

Training a Neural Network - A Numerical Example

Raphael B. Alampay

December 11, 2017

Abstract

Neural networks are models used to approximate a discriminative function for classification in a supervised learning fashion. You have a bunch of input in the form of n -dimensional numerical vectors that represent features of certain entities you'd like to teach the network to classify (example 1000 emails classified as spam and 1000 emails classified non-spam). This paper takes a look at a quantitative approach in training a neural network given actual numerical examples. This will allow the student/practitioner to understand the fundamentals of the training process namely feed forward and backpropagation. The paper is intended to be light in concept with specific examples for people getting into machine learning with neural networks.

1 Introduction

Neural networks are biologically inspired computational models that attempts to solve classification problems. It is composed on **neurons** which holds and processes values in the network. You can think of these values as signal strengths that aim to mimic how chemical reactions occur in the brain. The higher the value, the stronger the signal. In biology, neurons transmit and receive signals to and from other neurons by means of dendrites. These propagations are modelled in the neural network by means of weight values. Since a neuron may receive values from more than one neuron, it accounts all the weights connected to it before attempting to fire a signal thus simulating how we "react" to certain stimuli. Training the neural network roughly means looking for the optimal values for these weights based on what we already know in order for the model to properly "react" to a certain input.

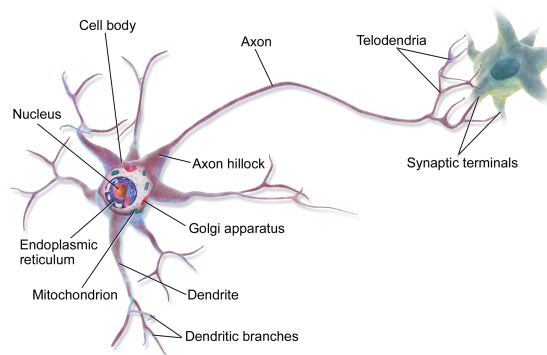


Figure 1: An example of a neuron taken from wikipedia.

There are typically 3 layers in a classical neural network - **input layer**, **hidden layer/s** and **output layer**. These layers are connected to one another via neurons' weights in an adjacent manner. This means that a layer can only be connected to at most two adjacent layers. The **input layer** will always be the leftmost layer, the **output layer** will always be the rightmost layer while everything in between will be **hidden layers**. The following example is an illustration of the classical neural network:

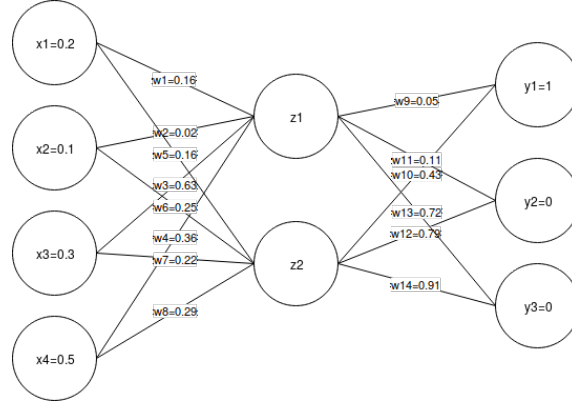


Figure 2: A neural network with 4 neurons in the input layer, 2 for hidden and 3 for output

1.1 Input Layer

The input layer contains neurons that represent the input values or what the network initially receives from the real world. These values are features that represent a classification/label that we'd like to recognize. In the mathematical model $f(x) = y$, this would be the x as an n -dimensional vector. For example, if we'd like to tell the neural network that we're looking at an image of a face represented by a matrix of pixel values, and suppose the size of the image is 32×32 pixels, the input layer would have a total of 1024 neurons each one corresponding to a pixel value. We refer to this vector as a **feature vector**.

1.2 Hidden Layer/s

From the input layer, information is passed to a hidden layer which contains neurons that processes signals it receives from the its adjacent layer/s (either from the left or from the right). A neural network can have more than one hidden layer. Neurons in the hidden layer are often denoted as z_i which we refer to as **latent variables**.

