# Using a Deep Feed Forward Neural Network to Classify the Language of a Sentence

# Christian Yeganeh (cyeganeh@masonlive.gmu.edu)

# CS 484-001

# Final Project Report

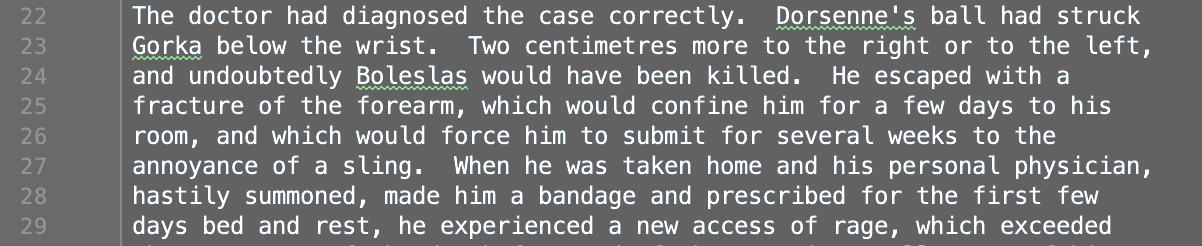
## Introduction:

The point of this product is to create a Neural network classifier that can take a sentence and determine which language the sentence is between, English, French, and German. The data to do this was taken from Project Gutenberg and transformed into usable words that were used to train a deep feed forward neural network with 3 hidden nodes. The output then outputs the chance that the inputted word is of each of the 3 languages. The use for this could be to have a portable and accurate language identifier that doesn’t rely on huge data sets or algorithmic approaches that take up a huge amount of memory or space with these look up tables, and instead replaces it with a trained neural network that classify a language with a 70% accuracy.

## Approach:

My approach to was to create a feed forward neural network for classifying the data. I started out by downloading my dataset that I was going to be using for this project. This turned out to be harder than anticipated, as Project Gutenberg does not like automated downloading of the books in their catalog. They do have a different system dedicated directly for robot web scrapping, and it is a little weird to use.

However, there was a library for quickly and easily downloading this automated site. The Gutenberg library was then used to download 20 books for each of the 3 languages that I chose to be in this classifier. I originally was going to do 5, and I believe the program could be done with 5 languages, however, the amount of processing, and the length of the training would probably have doubled if I would have had to include 2 more languages. Because of this, I dropped down to using 3 languages.



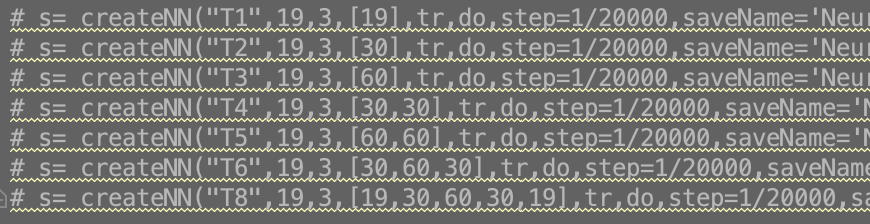
#### Figure 1. The books were downloaded raw and looked like this:

After this, I then went through all the books, and did some preprocessing including, making all characters lowercase, removing punctuation, and making the files all one line as Gutenberg just outputs it in a more readable way with each line being a certain length instead of putting new line characters at the end of paragraphs. I then used the TextBlob Library to take each of these new processed book files and load in all the sentences from each. It then randomly selected 160, and then created a Json file with all the data so the processing had to be done once. The sentences are then broken into individual words and then letters to create a list of all the words and letters that appear in all of the sentences with their true language stored with them.

