Multilayer Perceptron

Introduction to Machine Learning – GIF-7015

Professor : Christian Gagné

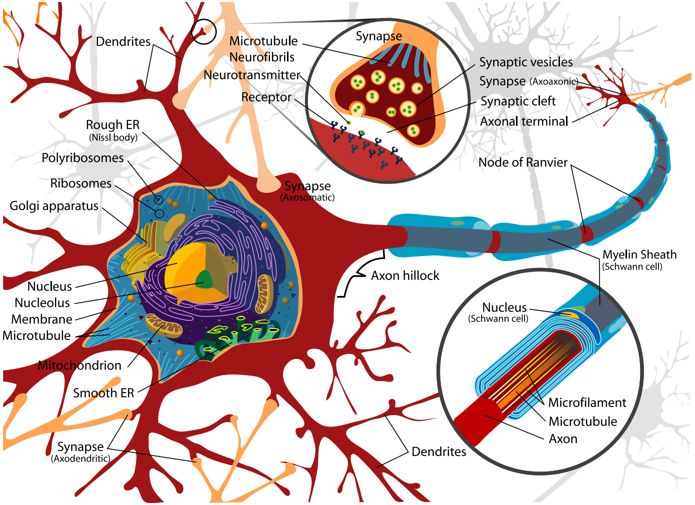
Week 7

7.1 Multilayer perceptron model

Natural intelligence

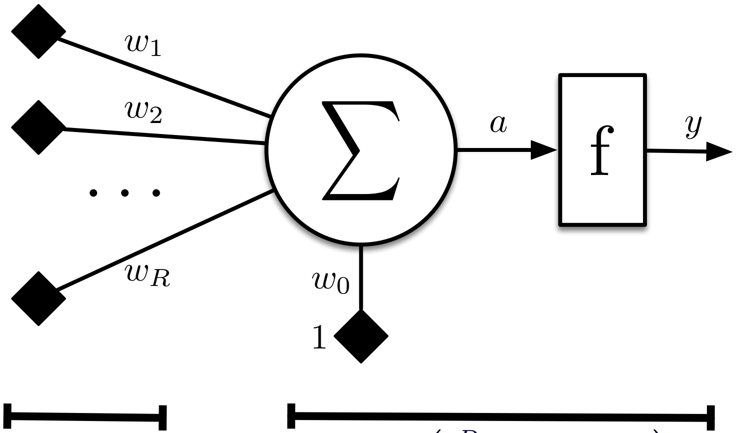
- Brain: natural intelligence
  - Parallel and distributed computing
  - Learning and generalization
  - Adaptation and context
  - Error-tolerant
  - Low energy consumption
- Biological computational machine!

Biological neuron



By LadyofHats, public domain, `https://commons.wikimedia.org/wiki/File:Complete_neuron_cell_diagram_en.svg`

Artificial neuron model



$$y = \mathrm{f}\left(\sum_{i=1}^{R} w_i x_i + w_0\right)$$

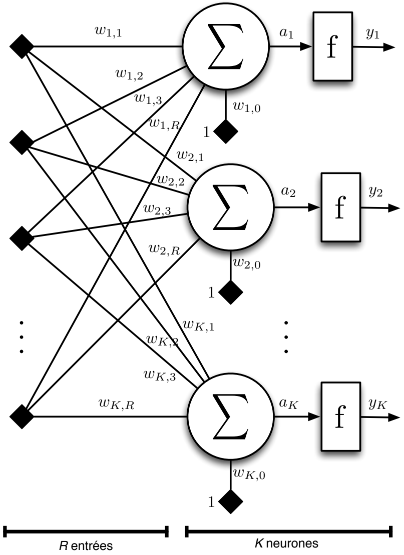
Neural network

- Each neuron is a linear discriminator with a transfer function f

$$y = \mathrm{f}\left(\sum_i w_i x_i + w_0\right) = \mathrm{f}(\mathbf{w}^\top \mathbf{x} + w_0)$$

- Examples of transfer functions
  - Linear function: $\mathrm{f}_{lin}(a) = a$
  - Sigmoid function: $\mathrm{f}_{sig}(a) = \frac{1}{1+\exp(-a)}$
  - Step function: $\mathrm{f}_{step}(a) = 1$ if $a \geq 0$ and $\mathrm{f}_{step}(a) = 0$ otherwise
- Several neurons connected together form a neural network
  - Single-layer network: neurons are connected to the inputs
  - Multilayer network: some neurons are connected to the outputs of other neurons

