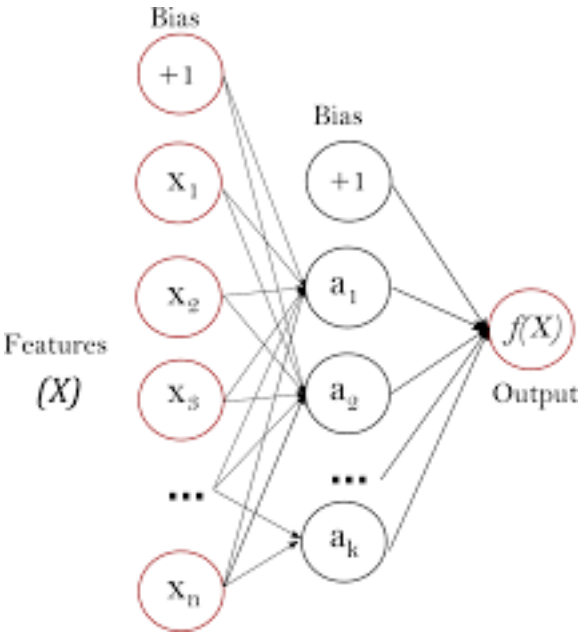
NN : Multilayer perceptron, Back-propagation principle

Olivier J.J. MICHEL, Florent CHATELAIN

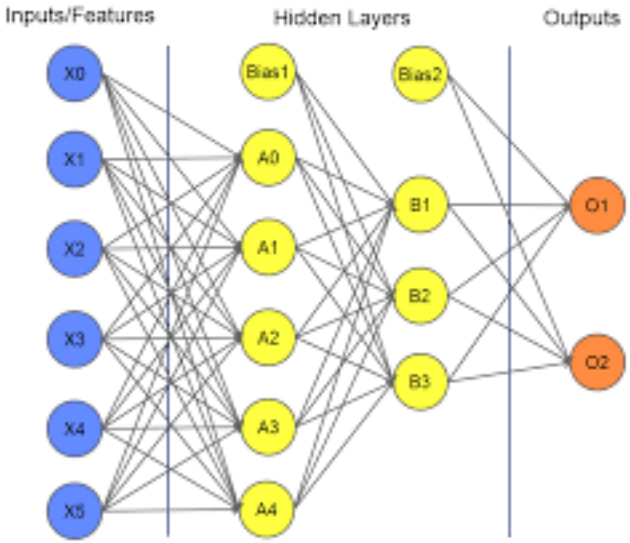
GIPSA-Lab, UMR 5216 CNRS

MLP, NN : Back-Propagation principles

Perceptron



Multi Layer Perceptron



See Perceptron_sonar_example.ipynb

See MLPC_logical_example.ipynb

www Sources for figures & examples
https://scikit-learn.org/stable/
https://gluon.mxnet.io/index.html
https://skymind.ai/wiki/

Learning with Back Propagation Algo :

Notations

w_{ij}^k: weight for node j in layer l_k for incoming node i
b_i^k: bias for node i in layer l_k
a_i^k: product sum plus bias (activation) for node i in layer l_k
o_i^k: output for node i in layer l_k
r_k: number of nodes in layer l_k
g: activation function for the hidden layer nodes
g_o: activation function for the output layer nodes

Activation function

$$a_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ji}^k o_j^{k-1} = \sum_{j=0}^{r_{k-1}} w_{ji}^k o_j^{k-1}$$

Backpropagation attempts to minimize

$$E(X,\theta) = \frac{1}{2N}\sum_{i=1}^{N}\left(\hat{y}_i - y_i\right)^2$$

with respect to the neural network's weights:

$\rightarrow$ for each weight w_{ij}^k, evaluate $\frac{\partial E}{\partial w_{ij}^k}$. By decomposing into a sum over individual error terms for each individual input-output pair

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial}{\partial w_{ij}^k}\left(\frac{1}{2}\left(\hat{y}_d - y_d\right)^2\right) = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial E_d}{\partial w_{ij}^k}.$$

Error function derivatives (index *d* is omitted below)

Apply the chain rule to the error function partial derivative

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k},$$

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k}.$$

Then $$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} \right) = o_i^{k-1}.$$

Output Layer (MLP with $M+1$ layers)

Assume a one-output neural network, so there is only one output node $j=1$). Expressing E in terms of the value a_1^m $\big($since δ_1^m is a partial derivative with respect to $a_1^m\big)$ gives

$$E = \frac{1}{2}\left(\hat{y} - y\right)^2 = \frac{1}{2}\big(g_o(a_1^m) - y\big)^2,$$

where $g_o(x)$ is the activation function for the output layer.

thus,

$$\delta_1^m = \left(g_0(a_1^m) - y\right) g_o'(a_1^m) = \left(\hat{y} - y\right) g_o'(a_1^m).$$

and finaly

$$\frac{\partial E}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = \left(\hat{y} - y\right) g_o'(a_1^m)\ o_i^{m-1}.$$

The Hidden Layers

One has for the error term δ_j^k in layer $1 \leq k < m$:

$$\delta_j^k = \frac{\partial E}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k},$$

where l ranges from 1 to r^{k+1}

(The bias input o_0^k corresponds to w_{0j}^{k+1} is fixed, does not depend on the outputs of previous layers, thus l does not take on the value 0.)

