Artificial Neural Networks
Multi Layer Perceptrons

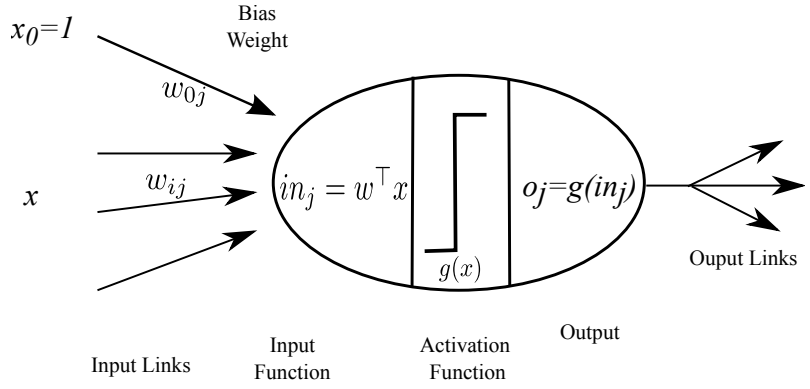
Slides from Javier Bejar

CS-FIB-UPC

Introduction

- Neurons are the building blocks of the brain
- Their interconnectivity defines the programming that allows us to solve all our everyday tasks
- They are able to perform parallel and fault tolerant computation
- Theoretical models of how the neurons in the brain work and how they learn have been developed since the beginning of Artificial Intelligence
- Many of these models are really simple (yet powerful) and have a slim resemblance to real brain neurons

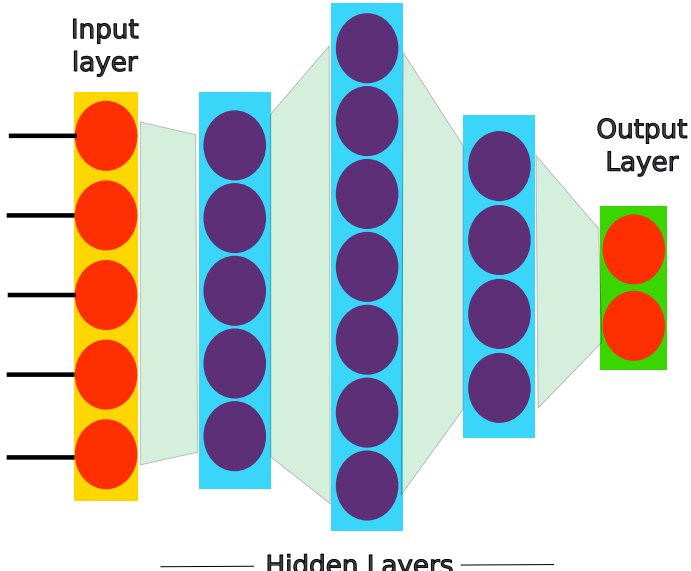
- The task that a single neuron performs is very simple
  - Given a set of inputs, compute an output using their combined value and a possibly non-linear transformation



- A neuron has a bias input (x_0) with value 1
- The weights of the neuron (w_i) are the parameters of the model
- The input is computed as a weighted sum of the inputs (linear combination)
- The output $\hat{y}$ is obtained by applying the activation function $g(x)$ to the input