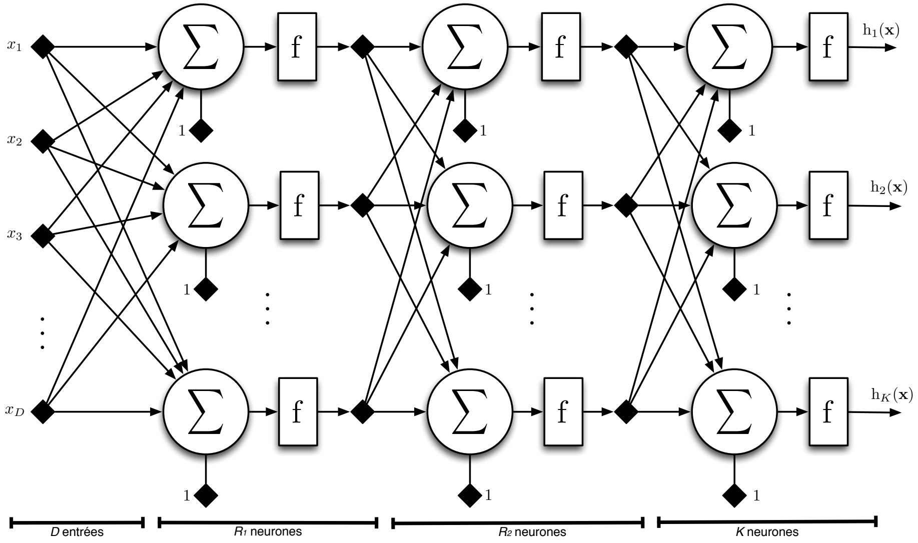
Neural network (one layer)



Multilayer perceptron

- Single-layer network: set of linear discriminants
  - Unable to correctly classify non-linearly separable data
- Multilayer network (multilayer perceptron)
  - Linear discriminants (neurons) cascaded at the output of other linear discriminants
  - Able to classify non-linearly separable data
  - Set of simple classifiers
  - Each layer makes a projection into a new space
- During data processing, information is propagated from inputs to outputs

Multilayer perceptron



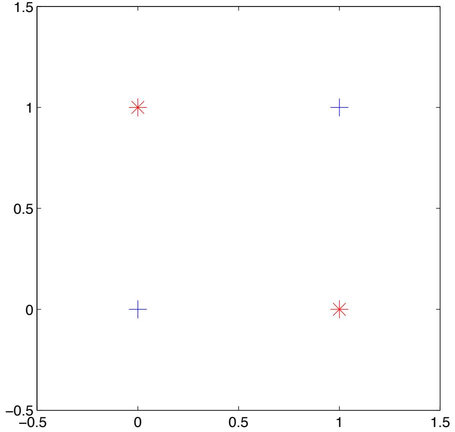
7.2 Topology and capacity of networks

XOR problem

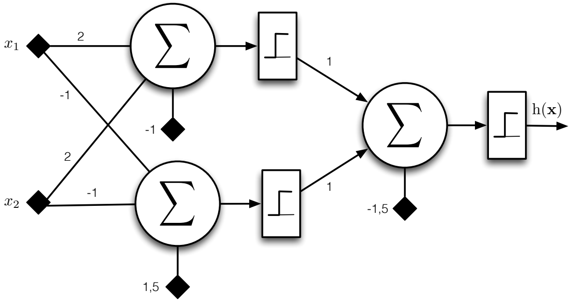
- XOR problem

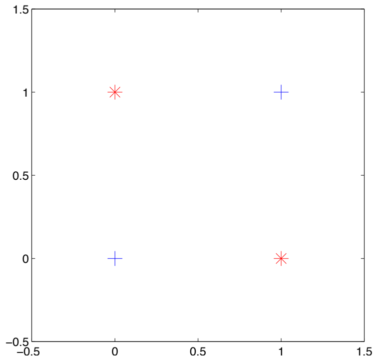
$$\begin{aligned} \mathbf{x}_1 &= [0\ 0]^\top & r_1 &= 0 \\ \mathbf{x}_2 &= [0\ 1]^\top & r_2 &= 1 \\ \mathbf{x}_3 &= [1\ 0]^\top & r_3 &= 1 \\ \mathbf{x}_4 &= [1\ 1]^\top & r_4 &= 0 \end{aligned}$$

