Language Detection Neural Network

Morgan Reilly
G00303598
Bsc.(Hons) Software Development
Artificial Intelligence
Galway-Mayo Institute of Technology

***Abstract*—A Language Detection Neural Network with Vector Hashing**

The purpose of this project was to construct an application which could generate a neural network model which was capable of language detection, and which could detect a given language from text file specified by a user. To do this it relied on the use of an Artificial Neural Network.

An Artificial Neural Network is a mathematical model which attempts to stimulate structural and functional aspects of biological neural networks.

The ANN used in this project is known as a Multi-layer Neural Network. These are feed-forward neural networks which consist of at least 3 layers. These layers are composed of an input layer of source neurons, a number of hidden layers consisting of computational neurons, and an output layer, which determines the classification, or output, of the model.

This project also uses Vector Hashing. Vector hashing is the mapping of feature values to indices in the feature vector set. This is done by applying a hashing function to each generated n-gram value from a parsed text file and then storing their hashed values directly as indices.

I. Vector Processor

The vector processor handles the generation of the training and testing data used for this application. The file used is the *WiLi-2018-small-11750.txt* which contains 11750 lines of different language samples pulled from various Wikipedia samples.

The size of the n-grams and vector size is set as a default of $2/300$, where *2 = n-gram size*, and *300 = input vector size* (my current machine isnt great for this). This can be changed in the configuration menu of the application, and generally yield's an accuracy of 75%.

The vector size correlates directly to the size of the input nodes in the Neural Network i.e. a vector size of 300 = an input node size of 300.

The vector processor first reads in line by line of the WiLi data set, and on each line it splits the line into 2 records, text and language.

The text is converted to lower case to keep consistency up, and the language is stored as per the WiLi data without any change.

After the line has been split an initialisation of the *vectorNgram[]* store must be completed. This is done simply by looping for the specified vector size (which can be specified in the console) and by setting each value at the index to 0.

The n-gram process is quite simple also, and integrates into the vector hashing. It first loops over the length of the text in the line being processed, making sure to stay within the bounds of the specified vector size. It then generates n-grams by creating sub-strings at the current index of the text, and the current index of the text plus the specified n-gram size. This will store a Character Sequence of n-grams.

This n-gram char sequence is then vector hashed by using the formula *index = ngram.hashCode() % vectorNgram.length*. After which the index of the vectorNgram is incremented *vectorNgram[index]++*. Note: This could have been done on a single line, but for my own clarity I felt it was better to have it broken out into more lines.

When the vectorNgram array has been filled with the correct vector hashes they are normalised between the range of -0.5 and 0.5. I chose this range since many NN projects seem to follow this heuristic, and also it yielded quite a high accuracy for this set up (75%).

After normalisation, the CSV file needs to be populated. This is done through the use of a

StringBuilder.
First the vectorNgram array is looped over and at each index is appended to the string-builder.
The same thing is done for all of the languages, except there is first a check to see which language is being processed. The processed language is appended with a 1.0, and every other value is appended with a 0.0. This allows for ease of classification in the NN.
The final check is to remove any unwanted commas from the csv file. This is simply done by setting the length of the builder and removing the index at -1.
On each iteration of the string-builder writes and populates the CSV file *data.csv*.

II. Neural Network

As outlined above, an Artificial Neural Network is a mathematical model which attempts to stimulate structural and functional aspects of biological neural networks. The neural network used in this application was fairly simple and used 3 layers.

A. Topology

The architecture of this neural network consists of 3 layers.
The first layer is the input layer, which is determined based on the vector size, consists of no activation function, a true bias, and input nodes = vector size. There is generally no activation function required for the input layer as the input nodes do not perform any computations. The bias being set to true means that its weighted sum of inputs will be adjusted to best fit the data.
The second layer, which is the the first and only hidden layer of this network, consists of a Sigmoidal activation function, a true bias, a pre-determined number of hidden nodes, and a drop-out layer. The activation function I used was purely down to experimentation. For many doing this project a TanH activation function suited, for others ReLu worked. However I found, for both time and accuracy, that Sigmoidal worked best. There is a bias in this layer, which provides the same functionality as outlined in the description of the first layer. The number of hidden nodes is calculated by using the Geometric Pyramid Rule, which is: $Nh = \sqrt{(input\_nodes * output\_nodes)}$. I based my choice for this rule on pre-determined calculations of the other available heuristics and came to my decision based solely on the outcomes of those calculations where I found the geometric pyramid rule to be the most appealing in terms of hidden node quantity.
The third, and final layer, is the output layer of this network. This layer outputs the classification of the data. The layer consists of a soft-max activation function, which is necessary to normalise the data into the desired output shape. It has no bias, and consists of 235 output nodes, which correlate to the number of languages enumerated *Language.java* class.

