Intelligent Systems: Reasoning and Recognition

James L. Crowley

Ensimag 2 Winter Semester 2018-2019
Lesson 7 6 mars 2019

Artificial Neural Networks

Outline

Notation

x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$
$\{\vec{x}_m\}$ $\{y_m\}$	Training samples for learning.
M	The number of training samples.
$a_j^{(l)}$	the activation output of the j^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	the weight from unit i of layer $l–1$ to the unit j of layer l.
b_j^l	bias for unit j of layer l.
η	A learning rate. Typically very small (0.01). Can be variable.
L	The number of layers in the network.
$\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$	Output Error of the network for the m^{th} training sample
$\delta_{j,m}^{(l)}$	Error for the j^{th} neuron of layer l, for the m^{th} training sample.
$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)}\delta_{j,m}^{(l)}$	Update for weight from unit i of layer $l–1$ to the unit j of layer l.
$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$	Update for bias for unit j of layer l.

Introduction

Key Equations

Feed Forward from Layer i to j:

$$a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right)$$

Feed Forward from Layer j to k:

$$a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \right)$$

Back Propagation from Layer j to i:

$$\delta_{i,m}^{(l-1)} = \frac{\partial f(z_i^{(l-1)})}{\partial z_i^{(l-1)}} \sum_{j=1}^{N^{(l)}} w_{ij}^{(l)} \delta_{j,m}^{(l)}$$

Back Propagation from Layer k to j:

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

Weight and Bias Corrections for layer j:

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)}$$
$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

Network Update Formulas:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij,m}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_{j,m}^{(l)}$$

Artificial Neural Networks

Artificial Neural Networks, also referred to as "Multi-layer Perceptrons", are computational structures composed a weighted sums of "neural" units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

Note that the term "neural" is misleading. The computational mechanism of a neural network is only loosely inspired from neural biology. Neural networks do NOT implement the same learning and recognition algorithms as biological systems.

The approach was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model. During the 1950's, Frank Rosenblatt developed the idea to provide a trainable machine for pattern recognition, called a Perceptron. The first Perceptron, constructed in 1956, was a room-sized analog computer that learned recognition functions. However, both the learning algorithm and the resulting recognition algorithm are easily implemented as computer programs, and future Perceptrons were implemented as programs.

In 1969, Marvin Minsky and Seymour Papert of MIT published a book entitled "Perceptrons", that claimed to document the fundamental limitations of the perceptron approach. Notably, they demonstrated that a one-level perceptron could not be constructed to perform an "exclusive OR".

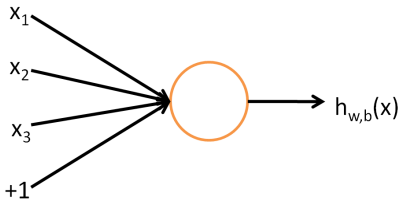
In the 1970s, frustrations with the limits of Artificial Intelligence research based on Symbolic Logic led a small community of researchers to explore the perceptron based approach. In 1973, Steven Grossberg, showed that a two layered perceptron could overcome the problems raised by Minsky and Papert, and solve many problems that plagued symbolic AI. In 1975, Paul Werbos developed an algorithm referred to as "Back-Propagation" that uses gradient descent to learn the parameters for perceptrons from classification errors with training data.

During the 1980's, Neural Networks went through a period of popularity with researchers showing that Networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, results were overtaken by more mathematically sound approaches for statistical pattern recognition such as support vector machines and boosted learning.

In 1998, Yves LeCun showed that convolutional networks composed from many layers could outperform other approaches for recognition problems. Unfortunately such networks required extremely large amounts of data and computation. Around 2010, with the emergence of cloud computing combined with planetary-scale data, training and using convolutional networks became practical. Since 2012, Deep Neural Networks have been shown to outperform other approaches for recognition tasks common to Computer Vision, Speech and Robotics. Training such networks is made possible by high-performance parallel computing (using GPUs) using massive scale data now available from the World Wide Web. A rapidly growing research community currently seeks to extend the application beyond recognition to generation of speech, images, video-sequences and robot actions.

