Hochschule Esslingen

Deep Learning Demonstration

Traffic Sign Recognition

Author:
Sergio Izquierdo

Supervisor:
Dr. Dominik Schoop

December 7, 2017

Hochschule Esslingen
University of Applied Sciences

Abstract

caca

Contents

From the begining of computing, the machines have solve probems incredibly difficult for a human, problems based on mathematicals rules. But what has been a really difficult task for computer was to solve some problems that are simple and easy for humans such as images or speech recognition. Tasks that a person perform easily but that are difficult to describe formally.

A person needs a big amount of knowledge to live and to understand, and many times this knowledge cannot be described in a formal way. That means that if we want the computers to performs the same tasks as an human, that computer should acquire that knowledge, by learning.

If it is said that a computer learns that means that it understands a complex concept as a hierachy of simpler concepts. The hierachy of concepts is deep, with many layers composed of concepts, and hence the term of deep learning.

To achieve this approach Neural Networks are going to be used. They are copying the behaviour of a brain neural network. Multiple layers of neurons that are connected to each others and pass information through those connections. This model, allows the computer to learn since the neurons adapt the output for a given input depending on the expected answer. That means that with a big amount of data to learn, the network could predict a correct answer for the input. [1]

Chapter 1

Neural Networks

The brain has been studied deeply during the last century. The researchers wanted to use the power of the brain in a mathematical or computational model. The first step in the Deep Learning was to copy the behaviour of the brain neurons to a circuit (1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts). The poor computing capacity of the time stopped the studies on the topic.

In the 80s the field became again interesting with the inclusion of multiple layered neural networks. In 1986 researchers modeled the idea of back propagation. This idea helps the network to distribute pattern recognition errors throughout the whole network. The problem is that with this algorithm the net learn slowly, so many iterations are needed.

Nowadays Neural Networks are incredibly important. The big data and the computing capacity are a perfect base for the networks. Also with new types of models such as Convolutional Neural Networks or Recurrent Neural Networks they are solving amazing problems beyond the image recognition. [7] [9]

1.1 What is a Neural Network

You could think of a Neural Network as a box with a certain number of inputs and outputs. For a given input the box answer with a output, that is the network prediction.

Let us imagine that we have one box of this kind, and if you input a image of a handwritten digit, the output of the box will be the digit of the picture. You could think that if you create one box like this you have all your work done, but the box by itself is stupid. If you create a box and feed it with the picture of a '1' the output could be whatever.

What we should do is to teach the net so it could answer correctly. To achieve it our box has several knobs that we can regulate in order to tell the box how bad was the prediction it said. In that way, after correcting the knobs several times, the box has learned, what means that it is ready for answer right. Now we have a box with the knowledge to make good predictions, but since it is a box we cannot outcome any knowledge for us. We cannot get any method to create an algorythm that solve our task more efficiently. We just can use our box, and get our answer from our box.

1.2 Under the Hood

Now that we have understood our box it is time to start knowing what is inside it. The neural network is composed by several layers. At least we have the input layer (receive the data from outside) and the output layer (give us the prediction). Between those two could be more hidden layers. Each of them has a certain number of neurons $n_1, n_2, .., n_m$ where m is the number of layers. That means that the input layer has n_1 neurons while the output layer has n_m neurons. Each neuron of the first layer is connected with each neuron of the second layer, each neuron of the second layer is connected with each neuron of the third layer and so on (See Figure 1.1).

Each of those connections has a weight w. To denote these weights we are going to use the form $w_{ij}^{(k)}$ where k means which layer the connection departs from, i means which neuron of layer k the connection departs from and j means which neuron of layer $k+1$ the connections arrives to (See Figure 1.1). Therefore w is:

$$\begin{aligned} & w_{ij}^{(k)} \\ & i = 1, \ldots, n_k \\ & j = 1, \ldots, n_{k+1} \\ & k = 1, \ldots, m \end{aligned} \tag{1.1}$$

