Language Detection Neural Network with Vector Hashing

David Gallagher

May 21, 2020

1 Introduction and Overview

This document outlines the steps take to produce the Neural Network.
These steps include testing of various activation functions, ngram sizes, text pre-processing, and identify the reasons for why the attributes of the final Network were decided upon. Throughout the document default values refer to the most optimized values discovered during testing and development.
The document aims to discuss the following points:

- N-grams hashing approach used within the application.
  - How is the hashing implemented?
  - How many ngrams are used?
  - Reasons for these choices.
- The size of the hashing feature vector.
  - What size is the hashing vector
  - Why?
  - Results of any testing/experimentation etc
- The overall neural network topology.
  - How many layers are in use?
  - How many nodes are used in each layer?
  - Why are we using those numbers?
- The activation functions used in each layer.
  - Which activation functions are used in each layer.
  - Why use those?
  - Testing results and conclusions

2 Running the Application

The user may modify the network slightly, setting preferred ngram and vector sizes. Also there is optional activation function modification and optional set the epoch, though the recommended settings are suggested in the event of wanting to keep some settings at default. The network will run at its best while using the suggested settings.
Its worth mentioning also, once the user has defined any preferences, and built a network, if then testing live data the same settings should be entered to keep the prediction and evaluation consistent with the network architecture.

No Other extras exist aside from an optional 'test files in a directory' feature which is available from the menu along with the setting preferences. This is to reduce manual file path entry for testing and will run predictions against any text files in a directory.
Choosing a test file may be entered by typing in a file or directory path or using a basic gui to select a directory or file.

3 Ngrams and Vector Hashing

Ngrams are generated using a 'lazy iterator' of sorts. The NgramProducer takes a ngram size and a string as it's parameters and returns the vectorized array of values. The VectorProcessor Class implementation will use the getNormalizedVector method to iterate over an input text, either the source training data text or a test text and strip it down to ngram size chunks and return a vectorized array of doubles.
Prior to returning the array of doubles the values are normalised to between 0-1 which lends it's self to the network output layer's activation function.
Once complete, the VectorProcessor, depending on which constructor has been called will either create a csv of training data for the Neural Network, or produce an array of doubles to use as testing data to evaluate the predictive accuracy of the Network.
Ultimately I found that lower ngram sizes worked best, with the vector size varying from 300 - 1000 in size. At some point one must decide on a trade-off regarding time and accuracy. During my testing I found that for every network I tested the accuracy would reach its peak at around epoch 10-14, with a bit of variation when using dramatically different vector sizes.

4 Network Topology

I tested various structural approaches to the network architecture. This included testing a variety of different input nodes, a variety of different hidden nodes and hidden layers, and both with and without drop out.
In each case the output layer was assigned an activation function of softmax as this function outputs number between 0 and 1. Hence the choice of normalization values (0-1) mentioned previously.
By far the leading success i found in relation to activation functions was that there are 6 high performers. At least, those 6 I found to be the best performing, are the most suited both to the task at hand and the architecture of my particular network.
Table 1 below illustrates the top 6 performing Activation Functions I found to be effective using my topology.

ACTIVATION FUNCTION	ACCURACY	TRAIN TIME	TEST TIME
ElliottSymmetric	85.663	82450 ms	1906 ms
LOG	84.828	82345 ms	2160 ms
TANH	82.630	85024 ms	2215 ms
ClippedLinear	81.412	81963 ms	1909 ms
BipolarSteepenedSigmoid	78.703	64872 ms	2070 ms
BiPolar	73.447	81465 ms	2009 ms

Table 1: Comparison of Activation Functions.

These tests were carried out with a vector size of 300 and a drop out rate of 0.8. The drop out rate was calculated as seen in the following Network Nodes snippet in listing 1, in accordance with a few different sources.
Most notably from Machine Learning Mastery's section on dropout for deep learning neural networks which I used as a guide during development. Their suggestion for setting the dropout rate:
'A good value for dropout in a hidden layer is between 0.5 and 0.8. Input layers use a larger dropout rate,

such as of 0.8'.
I set the dropout rate as a percentage of the amount of hidden nodes. The hidden nodes are calculated using the Geometric Pyramid Rule and as such scale with the amount of input nodes. This is to mitigate the possibility of over or under fitting the hidden layer with nodes.

Listing 1: Network Nodes

```
// Geometric Pyramid Rule to calculate hidden layer nodes
double hiddenGPR = Math.sqrt(vectorSize * LANGUAGES);
// recommended dropout rate of 0.8
int dropoutRate = (int) (hiddenGPR*0.8);
```

The input nodes were determined after testing with different values and finding a good balance of criteria and results as outlined in the Training and Testing section. It became clear that higher amounts of input nodes were more successful, with a lower threshold of 300 and a higher threshold of up to 1000 offering reasonable results. Tweaking the epochs or error threshold is necessary to prevent the network from training for vast periods of time for the larger input vectors. This inevitably comes down to a decision of trading time for marginal error reduction and accuracy increase versus training time.
Similarly, the output nodes are hard coded to 235 as there are 235 languages being evaluated within the network, with each language having an output node so that the network may evaluate a test to any language in the set.

