How to build a three-layer neural network from scratch

Daphne Cornelisse Follow
Feb 18, 2018 · 10 min read

In this post, I will go through the steps required for building a **three layer neural network.** I'll go through a problem and explain you the process along with the most important concepts along the way.

The problem to solve

A farmer in Italy was having a problem with his labelling machine: it mixed up the labels of three wine cultivars. Now he has 178 bottles left, and nobody knows which cultivar made them! To help this poor man, we will build a **classifier** that recognizes the wine based on 13 attributes of the wine.

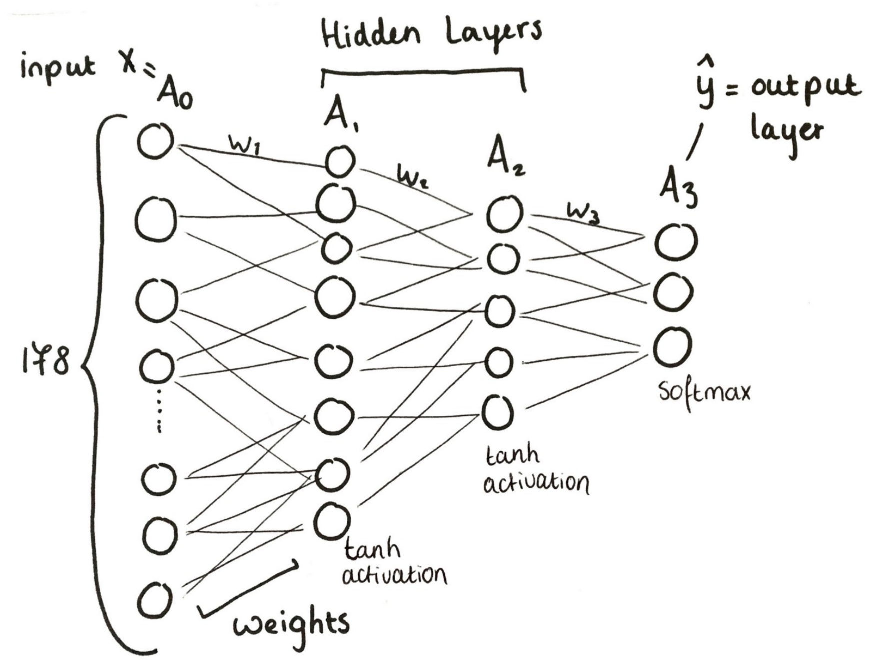
The fact that our data is labeled (with one of the three cultivar's labels) makes this a **Supervised learning** problem. Essentially, what we want to do is use our input data (the 178 unclassified wine bottles), put it through our neural network, and then get the right label for each wine cultivar as the output.

We will train our algorithm to get better and better at predicting (y-hat) which bottle belongs to which label.

Now it is time to start building the neural network!

Approach

Building a neural network is almost like building a very complicated function, or putting together a very difficult recipe. In the beginning, the ingredients or steps you will have to take can seem overwhelming. But if you break everything down and do it step by step, you will be fine.



Overview of the 3 Layer neural network, a wine classifier

In short:

- The input layer (x) consists of 178 neurons.
- A1, the first layer, consists of 8 neurons.
- A2, the second layer, consists of 5 neurons.
- A3, the third and output layer, consists of 3 neurons.

Step 1: the usual prep

Import all necessary libraries (NumPy, skicit-learn, pandas) and the dataset, and define x and y.

```
#importing all the libraries and dataset

import pandas as pd
import numpy as np

df = pd.read_csv('../input/W1data.csv')
df.head()

# Package imports

# Matplotlib
import matplotlib
import matplotlib.pyplot as plt

# SciKitLearn is a machine learning utilities library
import sklearn

# The sklearn dataset module helps generating datasets


import sklearn.datasets
import sklearn.linear_model
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score
```

Step 2: initialization

Before we can use our weights, we have to initialize them. Because we don't have values to use for the weights yet, we use random values between 0 and 1.