The Artificial Neuron

The simplest possible neural network is composed of a single neuron.



A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of a hypothesis, $h_{w,b}()$

$$a = h_{\vec{w},b}(\vec{X})$$

The neuron is composed of a weighted sum of input values

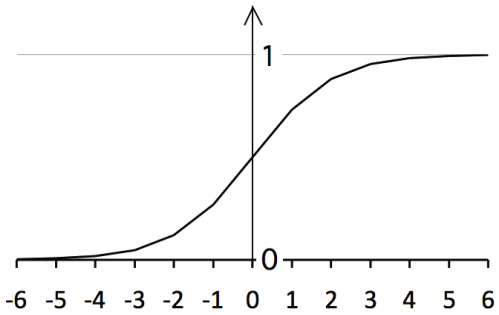
$$z = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b$$

followed by a non-linear "activation" function, $f(z)$ (sometimes written $\phi(z)$)

$$a = h_{\vec{w},b}(\vec{X}) = f(\vec{w}^T \vec{X} + b)$$

Many different activation functions may be used.

A popular choice for activation function is the sigmoid: $f(z) = \frac{1}{1+e^{-z}}$



This function is useful because the derivative is: $\frac{df(z)}{dz} = f(z)(1 - f(z))$

This gives a decision function: if $h_{\vec{w},b}(\vec{X}) > 0.5$ POSITIVE else NEGATIVE

Other popular decision functions include the hyperbolic tangent and the softmax.

The hyperbolic Tangent: $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

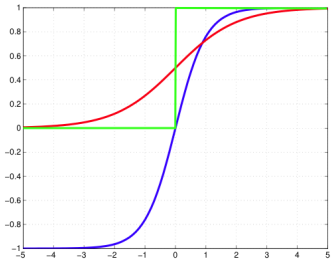
The hyperbolic tangent is a rescaled form of sigmoid ranging over [-1,1]

We could use the step function: $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$

Or the sgn function: $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$

However these is not differentiable, and we need a derivative for backpropagtion or for gradient descent.

Plot of Sigmoid (red), Hyperbolic Tangent (Blue) and Step Function (Green)



The softmax function is often used for multi-class networks. For K classes:

$$f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

The rectified linear function is popular for deep learning because of a trivial derivative:

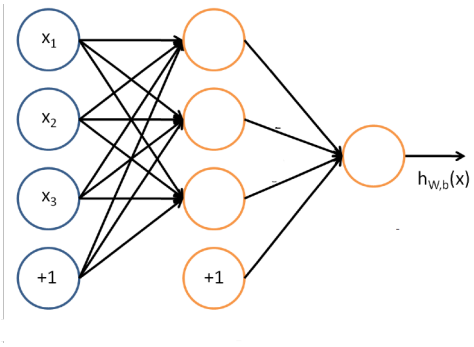
Relu: $f(z) = \max(0, z)$

While Relu is discontinuous at z=0, for $z > 0$: $\frac{df(z)}{dz} = 1$

Note that the choice of decision function will determine the target variable “y” for supervised learning.

The Neural Network model

A neural network is a multi-layer assembly of neurons. For example, this is a 2-layer network:



The circles labeled +1 are the bias terms.
The circles on the left are the input terms. Some authors, notably in the Stanford tutorials, refer to this as Level 1.

We will NOT refer to this as a level (or, if necessary, level L=0).
The rightmost circle is the output layer, also called L.
The circles in the middle are referred to as a "hidden layer". In this example there is a single hidden layer and the total number of layers is L=2.