Plugging in the error term :

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}.$$

From the definition of a_l^{k+1}

$$a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} g\left(a_j^k\right),$$

where $g(x)$ is the activation function for the hidden layers,

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'\left(a_j^k\right).$$

Plugging this into the latter equation yields a final equation for the error term δ_j^k in the hidden layers, called the *backpropagation* formula:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'\left(a_j^k\right) = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

And putting all equations together :

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\left(a_j^k\right) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

BACKPROPAGATION ALGO
Main equations

For the partial derivatives,

$$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

For the final layer's error term,

$$\delta_1^m = g_o'(a_1^m)\left(\hat{y_d} - y_d\right).$$

For the hidden layers' error terms,

$$\delta_j^k = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

For combining the partial derivatives for each input-output pair,

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} \left(\hat{y_d} - y_d\right)^2 \right) = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial E_d}{\partial w_{ij}^k}.$$

For updating the weights,

$$\Delta w_{ij}^k = -\alpha \frac{\partial E(X,\theta)}{\partial w_{ij}^k}.$$

The General algorithm

1) Calculate the **forward phase** for each $(\vec{x_d}, y_d)$; store the results $\hat{y_d}$, a_j^k, and o_j^k for each node j in layer k by proceeding from layer 0, to layer m, the output layer.

2) Calculate the **backward phase** for each $(\vec{x_d}, y_d)$; store $\frac{\partial E_d}{\partial w_{ij}^k}$ for each weight w_{ij}^k. Proceedi from output layer m, to layer 1, the input layer.

a) Evaluate δ_1^m

b) Backpropagate the error terms for the hidden layers δ_j^k, working backwards

c) Evaluate the partial derivatives of the individual error E_d with respect to w_{ij}^k

3) Combine the individual gradients for each input-output pair to get the total gradient for the entire set $X = \left\{(\vec{x_1}, y_1), \ldots, (\vec{x_N}, y_N)\right\}$ (a simple average of the individual gradients).

4) Update the weights according to the learning rate α and total gradient $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$

In the classic formulation,
for hidden nodes $\big(g(x) = \sigma(x)\big)$ (sigmoidal function)
and the output activation function is $\big(g_o(x) = x\big)$

$$g'(x) = \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\big(1 - \sigma(x)\big).$$

$$g'_o(x) = \frac{\partial g_o(x)}{\partial x} = \frac{\partial x}{\partial x} = 1.$$

$\rightarrow$ No need to remember the activation values a_1^m and a_j^k in addition to the output values o_1^m and o_j^k, greatly reducing the memory footprint of the algorithm.

BUT gradient descent algorithm may be infeasible when the training data size is huge.
Thus, a stochastic version of the algorithm is often used instead.

Stochastic Gradient method

At each iteration, rather than computing

$$\nabla_\theta E(\mathbf{X}) = \nabla_\theta(\sum_{d=1}^{N} E(x_d)) = \sum_{d=1}^{N} \nabla_\theta E(x_d)$$

stochastic gradient descent randomly samples d at uniform and computes $\nabla_\theta E(x_d)$ instead :
SGD uses $\nabla E(x_d)$ as an unbiased estimator of $\nabla E(\mathbf{X})$;

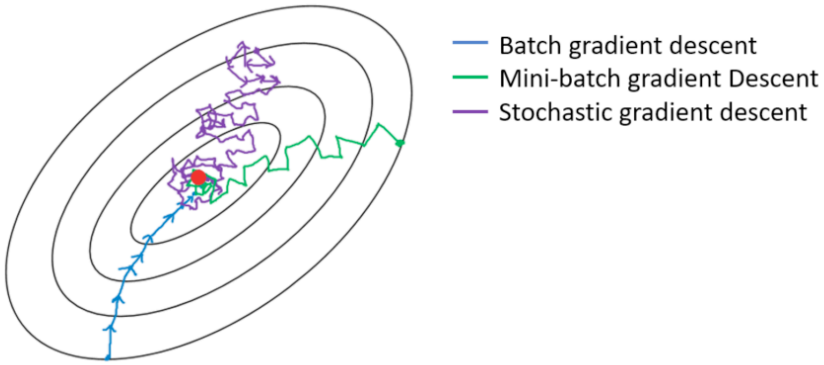
$$\mathbb{E}\left[\nabla_\theta E(x_d)\right] = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} \nabla E(x_k)\right] = \nabla E(\mathbf{X}).$$

In a generalized case, at each iteration a mini-batch $\mathcal{B}$ that consists of indices for training data instances may be sampled at uniform with replacement.

$$\nabla f E_{\mathcal{B}}(\mathbf{X}) = \frac{1}{|\mathcal{B}|}\sum_{d\in\mathcal{B}} \nabla E(x_d)$$

update θ as

$$\theta := \theta - \eta \nabla E_{\mathcal{B}}(\mathbf{X})$$



The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the mini-batch size is small, the computational cost at each iteration is light.

If the training data set has many redundant data instances, stochastic gradients may be so close to the true gradient $\nabla_\theta E(\mathbf{X})$ that a small number of iterations will find useful solutions to the optimization problem

Stochastic gradient descent can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling

Certain hardware processes mini-batches of specific sizes more efficiently.

See MLPC_sonar_example.ipynb