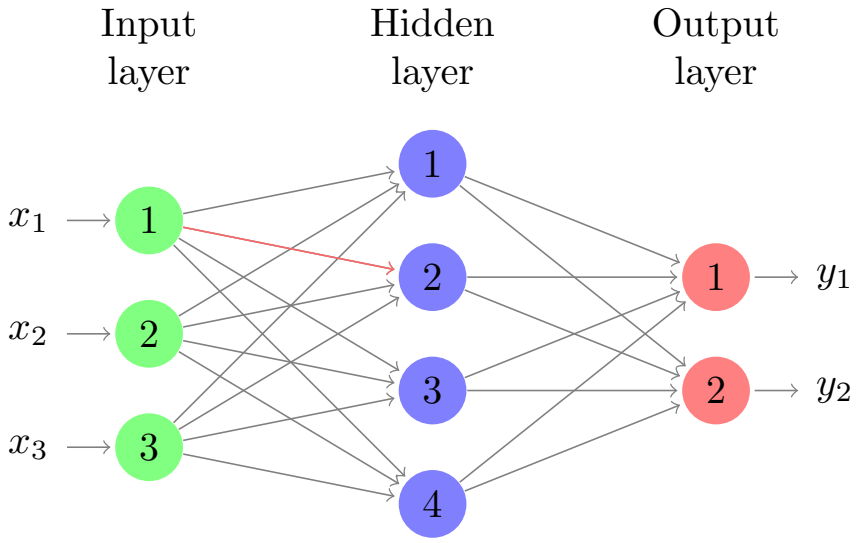

Figure 1.1: Example of Neural Network with 3 layers. Here $n_1 = 3$, $n_2 = 4$ and $n_3 = 2$. The connection in red is denoted as $w_{12}^{(1)}$ (according to Equation 1.1)

Each neuron has a bias that allow set how excited is a neuron. A really excited neuron always output '1' while a non-excited neuron always give us '0'. The bias helps the network to adjust some neurons as important since not every neuron output is equally interesting. The work of a single neuron is as simple as add the bias and the weighted inputs and apply a function to the result [2, Chapter 27]. The output o of a neuron is:

$$o_i^{(k)} = \begin{cases} x_i, & k = 1 \\ f\left(u_i^{(k)} + \sum_{j=1}^{n_{k-1}} w_{ji}^{(k-1)} \cdot o_j^{(k-1)}\right), & k > 1 \end{cases} \tag{1.2}$$

For example, for the neuron 3 in blue of Figure 1.1 the output will be $o_3^{(2)} = f(u_3^{(2)} + w_{13}^{(1)} \cdot o_1^{(1)} + w_{23}^{(1)} \cdot o_2^{(1)} + w_{33}^{(1)} \cdot o_3^{(1)})$. For the same neuron the scheme is showed in Figure 1.2.

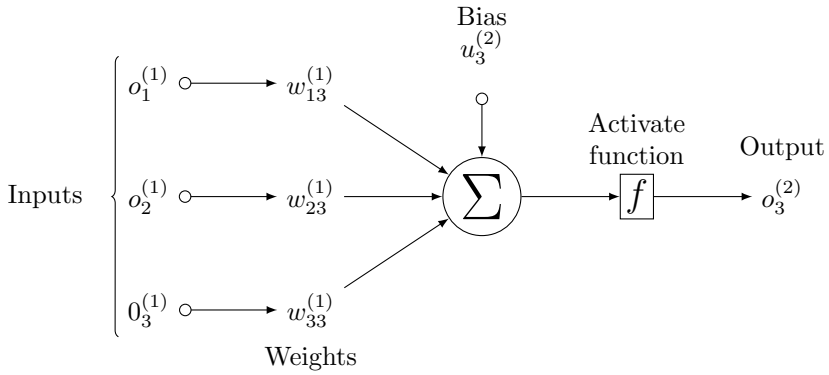

Figure 1.2: Example of the task of a single neuron [3]

1.3 Back Propagation

Now a Neural Network could be programmed with all showed on 1.2, a program that output y_1, and y_2 could be developed. But, where is the "learning" of deep learning here? Nowhere, yet. Is the moment of teaching the net with examples, giving it some inputs and showing it the expected outputs. The *Back Propagation Algorithm* is going to achieve it. The goal of this method is to minimize the error using the method of gradient descent.

1.3.1 Activation Function

As explained before each neuron has to apply one function to the result of its computing. One of the most popular functions used for this purpose is the sigmoid:

$$s(x) = \frac{1}{1 + e^{-cx}} \tag{1.3}$$

The shape of this function depends on the value of c, but for simplicity $c = 1$ is going to be used. The derivate of the sigmoid function respect of x is $s(x)' = s(x)(1 - s(x))$. One of the reasons why this function is used is the simplicity of the derivate, because it allows to make all the derivates later more easily [6, Chapter 7].

