Troi Chua

May 6, 2017

COGS 103

The Exclusive Or, or is it Inclusive?

There are many things in human communication that are highly ambiguous. This is to be expected as all forms of natural language are ambiguous (Russel and Norvig, 861). We use metaphors and analogies to explain concepts that are hard to convey, such as defining red and orange colors as warm and grays as cold. When people talk to each other, they use gestures to assist with convey expression, sense of time, direction, reference, and much more. When people talk to each other, they do not have to be explicit in their speech to the point that complete sentences and directly talking about the subject hand can be completely unnecessary. The ability to comprehend ambiguity in language can be attributed to knowledge through experiencing the world for the context of the subject at hand. While it may not be exactly as the human mind accomplishes it, an artificial neural network can be taught to process and comprehend this ambiguity.

One of the most famous problems in philosophical logic, computer science, and the cognitive sciences is that of the exclusive or problem. In its simplest terms, an exclusive or occurs when options are presented but the choice is limited such that not all the options can be picked at once. There are many factors to this, such as to the degree of freedom, quantum/limit in choice of options, and the number of options. However, one of the most subtle and difficult problems of the exclusive or is identifying when an or is exclusive or inclusive.

The “clusivity” of an or statement is not to be confused with the “ternality” of the statement. Whether an or statement is internal or external depends on whether the options

presented are the only options that can be chosen. The simplest example of this is a multiple-choice exam question. Most multiple-choice questions come in the format of “pick a, b, c, or d”, an inclusive or. The choices presented are the only options. An alternate question that may include “None of the above”, implying that there may be an answer not in the options. Another example is in surveys when they offer a fill in an option that is labelled “Other” and comes with a fill in the blank option. In most surveys that ask for a college major, “Cognitive Science” is not listed and must be input in the “other” section. Again, these are examples of or statements that focus on being either internal or external, not specifically inclusive or exclusive.

The “clusivity” of an or statement is all about how many options are available to be chosen within the options presented. An inclusive or allows for all the available options to be chosen. In logic terms for an or statement with two options, a and b, it is considered true if at least one of them is true, which means it is also true if both are true. In an exclusive or statement, it is false if both a and b or true: you can have one, the other, but not both.

What this ultimately boils down to is a simple binary choice of whether an or statement is exclusive or inclusive, making this classification an exclusive or problem. A most basic neural network framework design would take in a statement, be processed through the neurons of the network, and spit out either a one or zero, for exclusive and inclusive respectively. Binary classification is a problem that has been solved many times over with neural networks, but doing anything in the realm of natural language processing is a challenge in is it itself. Natural language processing is computationally expensive for any system, and how the data is fed into the system also important.

Luckily, modern machine learning libraries have framework designs that allow for simplified data input and leveraging graphical processing units (GPUs) for the heavy

computations. Most of these libraries will accomplish the same tasks with certain degrees of success based on how the network is designed, so picking one that you are most comfortable with. For this project, Google's Tensorflow was used for its choice of interface being Python, an interpreted programming language with simple syntax in comparison to fast compiled languages like C++, its portability, and its plethora of tutorials and project examples that have been made due to its popularity surging popularity since its release in November of 2015. Tensorflow, like any framework, is difficult to learn, but intuitive once known. There are many complications when it comes to researching how the library functions. Following the standard tutorials do give some general idea of what is going on, but they are highly specialized to the task they are accomplishing. For example, almost all the tutorials on Tensorflow use preexisting datasets in specific formats, such as MNIST, with built in functions to extract that specific data. There almost nothing explicit on feeding in random raw data.

Tensorflow programs have two main parts: preprocessing setup and processing sessions. Preprocessing is where all the important defining of the model and its components is done and the data is loaded into the program. The Tensorflow session is where the data is run through the model and the network is train. If the GPU version is utilized, the data is sent to be computed on the graphics card rather than on the CPU to leverage greater parallelization. As this is data transfer is all done in the background the same code can be used and the version of Tensorflow will do all the work. This is what makes Tensorflow portable like python: so long as someone has the library installed in their python library, it will run. There are also two versions: a CPU version that can run almost any computers, and a GPU version that can leverage an NVIDIA graphics card if the CUDA library is installed. As stated earlier, the GPU version is faster as it

can run more calculations in parallel due to having an exponentially higher core count than a CPU.