The parameters carry a superscript, referring to their layer.
We will use the following notation:

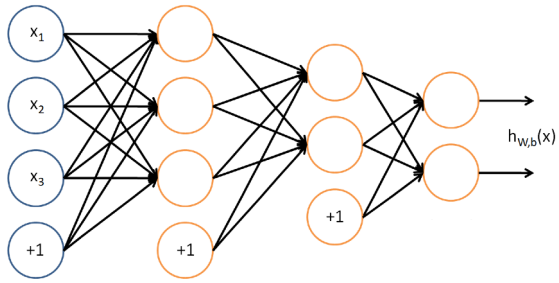
L	The number of layers (Layers of non-linear activations).
l	The layer index. l ranges from 0 (input layer) to L (output layer)
$N^{(l)}$	The number of units in layer l. $N^{(0)}=D$
$a_j^{(l)}$	The activation output of the j^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	The weight from the unit i of layer l-1 for the unit j of layer l.
$b_j^{(l)}$	The bias term for j^{th} unit of the l^{th} layer
$f(z)$	A non-linear activation function, such as a sigmoid, tanh, or soft-max

For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer.
$W_{13}^{(2)}$ is the weight for neuron 1 from the first level to neuron 3 in the second level.

The above network would be described by:

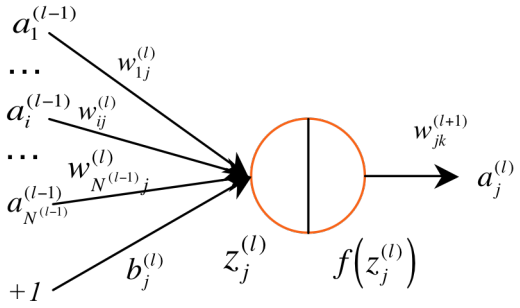
$$a_1^{(1)} = f(w_{11}^{(1)}X_1 + w_{21}^{(1)}X_2 + w_{31}^{(1)}X_3 + b_1^{(1)})$$
$$a_2^{(1)} = f(w_{12}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{32}^{(1)}X_3 + b_2^{(1)})$$
$$a_3^{(1)} = f(w_{13}^{(1)}X_1 + w_{23}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$
$$h_{\vec{w},b}(\vec{X}) = a_1^{(2)} = f(w_{11}^{(2)}a_1^{(1)} + w_{21}^{(2)}a_2^{(1)} + w_{31}^{(2)}a_3^{(1)} + b_1^{(2)})$$

This can be generalized to multiple layers. For example:



$\vec{h}(\vec{x}_m)$ is the vector of network outputs (one for each class).

Each unit is defined as follows:



The notation for a multi-layer network is

$\vec{a}^{(0)} = \vec{X}$ is the input layer. $a_i^{(0)} = X_d$

l is the current layer under discussion.

$N^{(l)}$ is the number of activation units in layer l. $N^{(0)} = D$

i,j,k Unit indices for layers l-1, l and l+1: $i \rightarrow j \rightarrow k$

$w_{ij}^{(l)}$ is the weight for the unit i of layer l-1 feeding to unit j of layer l.

$a_j^{(l)}$ is the activation output of the j^{th} unit of the layer l

$b_j^{(l)}$ the bias term feeding to unit j of layer l.

$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$ is the weighted input to j^{th} unit of layer l

$f(z)$ is a non-linear decision function, such as a sigmoid, tanh(), or soft-max

$a_j^{(l)} = f(z_j^{(l)})$ is the activation output for the j^{th} unit of layer l

For layer l this gives:

$$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \qquad a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right)$$

and then for l+1 :

$$z_k^{(l+1)} = \sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \qquad a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \right)$$

It can be more convenient to represent the network using vectors:

$$\vec{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{N_l}^{(l)} \end{bmatrix} \qquad \vec{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{N_l}^{(l)} \end{bmatrix}$$

and to write the weights and bias at each level l as a k by j Matrix,

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{1i}^{(l)} & \cdots & w_{1N^{(l-1)}}^{(l)} \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ w_{j1}^{(l)} & \cdots & w_{ji}^{(l)} & \cdots & w_{jN^{(l-1)}}^{(l)} \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ w_{N^{(l)}1}^{(l)} & \cdots & w_{N^{(l)}i}^{(l)} & \cdots & w_{N^{(l)}N^{(l-1)}}^{(l)} \end{pmatrix} \vec{b}^{(l)} = \begin{pmatrix} b_1^l \\ \vdots \\ b_i^l \\ \vdots \\ b_{N^{(l-1)}}^l \end{pmatrix}$$

(note: To respect matrix notation, we have reversed the order of i and j in the subscripts.)