1.3 Output Layer

The output layer contains neurons that represent the output of the network or the result/reaction of the network after receiving and processing the input (from input layer to hidden layer then finally to the output layer). The easiest way to model neurons in the output layer is to treat each one (neuron) as a classification/label that the network is trying to recognize with a value ranging from 0 to 1. The closer the value to 1 for an output neuron, the closer it is to thinking that it is that classification or label for a given input x . For example, let's say we're trying to learn how to differentiate cats from dogs from any other animal. The set of possible outputs (cat, dog or others) can be represented by:

$$Y = [y_1 \quad y_2 \quad y_3] \quad (1)$$

where y_1 represents the label cat, y_2 dog and y_3 others. A cat then for a neural network would look like $f(x) = [1 \quad 0 \quad 0]$, a dog would be $f(x) = [0 \quad 1 \quad 0]$ and finally any other animal $f(x) = [0 \quad 0 \quad 1]$.

2 Training

To train a neural network means to optimize the set of weights (connections between neurons) in such a way that when we give it a feature vector representing a cat, the output should be close to $f(x) = [1 \quad 0 \quad 0]$. In order to do this, at the beginning of training, the weights are randomly initialized. We then feed it a feature vector relating to cat and see what the Y value is — how close did the network get to the actual answer. This closeness value can be measured by a **loss function**. A higher value for this function means that the network yielded a higher error while smaller value means that the network is getting pretty close to recognizing what the input is. We

will use this error/closeness value to adjust the weights accordingly. The first step in this entire training process is called **feed forward**.

For the rest of the discussion, we will be referring to figure 2 as reference. Initially, the weights of the network will be as follows:

$$W_1 = \begin{bmatrix} 0.16 & 0.16 \\ 0.02 & 0.25 \\ 0.63 & 0.22 \\ 0.36 & 0.29 \end{bmatrix} \quad (2)$$

$$W_2 = \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix} \quad (3)$$

2.1 Feed Forward

The feed forward process can be thought of the movement of information from the input layer in the form of the feature vector's values to the first hidden layer and finally making the guess (classification) in the output layer. Given two consecutive layers, information is passed from the left layer to the right layer by performing a matrix multiplication operation between the left layer's neurons and the weight matrix between the left and right layer. The resulting matrix should have the same size as the neurons in the right layer (1 row with n columns where n is the size of the neurons in the right layer). Let's take the following illustration:

We're trying to solve for the values of Z by matrix multiplying X with W_1 . The result will then be passed to an activation function such as the **sigmoid function** given by:

$$a(x) = \frac{1}{(1 + e^{-x})} \quad (4)$$

Mathematically, we can then represent information flow from X to Z :

$$Z = a((X)(W_1)) \quad (5)$$

Numerically we have the following:

$$Z = \begin{bmatrix} 0.2 & 0.1 & 0.3 & 0.5 \end{bmatrix} \begin{bmatrix} 0.16 & 0.16 \\ 0.02 & 0.25 \\ 0.63 & 0.22 \\ 0.36 & 0.29 \end{bmatrix} \quad (6)$$

$$Z = \begin{bmatrix} 0.403 & 0.268 \end{bmatrix} \quad (7)$$

$$Z = a(\begin{bmatrix} 0.403 & 0.268 \end{bmatrix}) \quad (8)$$

$$Z = \begin{bmatrix} 0.60 & 0.57 \end{bmatrix} \quad (9)$$

Using the same approach, we feed forward Z 's values towards Y by matrix multiplying it with W_2 .

Mathematically:

$$Y = a((Z)(W_2)) \quad (10)$$

Numerically:

$$Y = \begin{bmatrix} 0.60 & 0.57 \end{bmatrix} \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix} \quad (11)$$

$$Y = \begin{bmatrix} 0.28 & 0.65 & 0.95 \end{bmatrix} \quad (12)$$

$$Y = a(\begin{bmatrix} 0.28 & 0.65 & 0.95 \end{bmatrix}) \quad (13)$$

$$Y = \begin{bmatrix} 0.57 & 0.66 & 0.72 \end{bmatrix} \quad (14)$$

2.2 Loss Function

The loss function determines how far (or close) the guess of the network (\hat{Y}) to the actual classification value (Y). Remember that we want to teach the network how to recognize a certain classification by adjusting the its weights. The amount of adjustment we do will largely depend on the value given by the **loss function**. For this paper, we will be using a very simple loss function:

$$E(Y, \hat{Y}) = \sum_i^n \frac{1}{2} (Y_i - \hat{Y}_i)^2 \quad (15)$$

In the case of our example, $\hat{Y} = [1 \ 0 \ 0]$, the value of our loss function will be computed as:

$$E(Y, \hat{Y}) = \frac{1}{2}(0.57 - 1)^2 + \frac{1}{2}(0.66 - 0)^2 + \frac{1}{2}(0.72 - 0)^2 \quad (16)$$

$$e_1 = \frac{1}{2}(0.57 - 1)^2 \quad (17)$$

$$e_2 = \frac{1}{2}(0.66 - 0)^2 \quad (18)$$

$$e_3 = \frac{1}{2}(0.72 - 0)^2 \quad (19)$$

$$E(Y, \hat{Y}) = e_1 + e_2 + e_3 \quad (20)$$

$$E(Y, \hat{Y}) = 0.09245 + 0.2178 + 0.2592 \quad (21)$$

$$E(Y, \hat{Y}) = 0.56945 \quad (22)$$

2.3 Back Propagation

Once we get the error value from the loss function, we can use this value to determine how much change we would apply to the weights to minimize this error value. This is done by a process called back propagation which takes the individual error values and cascades it from the output layer to the input layer seeing how much damage the weight values contributed to the overall error. Integral to this process is to solve for **gradients**. These gradients are approximation of derivatives. As with derivatives, gradient values dictate the direction of the error function of the network. If we know these values, we can roughly determine how to adjust our weights to minimize the error. To simplify things, since weight values drive the error value, we'd like to determine the gradient values for each neuron (since weights are attached to neurons) starting from the output layer (thus backward propagation) to give us the **delta weights** or how much magnitude should we adjust the original weights to lower the error. Numerically, this means that the size of our gradient vector will be the same size of neurons in a given layer.

For a more specific example, we'll break down this process into two major operations. The first part will perform back propagation starting from the output to the last hidden layer. And the second part will be from the last hidden layer down to the input layer. Similar to feed forward, we will be computing values with 3 inputs – 2 layers and the weight matrix in between them. For each pair of layers, the gradient values to be computed for the right layer.

2.3.1 BP from Output to Last Hidden Layer

To start off, we compute the gradients for the right layer in this part of BP which in this case is the output layer (Y). Gradients are computed by taking the product of the first derivatives of equations used in the model. Take the **loss function** for each y_i for instance:

$$e_i = \frac{1}{2}(y_i - \hat{y}_i)^2 \quad (23)$$

The first derivative e'_i for the **loss function** would be:

$$e'_i = \hat{y}_i - y_i \quad (24)$$

We also take the derivative for the neurons in the right layer (output layer in this case) which we will refer to as Y' . The activation function we used was:

$$a(x) = \frac{1}{1 + e^{-x}} \quad (25)$$

The derivative a given output y'_i would approximately be:

$$y'_i = a(x)(1 - a(x)) \quad (26)$$

$$y'_i = y_i(1 - y_i) \quad (27)$$

Given these derivatives, we can then compute for our gradients G_h (with a one to one correspondence to the right layer / output layer in this case). We can do this by using the following equation:

$$G = [(e'_1)(y'_1) \quad (e'_2)(y'_2) \quad (e'_3)(y'_3)] \quad (28)$$

Plugging in the necessary values we have:

$$G = [(0.09245)(0.2451) \quad (0.2178)(0.2244) \quad (0.2592)(0.2016)] \quad (29)$$

$$G = [0.022679495 \quad 0.04887432 \quad 0.052255] \quad (30)$$

Once we have the gradient values, we can use these to compute for the the change in weight δW_2 which will subtract from the original weights to get the updated weights \widehat{W}_2 (we're at index 2 because again we're moving from the last layer down to the input layer). At this part of the process, delta weights can be computed by multiplying the transpose of our gradients G with the output of the left layer – in this case Z . We then transpose the result to have the same shape as W_2 .