Getting the data that needs to be processed took place before designing the model as a dataset to train and set the network on would be necessary. Sadly, within the time constraints of this project, an ideal quantity of data was not obtainable. There was no dataset that focused specifically on exclusive or classification, so a dataset had to be hand built. For the exclusive or classification problem, the data is text that had to be converted into number representations that could be processed in a regression model and spit out the output.

Getting suitable text was simple as text data is everywhere. For this project, the Natural Language Toolkit (NLTK) datasets were chosen for NLTK's ease of extracting sentences from their database and the datasets include some famous books, like Alice in Wonderland, Macbeth, and a few works by Jane Austen. An extractor script was written to parse through the data and isolate sentences that have the expression ' or ' in them, to ensure that only the sentences with the full word 'or' were extracted and not sentences with 'or' as a substring in larger words.

Five specific texts were chosen and assigned numbers. For the training set, the first 40 samples are divided into five partitions, each one containing the first eight lines of each respective datasets. For the testing set, the first 10 samples are divided into five partitions, each one containing the first two lines of each respective datasets. The rest of each set was filled using a dice script, run for each set of four lines for training and each line for testing, that matched the text with the number of lines in that text. When run, the script would spit out the text number and a line number. The corresponding set of four that the line number belonged were assigned to the next set 4 lines of the training set. For the testing set, the line that was spit out was chosen to be added. This culminates to 80 training samples and 20 testing samples. These sentences were

stored in separate text files than the raw data. Two different scripts were then used to quantify the word values of the training and testing data text files. Oddly enough, these scripts did not work when run in the command line, but work in an IPython session.

The first script simply assigned an index value to each individual word based on a lexicon size of 50,000. This process is data compression as it assigns the symbolic patterns that composes makes up the word and assigns a numeric symbol (index) for it (Russel and Norvig, 866). The word itself is constant with all versions of it throughout the data set (if the first 'or' is 10, all 'or's are 10). To normalize things into a scale of zero to one, the function was told to assign them an index of 1/50,000. The script utilized a preprocessing method in Keras, a higher-level machine learning library with backend interfacing with Tensorflow. While this module was utilized, Keras itself was not used for the actual network. To make sure each sentence was the same length, each sentence vector was padded with a value of -1 until they were the same length as the longest sentence. These values were then stored in separate text files for consistency in training. These single number to word representation was what was utilized in the network.

The second script utilizes Gensim's implementation of Word2Vec to quantify each individual word in each sentence into a floating-point vector of length 100. Before the vectorization, each sentence is padded with a random string that is guaranteed to not appear in any of the texts, like a password. For this dataset, a made-up word was used. The same words have the same vector, so that made up word will have the same vector value across all the datasets. Like the simple number assignments of the first script, the vector values were stored in text files for a consistent dataset. Unfortunately, due to this producing a three-dimensional array, each sentence had to be stored in its own text file with each word taking up a line. While using this data would be more ideal as most natural language processing data vectorizers words, using

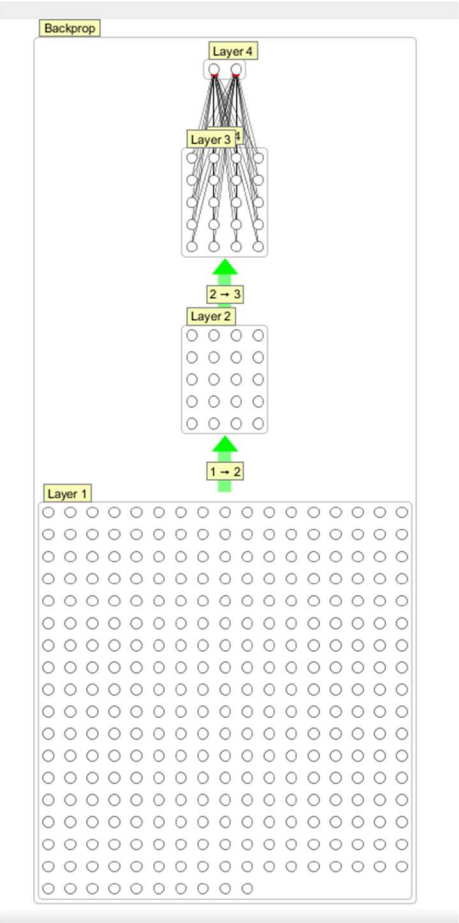
this set would involve using some of the more complex aspects of Tensorflow, such as an input pipeline. Tensorflow's tutorial on the subject is not very clear or intuitive to learn and apply within the timeframe of this project. This dataset is there, but not used.