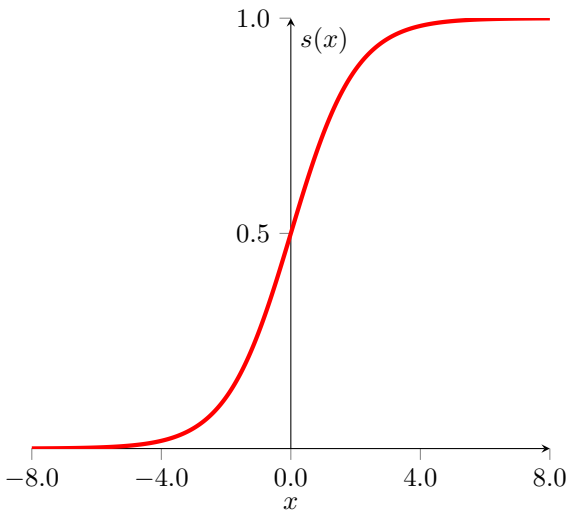

Figure 1.3: Shape of the sigmoid with $c = 1$

1.3.2 The Error

The error could be measured in differents ways. The most common and apparently most effective method is to calculate the Euclidean distance. It allows to measure the error in different dimensions, for example, the error in $\mathbb{R}^1$ between 4 and 1 is $E = 4-3 = 1$. For $\mathbb{R}^2$ you should use the Pythagorean theorem, and for $\mathbb{R}^n$ the distance is:

$$E_{(p,q)} = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2} \quad (1.4)$$

Is important to know how to size the error that the net has because you have to tell the network how far was the result from the expected output. And only once you have the distance between the target and the output you can adjust the knobs to achieve a better result.

How much the knob must be changed? It depends on how important is that knob for the result. For example it might be possible that changing one knob will get the same result. In this case it does not matter if we adjust the knob, but, what about one knob that when adjusted change the result drastically? Here the new position of the knob is realy important. Hence we need the partial error with respect a specific weight or bias [4, Chapter 2]:

$$\frac{\partial e}{\partial w_{ij}^{(k)}} \quad \text{or} \quad \frac{\partial e}{\partial u_i^{(k)}} \quad (1.5)$$

1.3.3 The Correction

Once we can calculate the error depending on one parameter we are ready to make the correction to the network. The process is simple, first you feed the input layer with some data, then you get the result and compare it with the target and finally you make this adjustment for each weight and bias (the α is the learning constant and represent how far is the next step in the gradient descent) [6, Chapter 7]:

$$\begin{aligned} w_{ij}^{(k)} &\Rightarrow \quad w_{ij}^{(k)} - \alpha \frac{\partial e}{\partial w_{ij}^{(k)}} \\ u_{i}^{(k)} &\Rightarrow \quad u_{i}^{(k)} - \alpha \frac{\partial e}{\partial u_{i}^{(k)}} \end{aligned} \tag{1.6}$$

To develop a working neural network is now possible with all this knwoledge. First we have to declare the architecture of our net. Then we set random weights and biases and we start teaching the net. Imagine you have a 10.000 images dataset, then you take 90% of those images to train the net, and the other 10% for testing how well we have developed it. We feed the network with one of those images from the train set and make the correction as we have seen in 1.6. Ideally we should do it with all the train set but as the size of that set could be (and should be) huge, we just take a small batch representative and apply the correction for each image of the batch. Once we have done it we have complete one epoch. With just one epoch the network still giving us weird answers, so what we have to do is to train the network with several epochs and then our network is trained and ready.

To measure how intelligent our network is you can use the test set of images to calculate the accuracy.

1.4 Implementation

With the full algorithm of the back propagation a full Neural Network could be developed. You can do it by following the steps and code it in *Python*, *Haskell*, *Matlab* ... But as you develop your net you will notice that it is only valid for a certain network architecture, and it is pretty slow because of the languages. You can develop it on *C++* or even in *C* what will need a lot of effort for achieving more speed.

But as Neural Networks are not new and as they are well known and famous, there are many tools you can choose for creating your model. It will be much faster to implement it and the net will run also faster since the tools are optimized. One of the most famous tools is developed by *Google*. They are the kings of de big data and they are using the data for machine learning so they have created *TensorFlow* carefully to fullfill all their needs.

