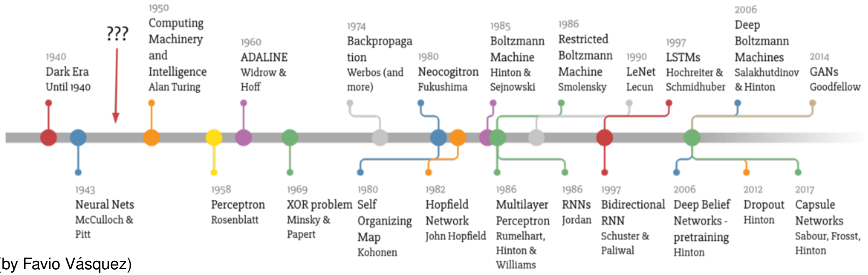
A^3 - Aprendizagem Automática Avançada

An Introduction to Artificial Neural Networks

𝔾. Marques

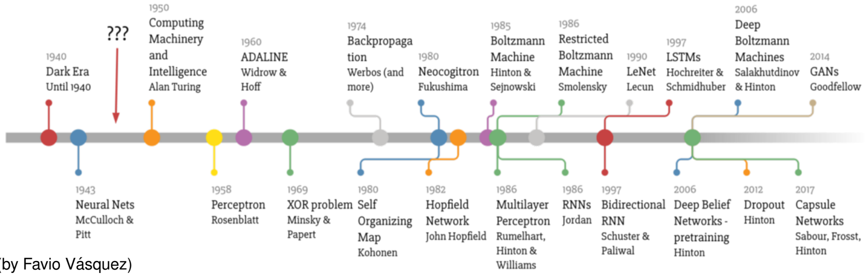
NN an Historical Perspective



(by Favio Vásquez)

1943 Warren McCulloch and Walter Pitts presented a simplified computational model of a biological neuron.

NN an Historical Perspective

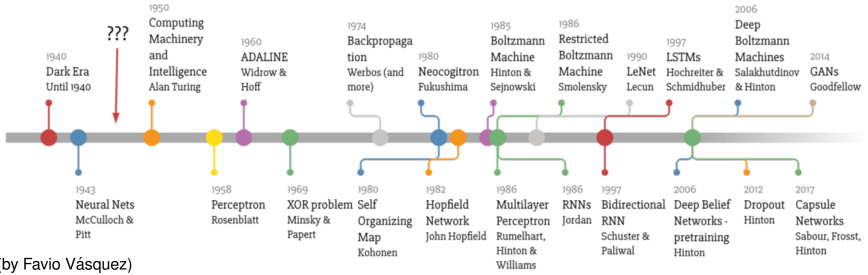


1940 Dark Era Until 1940
???
1950 Computing Machinery and Intelligence Alan Turing
1960 ADALINE Widrow & Hoff
1974 Backpropagation Werbos (and more)
1980 Neocogitron Fukushima
1985 Boltzmann Machine Hinton & Sejnowski
1986 Restricted Boltzmann Machine Smolensky
1990 LeNet Lecun
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2014 GANs Goodfellow
1943 Neural Nets McCulloch & Pitt
1958 Perceptron Rosenblatt
1969 XOR problem Minsky & Papert
1980 Self Organizing Map Kohonen
1982 Hopfield Network John Hopfield
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 RNNs Jordan
1997 Bidirectional RNN Schuster & Paliwal
2006 Deep Belief Networks - pretraining Hinton
2012 Dropout Hinton
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1949 Donald Hebb proposed a learning rule based on neural plasticity.

NN an Historical Perspective

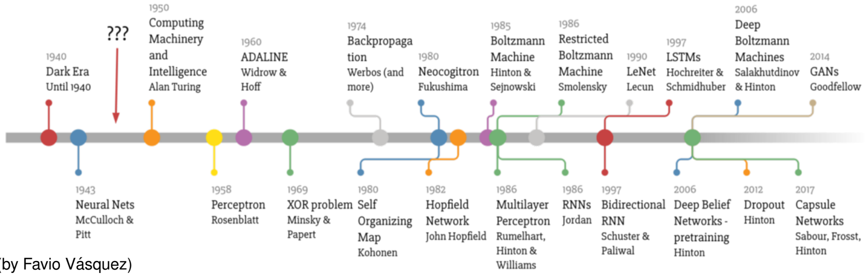


1940 Dark Era Until 1940
???
1950 Computing Machinery and Intelligence Alan Turing
1960 ADALINE Widrow & Hoff
1974 Backpropagation Werbos (and more)
1980 Neocogitron Fukushima
1985 Boltzmann Machine Hinton & Sejnowski
1986 Restricted Boltzmann Machine Smolensky
1990 LeNet Lecun
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2014 GANs Goodfellow
1943 Neural Nets McCulloch & Pitt
1958 Perceptron Rosenblatt
1969 XOR problem Minsky & Papert
1980 Self Organizing Map Kohonen
1982 Hopfield Network John Hopfield
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 RNNs Jordan
1997 Bidirectional RNN Schuster & Paliwal
2006 Deep Belief Networks - pretraining Hinton
2012 Dropout Hinton
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1958 Frank Rosenblatt, inspired on the Hebb learning rule and improvements on the McCulloch Pitts model, proposed the perceptron that was able to learn from data.

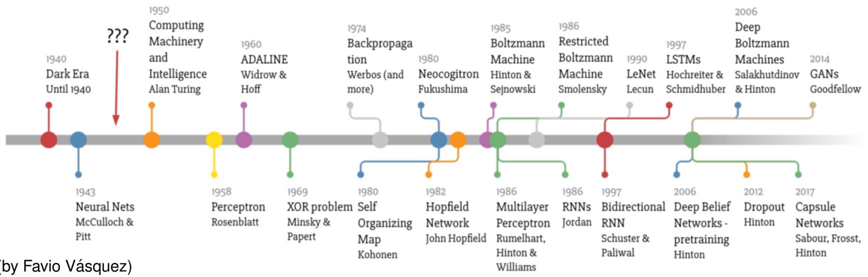
NN an Historical Perspective



(by Favio Vásquez)

1960 Bernard Widrow and Ted Hoff proposed the Adaline (Adaptive Linear Element), an improvement the perceptron learning rule.

NN an Historical Perspective

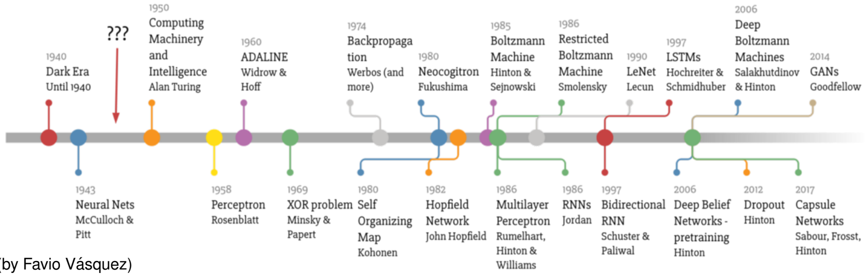


1940 Dark Era Until 1940
???
1950 Computing Machinery and Intelligence Alan Turing
1960 ADALINE Widrow & Hoff
1974 Backpropagation Werbos (and more)
1980 Neocogitron Fukushima
1985 Boltzmann Machine Hinton & Sejnowski
1986 Restricted Boltzmann Machine Smolensky
1990 LeNet Lecun
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2014 GANs Goodfellow
1943 Neural Nets McCulloch & Pitt
1958 Perceptron Rosenblatt
1969 XOR problem Minsky & Papert
1980 Self Organizing Map Kohonen
1982 Hopfield Network John Hopfield
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 RNNs Jordan
1997 Bidirectional RNN Schuster & Paliwal
2006 Deep Belief Networks - pretraining Hinton
2012 Dropout Hinton
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1969 Marvin Minsky and Seymour Papert showed the limitations of the perceptron (unable to solve the XOR problem), and contributed to the “AI winter”.