#### 

#### Figure 2. The processed and stored data in the json file

Then came the fun part of created the neural network. I created a function that could create a neural network of any shape given the number of nodes in each of the layers. Because I was not sure what would be the optimal shape for the network to get accurate results, I started out by creating around 8 different shapes for the neural networks. All having 19 input nodes and 3 output nodes. A note the number of input nodes was based on the longest word in the list of words. And the 3 output nodes represented the probability of each class. I then created the different shaped hidden layers, which some have just one hidden layer and looking like [19,30,3] or [19,60,3] and some having more layers such as [19,60,60,3] or what turned out to be the most accurate [19, 60,30,19,3]. I then left all 8 of these neural to train overnight. The result in the morning was that [19,60,30,19,3] had the smallest loss. I have included a comparison of a few different shapes that the loss function was graphed for each run in the Graphs section.



*Figure 3. The different neural networks that were created.*

My next step was to then take each of these neural networks and create a testing data. Rather than use the same data but in different sections, I just created a new set of sentences in the exact same way as before, only these would be 160 new random sentences from each of the books. These sentences were then run through the neural network and recorded how many for each language the neural network got correct and including the total number of correct. Each of the networks were at least 50% accurate in guessing overall, which my golden shape having an overall accuracy of 70%.

## Experiments:

### Data:

The raw data was 60 random books from Project Gutenberg in 3 different languages, processed into the sentences for each book. You can view the exact sentences used for training in the Sentences.dat file. And you can view the Sentences used for testing by viewing the TestSentences.dat file. Both are located in the main directory with the code. These books can be redownloaded by calling the getNBooks(numBooks, lang, location) function where lang is a language returned from the Gutenberg types (in this case for these lanuages ‘en’,’fr’,’gr’) and location is where the file will be saved. Almost everything else is handled automatically in DataGenerator.py. If it doesn’t find the Sentences.dat or TestSentences.dat files, it will recreate them from the books.

### Experimental setup:

The program is set up and run by running the Tester.py file. This program will generate the training data, generate the neural network, and then finally create testing data and run it through the neural network and collect the results. If you would like to change the shape of the neural network, it will be created in this file.

A note about creating the neural network. Due to trying to create a very flexible and powerful neural network creator, it has a lot of parameters that can be passed in. The full function looks as follows:

#### createNN(n, initSize, outSize, layers, train, expected, step= 0 .00005, restore=None,

#### numSteps=-1, saveName=None, printLoss=True,saveForGraph=False, optimize=True)

#### Figure 4. Inputs for the createNN function

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| n | The name of the neural network. Used in printing loss only. For keeping track |
| initSize | The number of nodes in the initial layer |
| outSize | The number of nodes in the output layer |
| layers | An array that contains number of nodes in hidden layers. Can be any length |
| train | The training values to be used |
| expected | The expected outcome for each of the training values |
| step | Optional. The step size for the gradient descent Default: 0.00005 |
| restore | Optional. If provided it will load a previous NN from here. Default: None |
| numSteps | Optional. The number of training iterations. If -1, it will go forever. Default: -1 |
| saveName | Optional. If provided it will save the NN to here when done. Default: None |
| printLoss | Optional. Whether to print the Loss after each iteration Default: True |
| saveForGraph | Optional. Whether to generate a graph of Loss at end Default: False |
| optimize | Optional. Whether to try to change step to improve lose Default: True |