$$\hat{y} = f(x, w) = g\left(\sum_{i=0}^{I} w_i x_i\right)$$

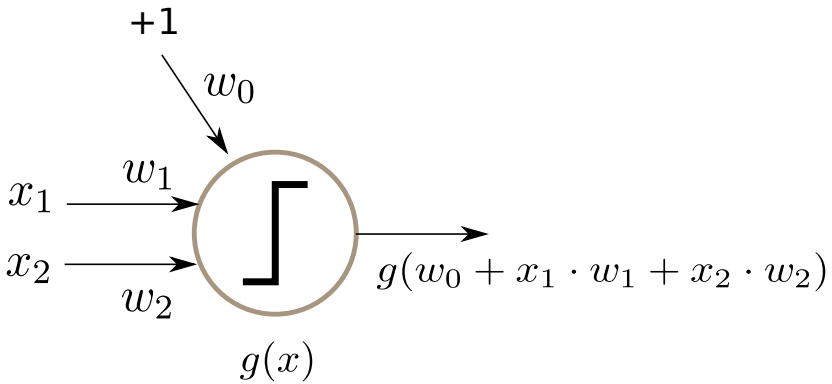
- Feed forward networks are organized in layers, each fully connected to the next, forming a DAG (no cycles)
  - Single layer neural networks (perceptron networks): input layer, output layer
  - Multiple layer neural networks (MLP): input layer, hidden layers, output layer
- The input layer has as many units as inputs in the problem
- The output layer has as many units as needed for the task
- Each hidden layer has several units (possibly a different number per layer)
- The input of each unit is all the outputs of the units from the previous layer

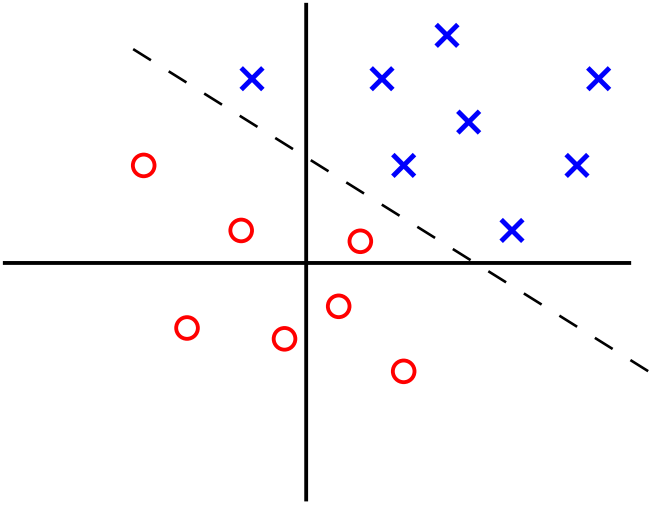

Input
layer
Output
Layer
Hidden Layers

Single Layer Neural Networks

- A single output unit perceptron can be used for classification when we have two classes (several units can be used together for more classes) and for regression
- The model with one unit and linear output for classification is a linear discriminant function that divides the examples in two classes (a linear function of the inputs)
- We can use different activation functions in the output layer for classification
  - Linear output = Linear discriminant (the sign is the class)
  - Sigmoid or hyperbolic tangent output $\approx$ Logistic regression (probability of the class)

Preceptron unit for two inputs

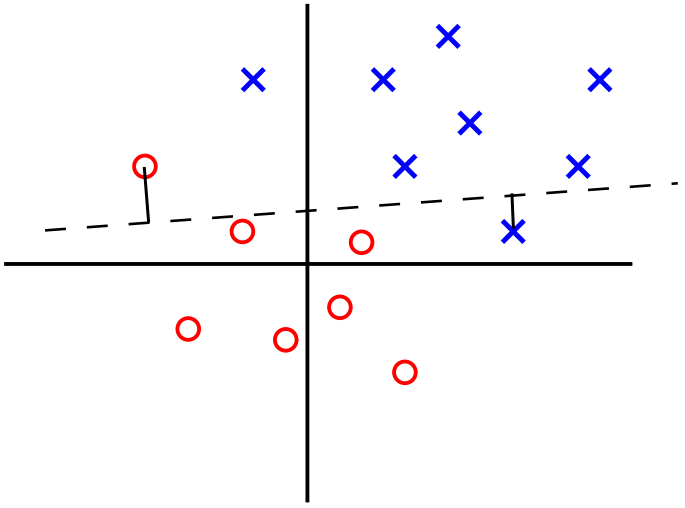




- To learn a linear discriminant with the perceptron we use as labels the values $y_n \in \{-1, +1\}$
- We minimize the squared error loss function to minimize the square distance from the missclassified instances to the linear discriminant function that defines the perceptron weights
- We want to minimize:

$$\ell(y_n, \hat{y}_n) = \frac{1}{2} \sum_{n=1}^{N} (y_n - \hat{y}_n)^2$$

where y_n is the label of example x_n and $\hat{y}_n$ is the output obtained by the perceptron $f(x_n, w) = w^\top x + w_0$