B. Loading Data

After preparation of the data through the use of *VectorProcessor.java*, the data is read into a codec which handles CSV format. With that codec that data is converted into an Encog specific data type which correctly stores the codec CSV values.

C. Train Neural Network

When the data has been correctly loaded, it is fed into a *FoldedDataSet*. A folded data set allows the data to fold into several equal data-sets, this is used primarily for cross-validation.
After the data is folded, it is trained using Resilient propagation. Back-Propagation requires the NN to be fully connected, and is where the weights are modified as the error rate is propagated backwards. Resilient-Propagation is a learning heuristic which is used in a feed-forward network. Real numbers are propagated forward, while errors are propagated backwards.
After the propagation is completed the data is cross validated. This is a sampling procedure used to evaluate the data, and to assess the predictive performance of the model. In this case we use 5-fold cross validation.
The model is then trained by calling *EncogUtility.trainToError()* which trains the model based on the minimum error rate which the user can specify. Optimal results piqued with an error rate of 0.0001%, however this seems to be subject to the back-propagation configuration.
The model is then saved to system for future use with user input.

D. Test Neural Network

The neural network is tested by first getting pairs of data. This data consists of the actual and the expected, as per the *data.csv* file.
For each pair in this set we get the actual by computing the value with the function *network.compute(pair.getInput())*. Expected is set with *pair.getIdeal()*.
Once these values are set we should loop over both of them respectfully, comparing the data, and recording the number of times the result is found in the actual and expected. Should the result be found, then the number of correct values is incremented by 1.
The total is found by incrementing at the end of each loop.
The accuracy is found by using *accuracy = (correct / total) * 100*.

E. Generate NN Prediction

When the neural network has been fully trained and saved to a desired accuracy the user can then get a prediction by passing in a text file to the Language Classifier.
The Language Classifier acts in a similar set of steps as the Vector Processor, the only difference being that it doenst have a language attached to it, so some of the steps are relating to language are omitted.
The prediction itself follows a similar format to the testing of the neural network, outlined above, by computing the output and comparing it with a lower bound value. The closer the value, the more likely the language output.

III. Activation Functions

In this project I had experimented with 4 types of activation functions:

1) Sigmoid: This activation function squashes the values between a range of 0 and 1. I found it to yield the highest accuracy for my model set up.
2) TanH: This activation function squashes the values between a range of-1 and 1. This yielded a decent accuracy and had a relatively quick performance time with my model set up.
3) Rectified Linear Unit (ReLU): This activation function squashes the values in a form of $f(x) = max(0, x)$, and leads positives to be set to 1, and negatives to be set to 0.
4) Softmax: This activation normalises the data into the desired output shape.

IV. Hidden Node Calculations

In calculation of the hidden nodes I took an input value of 150, and an output value of 235. With these I calculated the following based on common heuristics:

- $Nh = 150 + 235 = 385$
- $Nh = (150 * 0.66) + 235 = 334$
- $Nh = \underline{150 + 235 = 385}$
- $Nh = \sqrt{(150 * 235)} = 187.74$ (Geometric Pyramid Rule)

V. Conclusions

In conclusion of this project I found it to be a welcome challenge. The concepts of NN building were not new, but the Encog framework and the lower level configurations of the Neural Network made for a satisfying learning outcome. I think my network could have been definitely calibrated a little better for a higher accuracy but I feel a lot was down to so hardware limitations on my device vs the specified training time in the project. It is a topic I wish to pursue further and I was glad to receive this project as the final assessment of my undergrad course. Thanks for all your help throughout the years John.