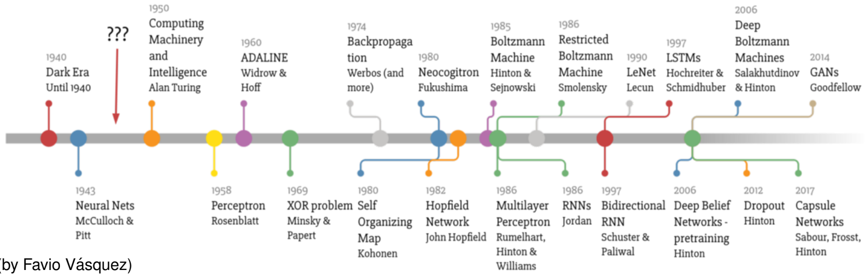
NN an Historical Perspective



(by Favio Vásquez)

1970-80 1st “AI winter”.

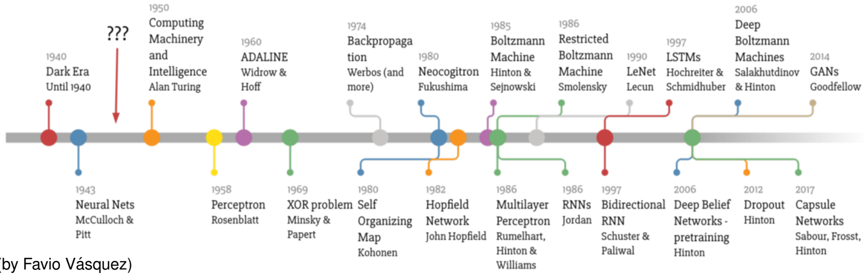
NN an Historical Perspective



(by Favio Vásquez)

1986 David Rumelhart, Geoffrey Hinton and Ronald Williams proposed the Multi-Layer Perceptron (MLPs) and the back propagation learning rule (this rule had already been proposed by Paul Werbos (1974) and others). This marked the end of the “AI winter”.

NN an Historical Perspective

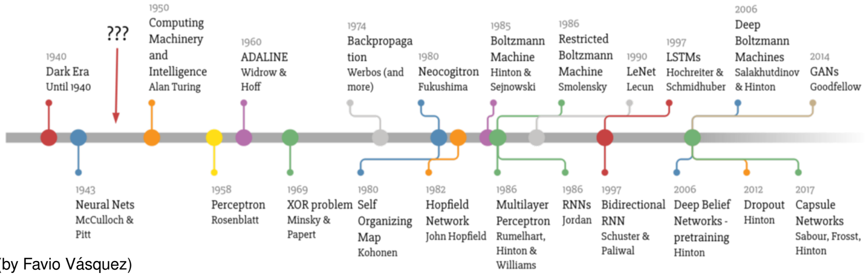


1940 Dark Era Until 1940
???
1950 Computing Machinery and Intelligence Alan Turing
1960 ADALINE Widrow & Hoff
1974 Backpropagation Werbos (and more)
1980 Neocogitron Fukushima
1985 Boltzmann Machine Hinton & Sejnowski
1986 Restricted Boltzmann Machine Smolensky
1990 LeNet Lecun
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2014 GANs Goodfellow
1943 Neural Nets McCulloch & Pitt
1958 Perceptron Rosenblatt
1969 XOR problem Minsky & Papert
1980 Self Organizing Map Kohonen
1982 Hopfield Network John Hopfield
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 RNNs Jordan
1997 Bidirectional RNN Schuster & Paliwal
2006 Deep Belief Networks - pretraining Hinton
2012 Dropout Hinton
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1986 Micheal Jordan proposed Recurrent Neural Networks (RNNs).

NN an Historical Perspective

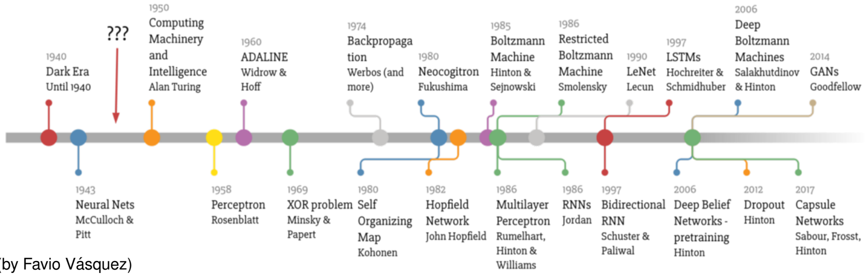


1940 Dark Era Until 1940
???
1943 Neural Nets McCulloch & Pitt
1950 Computing Machinery and Intelligence Alan Turing
1958 Perceptron Rosenblatt
1960 ADALINE Widrow & Hoff
1969 XOR problem Minsky & Papert
1974 Backpropagation Werbos (and more)
1980 Self Organizing Map Kohonen
1980 Neocogitron Fukushima
1982 Hopfield Network John Hopfield
1985 Boltzmann Machine Hinton & Sejnowski
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 Restricted Boltzmann Machine Smolensky
1986 RNNs Jordan
1990 LeNet Lecun
1997 Bidirectional RNN Schuster & Paliwal
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Belief Networks - pretraining Hinton
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2012 Dropout Hinton
2014 GANs Goodfellow
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1989 Yann LeCun proposed a Convolutional Neural Network (CNNs) for handwritten digit recognition.

NN an Historical Perspective

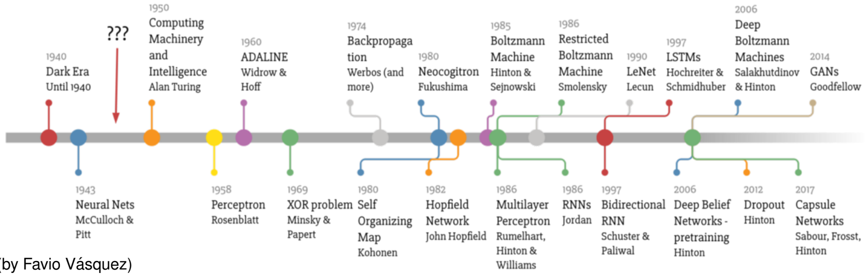


1940 Dark Era Until 1940
???
1950 Computing Machinery and Intelligence Alan Turing
1960 ADALINE Widrow & Hoff
1974 Backpropagation Werbos (and more)
1980 Neocogitron Fukushima
1985 Boltzmann Machine Hinton & Sejnowski
1986 Restricted Boltzmann Machine Smolensky
1990 LeNet Lecun
1997 LSTMs Hochreiter & Schmidhuber
2006 Deep Boltzmann Machines Salakhutdinov & Hinton
2014 GANs Goodfellow
1943 Neural Nets McCulloch & Pitt
1958 Perceptron Rosenblatt
1969 XOR problem Minsky & Papert
1980 Self Organizing Map Kohonen
1982 Hopfield Network John Hopfield
1986 Multilayer Perceptron Rumelhart, Hinton & Williams
1986 RNNs Jordan
1997 Bidirectional RNN Schuster & Paliwal
2006 Deep Belief Networks - pretraining Hinton
2012 Dropout Hinton
2017 Capsule Networks Sabour, Frosst, Hinton

(by Favio Vásquez)

1997 Sepp Hochreiter and Jürgen Schmidhuber proposed the LSTM RNN (Long Short-Term Memory).

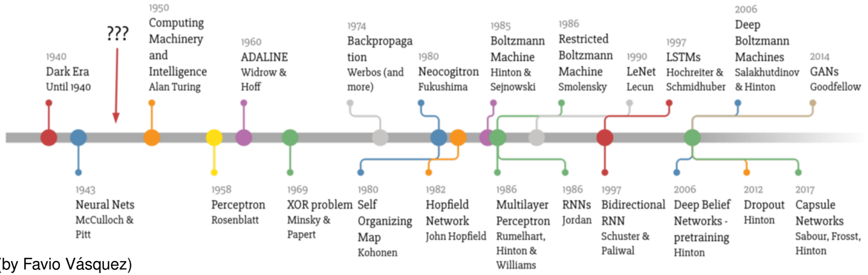
NN an Historical Perspective



(by Favio Vásquez)

1990s-12 2nd “AI winter”.

NN an Historical Perspective



(by Favio Vásquez)

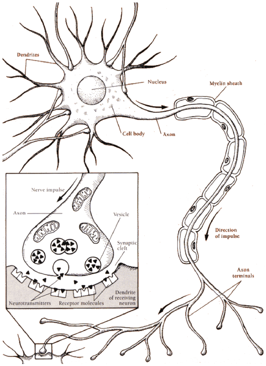
2012 Geoffrey Hinton ImageNet Deep CNNs. Beginning of the “AI Spring”.