In Python, the `random.seed` function generates "random numbers." However, random numbers are not truly random. The numbers generated are **pseudorandom**, meaning the numbers are generated by a complicated formula that makes it look random. In order to generate numbers, the formula takes the previous value generated as its input. If there is no previous value generated, it often takes the time as a first value.

That is why we seed the generator—to make sure that we always get **the same random numbers**. We provide a fixed value that the number generator can start with, which is zero in this case.

```
np.random.seed(0)
```

Step 3: forward propagation

There are roughly two parts of training a neural network. First, you are propagating forward through the NN. That is, you are “making steps” forward and comparing those results with the real values to get the difference between your output and what it should be. You basically see how the NN is doing and find the errors.

After we have initialized the weights with a pseudo-random number, we take a linear step forward. We calculate this by taking our input A0 times the **dot product** of the random initialized weights plus a **bias**. We started with a bias of 0. This is represented as:

$$z_1 = A_0 . W_1 + b$$

Now we take our z1 (our linear step) and pass it through our first **activation function**. Activation functions are very important in neural networks. Essentially, they convert an input signal to an output signal — this is why they are also known as *Transfer functions.* They introduce **non-linear properties** to our functions by converting the linear input to a non-linear output, making it possible to represent more complex functions.

There are different kinds of activation functions (explained in depth in this article). For this model, we chose to use the **tanh** activation function for our two hidden layers — A1 and A2 — which gives us an output value between -1 and 1.

Since this is a **multi-class classification problem** (we have 3 output labels), we will use the **softmax** function for the output layer — A3 — because this will compute the probabilities for the classes by spitting out a value between 0 and 1.

$$tanh(x) = \frac{e^{2*x} - 1}{e^{2x} + 1}$$

tanh function

By passing z1 through the activation function, we have created our first hidden layer — A1 — which can be used as input for the computation of the next linear step, z2.

$$A_1 = tanh(z_1)$$

In Python, this process looks like this:

```
# This is the forward propagation function
def forward_prop(model,a0):

    # Load parameters from model
    W1, b1, W2, b2, W3, b3 = model['W1'], model['b1'],
model['W2'], model['b2'], model['W3'],model['b3']

    # Do the first Linear step
    z1 = a0.dot(W1) + b1

    # Put it through the first activation function
    a1 = np.tanh(z1)

    # Second linear step
    z2 = a1.dot(W2) + b2

    # Put through second activation function
    a2 = np.tanh(z2)

    #Third linear step
    z3 = a2.dot(W3) + b3

    #For the Third linear activation function we use the
softmax function
    a3 = softmax(z3)

    #Store all results in these values
    cache =
{'a0':a0,'z1':z1,'a1':a1,'z2':z2,'a2':a2,'a3':a3,'z3':z3}
    return cache
```

In the end, all our values are stored in the cache.

Step 4: backwards propagation

After we forward propagate through our NN, we backward propagate our error gradient to update our weight parameters. We know our error, and want to minimize it as much as possible.