- We can obtain the linear discriminant using a gradient descent algorithm
- In order to obtain the update for the gradient descent algorithm we have to obtain the derivative of the error respect to the parameters $\nabla \ell(x, w)$
- The update rule will be:

$$w \leftarrow w + \Delta w$$

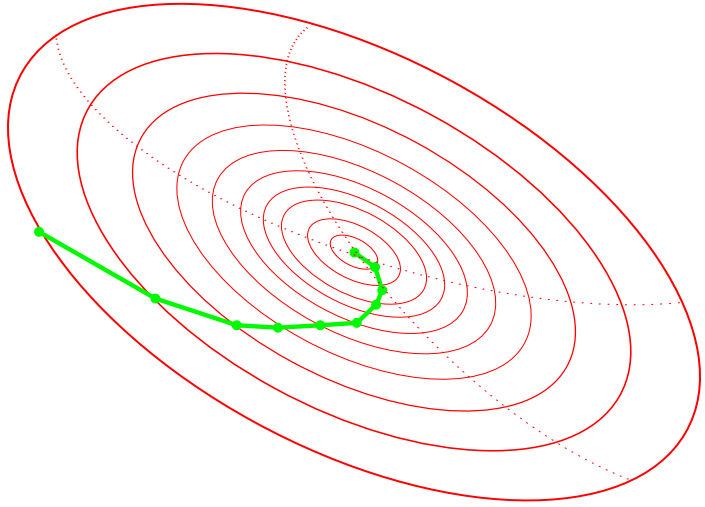
  where $\Delta w = -\alpha \nabla \ell(w)$ (steepest descent)
- This update is repeated several times until convergence (each iteration is called epoch)
- Depending on how the optimization is performed, each iteration uses the average of the gradient of several examples as the direction to move

$$
\begin{aligned}
\frac{\partial \ell}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\frac{\partial}{\partial w_i}(y_n - \hat{y}_n)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}2(y_n - \hat{y}_n)\frac{\partial}{\partial w_i}(y_n - \hat{y}_n)(chain\ \ rule) \\
&= \sum_{n=1}^{N}(y_n - \hat{y}_n)\frac{\partial}{\partial w_i}(y_n - w^\top x_n) \\
&= -\sum_{n=1}^{N}(y_n - \hat{y}_n)(x_{ni})
\end{aligned}
$$

- Now the update rule:

$$\Delta w_i = \alpha \cdot \sum_{n=1}^{N} (y_n - \hat{y}_n) x_{ni}$$

- The space of parameters for linear units is formed by a hyperparaboloid surface (quadratic function), this means that a global minimum exists
- The parameter α (learning factor) has to be small enough, so we do not miss the minimum, but not too small so it takes forever to reach it
- Usually this parameter α is decreased with the iterations (decay)



Algorithm: Gradient Descent Linear Perceptron

Input : $(\mathcal{X}, \mathcal{Y}) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ (training set, with binary labels $y_i \in \{+1, -1\}$)

Output: w (vector of weights)

$t \leftarrow 0$

$w(t) \leftarrow random\ weights$

repeat

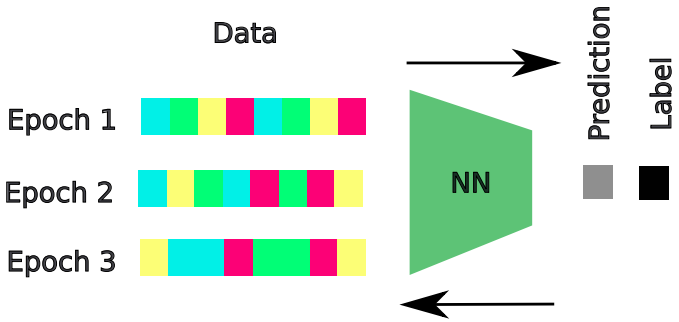
$\Delta w(t) = \alpha \cdot \frac{1}{N} \sum_{n \in \mathcal{X}} (y_n - \hat{y}_n) x_n$