- Example of non-linearly separable data



Network for the XOR problem

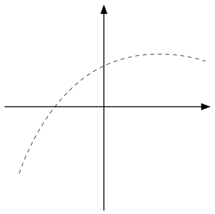


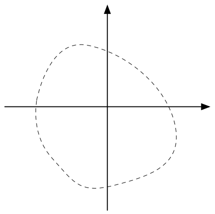


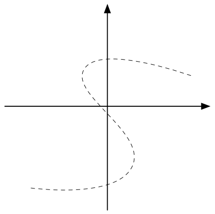
Networks topologies

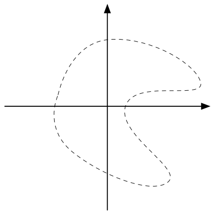
- Depending on the network topology used, different decision boundaries are possible
  - Network with a hidden layer and an output layer: convex boundaries
  - Two or more hidden layers: concave boundaries
    - The neural network is then a universal approximator
- Number of weights (therefore of neurons) directly determines the complexity of the classifier
  - Determining the right topology is often a matter of trial and error

Types of decision boundaries

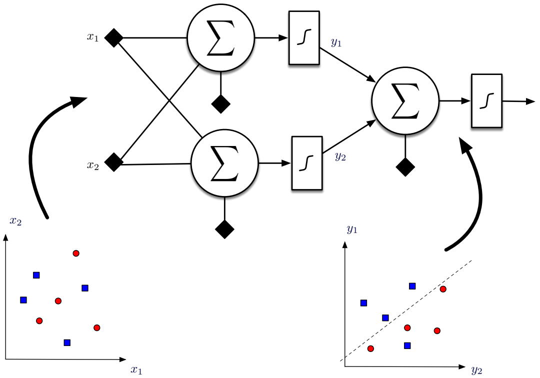

Convex open


Convex close


Concave open

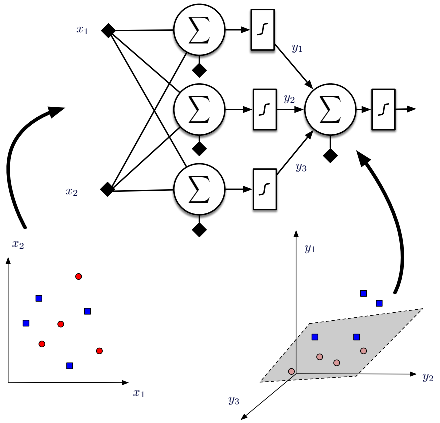

Concave close

Number of neurons on the hidden layer (classification)



2 neurons on the hidden layer: non-optimal

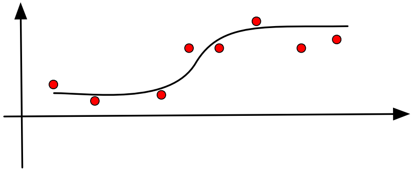
Number of neurons on the hidden layer (classification)



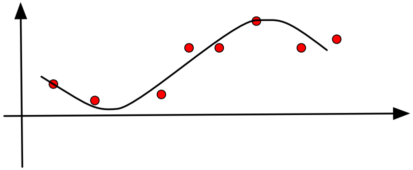
3 neurons on the hidden layer: no error

Number of neurons on the hidden layer (regression)