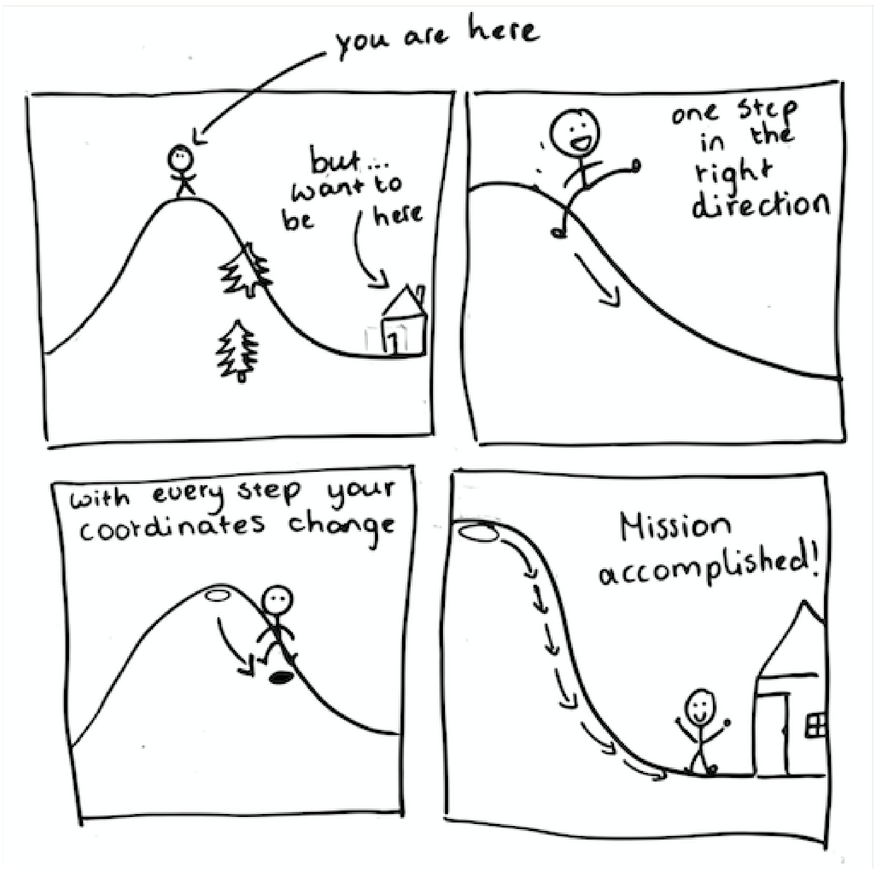
We do this by taking the **derivative of the error function,** with respect to the weights (W) of our NN, using **gradient descent**.

Lets visualize this process with an analogy.

Imagine you went out for a walk in the mountains during the afternoon. But now its an hour later and you are a bit hungry, so it’s time to go home. The only problem is that it is dark and there are many trees, so you can’t see either your home or where you are. Oh, and you forgot your phone at home.

But then you remember your house is in a valley, the lowest point in the whole area. So if you just walk down the mountain step by step until you don’t feel any slope, in theory you should arrive at your home.

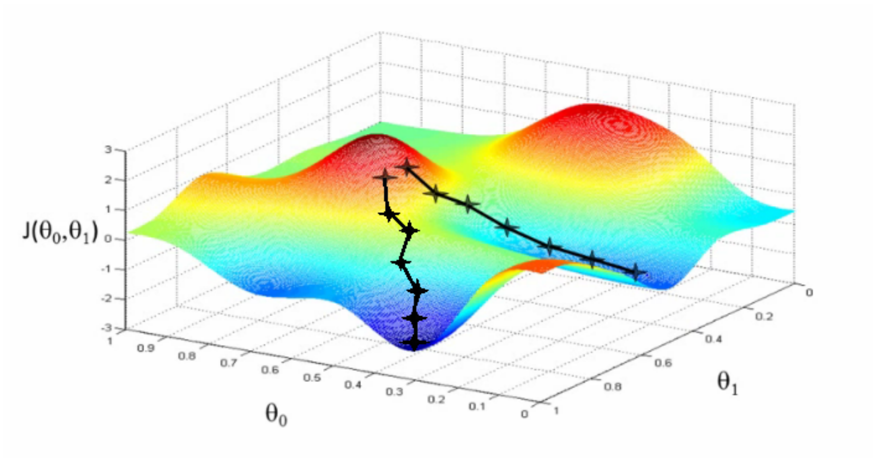
So there you go, step by step carefully going down. Now think of the mountain as the loss function, and you are the algorithm, trying to find your home (i.e. the **lowest point**). Every time you take a step downwards, we update your location coordinates (the algorithm **updates the parameters**).


you are here
but... want to be here
one step in the right direction
with every step your coordinates change
Mission accomplished!

The loss function is represented by the mountain. To get to a low loss, the algorithm follows the slope—that is the derivative—of the loss function.