$w(t+1) = w(t) + \Delta w(t)$

$t++$

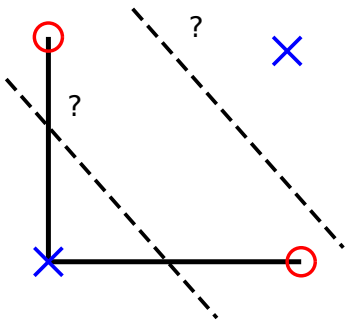
until *no change in* w

- If we have a large dataset, to compute the update with all the data can be costly
- The learning can be performed on-line, updating the weights using one example at a time, this is known as Stochastic Gradient Descent
- The examples are shuffled randomly each iteration (this prevents cycles)



- To use only one example could mean to have large variance on the gradient updates
- A compromise is to group the examples in random subsamples and average their gradients, this is known as Mini Batch Stochastic Gradient Descent
- The idea is that each example's gradient is a random variable that has as expectancy the true gradient, so to use only a few random examples is a good estimate

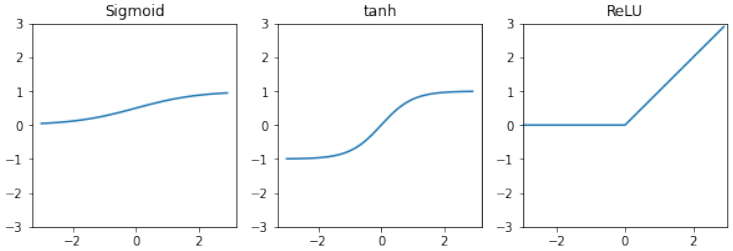
- With linear perceptrons we can only classify correctly linearly separable problems, they do not converge otherwise
- The hypothesis space is not powerful enough for real problems
- Example, the XOR function:



x_1	x_2	f(x)
0	0	1
0	1	0
1	0	0
1	1	1

Multiple Layer Neural Networks

- Perceptron units can be organized in layers forming feed forward networks
- Each layer performs a transformation that is passed to the next layer as input
- The combination of linear units only allows to learn linear functions
- To learn non linear functions we have to use non linear activation functions
- The most used are: sigmoid, tanh, Rectified Linear Unit (ReLU)



- We are using non linear transformations that are tuned during the learning

$$f_i(f_j(x, w_j), w_i) = g_i(w_{i0} + \sum_{m=1}^{M} w_{im} f_j(x, w_{jm})))$$

  i is a perceptron unit in the current layer, f_j is the transformation that is obtained from all the previous layers, that are the input of i

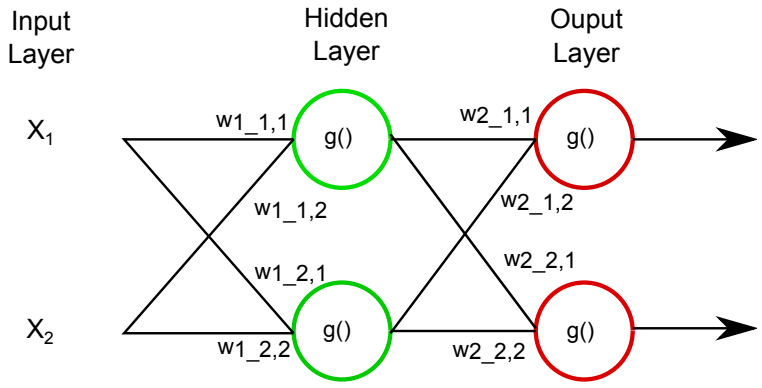
- The basis functions are not fixed as we assumed with the previous methods, we learn the basis functions ϕ