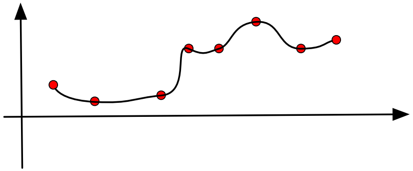
1 neuron



3 neurons



9 neurons

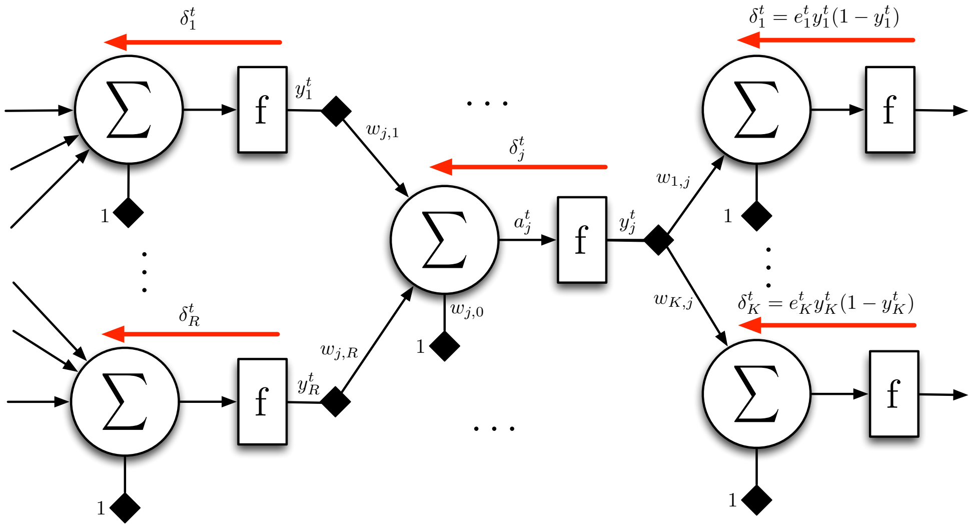


7.3 Error backpropagation

Error backpropagation

- Learning with the multilayer perceptron: determining the weights $\mathbf{w}, w_0$ of all neurons
- Error backpropagation
  - Learning by gradient descent
  - Output layer: error-guided correction between desired and achieved outputs
  - Hidden layers: correction according to sensitivities (influence of the neuron on the error in the output layer)

Error backpropagation



Neuron output values

- Value y_j^t of the neuron j for the data $\mathbf{x}^t$

$$y_j^t = \mathrm{f}(a_j^t) = \mathrm{f}\left(\sum_{i=1}^{R} w_{j,i} y_i^t + w_{j,0}\right)$$

  - f: neuron activation function
  - $a_j^t = \sum_{i=1}^{R} w_{j,i} y_i^t + w_{j,0}$: weighted summation of neuron inputs
  - $w_{j,i}$: weight of the link connecting the neuron j to the neuron i of the previous layer
  - $w_{j,0}$: bias of the neuron j
  - y_i^t: output of the neuron i of the previous layer for the data $\mathbf{x}^t$
  - R: number of neurons on the previous layer

Output layer error

- A dataset $\mathcal{X} = \{\mathbf{x}^t, \mathbf{r}^t\}_{t=1}^N$, with $\mathbf{r}^t = [r_1^t \; r_2^t \; \ldots \; r_K^t]^\top$, where $r_j^t = 1$ if $\mathbf{x}^t \in C_j$, otherwise $r_j^t = 0$
- Error observed for data $\mathbf{x}^t$ on neuron j of the output layer

$$e_j^t = r_j^t - y_j^t$$

- Quadratic error observed for data $\mathbf{x}^t$ on the K neurons of the output layer (one neuron per class)

$$E^t = \frac{1}{2} \sum_{j=1}^{K} (e_j^t)^2$$

- Observed mean squared error for the data in dataset $\mathcal{X}$

$$E = \frac{1}{N} \sum_{t=1}^{N} E^t$$

Error correction for the output layer

- Weight correction by gradient descent of the mean squared error

$$\Delta w_{j,i} = -\eta \frac{\partial E}{\partial w_{j,i}} = -\frac{\eta}{N} \sum_{t=1}^{N} \frac{\partial E^t}{\partial w_{j,i}}$$

- Error of neuron j depends on the neurons of the previous layer
    - Development using the derivative chain rule ($\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \frac{\partial y}{\partial x}$)

$$\begin{aligned} \frac{\partial E^t}{\partial w_{j,i}} &= \frac{\partial E^t}{\partial e_j^t} \frac{\partial e_j^t}{\partial y_j^t} \frac{\partial y_j^t}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{j,i}} \\ \frac{\partial E^t}{\partial w_{j,0}} &= \frac{\partial E^t}{\partial e_j^t} \frac{\partial e_j^t}{\partial y_j^t} \frac{\partial y_j^t}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{j,0}} \end{aligned}$$

Calculation of partial derivatives

- Development with sigmoid activation function ($y_j^t = \frac{1}{1+\exp(-a_j^t)}$)

$$
\begin{aligned}
\frac{\partial E^t}{\partial e_j^t} &= \frac{\partial}{\partial e_j^t} \frac{1}{2} \sum_{l=1}^{K} (e_l^t)^2 = e_j^t \\
\frac{\partial e_j^t}{\partial y_j^t} &= \frac{\partial}{\partial y_j^t} r_j^t - y_j^t = -1 \\
\frac{\partial y_j^t}{\partial a_j^t} &= \frac{\partial}{\partial a_j^t} \frac{1}{1+\exp(-a_j^t)} = \frac{\exp(-a_j^t)}{[1+\exp(-a_j^t)]^2} \\
&= \frac{1}{1+\exp(-a_j^t)} \frac{\exp(-a_j^t)+1-1}{1+\exp(-a_j^t)} = y_j^t(1-y_j^t) \\
\frac{\partial a_j^t}{\partial w_{j,i}} &= \frac{\partial}{\partial w_{j,i}} \sum_{l=1}^{R} w_{j,l} y_l^t + w_{j,0} = y_i^t \\
\frac{\partial a_j^t}{\partial w_{j,0}} &= \frac{\partial}{\partial w_{j,0}} \sum_{l=1}^{R} w_{j,l} y_l^t + w_{j,0} = 1
\end{aligned}
$$