$$\delta W_2 = (G^T Z)^T \quad (31)$$

Plugging in the values we have:

$$\delta W_2 = \left(\begin{bmatrix} 0.022679495 \\ 0.04887432 \\ 0.052255 \end{bmatrix} [0.60 \quad 0.57] \right)^T \quad (32)$$

$$\delta W_2 = \left(\begin{bmatrix} 0.013596 & 0.012916 \\ 0.029325 & 0.027858 \\ 0.031353 & 0.029785 \end{bmatrix} \right)^T \quad (33)$$

$$\delta W_2 = \begin{bmatrix} 0.013596 & 0.029325 & 0.031353 \\ 0.012916 & 0.027858 & 0.029785 \end{bmatrix} \quad (34)$$

Finally we can update the weights from Z to Y :

$$\widehat{W}_2 = W_2 - \delta W_2 \quad (35)$$

$$\widehat{W}_2 = \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix} - \begin{bmatrix} 0.013596 & 0.029325 & 0.031353 \\ 0.012916 & 0.027858 & 0.029785 \end{bmatrix} \quad (36)$$

$$\widehat{W}_2 = \begin{bmatrix} 0.036404 & 0.300675 & 0.688647 \\ 0.417084 & 0.762142 & 0.880215 \end{bmatrix} \quad (37)$$

2.3.2 BP from Last Hidden to Input Layer

Computing the gradients and updated weights from the last hidden layer Z down to the input layer X is a bit different but generally follows the same process. Probably the major difference in this step of back propagation is to apply the gradients computed in the last operation (gradients corresponding to Y) as part of the computation. We shall refer to this as G_p where:

$$G_p = G \quad (38)$$

We then have to solve for a new G which corresponds to the gradients of the right layer in this operation (Z). Aside from taking the previous operation's gradients, we have to also account for the old weights in the previous operation (W_2). We then matrix multiply G_p with the transpose of W_2 to give us the same shape as the right layer Z . This result will then be multiplied element-wise with the derivatives of $Z - Z'$. The operation would then be as follows:

$$G = (G_p W_p^T) \times Z' \quad (39)$$

Let's solve for the derivatives Z' first using the same derivative equation as Y' :

$$Z' = [z'_1 \quad z'_2] = [(0.60)(1 - 0.60) \quad (0.57)(1 - 0.57)] = [0.24 \quad 0.2451] \quad (40)$$

Next we solve for $G_p W_p^T$:

$$G_p = [0.022679495 \quad 0.04887432 \quad 0.052255] \quad (41)$$

$$W_p^T = \begin{bmatrix} 0.05 & 0.43 \\ 0.33 & 0.79 \\ 0.72 & 0.91 \end{bmatrix} \quad (42)$$

$$G_p W_p^T = [0.022679495 \quad 0.04887432 \quad 0.052255] \begin{bmatrix} 0.05 & 0.43 \\ 0.33 & 0.79 \\ 0.72 & 0.91 \end{bmatrix} \quad (43)$$

$$G_p W_p^T = [0.054885 \quad 0.095906] \quad (44)$$

Finally putting them all together to solve for $G = (G_p W_p^T) \times Z'$:

$$G = [0.054885 \quad 0.095906] \times [0.24 \quad 0.2451] = [0.0131723757 \quad 0.02350658] \quad (45)$$

We can then extract the delta weights δW_1 using the following:

$$\delta W_1 = X^T G_h \quad (46)$$

Plugging in the values we have:

If you have a deeper neural network, then we can apply the equations above to the next consecutive layers.

To test if we have indeed trained the network, we'll use the same input and perform a feed forward using the updated weights. Our neural network will now look like the following:

This can be mathematically expressed:

$$Z = a(XW_1) \quad (47)$$

$$Y = a(ZW_2) \quad (48)$$

Plugging in the values we have:

Finally we use the **loss function** to see if the network has improved (lower total error will indicate improvement):