The training and testing set need to be analyzed and labeled by hand, which means that the labeling is based on the labeler's interpretation. This presents an interesting problem as many of the sentences have stacked or statements. For the sake of consistency, it was decided that these sentences would be labeled based on the highest order of or statement. For example: (a xor b) or (c or d) is inclusive because the most globally applied or (the one in the middle) is inclusive. Zero and one are assigned to inclusive and exclusive or respectively and the labels were stored in their own text files.

With all the data and labels in text files that can be accessed and loaded into the network, it is time to start designing it. Unfortunately, most tutorials for Tensorflow only deal with the MNIST, which is so basic that it is considered the "Hello World!" of machine learning. To make matters worse, Tensorflow has built in functions to extract, parse, and batch the data for the program automatically that it is not clear how someone would do so with their own raw data. Luckily, the opensource nature of Tensorflow projects leads to there being a lot of sample code on Github. User sorendip had written an IPython Notebook with a binary classification model that used dummy data, so it needed to be sorted, batched and fed into the network in batches manually. The network used for this exclusive or classification problem is based heavily on his implementation.

As stated earlier, a Tensorflow program is split into two parts. The first part, preprocessing, is where everything is defined. The quantified data and labels needs to be loaded into the program and reshaped so that it plays nice with Tensorflow's interface. Placeholders

need to be made as variables for the data to be passed in through. Next, we define the network architecture: a four-layer feed forward network. While the Simbrain software was not used for training the network due to the computational expense, it can still be used to produce visualizations on what the network looks like. There are many ways to implement this, but one, Pythonic, way to keep things organized is to do so with dictionaries. A dictionary containing the attributes of the weights for the individual layers as well as their biases will help when implementing the model. For this network, two hidden layers with 20 sigmoidal neurons are between the input and output layers. At least one hidden layer is needed as this classification problem is an exclusive or problem, it is linearly inseparable and cannot be solved by networks that are only 2 layers (Yoshimi, 100). A network is considered deep if it has more than a single hidden layer. The network is defined as:



$$layer_1 = input_{ij} * W_{ij} + b_{ij}$$

$$layer_2 = in_{1ij} * W_{ij} + b_{ij}$$

$$output = layer_{2_{ij}} * W_{ij} + b_{ij}$$

The cost error function is Tensorflow's built in sigmoid cross-entropy with logits, the inverse of sigmoid, because that was consistently the recommended function for binary classification (Polamuri, 2017). The cost function is minimized by Tensorflow's gradient descent optimizer with a learning rate of 0.01 for a balance of swift and accurate learning. Combined,

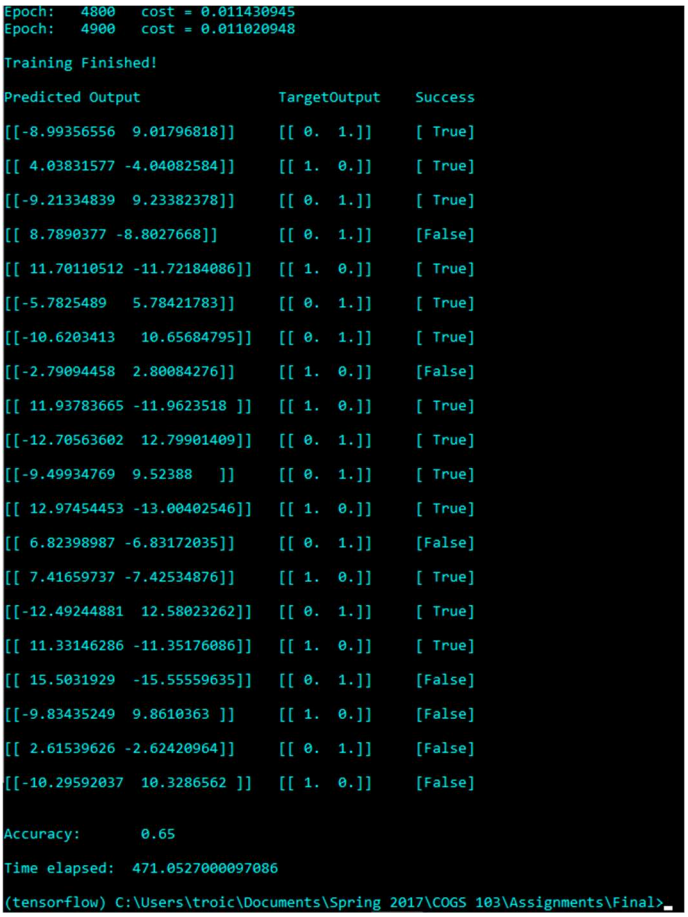
these two are apply backpropagation to the network so that the weights are updates. These functions in the model implementation use Tensorflow's inbuilt functions and data structures, and are simple containers until fed data in a Session.