- We have functions that are non-linear respect to the parameters

- Networks with several layers allow to approximate any function given enough units in a layer and enough layers (Universal Approximator functions)
  - Any boolean function can be represented as a two layer MLP (including XOR) (number of hidden units grows exponentially with the number of inputs)
  - Any continuous function can be approximated by a two layer network (sigmoid hidden units, linear output units)
  - Any arbitrary function can be approximated by a three layer network (sigmoid hidden units, linear output units)
- The architecture of the network defines the functions that can be approximated
- But it is difficult to characterize for a particular network structure what functions can be represented and what functions can not

- The output of a MLP determines its task:
  - Univariate Regression: one output neuron with a linear activation function
  - Multivariate Regression: as many output neurons as output functions with linear activation function
  - Binary Classification: one output neuron with a sigmoid activation function
  - Multi Class Classification: as many outputs as classes with a softmax activation
- Regression tasks normally use the squared loss and classification tasks the cross entropy loss, but other losses can also be used depending on the application

- For single layer networks when we have multiple outputs, we can learn each output separately
- In the multilayer case, we have a set of parameters for each unit of the layer and each layer is fully connected to the next layer
- This means that the outputs of the network are all interconnected by the interdependence generated by the hidden layers
- The upgrade of the weights from one output affect the connections with other outputs


Input Layer
Hidden Layer
Ouput Layer
X1
X2
w1_1,1
w1_1,2
w1_2,1
w1_2,2
w2_1,1
w2_1,2
w2_2,1
w2_2,2
g()
g()
g()
g()

- Any neural network is basically a directed acyclic graph (DAG) and the computations for training must follow that graph
- The algorithm used for training is called Backpropagation
- This is a dynamic programming algorithm that uses the topological sort of the graph to compute the gradients of the function given a set of examples

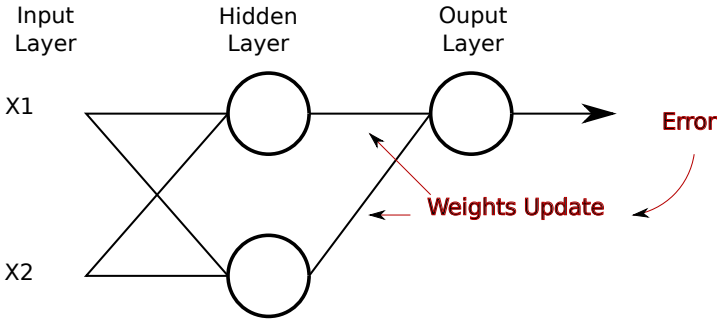
- The dynamic programming is needed to avoid repeated computation given that some operations are shared by several paths in the graph
- The parameters of each layer are computed and updated at the same time
- It combines two steps
  1. The forward pass evaluates the function for the examples for computing the error
  2. The backward pass computes the gradient of the function and the update of the weights

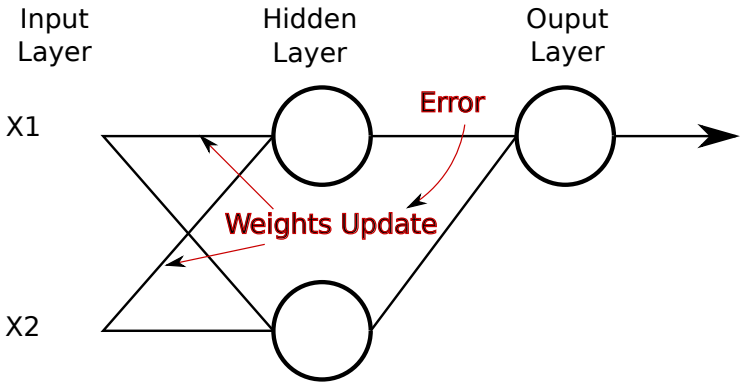
1. Propagate the examples through the network to obtain the output (forward propagation)
   - For each layer, each unit computes the linear combinations of the inputs and the weights
   - Each unit computes the activation functions and passes the output to all the units of the next layer

2. Propagate the output error layer by layer updating the weights of the neurons (back propagation)
   - Each unit computes the difference between the estimated value obtained by the next layer and the actual value of the unit
   - The difference is propagated to each unit of the previous layer proportionally to their weight and the weights are updated