Learning for the output layer

- Learning the output layer weights

$$\begin{aligned}\Delta w_{j,i} &= -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial w_{j,i}} = -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial e_j^t}\frac{\partial e_j^t}{\partial y_j^t}\frac{\partial y_j^t}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{j,i}} \\ &= \frac{\eta}{N}\sum_{t=1}^{N} e_j^t y_j^t(1-y_j^t)y_i^t\end{aligned}$$

- Learning the biases of the output layer

$$\begin{aligned}\Delta w_{j,0} &= -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial w_{j,0}} = -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial e_j^t}\frac{\partial e_j^t}{\partial y_j^t}\frac{\partial y_j^t}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{j,0}} \\ &= \frac{\eta}{N}\sum_{t=1}^{N} e_j^t y_j^t(1-y_j^t)\end{aligned}$$

7.4 The delta rule

The delta rule

- Let a delta δ_j^t, which corresponds to the *local gradient* of the neuron j for the data $\mathbf{x}^t$

$$\begin{aligned}
\delta_j^t &= e_j^t y_j^t (1 - y_j^t) \\
\Delta w_{j,i} &= \frac{\eta}{N} \sum_{t=1}^{N} \delta_j^t y_i^t \\
\Delta w_{j,0} &= \frac{\eta}{N} \sum_{t=1}^{N} \delta_j^t
\end{aligned}$$

- Useful formulation for hidden layer error correction

Hidden layer error correction

- Error gradient for hidden layers

$$\frac{\partial E^t}{\partial w_{j,i}} = \frac{\partial E^t}{\partial y_j^t}\frac{\partial y_j^t}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{j,i}}$$

- Only $\frac{\partial E^t}{\partial y_j^t}$ changes, $\frac{\partial y_j^t}{\partial a_j^t}$ and $\frac{\partial a_j^t}{\partial w_{j,i}}$ are the same as on the output layer
  - Error for a neuron of the hidden layer depends on the error of the k neurons of the next layer (error backpropagation)

$$E^t = \frac{1}{2}\sum_k (e_k^t)^2$$

$$\frac{\partial E^t}{\partial y_j^t} = \frac{\partial}{\partial y_j^t}\frac{1}{2}\sum_k (e_k^t)^2 = \sum_k e_k^t \frac{\partial e_k^t}{\partial y_j^t}$$

Hidden layer error correction

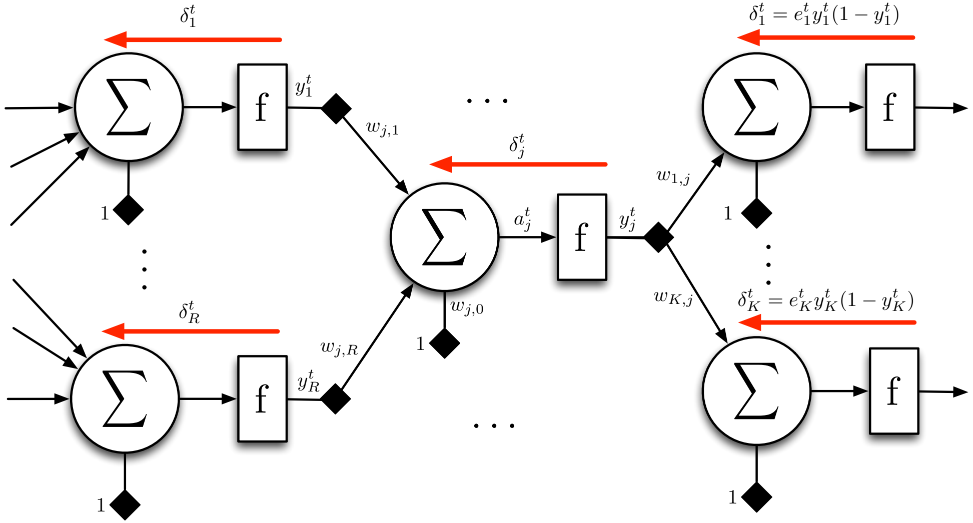
$$
\begin{aligned}
\frac{\partial E^t}{\partial y_j^t} &= \frac{\partial}{\partial y_j^t}\frac{1}{2}\sum_k (e_k^t)^2 = \sum_k e_k^t \frac{\partial e_k^t}{\partial y_j^t} \\
&= \sum_k e_k^t \frac{\partial e_k^t}{\partial a_k^t}\frac{\partial a_k^t}{\partial y_j^t} \\
&= \sum_k e_k^t \frac{\partial (r_k^t - y_k^t)}{\partial a_k^t}\frac{\partial\left(\sum_l w_{k,l} y_l^t + w_{k,0}\right)}{\partial y_j^t} \\
&= \sum_k e_k^t[-y_k^t(1-y_k^t)]w_{k,j} \\
\delta_k^t &= e_k^t[y_k^t(1-y_k^t)] \\
\frac{\partial E^t}{\partial y_j^t} &= -\sum_k \delta_k^t w_{k,j}
\end{aligned}
$$