We can see that the weights are a 3rd order Tensor or vector of matrices, with one matrix for each layer, The biases are a matrix (vector of vectors) with a vector for each level.

$$\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)} \quad \text{and} \quad \vec{a}^{(l)} = f(\vec{z}^{(l)}) = f(W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)})$$

We can assemble the set of matrices $W^{(l)}$ into an 3rd order Tensor (Vector of matrices), W, and represent $\vec{a}^{(l)}$, $\vec{z}^{(l)}$ and $\vec{b}^{(l)}$ as matrices (vectors of vectors): A, Z, B.

So how to do we learn the weights W and biases B?

We could train a 2-class detector from a labeled training set $\{\vec{x}_m\},\{y_m\}$ using gradient descent. For more than two layers, we will need to use the more general "back-propagation" algorithm.

Backpropagation

Back-propagation adjusts the network the weights $w_{ij}^{(l)}$ and biases $b_j^{(l)}$ so as to minimize an error function between the network output $\vec{a}_m^L$ and the target value $\vec{y}_m$ for the M training samples $\{\vec{x}_m\}$, $\{\vec{y}_m\}$.

This is an iterative algorithm that propagates an error term back through the hidden layers and computes a correction for the weights at each layer so as to minimize the error term.

This raises two questions:
1) How do we initialize the weights?
2) How do we compute the error term for hidden layers?

1) How do we initialize the weights?

A natural answer for the first question is to initialize the weights to 0.

By experience this causes problems. If the parameters all start with identical values, then the algorithm can end up learning the same value for all parameters. To avoid this, we initialize the parameters with a small random variable that is near 0, for example computed with a normal density with variance ε (typically 0.01).

$\forall_{i,j} w_{ji}^{(l)} = \mathcal{N}(X;0,\varepsilon)$ and $\forall_{j} b_j^{(l)} = \mathcal{N}(X;0,\varepsilon)$ where $\mathcal{N}$ is a sample from a normal density.

An even better solution is provided by Xavier GLORIOT's technique (see course web site on Xavier normalization). However that solution is too complex for today's lecture.

2) How do we compute the error term?

Back-propagation propagates the error term back through the layers, using the weights. We will present this for individual training samples. The algorithm can easily be generalized to learning from sets of training samples (Batch mode).

Given a training sample, $\vec{x}_m$, we first propagate the $\vec{x}_m$ through the L layers of the network (Forward propagation) to obtain a hypothesis $\vec{h}(\vec{x}_m;W,B) = \vec{a}^{(L)}$.

We then compute an error term. In the case, of a multi-class network, this is a vector, with k components, one output for each hypothesis. In this case the indicator vector would be a vector, with one component for each possible class:

$$\vec{\delta}_m^{out} = -\left(\vec{y}_m - \vec{a}_m^{(L)}\right) \quad \text{or for each class k:} \quad \delta_{k,m}^{out} = -\left(y_{k,m} - a_{k,m}^{(L)}\right)$$

The error term $\vec{\delta}_m^{out}$ is the total error for the whole network for sample m.

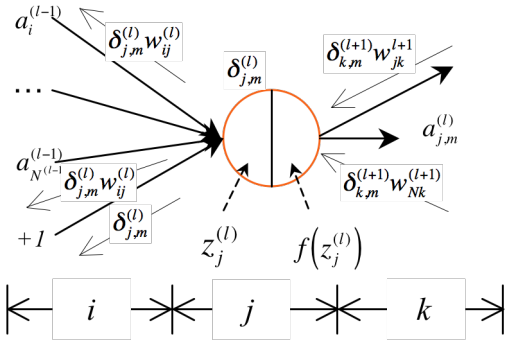
To keep things simple, let us consider the case of a two class network, so that δ_m^{out}, $h(\vec{X}_m)$, $a_m^{(L)}$, and y_m are scalars. The results are easily generalized to vectors for multi-class networks. At the output layer, the "error" for each training sample is:

$$\delta_m^{out} = -\left(y_m - a_m^{(L)}\right) = \left(a_m^{(L)} - y_m\right)$$

The error term for layer L is then:

$$\delta_m^{(L)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \delta_m^{out}$$

For the hidden units in layers $l < L$ the error $\delta_j^{(l)}$ is based on a weighted average of the error terms for $\delta_k^{(l+1)}$.