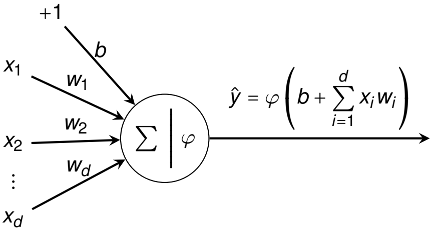
NN an Historical Perspective

Why Deep Learning? Why Now?

- Hardware
  CPUs, GPUs and TPUs, and computer clusters.
- Data
  Internet allows the collections and distribution of large (huge) datasets.
- Algorithmic Advances
  Better initializations, activations and optimizations
- Deep Learning automates the step of *feature engineering*.

The Perceptron

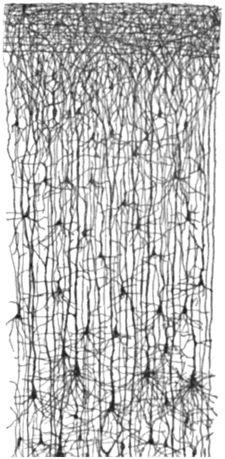




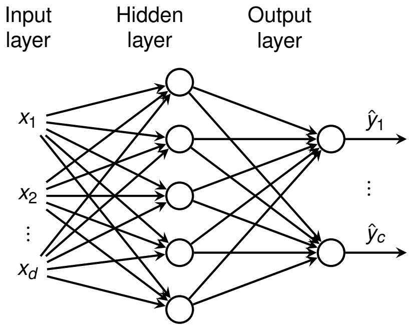
- In the perceptron, the activation function is a step function.
- In log-linear classifiers, φ is either a sigmoid or hyperbolic tangent.
- In neural networks , φ can also be the previous two functions, and also many others (not biologically inspired).

NOTE: The perceptron and log-linear classifiers have several limitations. These are overcome by multi-layer perceptrons, which are basically stacked layers of log-linear classifiers (also inspired by the connections between neurons in the brain).

Multi-Layer Perceptron

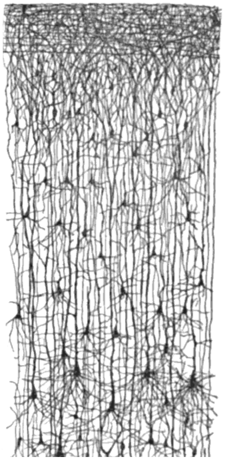


- Simple, three layer MLP networks:

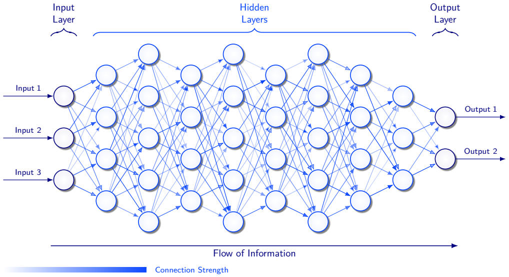


- Each circle represents a non-linear, perceptron-like operation.
- Such networks can solve the XOR problem and other non-linear ones.

Multi-Layer Perceptron



- Deep neural network:



- Capable of solving highly complex problems.

Training MLPs
Backpropagation Algorithm

- A MLP network consists of one input layer, one or more hidden layers, and output layer. In classification the number of outputs, $\hat{y}_i$ corresponds to the number of classes.

ALGORITHM:

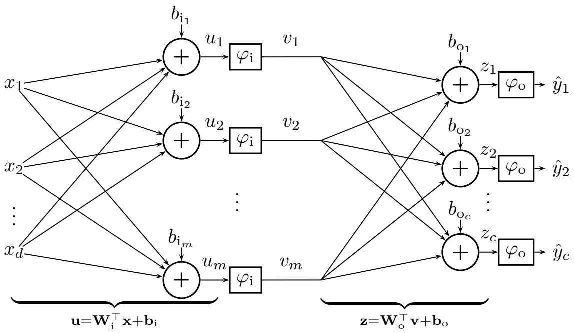
- FORWARD PASS:
  Each training instance, $\mathbf{x}[n]$ is fed through the network to produce an output $\hat{\mathbf{y}}[n]$.

- BACKWARD PASS:
  The error for each instance is computed and the gradient is fed back through the network. A typical error function is the squared difference between the output and desired response for that instance: $\mathbf{e}[n] = \left\|\mathbf{y}[n] - \hat{\mathbf{y}}[n]\right\| = \sum_i (y_i - \hat{y}_i)^2$
  It is necessary to calculate the gradient for all the weights of each neuron in the network and adapt them through gradient descent.

Training MLPs

Backpropagation Algorithm

EXAMPLE: MLP network with *d* inputs, *m* hidden units and *c* outputs.



- Error function: Mean Squared Error (MSE) $\mathcal{E} = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{y}_n - \hat{\mathbf{y}}_n\|^2 = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{e}_n\|^2$
- Weight adaptation through steepest gradient descent (*i.e.* calculate $\frac{\partial\mathcal{E}}{\partial\mathbf{W}_i}, \frac{\partial\mathcal{E}}{\partial\mathbf{b}_i}, \frac{\partial\mathcal{E}}{\partial\mathbf{W}_o}, \frac{\partial\mathcal{E}}{\partial\mathbf{b}_o}$).

Training MLPs

Backpropagation Algorithm

EXAMPLE: MLP network with d inputs, m hidden units and c outputs.

- Weights and parameter dimensions:
  - $\mathbf{y}$, $\hat{\mathbf{y}}$, $\mathbf{z}$ and $\mathbf{b}_o$ are c dimensional vectors
  - $\mathbf{v}$, $\mathbf{u}$ and $\mathbf{b}_i$ are m dimensional vectors
  - input training vector $\mathbf{x}$ is a d dimensional vector
  - $\mathbf{W}_i$ is a $d \times m$ dimensional matrix
  - $\mathbf{W}_o$ is a $m \times c$ dimensional matrix

Gradients for output biases $\mathbf{b}_\mathrm{o}$:

$$\frac{\partial \|\mathbf{e}\|^2}{\partial \mathbf{b}_\mathrm{o}} = \left[\frac{\partial \|\mathbf{e}\|^2}{\partial b_{\mathrm{o}_1}}, \ldots, \frac{\partial \|\mathbf{e}\|^2}{\partial b_{\mathrm{o}_c}} \right]^\mathsf{T}$$

$$\frac{\partial \|\mathbf{e}\|^2}{\partial b_{\mathrm{o}_j}} = \frac{\partial \|\mathbf{e}\|^2}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j} \frac{\partial z_j}{\partial b_{\mathrm{o}_j}} = -2(y_j - \hat{y}_j)\varphi_\mathrm{o}'(z_j)$$

- $$\frac{\partial \|\mathbf{e}\|^2}{\partial \hat{y}_j} = -2e_j = -2(y_j - \hat{y}_j)$$
- $$\frac{\partial \hat{y}_j}{\partial z_j} = \varphi_\mathrm{o}'(z_j) = \begin{cases} \hat{y}_j - \hat{y}_j^2 & \text{for sigmoids} \\ 1 - \hat{y}_j^2 & \text{for hyperbolic tangents} \end{cases}$$
- $$\frac{\partial z_j}{\partial b_{\mathrm{o}_j}} = 1$$

Training MLPs

Backpropagation Algorithm

EXAMPLE: MLP network with *d* inputs, *m* hidden units and *c* outputs.

- Weights and parameter dimensions:
  - $\mathbf{y}$, $\hat{\mathbf{y}}$, $\mathbf{z}$ and $\mathbf{b}_o$ are c dimensional vectors
  - $\mathbf{v}$, $\mathbf{u}$ and $\mathbf{b}_i$ are m dimensional vectors
  - input training vector $\mathbf{x}$ is a d dimensional vector
  - $\mathbf{W}_i$ is a $d \times m$ dimensional matrix
  - $\mathbf{W}_o$ is a $m \times c$ dimensional matrix

Gradients for output weights $\mathbf{W}_o$.