The second half of a Tensorflow program is the session which is where all the computations occurs. The session then splits up the data and labels into equivalent sized to train the network in iterations. This batch learning method computes the total average error made over a training session and applies its change at the end (Alpaydin, 285). Because of the way the data is stored, an inner for loop within the loop that iterates through the batches is needed so that each sentence of 299 words is passed in. This continues until all steps have been run through. When it is done, the session then iterates through the testing data and sees how accurately it was trained.

Through working with this data and network, many interesting quirks about Tensorflow came to light. For example, at one point the network would only output ones. The network would originally calculate the accuracy by how far the network was from the target. However, for classification, the library estimates how close things are based on how many options there are. For discrete binary classification, there must be two classes of output, 0 and 1. To account for this, the labels need to be made into pairs so that it could be one hot encoded. The labels had to be turned into (0,1) for inclusive and (1,0) for exclusive.

Training the network is in itself an adventure. On an Nvidia GTX 970, the program runtime is almost 10 minutes for the 5000 training cycles the network runs through, showing how computationally time consuming it is. However, it is very clear that the size of the total number of neurons in the network is what influences the load the most. Numerous quantities of layers and hidden units were tested, but after yielding insignificant gains at the cost of

computational load, the current architectural design of a single hidden unit with 10 neurons was decided on for simplicity of meeting a minimum to accomplish binary classification.



```
Epoch:   4800   cost = 0.011430945
Epoch:   4900   cost = 0.011020948

Training Finished!

Predicted Output                TargetOutput    Success

[[-8.99356556  9.01796818]]     [[ 0.  1.]]     [ True]

[[ 4.03831577 -4.04082584]]     [[ 1.  0.]]     [ True]

[[-9.21334839  9.23382378]]     [[ 0.  1.]]     [ True]

[[ 8.7890377 -8.8027668]]       [[ 0.  1.]]     [False]

[[ 11.70110512 -11.72184086]]   [[ 1.  0.]]     [ True]

[[-5.7825489   5.78421783]]     [[ 0.  1.]]     [ True]

[[-10.6203413   10.65684795]]   [[ 0.  1.]]     [ True]

[[-2.79094458  2.80084276]]     [[ 1.  0.]]     [False]

[[ 11.93783665 -11.9623518 ]]   [[ 1.  0.]]     [ True]

[[-12.70563602  12.79901409]]   [[ 0.  1.]]     [ True]

[[-9.49934769  9.52388   ]]     [[ 0.  1.]]     [ True]

[[ 12.97454453 -13.00402546]]   [[ 1.  0.]]     [ True]

[[ 6.82398987 -6.83172035]]     [[ 0.  1.]]     [False]

[[ 7.41659737 -7.42534876]]     [[ 1.  0.]]     [ True]

[[-12.49244881  12.58023262]]   [[ 0.  1.]]     [ True]

[[ 11.33146286 -11.35176086]]   [[ 1.  0.]]     [ True]

[[ 15.5031929  -15.55559635]]   [[ 0.  1.]]     [False]

[[-9.83435249  9.8610363 ]]     [[ 1.  0.]]     [False]

[[ 2.61539626 -2.62420964]]     [[ 0.  1.]]     [False]

[[-10.29592037  10.3286562 ]]   [[ 1.  0.]]     [False]

Accuracy:       0.65

Time elapsed:  471.0527000097086

(tensorflow) C:\Users\troic\Documents\Spring 2017\COGS 103\Assignments\Final>
```

The results that the network produced were disappointing, but not unexpected. Through the many runs of training, the cost produced by a run would decrease significantly, but could be rather inconsistent and ranging from being below 0.05 to being fixated at a point well above 1.00. However, this learning from the data set did not translate well to performance when tested on the testing set. The network consistently gets between 45 to 55 percent accuracy, and occasionally hits a high of 65% as seen in the figure to the right.