Hidden layer error correction

- Correction of the corresponding error

$$
\begin{aligned}
\frac{\partial E^t}{\partial w_{j,i}} &= \frac{\partial E^t}{\partial y_j^t}\frac{\partial y_j^t}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{j,i}} \\
&= -\left[\sum_k \delta_k^t w_{k,j}\right] y_j^t(1-y_j^t)y_i^t \\
\delta_j^t &= y_j^t(1-y_j^t)\sum_k \delta_k^t w_{k,j} \\
\Delta w_{j,i} &= -\eta\frac{\partial E}{\partial w_{j,i}} = -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial w_{j,i}} = \frac{\eta}{N}\sum_{t=1}^{N}\delta_j^t y_i^t \\
\Delta w_{j,0} &= -\eta\frac{\partial E}{\partial w_{j,0}} = -\frac{\eta}{N}\sum_{t=1}^{N}\frac{\partial E^t}{\partial w_{j,0}} = \frac{\eta}{N}\sum_{t=1}^{N}\delta_j^t
\end{aligned}
$$

Error backpropagation



7.5 Backpropagation algorithm

Batch and online learning

- Batch learning
  - Guided by mean squared error ($E = \frac{1}{N}\sum_t E^t$)
  - Weight correction once at each epoch, calculating the error for the whole dataset
  - Relatively stable learning
- Online learning
  - Weight correction for each data presentation, so N weight corrections per epoch
  - Guided by the quadratic error of each data (E^t)
  - Requires permutation of the processing order at each epoch to avoid bad sequences
  - Online learning faster than batch, but with greater instabilities
- Mini-batch learning
  - Trade-off between online learning and batch learning, using mini batches of a predefined size

Neuron saturation

- Operating range of neurons with sigmoid function around 0
  - For low a values $\mathrm{f}_{sig}(a) \rightarrow 0$, and for high a values, $\mathrm{f}_{sig}(a) \rightarrow 1$

$$\mathrm{f}_{sig}(1) = 0.7311, \quad \mathrm{f}_{sig}(5) = 0.9933, \quad \mathrm{f}_{sig}(10) \approx 1$$

- For large/small values, say $x < -10$ or $x > 10$, gradient almost equal to zero
  - Extremely slow learning
- Input values, the $\mathbf{x}^t$, must be normalized beforehand in $[-1, 1]$
  - Typically, normalization according to min and max values of the dataset for each dimension
  - Apply the same normalization to the evaluated data (do not recalculate the normalization)

Target output values

- In classification, target values $r_i^t \in \{0, 1\}$
  - Also suffers from the problem of neuron saturation with sigmoid function
  - We aim to approximate the r_i^t with the neurons of the output layer

$$\mathrm{f}_{sig}(a) = 0 \;\Rightarrow\; a \rightarrow -\infty, \quad \mathrm{f}_{sig}(a) = 1 \;\Rightarrow\; a \rightarrow \infty$$

- Solution: transform the desired values into values $\tilde{r}_i^t \in \{0.05, 0.95\}$
  - If $\mathbf{x}^t \in C_i$ then $\tilde{r}_i^t = 0.95$
  - Otherwise $\tilde{r}_i^t = 0.05$

Weights initialization

- The weights and biases of a multilayer perceptron are randomly initialized
    - Typically, weights and biases are initialized uniformly in $[-0.5, 0.5]$

$$w_{j,i} \sim \mathcal{U}(-0.5,\ 0.5),\ \forall i,j$$

- Multilayer Perceptron is thus a stochastic algorithm
    - From one run to another, we do not necessarily obtain the same results