5 Training and Testing

I tested the various iterations of network based on the following criteria:

- Epoch training time.
- Data-set testing time.
- Overall running time.
- Accuracy results from testing the kfold partitions.

In an extensive testing file included in the project repository and titled 'results.txt', I have documented testing singular variable changes. By which I mean, keeping all test parameters the same such as all network topology, but changing only vector size and thereafter comparing results.
What I discovered while testing networks including drop out, was the error decrease was marginally improved. The overall training and testing however did not change much, irrespective of drop out rate. This alerted me to a few different possibilities.
I tabled some of the dropout test results below in tables 2 and 3. The statistics results 1 and 2 below clearly show only minimal impact on the current topology.

Epoch	**Run Time**	**Error**	**Error Reduction**
1	21915 ms	0.0034401	0.0034401
2	20335 ms	0.0017552	-0.0016849
3	21524 ms	0.0013319	-0.0004233
4	20786 ms	0.0012532	-0.0000787
5	22786 ms	0.0011959	-0.0000573
6	25342 ms	0.001153	-0.0000429
7	21450 ms	0.001118	-0.000035
8	21322 ms	0.0010914	-0.0000266
9	20897 ms	0.0010727	-0.0000186

Table 2: **2Gram, 500 Vector, 9 Epoch - No Drop Out.**

Epoch	**Run Time**	**Error**	**Error Reduction**
1	24491 ms	0.0035231	0.0035231
2	22377 ms	0.0016861	-0.0018371
3	23430 ms	0.0013046	-0.0003814
4	21168 ms	0.0012306	-0.000074
5	20099 ms	0.0011782	-0.0000524
6	21159 ms	0.0011377	-0.0000404
7	20749 ms	0.0011033	-0.0000344
8	20965 ms	0.001072	-0.0000313
9	21223 ms	0.0010489	-0.0000231

Table 3: **2Gram, 500 Vector, 9 Epoch - With Drop Out.**

```
INFO: Training Complete.
Total Train Time: 196357 ms
INFO: Correct Predictions: 10564.0
Total Predictions: 11739.0
Total Test time: 4335
INFO: Accuracy = 89.99062952551324
INFO: Total Run Time: 200692 ms
```

Figure 1: No Dropout Stats

```
INFO: Training Complete.
Total Train Time: 195660 ms
INFO: Correct Predictions: 10519.0
Total Predictions: 11739.0
Total Test time: 4207
INFO: Accuracy = 89.60729193287332
INFO: Total Run Time: 199867 ms
```

Figure 2: With Drop Out Stats

Issues with hashing the initial text? Issues with the network topology used? What about different activation functions? The table in the topology section is the result of the following research and testing.
Upon further research I read through the available activation functions and their proposed usages.
The Encog ActivationFunction Interface provides a list of the available activation functions. Of particular interest to me having discovered the efficiency and performance of the ActivationElliottSymmetric, was the description regarding it's promise in classification tasks as opposed to the TANH function. This was interesting as in my results table 1, we can see that Elliot takes longer than TANH to train the same network, yet less time to self test with the crossvalidation kfold partitions.
The ElliotSymmetric function is reffered to as a 'Computationally efficient alternative to ActivationTANH'. Perhaps with more extensive testing this would become more apparent, I am inexperienced with how these functions ought to be tested but my results are similar with regards to time and performance.
Further to the discovery of highly performant functions, I found other activation functions to be extremely poor and inappropriate for the task of classification. For example the SIN, Gaussian, Linear functions were so poor they would have less correct guesses than some seemingly random guesses. This shoe that the network was incapable of learning anything of value using these functions.
In fact, when reading through the documentation for these functions, the Linear for example is documented as follows:
'This activation function is primarily theoretical and of little actual use.'
While the others are more suited to NEAT or CPPN networks.

6 Predicting

In my testing the predicting ability of the network, It has been poor.
I have found that even with an extensive directory of multiple languages, the network fails to accurately predict languages most of the time. This is an interesting outcome as while the network shows a very high percentage of accuracy after 3 minutes of training (often within 1 degree of 90%), the model fails when tested against live data.

There are certainly areas where the network might be improved. For example, preprocessing the text both training and live, might improve overall accuracy. I am also certain after more testing that I would find more optimal settings for the topology.

7 Conclusion

I decided to stick with training the best network I could which had a 3 layer topology, as there are many different topologies which may perform equally or better than this one.
The network *has* to be a multi-layer perceptron architecture in order to be able to classify languages, such tasks are not linearly separable and as such require multiple layers as opposed to a single layer or single perceptron.
I found that configuring the network with certain activation functions showed no promise. As documented in my testing training section file, certain functions appear to be more suited to classifying this type of data.