$\dfrac{\partial \|\mathbf{e}\|^2}{\partial \mathbf{W}_\text{o}}$ is a $m \times c$ matrix.

$$\begin{aligned}\frac{\partial \|\mathbf{e}\|^2}{\partial w_{\text{o}_{ij}}} &= \frac{\partial \|\mathbf{e}\|^2}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j} \frac{\partial z_j}{\partial w_{\text{o}_{ij}}} \\ &= -2(y_j - \hat{y}_j)\varphi'_\text{o}(z_j) v_i\end{aligned}$$

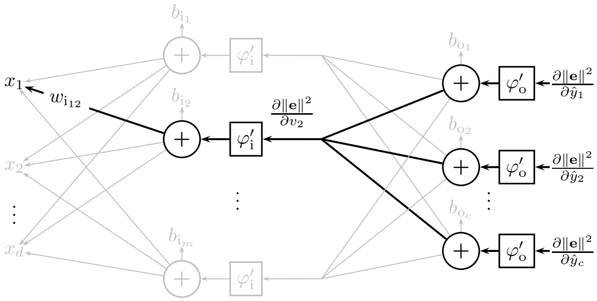
- $\dfrac{\partial \|\mathbf{e}\|^2}{\partial \hat{y}_j} = -2e_j = -2(y_j - \hat{y}_j)$
- $\dfrac{\partial \hat{y}_j}{\partial z_j} = \varphi'_\text{o}(z_j) = \begin{cases} \hat{y}_j - \hat{y}_j^2 & \text{for sigmoids} \\ 1 - \hat{y}_j^2 & \text{for hyperbolic tangents} \end{cases}$
- $\dfrac{\partial z_j}{\partial w_{\text{o}_{ij}}} = v_i$

Training MLPs

Backpropagation Algorithm

EXAMPLE: MLP network with *d* inputs, *m* hidden units and *c* outputs.

Gradients for intput weights $\mathbf{W}_i$ and $\mathbf{b}_i$:



b_{i_1}
b_{i_2}
b_{i_m}
x_1
x_2
x_d
$w_{i_{12}}$
φ'_i
$\frac{\partial\|\mathbf{e}\|^2}{\partial v_2}$
b_{o_1}
b_{o_2}
b_{o_c}
φ'_o
$\frac{\partial\|\mathbf{e}\|^2}{\partial \hat{y}_1}$
$\frac{\partial\|\mathbf{e}\|^2}{\partial \hat{y}_2}$
$\frac{\partial\|\mathbf{e}\|^2}{\partial \hat{y}_c}$

- All the paths from the output to the parameter to adapt, w_{12}, must be considered:
- $$\frac{\partial\|\mathbf{e}\|^2}{w_{i_{12}}} = \sum_{k=1}^{c} \frac{\partial\|\mathbf{e}\|^2}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_k} \frac{\partial z_k}{\partial v_2} \frac{\partial v_2}{\partial u_2} \frac{\partial u_2}{\partial w_{i_{12}}}$$

Training MLPs

Backpropagation Algorithm

EXAMPLE: MLP network with *d* inputs, *m* hidden units and *c* outputs.

Gradients for input biases $\mathbf{b}_\mathrm{i}$:

$$\frac{\partial\|\mathbf{e}\|^2}{\partial\mathbf{b}_\mathrm{i}} = \left[\frac{\partial\|\mathbf{e}\|^2}{\partial b_{\mathrm{i}_1}}, \dots, \frac{\partial\|\mathbf{e}\|^2}{\partial b_{\mathrm{i}_m}}\right]^\mathsf{T}$$

$$\frac{\partial\|\mathbf{e}\|^2}{b_{\mathrm{i}_j}} = -2\frac{\partial v_j}{\partial u_j}\frac{\partial u_j}{\partial b_{\mathrm{i}_j}}\sum_{k=1}^{c} e_k\varphi_\mathrm{o}'(z_k)\frac{\partial z_k}{\partial v_j}$$

$$= -2\varphi_\mathrm{i}'(u_j)\sum_{k=1}^{c} w_{\mathrm{o}_{jk}} e_k\varphi_\mathrm{o}'(z_k)$$

Gradients for input weights $\mathbf{W}_i$:

$$\frac{\partial\|\mathbf{e}\|^2}{w_{\mathrm{i}_{ij}}} = -2x_i\frac{\partial v_j}{\partial u_j}\frac{\partial u_j}{\partial w_{\mathrm{i}_{ij}}}\sum_{k=1}^{c} e_k\varphi_\mathrm{o}'(z_k)\frac{\partial z_k}{\partial v_j}$$

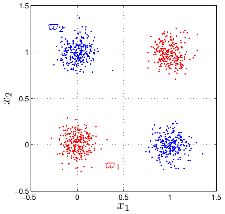
$$= -2x_i\varphi_\mathrm{i}'(u_j)\sum_{k=1}^{c} w_{\mathrm{o}_{jk}} e_k\varphi_\mathrm{o}'(z_k)$$

- $\frac{\partial\|\mathbf{e}\|^2}{\partial\hat{y}_k} = -2e_k = -2(y_k - \hat{y}_k)$
- $\frac{\partial\hat{y}_k}{\partial z_k} = \varphi_\mathrm{o}'(z_k) = \begin{cases} \hat{y}_k - \hat{y}_k^2 & \text{sigmoids} \\ 1 - \hat{y}_k^2 & \text{hyper. tang.} \end{cases}$
- $\frac{\partial z_k}{\partial v_j} = w_{\mathrm{o}_{jk}}$
- $\frac{\partial v_j}{\partial u_j} = \varphi_\mathrm{i}'(u_j) = \begin{cases} v_j - v_j^2 & \text{sigmoids} \\ 1 - v_j^2 & \text{hyper. tang.} \end{cases}$
- $\frac{\partial u_j}{\partial b_{\mathrm{i}_j}} = 1$
- $\frac{\partial u_j}{\partial w_{\mathrm{i}_{ij}}} = x_i$

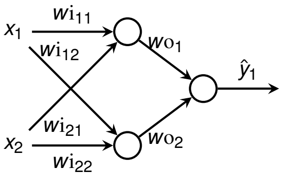
Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)



Consider the following NN with activation functions $\varphi(x) = \tanh(x)$.



NOTE: There is also the biases b_1 and b_2 of the hidden nodes (admit that the output node has no bias).

- Output of the first hidden node: $v_1 = \tanh\left(w_{i_{11}} x_1 + w_{i_{21}} x_2 + b_1\right)$
- Output of the second hidden node: $v_2 = \tanh\left(w_{i_{12}} x_1 + w_{i_{22}} x_2 + b_2\right)$
- Output of the network: $\hat{y} = \tanh\left(v_1 w_{o_1} + v_2 w_{o_2}\right)$

Objective: Write a Python program that adapts the parameters of the network by minimizing the MSE through steepest gradient descent.

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

PROGRAMING CONSIDERATIONS:

1. Initializations: weights (*e.g* Gaussian), adaptation step η, number of iterations.
2. Adaptation - for the whole training set :
   - Calculate error
   - Calculate gradients
   - Adapt weights
3. Repeat until maximum number of iterations is reached.

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- MSE function: $\mathcal{E} = \frac{1}{N}\sum_{n=0}^{N-1}(y[n]-\hat{y}[n])^2 = \frac{1}{N}\sum_{n=0}^{N-1}\left(y[n]-\tanh\left(w_{o_1}v_1[n]+w_{o_2}v_2[n]\right)\right)^2$
- Network operations:
  - Hidden layer:

$$\begin{aligned} u_1 &= w_{i_{11}}x_1 + w_{i_{21}}x_2 + b_1 & u_2 &= w_{i_{12}}x_1 + w_{i_{22}}x_2 + b_1 \\ v_1 &= \tanh\left(w_{i_{11}}x_1 + w_{i_{21}}x_2 + b_1\right) & v_2 &= \tanh\left(w_{i_{12}}x_1 + w_{i_{22}}x_2 + b_1\right) \end{aligned}$$

  - Output:

$$\begin{aligned} z &= w_{o_1}v_1 + w_{o_2}v_2 \\ \hat{y} &= \tanh(z) = \tanh\left(w_{o_1}v_1 + w_{o_2}v_2\right) \\ &= \tanh\left(w_{o_1}\tanh\left(w_{i_{11}}x_1 + w_{i_{21}}x_2 + b_1\right) + w_{o_2}\tanh\left(w_{i_{12}}x_1 + w_{i_{22}}x_2 + b_1\right)\right) \end{aligned}$$