Backpropagation algorithm

1. Normalize data $x_i^t \in [-1,1]$ and target output $\tilde{r}_j^t \in \{0.05, 0.95\}$
2. Initialize weights and bias randomly, $w_{i,j} \in [-0.5, 0.5]$
3. As long as the stop criterion is not reached, repeat:
   3.1 Calculate the observed outputs by propagating the data forward
   3.2 Calculate the observed errors on the output layer

$$e_j^t = \tilde{r}_j^t - y_j^t, \quad j = 1, \ldots, K, \quad t = 1, \ldots, N$$

   3.3 Adjust weights and bias by backpropagating the observed error

$$\begin{aligned} w_{j,i} &= w_{j,i} + \Delta w_{j,i} = w_{j,i} + \frac{\eta}{N} \sum_t \delta_j^t y_i^t \\ w_{j,0} &= w_{j,0} + \Delta w_{j,0} = w_{j,0} + \frac{\eta}{N} \sum_t \delta_j^t \end{aligned}$$

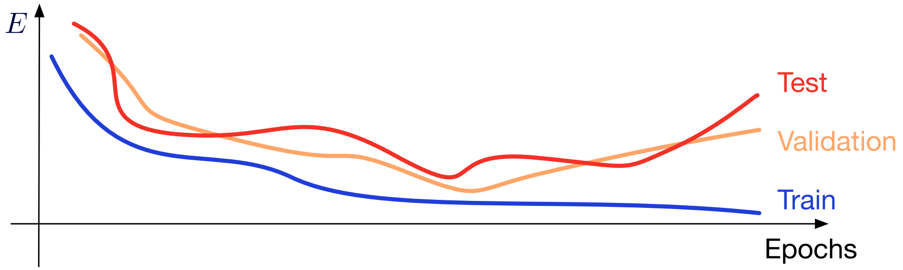
where the local gradient is defined by:

$$\delta_j^t = \begin{cases} e_j^t y_j^t (1 - y_j^t) & \text{if } j \in \text{output layer} \\ y_j^t (1 - y_j^t) \sum_k \delta_k^t w_{k,j} & \text{if } j \in \text{hidden layer} \end{cases}$$

7.6 Training techniques and tips

Overfitting and stop criterion

- Number of epochs: determining factor for overfitting
- Stop criterion: when the error on the validation set increases (generalization)
- Requires to use part of the dataset for validation



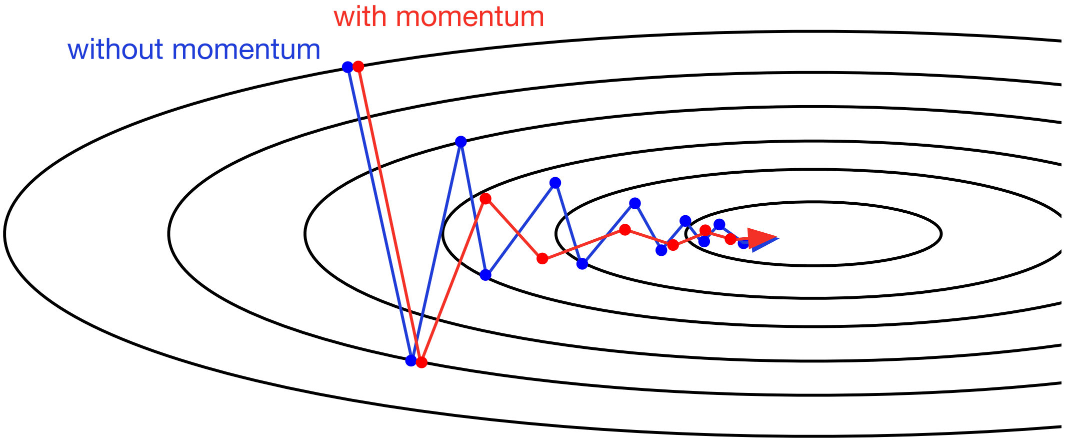
Momentum

- Generalized delta rule

$$
\begin{aligned}
w_{j,i}(n) &= w_{j,i}(n-1) + \frac{\eta}{N}\sum_t \delta_j^t y_i^t + \alpha \Delta w_{j,i}(n-1) \\
w_{j,0}(n) &= w_{j,0}(n-1) + \frac{\eta}{N}\sum_t \delta_j^t + \alpha \Delta w_{j,0}(n-1)
\end{aligned}
$$