The range of 45 to 55% falls in the range of the proportions between test cases. For this training set, the split between exclusive and inclusive or is 45/55 respectively. To verify that the network was not simply trying to guess all exclusive/inclusive for the runs that achieve 45 or 55%, the network was programmed to output the predicted values next to the expected values. As shown in the figure above, it is not. The output also shows that a 1 is a positive value and a 0 is a negative value for the model predicted output.

While the 65% result is approaching a decent accuracy, its infrequency does not afford it much significance in comparison to the common range stated above. Out of sheer curiosity, an alternate dataset that was created was fed into the network to see how the data may differ. That dataset never succeeded in breaking out of the 45 to 55% range. This is not a good accuracy to be

in because it is at the state in which a guess could be a coin flip. In spite of this, the variance in the predictions the trained model produces shows that the network is putting in effort to correctly predict whether the or statement was negative or positive.

There were many problems to the approaches taken in this project. The dataset is undoubtedly too small as there were only two sets of 80 training samples and two sets of 20 testing samples. This was due to time constraints: the data was easy and quick to gather, but each sample had to be hand analyzed and labeled. The analyzing and labeling was done by a none-linguistic expert whom had to judge whether the statement was exclusive or inclusive. The judgements may not be accurate, especially when interpreting complex statements with stacked or statement. There was also the set training and testing sets. Most machine learning use cross-validation on a single set of data to split the test and training datasets so that between runs, the sets have more variance and aren't overfit to a set. The time constraints also lead to an important oversight: a large part of the data that is filler words with little to no context, like "a" and "the". And as stated earlier, if time constraints could afford it, it would have also been ideal to use the word vectorized data sets, but would have required extra time to learn Tensorflow's vague input pipeline system.

The fundamental issues that arose with the results generally stem from insufficient data. A lack of preexisting datasets forced the need to gather data and for it to be labeled by the experimenter and network designer. While gathering data was neither difficult nor time consuming, labeling them was. The lack of quantity and variety in which parts of the data are for training and testing lead to limited results and stunted the potential for the network to possibly learn better.

Tensorflow came with its own set of problems. The library and systems were all operational and bug free, but utilizing the framework from a near fresh point of view fell short of being intuitive. There is a steep learning curve to finding all the resources on the libraries website and most tutorials that the open source community have produced are still built upon the basic examples the main site presents and comes with the same problems, such as expediting things with built in functions that do not show how things are done in the background. The expectation of prior knowledge is not unreasonable, but the documentation is still vague for providing how to do thing like load and organize raw data, and it is sparse when it comes explaining when different models are more useful. Tensorflow is still a solid platform that has the effect of hindsight: everything is there, can be used, and is simple to match tools to problems only after a user develops an intuitive knowledge base on which and what to use when.

Regardless of how poor the results may be, they show some semblance of the concept being possible. The exclusive or problem, also known as binary classification, has been solved through a variety of implementations, and doing so for classifying the difference between an inclusive and exclusive or statement should not be so difficult in comparison. Again, a lack of data is what stunted the possibility of further research on network architecture. Tensorflow's interface has some quirks to work around and is a powerful tool that can be used to implement a network capable of learning how to accurately identify whether an or statement is exclusive or inclusive. There just needs to be more data for further design research to be conducted and a network to be trained on.

Bibliography

Alpaydin, Ethem. *Introduction to Machine Learning*. Cambridge: MIT, 2014. 239-307. Print.

Polamuri, Saidmadhu. "Difference Between Softmax Function and Sigmoid Function." Dataaspirant. N.p., 7 Mar. 2017. Web. 10 May 2017.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Upper Saddle River: Prentice-Hall, 2010. 693-736,860-882. Print.

Soerendip. "Soerendip/Tensorflow-binary-classification." GitHub. N.p., n.d. Web. 7 May 2017.

Yoshimi, Jeff, Chelsea Gordon, Scott Hotton, and Zach Toshi. *Neural Networks in Cognitive* Science. N.p.: n.p., n.d. Print.