- Matrix notation:

$$\mathbf{x}=\begin{bmatrix}x_1\\x_2\end{bmatrix},\ \mathbf{b}=\begin{bmatrix}b_1\\b_2\end{bmatrix},\ \mathbf{W}_\mathrm{i}=\begin{bmatrix}w_{i_{11}} & w_{i_{12}}\\ w_{i_{21}} & w_{i_{22}}\end{bmatrix},\ \mathbf{u}=\begin{bmatrix}u_1\\u_2\end{bmatrix}=\mathbf{W}_\mathrm{i}^\top\mathbf{x}+\mathbf{b},\ \mathbf{v}=\begin{bmatrix}v_1\\v_2\end{bmatrix}=\tanh(\mathbf{u}),\ \mathbf{w}_\mathrm{o}=\begin{bmatrix}w_{o_1}\\w_{o_2}\end{bmatrix}$$

$$z = \mathbf{w}_\mathrm{o}^\top\mathbf{v}$$

$$\hat{y} = \tanh\left(\mathbf{w}_\mathrm{o}^\top\mathbf{v}\right) = \tanh\left(\mathbf{w}_\mathrm{o}^\top\tanh\left(\mathbf{W}_\mathrm{i}^\top\mathbf{x}+\mathbf{b}\right)\right)$$

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradients of output weights $\mathbf{w}_\mathrm{o}$:

$$\frac{\partial \mathcal{E}}{\partial w_{\mathrm{o}_1}} = \frac{-2}{N} \sum_{n=0}^{N-1} \left(y[n] - \hat{y}[n]\right) \frac{\partial \hat{y}[n]}{\partial z[n]} \frac{\partial z[n]}{\partial w_{\mathrm{o}_1}} = \frac{-2}{N} \sum_{n=0}^{N-1} e[n] \left(1 - \hat{y}[n]^2\right) v_1[n]$$

$$\frac{\partial \mathcal{E}}{\partial w_{\mathrm{o}_2}} = \frac{-2}{N} \sum_{n=0}^{N-1} \left(y[n] - \hat{y}[n]\right) \frac{\partial \hat{y}[n]}{\partial z[n]} \frac{\partial z[n]}{\partial w_{\mathrm{o}_2}} = \frac{-2}{N} \sum_{n=0}^{N-1} e[n] \left(1 - \hat{y}[n]^2\right) v_2[n]$$

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_{\text{o}}} = -\frac{2}{N}\sum_{n} e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases **b**:

$$
\begin{aligned}
\frac{\partial \mathcal{E}}{\partial b_1} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_1[n]}\frac{\partial v_1[n]}{\partial u_1[n]}\frac{\partial u_1[n]}{\partial b_1}\\
&= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right) w_{\text{o}_1}\left(1-v_1[n]^2\right)\times 1\\
\frac{\partial \mathcal{E}}{\partial b_2} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_2[n]}\frac{\partial v_2[n]}{\partial u_2[n]}\frac{\partial u_2[n]}{\partial b_2}\\
&= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right) w_{\text{o}_2}\left(1-v_2[n]^2\right)\times 1
\end{aligned}
$$

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\mathbf{W}_\mathrm{i}$:

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{\mathrm{i}_{11}}} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_1[n]}\frac{\partial v_1[n]}{\partial u_1[n]}\frac{\partial u_1[n]}{\partial w_{\mathrm{i}_{11}}}\\ &= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right)w_{\mathrm{o}_1}\left(1-v_1[n]^2\right)x_1[n]\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{\mathrm{i}_{21}}} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_1[n]}\frac{\partial v_1[n]}{\partial u_1[n]}\frac{\partial u_1[n]}{\partial w_{\mathrm{i}_{21}}}\\ &= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right)w_{\mathrm{o}_1}\left(1-v_1[n]^2\right)x_2[n]\end{aligned}$$

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\mathbf{W}_\mathrm{i}$:

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{\mathrm{i}_{12}}} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_2[n]}\frac{\partial v_2[n]}{\partial u_2[n]}\frac{\partial u_2[n]}{\partial w_{\mathrm{i}_{12}}}\\ &= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right)w_{\mathrm{o}_2}\left(1-v_2[n]^2\right)x_1[n]\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{\mathrm{i}_{22}}} &= \frac{-2}{N}\sum_{n=0}^{N-1}\left(y[n]-\hat{y}[n]\right)\frac{\partial \hat{y}[n]}{\partial z[n]}\frac{\partial z[n]}{\partial v_2[n]}\frac{\partial v_2[n]}{\partial u_2[n]}\frac{\partial u_2[n]}{\partial w_{\mathrm{i}_{22}}}\\ &= \frac{-2}{N}\sum_{n=0}^{N-1} e[n]\left(1-\hat{y}[n]^2\right)w_{\mathrm{o}_2}\left(1-v_2[n]^2\right)x_2[n]\end{aligned}$$

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\frac{\partial \mathcal{E}}{\partial \mathbf{W}_\mathrm{i}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{x}[n]\left(\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)\right)^\mathsf{T}$

PYTHON PSEUDO-CODE:

Consider that the data is in a `Numpy` array, `X`, of dimensions $2\times N$, and that the desired output (class labels) are store in an array `y` of N elements. Furthermore the network weights $\mathbf{w}_\mathrm{o}$, $\mathbf{b}$, $\mathbf{W}_\mathrm{i}$ are in arrays `wo,b,Wi` of dimensions 2×1, 2×1 and 2×2, respectively.

- Forward Pass:
- Error:

```
U=np.dot(Wi.T,X)+b      2× N matrix
V=np.tanh(U)            2× N matrix
Z=np.dot(wo.T,V)        1× N matrix
Yh=np.tanh(Z)           1× N matrix
E=Y-Yh                  1× N matrix
```

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_{\mathrm{o}}} = -\frac{2}{N} \sum_n e[n] \left(1 - \hat{y}[n]^2\right) \mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N} \sum_n e[n] \left(1 - \hat{y}[n]^2\right) \mathbf{w}_{\mathrm{o}} \left(1 - \mathbf{v}[n]^2\right)$
- Gradient for input weights $\frac{\partial \mathcal{E}}{\partial \mathbf{W}_{\mathrm{i}}} = -\frac{2}{N} \sum_n e[n] \left(1 - \hat{y}[n]^2\right) \mathbf{x}[n] \left(\mathbf{w}_{\mathrm{o}} \left(1 - \mathbf{v}[n]^2\right)\right)^{\mathsf{T}}$

PYTHON PSEUDO-CODE:
Consider that the data is in a `Numpy` array, `X`, of dimensions $2\times N$, and that the desired output (class labels) are store in an array `y` of N elements. Furthermore the network weights $\mathbf{w}_{\mathrm{o}}$, $\mathbf{b}$, $\mathbf{W}_{\mathrm{i}}$ are in arrays `wo,b,Wi` of dimensions 2×1, 2×1 and 2×2, respectively.

- Gradients:

```
dz=-2*E*(1-Yh**2)            1× N matrix
dwo=np.dot(V,dz.T)           2× 1 matrix
dgu=(1-V**2)*np.dot(wo,dz)   2× N matrix
db=np.sum(dgu,axis=1)        2× 1 matrix
dWi=np.dot(X,dgu.T)          2× 2 matrix
```

Training MLPs

Example: XOR problem

Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\frac{\partial \mathcal{E}}{\partial \mathbf{W}_\mathrm{i}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{x}[n]\left(\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)\right)^\top$

PYTHON PSEUDO-CODE:

Consider that the data is in a `Numpy` array, `X`, of dimensions $2\times N$, and that the desired output (class labels) are store in an array `y` of N elements. Furthermore the network weights $\mathbf{w}_\mathrm{o}$, $\mathbf{b}$, $\mathbf{W}_\mathrm{i}$ are in arrays `wo,b,Wi` of dimensions 2×1, 2×1 and 2×2, respectively.