- The error of the single layer perceptron links directly the transformation of the input in the output
- In the case of multiple layers each layer has its own error
- The error of the output layer is directly the error computed from the true values
- The error for the hidden layers is more difficult to define

- The idea is to use the error of the next layer to influence the weights of the previous layer
- We are propagating backwards the output error, hence the name of **backpropagation**
- The error of the units in the output layer is distributed to the previous layer proportionally to the weights of the connections, the process is repeated until the input layer is reached


Input
Layer
Hidden
Layer
Ouput
Layer
X1
X2
Error
Weights Update


Input Layer
Hidden Layer
Ouput Layer
X1
X2
Error
Weights Update

- An MLP computes in a unit the function:

$$a_j = \sum_i w_{ji} \mathrm{o}_i$$

where o_i is the output computed by the unit i from the previous layer and w_{ji} is the weight of the connection between the unit i and unit j

- The output of unit j is computed as:

$$\mathrm{o}_j = g(a_j)$$

where g is an activation function

- The Forward pass computes in order all this values until it reaches the output as $\hat{y}_n = f(x_n, w)$ so the error of the output layer can be computed

- In order to recompute the weights w_{ji} we need to obtain derivative of the error respect to the weights for each example

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}}$$

- The error depends on the weights only on the summation in a_j, so applying the chain rule:

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = \frac{\partial \ell(x_n, w)}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

where

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

- The derivative of the error respect to the unit j is called the error term

$$\delta_j = \frac{\partial \ell(x_n, w)}{\partial a_j}$$

- This term depends on the error function that we are using
- So the derivative of the error respect to the weights can be expressed as:

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = \delta_j z_i$$

- If we use as loss function the squared loss as we saw with the delta rule (see slide 14) we have that the error can be computed as:

$$\frac{\partial \ell(x_n, w)}{\partial a_j} = \frac{\partial \frac{1}{2}(y_n - a_j)^2}{\partial a_j} = -(y_n - a_j)$$
$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = -(y_n - a_j)\frac{\partial a_j}{\partial w_{ji}} = -(y_n - a_j)\,\mathbf{z}_i$$

- If we assume a network with only one layer and the identity function as activation function ($a_j = w_{ji}x_{ni}$) we obtain the delta rule

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = -(y_n - a_j)x_{ni}$$

- The error only depends on the actual output as there is no link with other units
- The gradient depends on the derivative of the activation function of the output and the error function
- This is what we are optimizing in the linear methods

- For the units of the hidden layers, each unit accumulates the derivatives of the units is connected to, applying the chain rule:

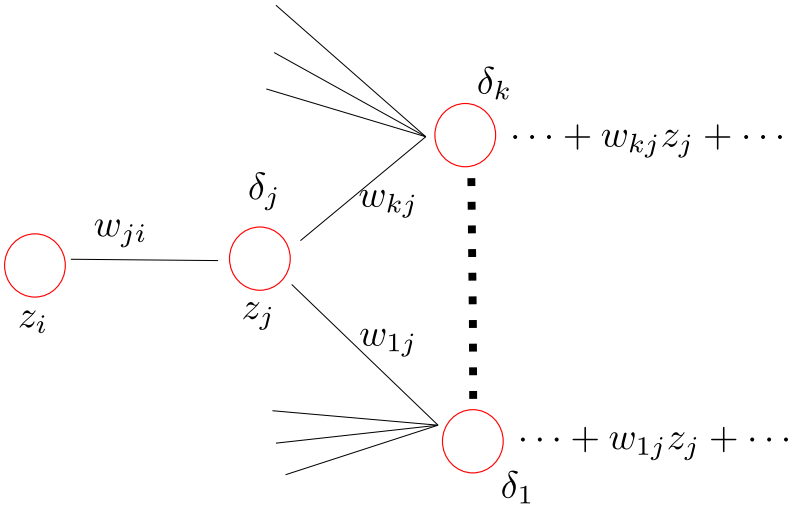
$$\delta_j = \frac{\partial \ell(x_n, w)}{\partial a_j} = \sum_k \frac{\partial \ell(x_n, w)}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

  we sum over all the units k from the next layer that j is connected with

- We obtain

$$\delta_j = \sum_k \delta_k \frac{\partial w_{kj} \mathbf{o}_j}{\partial a_j} = \sum_k \delta_k w_{kj} \frac{\partial g(a_j)}{\partial a_j} = g'(a_j) \sum_k w_{kj} \delta_k$$