1.4.1 Tensorflow

TensorFlow is an open source software library for numerical computation using data flow graphs. It was developed initially for *Google* machine learning purposes but it could be used in other domains. It was released for the first time in 2015 and now it is on version *1.4.0*. This tool gives you a APIs of different levels of abstraction that allow you to work with complex numerical computation in an abstract, easy and elegant way [8].

In TensorFlow **SIN ACABAR**

1.4.2 MNIST Data Set

Every time you start learning a new language you first start with *Hello World*. In the machine learning this program is the *MNIST* problem. The *MNIST* data set is composed by 70.000 images of handwritten digits (See Figure 1.4), and the whole data set has been studied several times as it has shown to fit very well with basic neural networks.

In order to be able to give the images to the input they are stored as flattened vector of $24 \cdot 24 = 784$ numbers. The output of the networ will be an array of dimension 10 where each position is 0 but one has the value 1. The position of the value 1 represents which number the network thinks the image is. For instance, if the input is the image of a 8 the perfect output is $[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]$.

1.4.3 Building the Model

As we have seen from the data the input is of dimension 784 while the output has 10 outcoming values. We are going to have two hidden layers of 256 neurons each. We are going to define the parameters of the network so we also can define the learning constant $\alpha = 0.01$, the number of epochs to

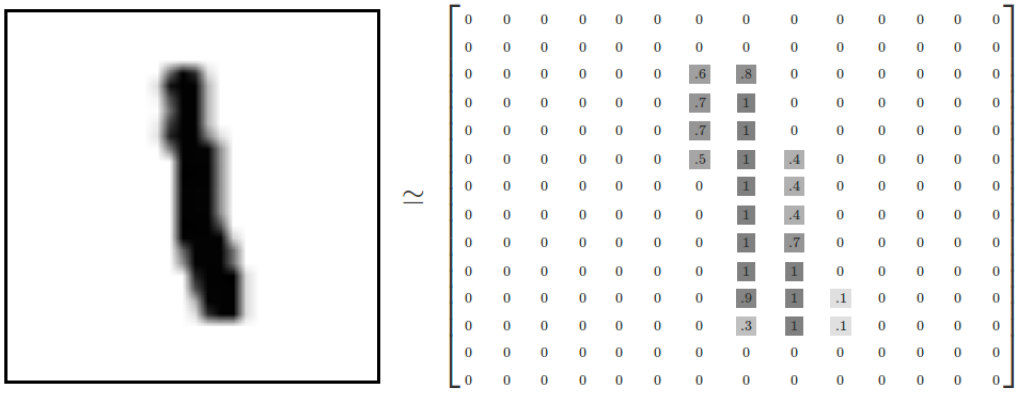

Figure 1.4: The images are $28x28$ pixels where each of them describes how dark is that point of the image. This figure is $14x14$ for simplicity [5]

1000 and the batch size (number of images the network train with in each epoch) to 100.

```
#Network parameters
n_hidden1 = 256
n_hidden2 = 256
n_input = 784
n_output = 10
#Learning parameters
learning_constant = 0.01
number_epochs = 1000
batch_size = 100
```

We also have to specify how we are going to input the data and how we expect the ouput, for that we are goin gto define the placehoders, defining that the input is a two dimensions array of floats, one dimension is every input (784) and as we do not know the number of images we are going to give we write *None* for the other dimension. In a similar way the output is defined.

```
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_output])
```

Each neuron has a bias and each pair of neurons of neighbours layers has a weight so we must define all of them:

```
#Biases first hidden layer
b1 = tf.Variable(tf.random_normal([n_hidden1]))
```

```
#Biases second hidden layer
b2 = tf.Variable(tf.random_normal([n_hidden2]))
#Biases output layer
b3 = tf.Variable(tf.random_normal([n_output]))

#Weights connecting input layer with first hidden layer
w1 = tf.Variable(tf.random_normal([n_input, n_hidden1]))
#Weights connecting first hidden layer with second hidden layer
w2 = tf.Variable(tf.random_normal([n_hidden1, n_hidden2]))
#Weights connecting second hidden layer with output layer
w3 = tf.Variable(tf.random_normal([n_hidden2, n_output]))
```