- Adaptations:

```
wo=wo-eta*dwo     2× 1 matrix
b=b-eta*db        2× 1 matrix
Wi=Wi-eta*dWi     2× 2 matrix
```

Training MLPs

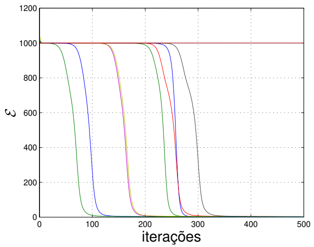
Example: XOR problem

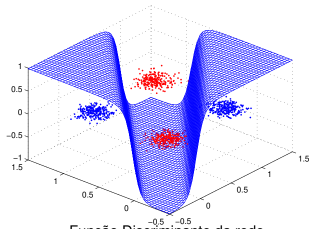
Binary classification problem (2D data with 500 pts/class)

- Gradient for output weights $\dfrac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\dfrac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\dfrac{\partial \mathcal{E}}{\partial \mathbf{W}_\mathrm{i}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{x}[n]\left(\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)\right)^\top$

RESULTS:

- Different initialization lead to different results.
- Important to visualize the error.





Função Discriminante da rede

Training MLPs

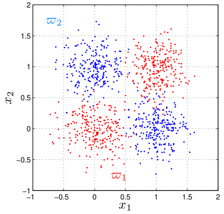
Example: XOR problem

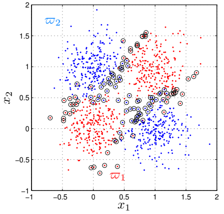
Binary classification problem (2D data with 500 pts/class)

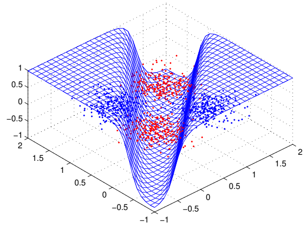
- Gradient for output weights $\frac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\frac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\frac{\partial \mathcal{E}}{\partial \mathbf{W}_\mathrm{i}} = -\frac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{x}[n]\left(\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)\right)^\top$

RESULTS:

- Noisy data. 2 hidden units MLP network.







Training MLPs

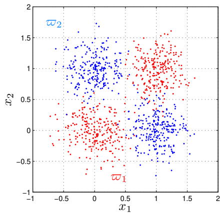
Example: XOR problem

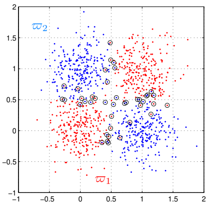
Binary classification problem (2D data with 500 pts/class)

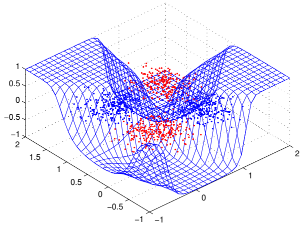
- Gradient for output weights $\dfrac{\partial \mathcal{E}}{\partial \mathbf{w}_\mathrm{o}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{v}[n]$
- Gradient for input biases $\dfrac{\partial \mathcal{E}}{\partial \mathbf{b}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)$
- Gradient for input weights $\dfrac{\partial \mathcal{E}}{\partial \mathbf{W}_\mathrm{i}} = -\dfrac{2}{N}\sum_n e[n]\left(1-\hat{y}[n]^2\right)\mathbf{x}[n]\left(\mathbf{w}_\mathrm{o}\left(1-\mathbf{v}[n]^2\right)\right)^\top$

RESULTS:

- Noisy data. 5 hidden units MLP network.







Training MLPs

Extra Considerations

In training MLPs there several points one needs to take in consideration. So far, we have seen a simple MLP, on hidden layer network, trained by minimizing the Mean Squared Error through steepest descend. The state of the art has evolved to a point where now there are many decisions to make when training a NN.

- There are many error functions one can choose from. The same also applies to activation functions, optimizations methods and weight initializations.
- Another important and often overlooked aspect is data normalization. Most of the times, it is essential for a successful training process to normalize the data (there are many ways to do this in `sklearn` and `tensorflow`).
- One more factor to take in account when adapting a MLP is to avoid over-fitting the training data. MLPs can easily do this through ℓ_1 and ℓ_2 regularization, but many other techniques should also be considered. such as *Early Stopping*, *Batch Normalization*, *Dropout Layer* and *Data Augmentation* (these will be covered later on in the course).
- Finally, examine our training and testing methodology. Note that for large networks and datasets, sometimes one has to revert to simple train/test schemes (or train/validation/test split of the data), since it is not possible to implement cross-validation or other more computational demanding approaches.

Neural Networks with Tensorflow-Keras

NN with Tensorflow-Keras

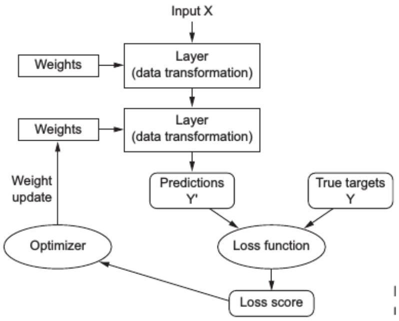
- Tensorflow is an open source machine learning library developed by Google Brain Team, specially suited for deep neural networks. First released in 2015, it has since then undergone major changes, namely incorporating the Keras high level API.

  Check out:
  - TensorFlow
  - Keras
- Online courses:
  - Deep Learning AI
  - Coursera
  - Udacity
  - Standford Online
  - PyImageSearch
  - ...

NN with Tensorflow-Keras

Constructing a neural network with `tf-keras` revolves around the following programing blocks:

- Defining the *layers* and combining them into a *network* (or *model*).
- Choosing a *cost* function to be minimized by the network
- Choosing an *optimizer* for the weight adaptation.
- Training the network.



by François Chollet

NOTE: there are other processing blocks in machine learning project using neural networks, namely the data preparation, the evaluation, deploying and monitoring the developed system .

NN with Tensorflow-Keras

Building Neural Networks:

A *fully connected* neural network is a stack of layer. First one needs to create a model, then define the layers and their number of units and their *activation functions*, and finally compile the . Here is an example of the TF-Keras implementation of a 3 layer MLP network, for 20-dimensional input data:

```
import tensorflow as tf
import tensorflow.keras as keras
nn=keras.Sequential() or:  nn=keras.models.Sequential()
nn.add(keras.layers.Dense(units=32, \
input_shape=(20,),activation="tanh"))
nn.add(keras.layers.Dense(10,activation="softmax"))
nn.compile(optimizer="adam", \
loss="categorical_crossentropy",metrics=["accuracy"])
```

NOTE: The above code creates a model by adding two layers with the function `.add()`. For the first layer, the input data shape has to be specified. Like in many other cases in Python, this is not the only way to define a model. For example, the next lines of code define the same model as the above.

```
nn=keras.Sequential([keras.layers.Dense(32,input_shape=X.shape[0], \
activation="tanh") \
keras.layers.Dense(10,activation="softmax")])
```

NN with Tensorflow-Keras

Building Neural Networks:

A *fully connected* neural network is a stack of layer. First one needs to create a model, then define the layers and their number of units and their *activation functions*, and finally compile the . Here is an example of the TF-Keras implementation of a 3 layer MLP network, for 20-dimensional input data:

```
import tensorflow as tf
import tensorflow.keras as keras
nn=keras.Sequential() or:  nn=keras.models.Sequential()
nn.add(keras.layers.Dense(units=32, \
input_shape=(20,),activation="tanh"))
nn.add(keras.layers.Dense(10,activation="softmax"))
nn.compile(optimizer="adam", \
loss="categorical_crossentropy",metrics=["accuracy"])
```

NOTE: To check the configuration of a given model, use the `.summary()` method:

```
nn.summary()
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
dense_1 (Dense) (None, 32) 672
_________________________________________________________________
dense_2 (Dense) (None, 10) 330
=================================================================
Total params:  1,002
Trainable params:  1,002
```

NN with Tensorflow-Keras

Building Neural Networks:

A *fully connected* neural network is a stack of layer. First one needs to create a model, then define the layers and their number of units and their *activation functions*, and finally compile the . Here is an example of the TF-Keras implementation of a 3 layer MLP network, for 20-dimensional input data:

```
import tensorflow as tf
import tensorflow.keras as keras
nn=keras.Sequential() or:  nn=keras.models.Sequential()
nn.add(keras.layers.Dense(units=32, \
input_shape=(20,),activation="tanh"))
nn.add(keras.layers.Dense(10,activation="softmax"))
nn.compile(optimizer="adam", \
loss="categorical_crossentropy",metrics=["accuracy"])
```

Models:

The first step is to create a model. Keras has available two types of models: the *sequential models* and models defined with *Keras functional API*.