  So we just propagate backwards the δ_k from the next layer proportionally to the connection weights


δ_k
$\cdots + w_{kj} z_j + \cdots$
δ_j
w_{kj}
w_{ji}
z_i
z_j
w_{1j}
$\cdots + w_{1j} z_j + \cdots$
δ_1

- With the gradients for each variable we can recompute the weights using gradient descent (all/stochastic/minibatch stochastic)
- For the output layer:

$$w_{ji}^{t+1} = w_{ji}^{t} - \alpha \delta_j z_i$$

- For the hidden layers

$$w_{ji}^{t+1} = w_{ji}^{t} - \alpha \left(g'(a_j) \sum_k w_{kj} \delta_k \right) z_i$$

- For instance, using sigmoid activation functions so $g(a_j) = \mathrm{o}_j = \sigma(a_j)$:[1]

$$w_{ji}^{t+1} = w_{ji}^{t} - \alpha \left(\mathrm{o}_j (1 - \mathrm{o}_j) \sum_k w_{kj} \delta_k \right) z_i$$

[1] Remember that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

- Currently, Neural networks are defined directly as computational graphs
- A neural network can have in its nodes many kinds of computations interconnected with other nodes, each one needs a derivative and a way of computing the forward and backward pass
- All these computations are defined programmatically and combined on run time using automatic differentiation
- The Neural network definition language/runtime is in charge of building the computation graph and generating the forward and backward passes

- For multilayer neural networks the hypothesis space is the set of all possible weights values that the units can have
- The surface determined by these parameters and the error function can have several local optima
- This means that the solution can change depending on the initialization
- Convergence can be also very slow for certain problems

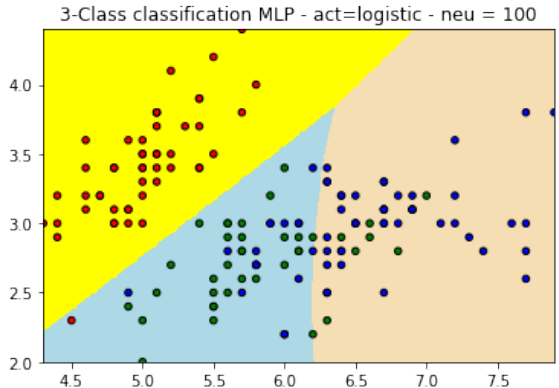
- One improvement introduced to obtain faster convergence is the use of the method called *momentum*
- This method alters the update of the weights including a weighted part of the previous update Δw_{ji}^{t-1}:

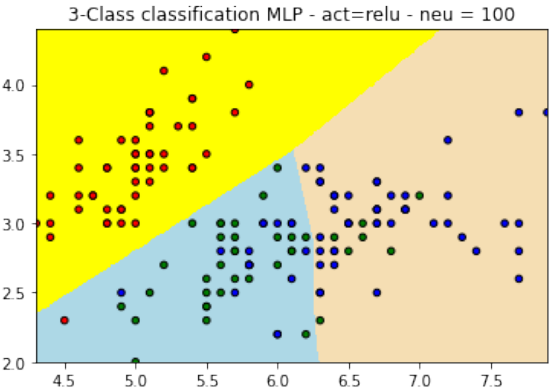
$$w_{ji}^{t+1} \leftarrow w_{ji}^{t} + \alpha \Delta w_{ji}^{t} + \mu \Delta w_{ji}^{t-1} \;\; with \;\; 0 \leq \mu < 1$$