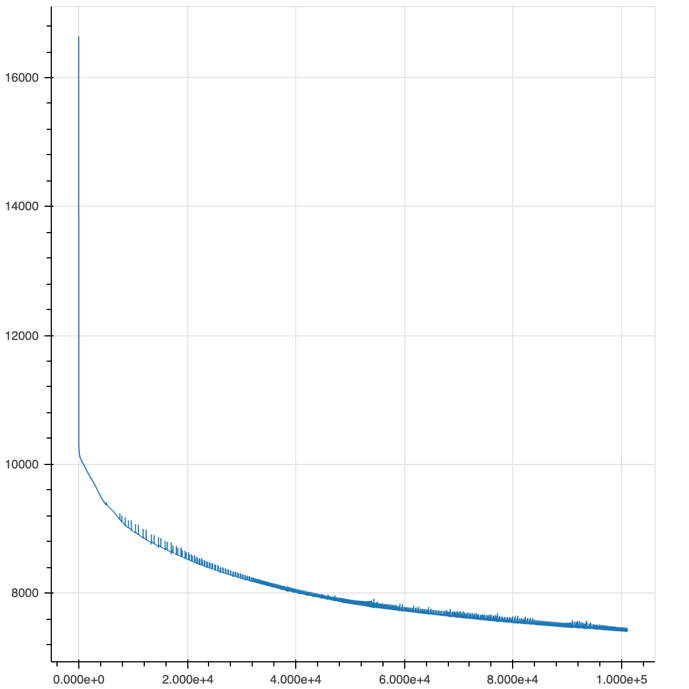
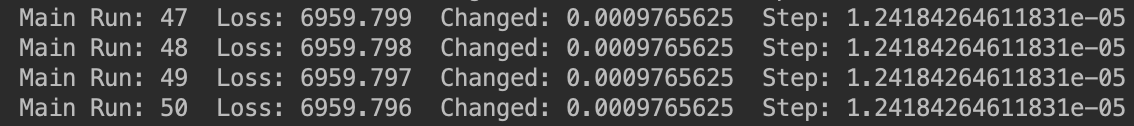
My main and most accurate neural network was created with the following parameters, keeping in mind a Sentence.dat max wordlength of 19.

createNN(19,3,[60,30,19],tr,do,step=.000002,restore=”model.nnnet”,numSteps=-1, saveName=”model.nnnet”,printLoss=True,saveForGraph=True)

It was then trained until loss went from around ~17,000 to ~7,200 which took about 7 hours overnight.

### Results and Analysis:

The loss stabilized off at around 6,950. I set it so that it would keep training and lowering the step size and it eventually got to the point where it wasn’t changing and was at 6,950. I didn’t have it running the graphLoss part. But the results can be seen below.



#### Figure 5. The graph of loss over time Figure 6. The live data showing change and loss

When I ran the initial 8 different layouts for the neural networks and ended them after 7 hours of training, all of them had results above 50% in all 4 categories of English % correct, French % correct, German % correct, and Total % correct. Unfortunately, the image that had their raw scores was overwritten so I don’t have exact numbers other than the main neural network as it is still saved and can be rerun to get the scores. I didn’t implement the saving neural network feature into the function yet then and was still manually saving the neural networks, so these have been lost. However, I still have their create calls in the Tester.

#### Figure 7. The Results from the best Neural Network in the initial 8 trained

|  |  |
| --- | --- |
| English % Correct | 72.852% |
| French % Correct | 69.916% |
| German % Correct | 69.228% |
| Total % Correct | 70.665% |

Overall, the Neural Network is very accurate. When compared to the raw chance of guessing the language at 33.333%, this is significantly better. I am very happy to have a Total % Correct above 70%. The results are fairly balanced, and it has a high accuracy of classifying any random sentence correctly.

### Brief Conclusion:

In summary, I was able to create a neural network that was accurate up to 70% in classifying a random sentence as either English, French, or German. The network took time to train to completion, but by the end could very reliably detect the language of a sentence.

## Code:

**BookAndWordProcessor.py:**

This code is all for getting the books and processing the books into a direct format. The document mostly is written by me, but I did use 2 libraries in this document. In getNBooks, it uses the Gutenberg library to download the actual books. This library did not do any processing or messing with the data. Only facilitating making the downloading of the books easier. The other library is in getSentences, I made use of TextBlob to automatically get the sentences. It appears like with all the preprocessing I did, I may have been able to just separate by a period to get roughly the same thing, but I originally was using it for more and ended up implementing my own functions for the same things. This last reference probably could be removed.

**DataGenerator.py:**

This file is all for creating the training data and letter list that’s used when making the word vectors. Everything here is written by me. With the exception of the use of the Json library to load in the sentences from a Json file. And this step was only as an optimization to prevent it having to make two new sentence dictionaries every time it ran, which took about a minute to do at best. If the max length for all the words in the TestSentences.dat file was not equal to the max length for all the words in the Sentences.dat file, it would have to recreate the file until they match. The Json just made it have to import from a file the already created sentences which took seconds for both.