When we walk down the mountain, we are updating our location coordinates. The algorithm updates the parameters of the neural network. By getting closer to the minimum point, we are approaching our goal of **minimizing our error.**

In reality, gradient descent looks more like this:



We always start with calculating **the slope of the loss function** with respect to z, the slope of the linear step we take.

Notation is as follows: dv is the derivative of the loss function, with respect to a variable v.

$$dz_3 = (A_3 - y)$$

Next we calculate the **slope of the loss function** with respect to our weights and biases. Because this is a 3 layer NN, we will iterate this process for z3,2,1 + W3,2,1 and b3,2,1. Propagating backwards from the output to the input layer.

$$dW_3 = \frac{1}{m} A_2^T . dz_3$$

$$db_3 = \frac{1}{m} \sum dz_3$$

This is how this process looks in Python:

```
# This is the backward propagation function
def backward_prop(model,cache,y):

# Load parameters from model
    W1, b1, W2, b2, W3, b3 = model['W1'], model['b1'],
model['W2'], model['b2'],model['W3'],model['b3']
```

```
    # Load forward propagation results
    a0,a1, a2,a3 =
cache['a0'],cache['a1'],cache['a2'],cache['a3']

    # Get number of samples
    m = y.shape[0]

    # Calculate loss derivative with respect to output
    dz3 = loss_derivative(y=y,y_hat=a3)

# Calculate loss derivative with respect to second layer
weights
    dW3 = 1/m*(a2.T).dot(dz3) #dW2 = 1/m*(a1.T).dot(dz2)

    # Calculate loss derivative with respect to second layer
bias
    db3 = 1/m*np.sum(dz3, axis=0)

    # Calculate loss derivative with respect to first layer
    dz2 = np.multiply(dz3.dot(W3.T) ,tanh_derivative(a2))

    # Calculate loss derivative with respect to first layer
weights
    dW2 = 1/m*np.dot(a1.T, dz2)

    # Calculate loss derivative with respect to first layer
bias
    db2 = 1/m*np.sum(dz2, axis=0)

    dz1 = np.multiply(dz2.dot(W2.T),tanh_derivative(a1))

    dW1 = 1/m*np.dot(a0.T,dz1)

    db1 = 1/m*np.sum(dz1,axis=0)

    # Store gradients
    grads = {'dW3':dW3, 'db3':db3,
'dW2':dW2,'db2':db2,'dW1':dW1,'db1':db1}
    return grads
```

Step 5: the training phase

In order to reach the **optimal weights and biases** that will give us the desired output (the three wine cultivars), we will have to **train** our neural network.

I think this is very intuitive. For almost anything in life, you have to train and practice many times before you are good at it. Likewise, a neural network will have to undergo many epochs or iterations to give us an accurate prediction.

When you are learning anything, lets say you are reading a book, you have a certain **pace**. This pace should not be too slow, as reading the book will take ages. But it should not be too fast, either, since you might miss a very valuable lesson in the book.

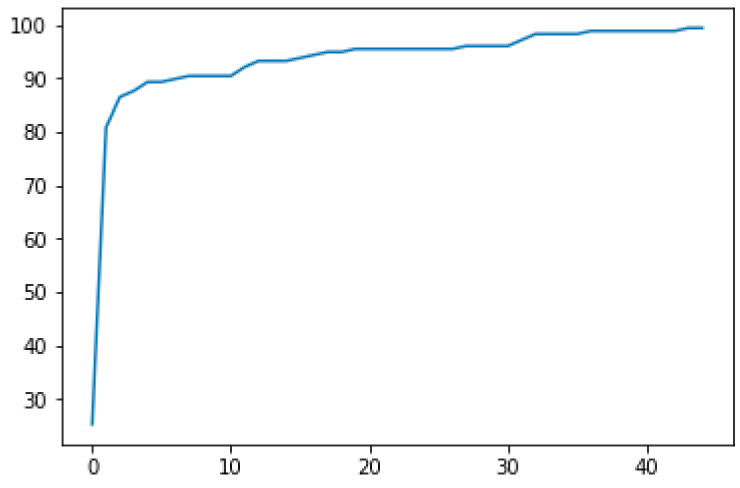
In the same way, you have to specify a “**learning rate**” for the model. The learning rate is the multiplier to update the parameters. It determines how rapidly they can change. If the learning rate is low, training will take longer. However, if the learning rate is too high, we might miss a minimum. The learning rate is expressed as:

$$a := a - \alpha * \frac{dL(w)}{da}$$

- **:=** means that this is a definition, not an equation or proven statement.
- ***a*** is the learning rate called *alpha*
- **dL(w)** is the derivative of the total loss with respect to our weight **w**
- **da** is the derivative of *alpha*