We compute error terms, $\delta_j^{(l)}$ for each unit j in layer l back to layer $l–1$ using the sum of errors times the corresponding weights times the derivative of the activation function.

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{l+1}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

This error term tells how much the unit j was responsible for differences between the activation of the network $\vec{h}(\vec{x}_m; w_{jk}^{(l)}, b_k^{(l)})$ and the target value $\vec{y}_m$.

For the sigmoid activation function. $f(z) = \frac{1}{1+e^{-z}}$ the derivative is: $\frac{df(z)}{dz} = f(z)(1 - f(z))$

For $a_j^{(l)} = f(z_j^{(l)})$ this gives:

$$\delta_{j,m}^{(l)} = a_{j,m}^{(l)}(1 - a_{j,m}^{(l)}) \cdot \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

This error term can then used to correct the weights and bias terms leading from layer j to layer i.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)}$$
$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

Note that the corrections $\Delta w_{ij,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are NOT applied until after the error has propagated all the way back to layer $l=1$, and that when $l=1$, $a_i^{(0)} = x_i$.

For "batch learning", the corrections terms, $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are averaged over M samples of the training data and then only an average correction is applied to the weights.

$$\Delta w_{ij}^{(l)} = \frac{1}{M} \sum_{m=1}^{M} \Delta w_{ij,m}^{(l)} \qquad \Delta b_j^{(l)} = \frac{1}{M} \sum_{m=1}^{M} \Delta b_{j,m}^{(l)}$$

then

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij}^{(l)} \qquad b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)}$$

where η is the learning rate.

Back-propagation is equivalent to computing the gradient of the loss function for each layer of the network. A common problem with gradient descent is that the loss function can have local minimum. This problem can be minimized by regularization. A popular regularization technique for back propagation is to use "momentum"

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij}^{(l)} + \mu \cdot w_{ij}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)} + \mu \cdot b_j^{(l)}$$

where the terms $\mu \cdot w_j^{(l)}$ and $\mu \cdot b_j^{(l)}$ serves to stabilize the estimation.

The back-propagation algorithm may be continued until all training data has been used. For batch training, the algorithm may be repeated until all error terms, $\delta_{j,m}^{(l)}$, are a less than a threshold.

Summary of Backpropagation

The Back-propagation algorithm can be summarized as:

1) Initialize the network and a set of correction vectors:

$$\begin{aligned}
&\forall_{i,j,l} w_{ji}^{(l)} = \mathcal{N}(X;0,\varepsilon) \\
&\forall_{i,l} b_j^{(l)} = \mathcal{N}(X;0,\varepsilon) \\
&\forall_{i,j,l} \Delta w_{ji}^{(l)} = 0 \\
&\forall_{i,l} \Delta b_j^{(l)} = 0
\end{aligned}$$

where $\mathcal{N}$ is a sample from a normal density, and ε is a small value.

2) For each training sample, $\vec{x}_m$, propagate $\vec{x}_m$ through the network (forward propagation) to obtain a network activation $a_m^{(L)}$. Compute the error and propagate this back through the network:

a) Compute the network error term: $\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$

b) Compute the error term at Layer L: $\delta_m^{(L)} = \dfrac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \delta_m^{out}$

c) Propagate the error back from *l=L-1* to *l=1*: $\delta_{j,m}^{(l)} = \dfrac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$

d) Use the error at each layer to set a vector of correction weights.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)} \qquad \Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

3) For all layers, *l=1 to L*, update the weights and bias using a learning rate, η

$$\begin{aligned}
w_{ij}^{(l)} &\leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij,m}^{(l)} + \mu \cdot w_{ij}^{(l)} \\
b_j^{(l)} &\leftarrow b_j^{(l)} - \eta \cdot \Delta b_{j,m}^{(l)} + \mu \cdot b_j^{(l)}
\end{aligned}$$

Note that this last step can be done with an average correction matrix obtained from many training samples (Batch mode), providing a more efficient algorithm.