**NeuralNetHandler.py**

This file contains the code for creating, testing, and dealing with the neural networks generated in this program. It uses Tensor flow for the actual stuff done with the neural network, but all the code to create and test the neural network was created by me. TensorFlow is a powerful neural network library, but it was awful to create a neural network in it until I made my function that would automatically create, start training, save/restore, and return the neural network. The testing functions were both purely made by me to use the TensorFlow neural network to classify an input. NumPy was also used here for processing of arrays and better data storage.

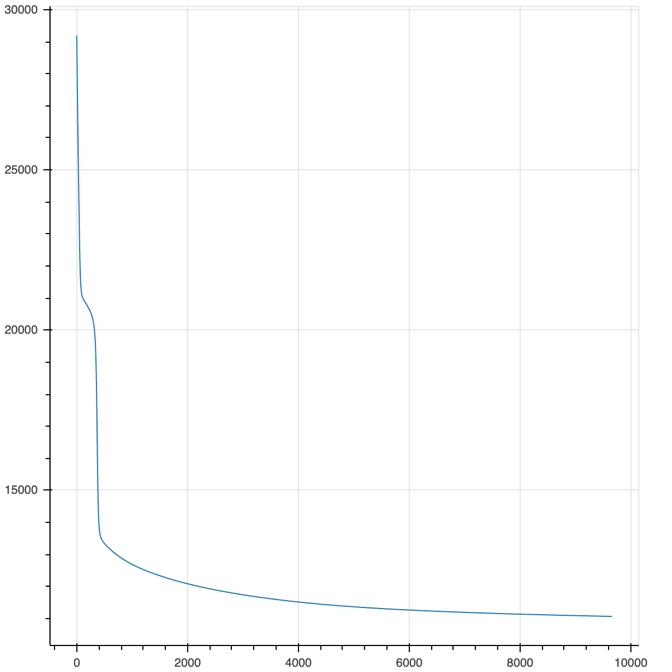
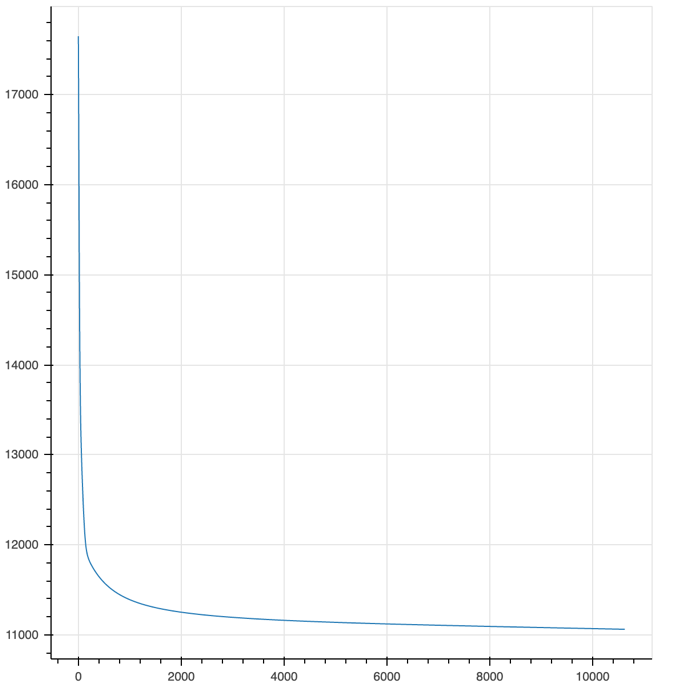
**Tester.py**