- Factor $\Delta w_{j,i}(n-1)$ is the correction made to the weight/bias at the previous epoch
- Parameter $\alpha \in [0.5, 1]$ is named *momentum*
- Gives inertia to the descent of the gradient, including a correction from the previous iterations
- With momentum, the factor $\Delta w_{j,i}(n-1)$ depends itself on the correction of the previous iteration $\Delta w_{j,i}(n-2)$, and so on

Momentum



Regression with multilayer perceptron

- Backpropagation algorithm developed for sigmoid transfer function for classification
  - Other transfer functions can be used
    - Linear function: $\mathrm{f}_{lin}(a) = a$
    - Hyperbolic tangent function: $\mathrm{f}_{tanh}(a) = \tanh(a)$
    - ReLU function (*rectified linear unit*): $\mathrm{f}_{ReLU}(a) = \max(0,a)$
  - In fact, all continuous functions derivable on $\mathbb{R}$ can be used
- Multilayer perceptron suitable for regression
  - Recommended topology: a hidden layer with a sigmoid function and an output layer with a linear function
  - Mean squared error criterion is appropriate for regression

Second order method

- The gradient descent is a first order method (first derivatives)
- Possibility to do better with second order methods
- Newton's method
  - Based on the expansion of the second order Taylor series, $\mathbf{x}' = \mathbf{x} + \Delta\mathbf{x}$ one point in the neighbourhood of $\mathbf{x}$

$$F(\mathbf{x}') = F(\mathbf{x} + \Delta\mathbf{x}) \approx F(\mathbf{x}) + \nabla F(\mathbf{x})^\top \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \nabla^2 F(\mathbf{x}) \Delta\mathbf{x} = \hat{F}(\mathbf{x})$$

  - Search for a plateau in the squared error $\hat{F}(\mathbf{x})$

$$\begin{aligned} \frac{\partial \hat{F}(\mathbf{x})}{\partial \mathbf{x}} &= \nabla F(\mathbf{x}) + \nabla^2 F(\mathbf{x}) \Delta\mathbf{x} = 0 \\ \Delta\mathbf{x} &= -(\nabla^2 F(\mathbf{x}))^{-1} \nabla F(\mathbf{x}) \end{aligned}$$

  - Calculation of the inverse of the Hessian matrix $((\nabla^2 F(\mathbf{x}))^{-1})$: high calculation costs
  - Conjugate gradient method avoids the calculation of the inverse of the Hessian matrix

7.7 Multilayer perceptron in scikit-learn

Scikit-learn

- Multilayer perceptron is available in scikit-learn
  - Scikit-learn uses some (but not all) of the deep network advances
  - No GPU acceleration for calculations, rigid models (not easily customizable)
- `neural_network.MLPClassifier`: multilayer perceptron for classification
  - Minimizes cross entropy for classification with gradient-based methods

$$E_{entr} = -\sum_t r^t \log y^t + (1 - r^t) \log(1 - y^t)$$

- `neural_network.MLPRegressor`: multilayer perceptron for regression
  - Minimizes the quadratic error with gradient-based methods

`MLPClassifier` and `MLPRegressor` parameters

- `hidden_layer_sizes` (tuple): number of neurons on each hidden layer (default: (100,))
- `activation` (string): 'identity' (linear), 'logistic' (sigmoid), 'tanh' and 'relu' (default: 'relu')
- `solver` (string): 'lbfgs' (quasi-Newton), 'sgd' (stochastic gradient descent), 'adam' (sgd with automatic determination of the learning rate) (default: 'adam')
- `alpha` (float): parameter of the L_2 regulation of the weights (default: 0.0001)
- `batch_size` (int): batch size for each update (default: min(200,N))
- `learning_rate_init` (float): initial learning rate (default: 0.001)
- `learning_rate` (string): 'constant', 'invscaling' (learning_rate_init / pow(t, power_t)), 'adaptive' (reduces current rate when learning stagnates) (default: 'constant')
- `max_iter` (int): maximum number of epochs (default: 200)
- `tol` (float): tolerance, stop learning if gain $<$ tolerance for more than two epochs (default: 10^{-4})
- `momentum` (float): momentum for gradient descent (default: 0.9)
- `early_stopping` (bool): stop when error on validation set does not go down anymore (default: False)
- `validation_fraction` (float): portion of the data used for validation with the *early stopping*. (default: 0.1)