first AI winter



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

- As noted by Marvin Minsky (Perceptrons)
  - we need to use MLPs even to represent simple nonlinear functions such as the XOR mapping
  - no one on earth had found a viable way to train MLPs good enough to learn such simple functions

Work restarts

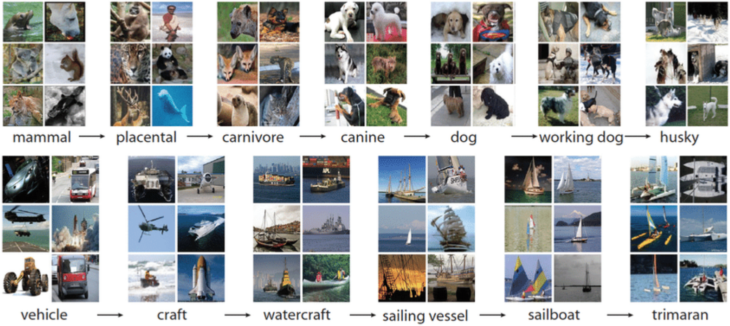
- 1982 J. Hopfield (a reknown physicist) advocates the use of neural networks
- 1986 : Rumelhart and McClelland apply the backpropagation algorithm to train **multi-layer neural networks**
- 1989 : universal approximatin theorem
- 1989 : first uses of the convolutional neural networks
- First success story : hand-written digit recognition [5]

5. LeCun et al. *Backpropagation Applied to Handwritten Zip Code Recognition (1989)*

Second winter (of neural networks)

- deep neural networks remain hard to train (vanishing gradient, weight initiallization)
- Support Vector Machines (SVM) dominate the machine learning world
- neural networks are undesirable (de facto excluded from AI conferences)

2009 ImageNet



15M images / 22 000 categories / 62 000 cats

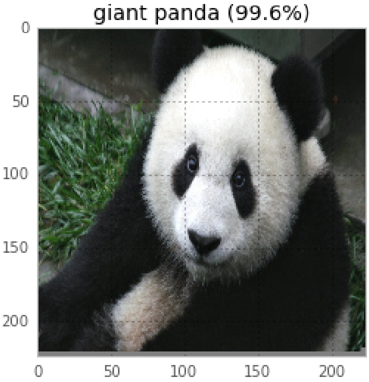
Early 2010s : the advent of deep learning

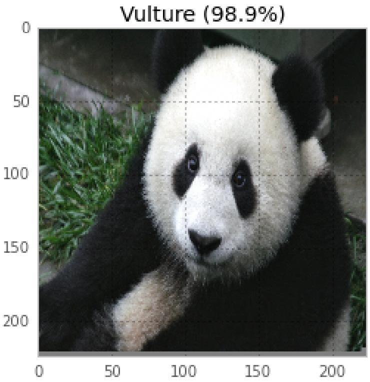
- Several breakthrough in the early 2010s :
  - weight initialization
  - Big Data
  - GPU
  - ReLU

- Deep neural networks become state of the art
  - 2012 : image classification [6]
  - 2015 : Natural language processing (NLP)

6. ImageNet Classification with Deep Convolutional Neural Networks (2012)

Some work left





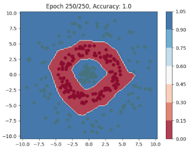
Subsection 2

In practice

How to use neural networks ?

Use existing libraries ! Also contains all elements to develop new machine learning methods (used in research) :

- Scikit-learn
- Keras + Tensorflow




```python
# Création du modèle de réseau de neurones
model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(2, activation='softmax')
])
# Compilation du modèle
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Entraînement sur data avec labels
model.fit(data, labels, epochs=250, verbose=0)
# Prédiction sur data_test
predicted_labels = model.predict(data_test)
```

To go further...

Introduction to Deep Learning : https://fidle.cnrs.fr/ / https://www.youtube.com/@CNRS-FIDLE