We chose a learning rate of 0.07 after some experimenting.

```
# This is what we return at the end
model = initialise_parameters(nn_input_dim=13, nn_hdim= 5,
nn_output_dim= 3)
model =
train(model,X,y,learning_rate=0.07,epochs=4500,print_loss=Tr
ue)
plt.plot(losses)
```



Finally, there is our graph. You can plot your accuracy and/or loss to get a nice graph of your prediction accuracy. After 4,500 epochs, our algorithm has an accuracy of 99.4382022472 %.

Brief summary

We start by feeding data into the neural network and perform several matrix operations on this input data, layer by layer. For each of our three layers, we take the dot product of the input by the weights and add a bias. Next, we pass this output through an **activation function** of choice.

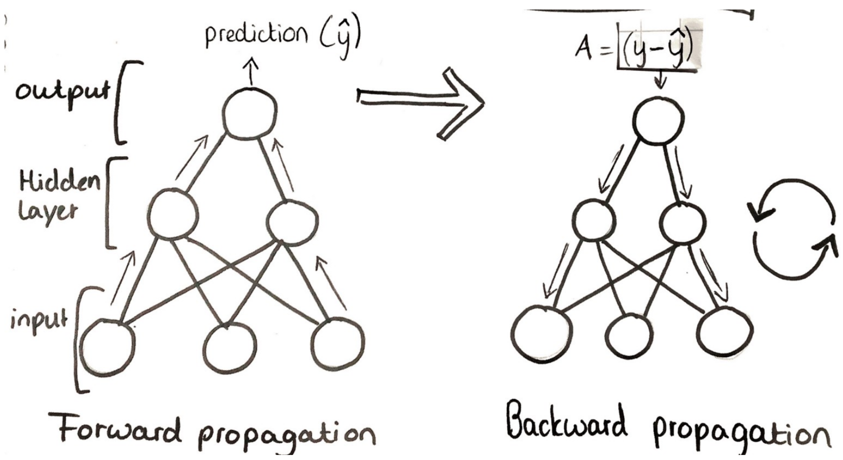
The output of this activation function is then used as an input for the following layer to follow the same procedure. This process is iterated three times since we have three layers. Our final output is **y-hat**, which is the ***prediction*** on which wine belongs to which cultivar. This is the end of the forward propagation process.

We then calculate the **difference** between our prediction (y-hat) and the expected output (y) and use this error value during backpropagation.

During backpropagation, we take our error — the difference between our prediction y-hat and y — and we mathematically push it back through the NN in the other direction. We are learning from our mistakes.

By taking the derivative of the functions we used during the first process, we try to discover what value we should give the **weights** in order to achieve the **best possible prediction**. Essentially we want to know what the relationship is between the value of our weight and the error that we get out as the result.

And after many epochs or iterations, the NN has learned to give us more accurate predictions by adapting its parameters to our dataset.



Overview of Forward and Backwards propagation

This post was inspired by the week 1 challenge from the Bletchley Machine Learning Bootcamp that started on the 7th of February. In the coming nine weeks, I'm one of 50 students who will go through the fundamentals of Machine Learning. Every week we discuss a different topic and have to submit a challenge, which requires you to really understand the materials.

If you have any questions or suggestions or, let me know!

Or if you want to check out the whole code, you can find it here on Kaggle.

• • •

Recommended videos to get a deeper understanding on neural networks:

- 3Blue1Brown's series on neural networks
- Siraj Raval's series on Deep Learning