Done, we have defined $728 \cdot 256 + 256 \cdot 256 + 256 \cdot 10 = 254464$ weights in three lines, TensorFlow is powerful. To finish we just have to tell how the four layers are connected, and the the model will be ready to be trained. To achieve it we have to define what is the task for each neuron. If one vector of data x is incoming to a neuron, it has to weight it (w) add the bias u and apply the activation function $f(x \cdot W + u)$. Making the product of each input data with their weight is as simple as applying *tf.matmul*, then add the bias with *tf.add* and finally apply the activation function *tf.nn.relu*.

```
#The incoming data given to the
#network is input_d
def multilayer_perceptron(input_d):
    #Task of neurons of first hidden layer
    layer_1 = tf.nn.relu(tf.add(tf.matmul(input_d, w1), b1))
    #Task of neurons of second hidden layer
    layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, w2), b2))
    #Task of neurons of output layer
    out_layer = tf.add(tf.matmul(layer_2, w3),b3)
    return out_layer
```

The full neural network model can be stored in a single variable in such a simple way as (X is the placeholder we have already defined as the input of the network):

```
# Create model
neural_network = multilayer_perceptron(X)
```

1.4.4 Training the Model

Is the moment of training the network. For that purpose we have to define what is the error and which method we are going to use to fix it. The error or the loss is calculated giving our model and the placeholder of the output.

For fixing that loss we are going to apply the gradient descent optimizer and we have to define what we want to reduce and what is the learning constant.

```
#Define the loss or the error
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
                         logits=neural_network,labels=Y))
#Define how to fix it
optimizer = tf.train.GradientDescentOptimizer(learning_constant)
                     .minimize(loss_op)
```

Next step is iterate trought all the epochs. In each one the program should take one batch of images and their expected output, feed the network with it, and finally apply the optimizer we have already defined. The task of running the epochs could not be done in such a simple way as we have defined the model. The train should be performed inside a TensorFlow session after all the declares variables are initialized. The session allows not just declare the model but perform operations.

```
#Initializing the variables
init = tf.global_variables_initializer()
#Create a session
with tf.Session() as sess:
    sess.run(init)
    #Training epoch
    for epoch in range(number_epochs):
        #Get one batch of images
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        #Run the optimizer feeding the network with the batch
        sess.run(optimizer, feed_dict={X: batch_x, Y: batch_y})
        #Display the epoch (just every 100)
        if epoch % 100 == 0:
            print("Epoch:", '%d' % (epoch))
```

1.4.5 Evaluating the Model

The network is already trained so it is the time of test how well we have designed it and how much it has learned. One part of the date set was reserved for testing. The outputs of the network are not as perfect as one vector with all 0 but one 1 pointing the solution. Actually it is more close to something like $[0.01, 0.94, 0.02, 0.025, 0.025, 0.06, 0.01, 0.01, 0.4, 0.04]$ because the network is not absolutely sure about the answer so the output is something like "I am 94% sure that this is a 1 but it could be a 8 in a 4% and a bit percentage for the other options". To transform that answers

the *softmax* function should be applied. It takes the bigger number and transform it to a 1 while change the other smallers numbers with 0s. Now we define what is a correct answer with *tf.equal*, one prediction is correct if it is equal to the expected output, easy. To finish we define the accuracy to print it and feed the network wiht the test subset.

```
# Test model
pred = tf.nn.softmax(neural_network)  # Apply softmax to logits
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print("Accuracy:", accuracy.eval({X: mnist.test.images, Y: mnist.test.labels}
```

If we execute the code we get an accuracy close to 90%. It might seem that is a great result, it answer correctly 9 out of 10 times. But in machine learning this result is pretty bad, it is normal because the network used is the most simple network but compared with other kind of models it is a bit shaming.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[2] Janusz Kacprzyk and Witold Pedrycz. *Springer handbook of computational intelligence*. Springer, 2015.

[3] Gonzalo Medina. *Diagram of an artificial neural network*. Tex Stack Exchange. URL:https://tex.stackexchange.com/a/132471/. (Visited on 30/11/2017).

[4] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[5] Christopher Olah. *Visualizing MNIST: An Exploration of Dimensionality Reduction*. Oct. 9, 2015. URL: http://colah.github.io/posts/2014-10-Visualizing-MNIST/ (visited on 04/12/2017).

[6] Raul Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[7] Standford. *Neural Networks*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/index.html (visited on 16/11/2017).

[8] Amy Unruh. *What is the TensorFlow machine intelligence platform?* URL: https://opensource.com/article/17/11/intro-tensorflow (visited on 06/12/2017).

[9] Neha Yadav, Anupam Yadav, and Manoj Kumar. *An introduction to neural network methods for differential equations*. Springer, 2015.