- Sequential models consist in a stack of layers. This model is enough for most applications and will be the preferred choice in this course.
- The Keras functional API allows to define more complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

NN with Tensorflow-Keras

Building Neural Networks - Layers:

- The previous code is a stack of 3 `Dense` layers. `Dense` layers are fully connected layers, in the process of building neural network, one has to decide **how many layers** to use and **how many units** in each layer.
- The more layers, and the more units per layer one uses, the more the network is capable of discriminating complex problems, but at the same time, the networks become more susceptible of **overfitting** the data.
- `Dense` layers perform a linear (affine) transformation followed by a non-linearity (the activation function): $\mathbf{z} = \varphi\left(\mathbf{W}\mathbf{v} + \mathbf{b}\right)$.
  Without the activation functions, a stack of `Dense` layers would simply result in a linear transformation: $\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$

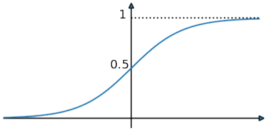
There are many types of layers in `tf-keras`, some used for pre-processing (*e.g* `Flatten`, `Masking`), others used for convolutional and recurrent networks, and others used for regularization purposes.

In the process of defining a layer, and once the number of units is set, there are still many decisions to make, such as weights and biases initializations and regularizations, and the choice of activation functions.

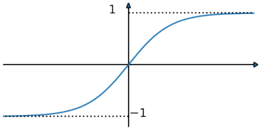
NN with Tensorflow-Keras

Activation Functions:

- The activation functions initially used in MLP networks were the sigmoid and the hyperbolic tangent. These functions are saturating functions (inspired from biological perspective).



$$\varphi(x) = \frac{1}{1+e^{-x}}$$



$$\varphi(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

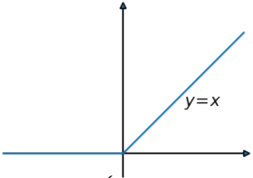
- These function have a severe limitations in terms of adapting the weights of a MLP network with more than 3 layers. The problem is *the vanishing gradient*. The backpropagation algorithm works from the output layer to the input layer, propagating the error gradients. Since the gradients of these activation functions are close to zero for most of their domain, the gradients often get smaller and smaller as the algorithm progresses down the lower layers, leaving the lower layers connection weights virtually unchanged. In these situations, the training never converges to a good solution.

NN with Tensorflow-Keras

Activation Functions:

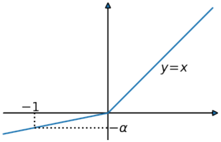
- The vanishing gradient problem is solved by using *non-saturating* activations functions. Among these, the most common are:

ReLU:
Rectified Linear Unit



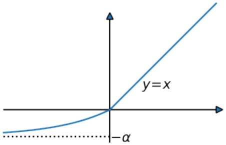
$$\varphi(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$$

Leaky ReLU:



$$\varphi(x) = \max(\alpha x, x)$$

ELU:
Exponential Linear Unit



$$\varphi(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$$

NN with Tensorflow-Keras

Activation Functions:

- There are other activation functions available under the module `keras.activations`. These include the `linear` activation (*i.e* identity transformation - used in regression), `softsign`, `hard_sigmoid`, scaled ELU and many others. The activations can be specified when creating the layers. For example, the next command adds a `Dense` layer with ReLU activations:
  ```
  nn.add(keras.layers.Dense(32,activation="relu"))
  ```
  or
  ```
  nn.add(keras.layers.Dense(32,activation=keras.activations.relu))
  ```

- There is also a special activation function used in the last layer of a MLP for multi-class classification problem: the **softmax** activation. The softmax converts the outputs of the network into class probabilities. Given a network with c outputs, $\hat{y}_i$ with $i = 1, \ldots, c$, the new (softmax) outputs are:
  $$\hat{z}_i = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}}$$
  After the applying the softmax normalization the outputs, $\hat{z}_i$ will be in the interval $[0, 1]$ and $\sum_i \hat{z}_i = 1$.

NN with Tensorflow-Keras

Regularization Techniques:

Regularization refers to methods that prevent the networks to overfit the training data.

- A standard approach is to apply a penalty to the network weights (usually not the biases). The available regularizations in `tf-keras` are ℓ_1, ℓ_2, and a combination of both. The type of regularization is specified in the `layer` command:
```
keras.layers.Dense(32,activation="relu",\
kernel_regularizer=keras.regularizer.l1(0.02))
```
- Another way to avoid the network from overfitting is to use **Dropout**. Dropout is a very popular regularization technique for deep learning networks (by G. Hinton 2012). In this approach, each units weights have a probability of being temporarily “dropped out” during each training step. In `tf-keras` this is done by using a `Dropout` layer after each `Dense` layer. The next commands add a dropout layer (with $p=0.5$) to dense layer. Note that dropout layers do not have any weights (since they are not part of the network) .
```
nn.add(keras.layers.Dense(32,activation="relu"))
nn.add(keras.layers.Dropout(0.5))
```
- Another technique to avoid overfitting is to stop the training is the error in the validation set starts to increase. This is known as **early stoping**. This can be easily implemented by saving the weights of the network every time the best performance on the validation set is attained, and use the winning model.

NN with Tensorflow-Keras

Compiling the Model:

Once the model defined, it needs to be compiled. This means choosing a loss function and an optimization method.

- **Loss Functions:**
  Loss functions are inversely proportional to the performance of the network. The functions are dependent on the outputs of the network, and they are to be minimized during the training by adjusting the weights and biases of a network. There are many loss functions (see Keras Losses) such as MSE, MAE, hinge, etc. For classification, the following are usually a good choice:
  - Binary cross-entropy for two class problems (binary classification)
    `nn.compile(loss="binary_crossentropy")`
  - Categorical cross entropy for multi-class classification
    `nn.compile(loss="categorical_crossentropy")`
- **Optimizers:**
  There are many optimization techniques available in Keras (see Keras Optimizers). These are based on ameliorations of the maximum gradient descent procedure. These include the stochastic gradient descent (`SGD`), the Nesterov, the Adam, the Adagrad and variants, RMSProp and others. ex.: `nn.compile(optimizer="Adam")`

NN with Tensorflow-Keras

Training the Model:

Once compiled, the model is ready to be trained. This is accomplished using the function `fit()`. There are several important parameters to take in account:

- Input dataset and labels.
- Validation set and labels.
  It is important to use a validation set to assess the performance of the network by visualizing the training/validation loss/accuracy evolution. The validation set also enables the use of early stopping. Training methodologies should, if possible, include cross-validation or shuffle/split approaches.
- Number of epochs.
  An epoch is a pass over the entire training set during training. The division of the training into epochs is useful for periodic evaluation and logging purposes. Keras also has the possibility to run **callbacks** at the end of each epoch, like learning rate changes or model saving.
- Batch size.
  The variable `batch_size` specifies the number of training samples used in each weight adaptation. Larger batch sizes usually result in better approximations at the expense of a longer training time.

Ex: training the previous network for 10 epochs:

```
train=nn.fit(Xtrain,ytrain,epochs=10,batch_size=1024,\
validation_data=(Xvalid,yvalid))
```

NN with Tensorflow-Keras

Training the Model:

Once compiled, the model is ready to be trained. This is accomplished using the function `fit()`. There are several important parameters to take in account:

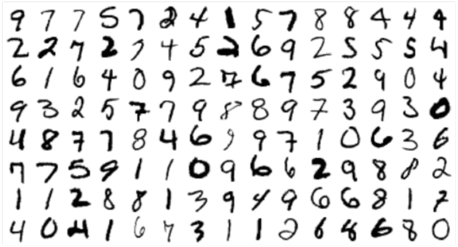
- Input dataset and labels.
- Validation set and labels.
  It is important to use a validation set to assess the performance of the network by visualizing the training/validation loss/accuracy evolution. The validation set also enables the use of early stopping. Training methodologies should, if possible, include cross-validation or shuffle/split approaches.
- Number of epochs.
  An epoch is a pass over the entire training set during training. The division of the training into epochs is useful for periodic evaluation and logging purposes. Keras also has the possibility to run **callbacks** at the end of each epoch, like learning rate changes or model saving.
- Batch size.
  The variable `batch_size` specifies the number of training samples used in each weight adaptation. Larger batch sizes usually result in better approximations at the expense of a longer training time.

Ex: training the previous network for 10 epochs:
It is important to specify a return variable to see the accuracy and loss evolutions during training.