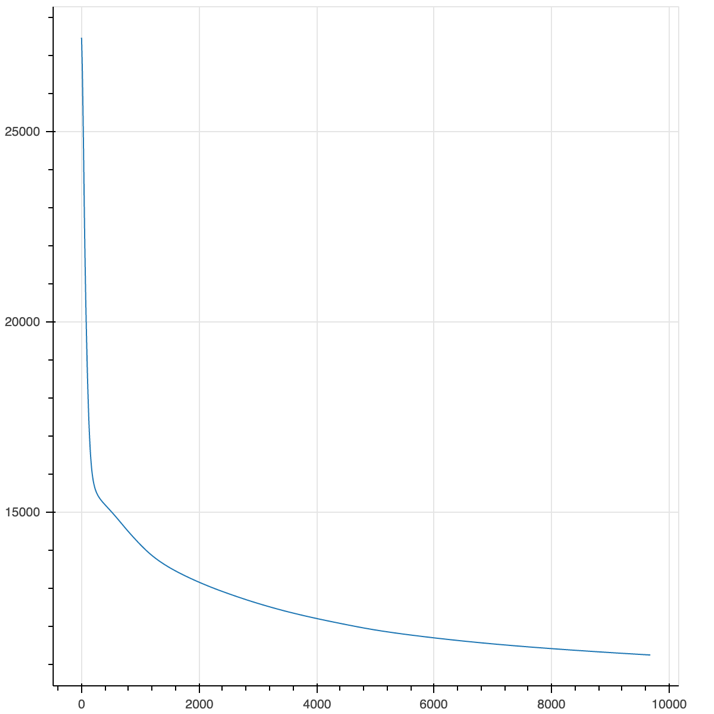
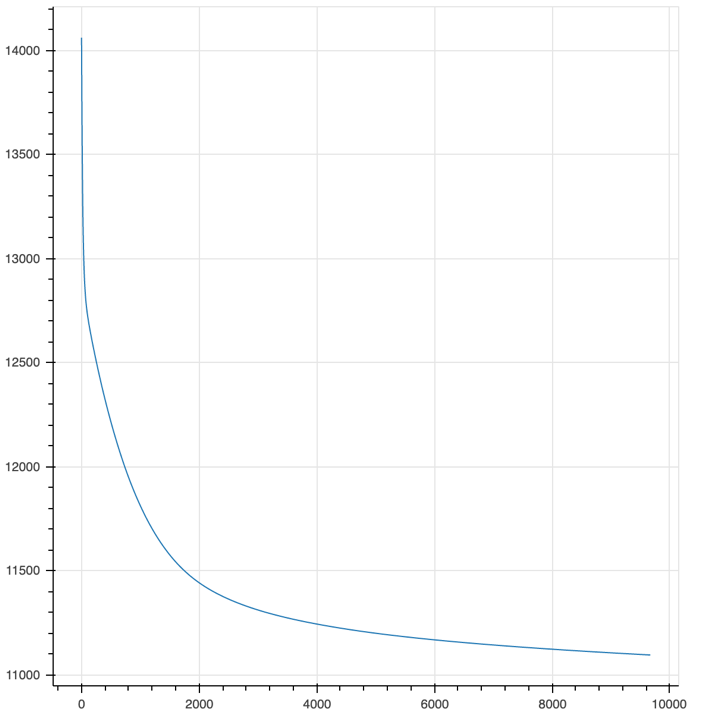
This is the last document. This code does not provide any direct code but can be thought of as the main file. It is the file that is run that will create the neural network and then run the tests on the data. The only library that was used here, was the bokeh plotting library. And this was how I generated graphs for things like loss or word length frequency. It just lets you create data plots. Everything else was written by me.

**Sentence.dat/TestSentences.dat**

These documents contain precomputed and made sentences for use in the training, and testing.

## Extra Graphs:

*Figure 8.Loss of [19,30,3] Neural Net Figure 9. Loss of [19,30,60,30,3] Neural Net*



*Figure 10. Loss of [19,30,30,3] Neural Net Figure 11. Loss of [19,60,60 ,3] Neural Net*