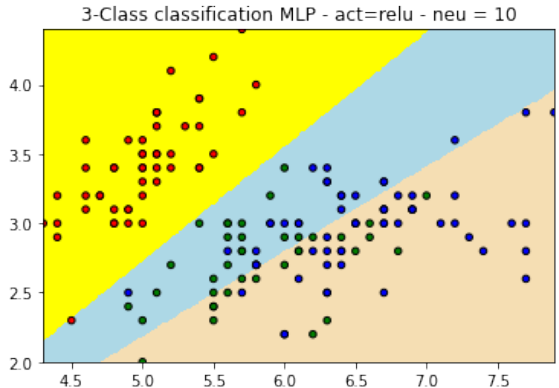
- The effect is to maintain the exploration in the same direction, allowing to skip local minima and performing longer steps, helping to faster convergence

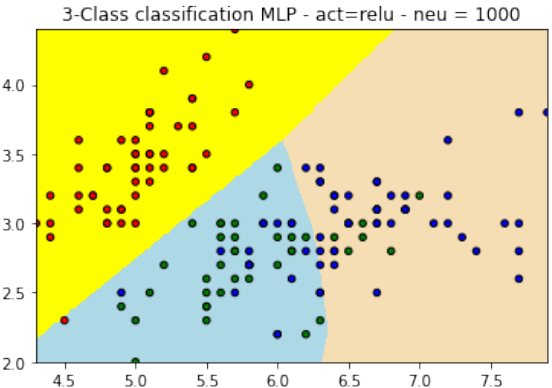
- The more obvious condition for terminating the backpropagation algorithm is when the error can not be reduced any more, usually this leads to overfitting
- This overfitting occurs during the latest iterations of the algorithm
- At this point the weights try to minimize the error fitting specific characteristics of the data (usually the noise)
- One possibility is to use regularization, for instance L_2 or L_1 (weight decay)
- Other possibility is to limit the number of iterations of the algorithm, this is called early termination
- This adds more hyperparameters to the model

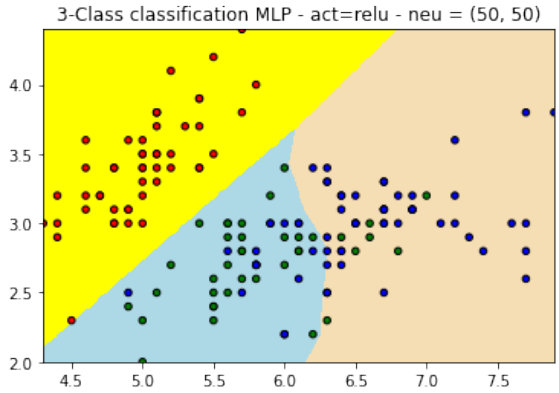
- To optimize a neural network is not an easy task
  - Decide the number of layers
  - Decide the units per layer
  - Decide the error function
  - Decide the activation function (per layer)
  - Decide optimization algorithm and its parameters
  - Use early termination?
  - Momentum?
  - Regularization?
  - ....
- The computational cost makes difficult to use cross validation
- The size of the dataset matters (many parameters to tune)

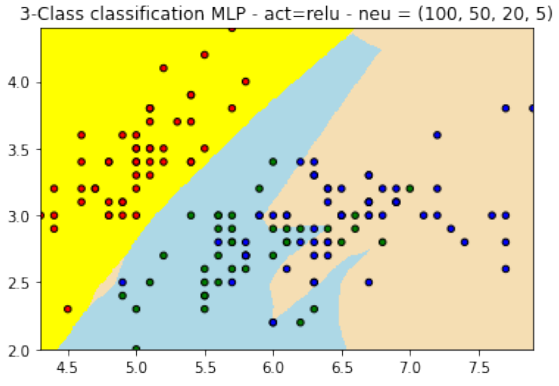

3-Class classification MLP - act=logistic - neu = 100
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


3-Class classification MLP - act=relu - neu = 100
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


3-Class classification MLP - act=relu - neu = 10
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


3-Class classification MLP - act=relu - neu = 1000
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


3-Class classification MLP - act=relu - neu = (50, 50)
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


3-Class classification MLP - act=relu - neu = (100, 50, 20, 5)
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5