NN with Tensorflow-Keras

MNIST Dataset:

Multi-class problem, with 10 classes (images of handwritten digits 0-9).
Training set: 60 000 images
Test set: 10 000 images



Objectives:

- Train a MLP network with this data.
- Use a validation set to evaluate the performance.
- Assess the quality of the network and spot any sings of overfitting.

NN with Tensorflow-Keras

MNIST Dataset:

Load needed Python modules, the MNIST dataset and prepare it for classification.

- Python modules:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.keras as keras
from tensorflow.keras.datasets import mnist
```

- Load data and divide the training data into train and validation sets.

```
(Xtrain,ytrain),(Xtest,ytest)=mnist.load_data()
Ntrain=40000
Xvalid=Xtrain[Ntrain:]*1.  # "*1."→ to convert to float
yvalid=ytrain[Ntrain:]
Xtrain=Xtrain[:Ntrain]*1.
ytrain=ytrain[:Ntrain]
Xtest=Xtest*1.
```

(suggestion: use the `StandardScaler` from `sklearn` to normalize the data).

- Convert labels into vectors (one hot encoding).

```
ytrainB=keras.utils.to_categorical(ytrain)
yvalidB=keras.utils.to_categorical(yvalid)
ytestB=keras.utils.to_categorical(ytest)
```

NN with Tensorflow-Keras

MNIST Dataset:

Build, compile and train a fully connected 3 layer MLP network, with 50 units in the hidden layer. Although the input data consists of images, in this approach (as in most of ML cases), the data is first converted into vector format (with 784 dimensions). Note that the same conversion process has to be done to the test data or to new data fed into the network.

- Defining the 3-layers MLP network:

  ```
  nn=keras.Sequential()
  nn.add(keras.layers.Flatten(input_shape=[28,28]))
  nn.add(keras.layers.Dense(50, activation="tanh"))
  nn.add(keras.layers.Dense(10,activation='softmax'))
  ```

  NOTE: 1) The first layer is just a reshape of the inputs (it does not have any parameters). This could also have been done in the pre-processing phase of the data.
  2) The last layers is a softmax activation (outputs = number of classes).

- Compiling the model

  ```
  nn.compile(optimizer="nadam",\
  loss="categorical_crossentropy",metrics=["accuracy"])
  ```

  (suggestion: try different optimizers).

NN with Tensorflow-Keras

MNIST Dataset:

Training the model.

- Training the model for 10 epochs (should be longer).

```
train=nn.fit(Xtrain, ytrainB, epochs=10\
            ,batch_size=1024,validation_data=(Xvalid, yvalidB))
```

```
Train on 40000 samples, validate on 20000 samples
Epoch 1/10
40000/40000 [==============================] - 1s 16us/sample - loss:  1.2974 -
acc:  0.5868 - val_loss:  0.7243 - val_acc:  0.7901
Epoch 2/10
40000/40000 [==============================] - 0s 12us/sample - loss:  0.5901 -
acc:  0.8301 - val_loss:  0.5206 - val_acc:  0.8492
Epoch 3/10
40000/40000 [==============================] - 0s 12us/sample - loss:  0.4705 -
acc:  0.8640 - val_loss:  0.4453 - val_acc:  0.8672

⋮

Epoch 10/10 40000/40000 [==============================] - 0s 12us/sample -
loss:  0.3213 - acc:  0.9043 - val_loss:  0.3271 - val_acc:  0.9040
```

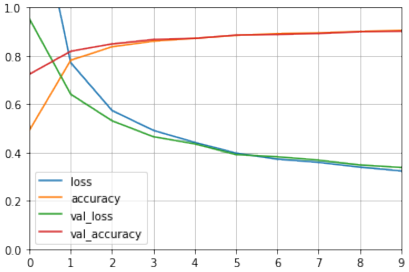
NN with Tensorflow-Keras

MNIST Dataset:

Training the model.

- Training the model for 10 epochs (should be longer).
```
train=nn.fit(Xtrain, ytrainB, epochs=10\
            ,batch_size=1024,validation_data=(Xvalid, yvalidB))
```

- Visualizing the loss and accuracy evolution during training (final accuracy≈ 90%)
```
h=train.history
plt.plot(h["loss"]),plt.plot(h["acc"])
plt.plot(h["val_loss"]),plt.plot(h["val_acc"])
```

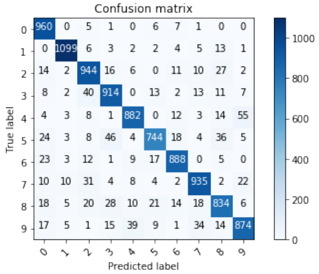


NN with Tensorflow-Keras

MNIST Dataset:

Making predictions and test-set evaluation:

- Predictions on the test set.
  The predicted outputs have to be converted back to integer labels (0-9).
  ```
  yOut=nn.predict(Xtest)
  ytestPred=np.argmax(yOut,axis=1)
  ```
- Print the confusion matrix.
  ```
  from sklearn.metrics import confusion_matrix
  print(confusion_matrix(ytest,ytestPred))
  ```



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	960	0	5	1	0	6	7	1	0	0
1	0	1099	6	3	2	2	4	5	13	1
2	14	2	944	16	6	0	11	10	27	2
3	8	2	40	914	0	13	2	13	11	7
4	4	3	8	1	882	0	12	3	14	55
5	24	3	8	46	4	744	18	4	36	5
6	23	3	12	1	9	17	888	0	5	0
7	10	10	31	4	8	4	2	935	2	22
8	18	5	20	28	10	21	14	18	834	6
9	17	5	1	15	39	9	1	34	14	874

- Accuracy on the test set (`nn.evaluate(Xtest,ytestB)`)
  ```
  10000/10000 [==========================] - 0s 29us/sample - loss:  0.3106 - acc:  0.9074
  ```

NN with Tensorflow-Keras

Saving and Loading Keras Models:

Models can be saved using the `.save(`HDF5 file name`)` function of the model class. The saved file contains:

- model architecture
- model weights
- the training configuration (loss, optimizer)
- the state of the optimizer,(this allows to resume training where it was left off).

In order to save models, Keras uses the Python package `h5py` which is a interface for the binary data format HDF5 (see `www.h5py.org`).
If you need to install the package: `pip -install h5py`.

Models can be loaded using the `load_model` function:

```
from tensorflow.keras.models import load_model
nn=load_model(HDF5 model file)
```

Model configurations and model weights can also be saved/loaded separately. To model's architecture can be saved to a JSON or YAML file format, via the model's `.to_json()` or `.to_yaml()` functions, and loaded using the `model_from_json()` or the `model_from_yaml()` functions. The weights are saved using the `.save_weights()` function and loaded using the `.load_weights()` function.

Check out the Keras Documentation on how to save models.

NN with Tensorflow-Keras

Callbacks:

Callbacks are functions that can be applied during the training stage. Callbacks have access to all the available data about the state of the model and its performance, and they can be used for various actions. Here are a few of its usages:

- **Logging** training and validation data during training. Among the the callbacks used for this purpose are `BaseLogger` (automatically applied to Keras models), the `progbarLogger` the `CSVLogger`, and the `History` callback (also automatically applied) .
- Creating **model checkpoints** by saving the model after every epoch:
```
tensorflow.keras.callbacks.ModelCheckpoint(fileName,\
monitor="val_loss", verbose=1, save_best_only=True,\
save_weights_only=False, mode="auto", period=1)
```
- **Early stopping** the training process. This callback is used to stop the training when the performance has stopped improving:
```
tensorflow.keras.callbacks.EarlyStopping(monitor="val_loss",\
patience=5, verbose=1, mode="auto", restore_best_weights=True)
```
- There are many callbacks and custom callbacks can be also created. Check out the Keras Documentation.

NN with Tensorflow-Keras

Callbacks:

Callbacks are passed as a list (with the keyword `callbacks`) to the `.fit()` function of the model. The following code trains the model of the previous example with three callbacks:

```
from tensorflow.keras import callbacks
CB1=callbacks.ModelCheckpoint(fileName1, period=1)
CB2=callbacks.EarlyStopping(monitor="val_loss",patience=5, \
verbose=1, mode="auto", restore_best_weights=True)
CB3=callbacks.CSVLogger(fileName2, separator=",", append=False)
CBlist=[CB1,CB2,CB3]

train=nn.fit(..., callbacks=CBlist)
```