Classifying Hate Speech:

An Exploration of Neural Net Parameter Considerations

# Introduction:

***A warning****: many language tasks may involve classifying content that is objectionable, and the data used in this project contain quite a bit of objectionable content, including racist, sexist, homophobic, or otherwise offensive text in the form of Tweets, and this report will contain examples.*

Deep learning is a powerful tool for analyzing and interpreting unstructured data. Here, I implemented a simple but customizable neural net framework for classifying natural language as objectionable or not objectionable. The data that is training and testing the neural network is a sample of Twitter tweets, which are numerous and maintain a convenient structure (maximum length and homogenous tweet style). By manipulating various parameters of an otherwise simple neural net, we see changes in a variety of performance metrics with the goal of optimizing a neural network on a given dataset.

The implications for such an application range from data mining for market or academic research to lightweight content classification for social networking or search engines. In this case, it is easy to imagine a pre-trained version of such a neural net being used to help moderate text posts on a social network. Alternatively, portions of a longer text can be fed through the net iteratively after to evaluate an entire document for possible objectionable content.

The framework itself is extensible and customizable. It is easy to apply new data sets, extend the topography of the hidden layers, incorporate semi-supervised applications by first training the set on a larger corpus and freezing layers, and it can be used as a tool to explore the computational limits of neural network implementations on particular hardware and more generally.

The findings in this case study demonstrate some very fundamental pieces of neural net implementation for Natural Language Processing tasks, and later its complexity and the various considerations surrounding the pre-processing, network topography, optimization parameters, sizing, and reuse choices.

Project Repository:

A Preview of some project elements: <https://youtu.be/65sUTDyaCWs>

# Why Neural Nets?

Neural Nets have been employed in Natural Language Processing contexts for a variety of applications, including both classifying and generating natural language according to training data. Original NLP techniques had been based on linear modeling approaches, but recently, neural nets in combination with powerful computing technology have allowed for significant breakthroughs in language modeling.

One application of neural nets today is the generation of language. Given a large set of mappings between natural language and output training labels, a neural network may be employed to generate never-before-seen output. Before neural networks, this was a challenge because the sequential nature of language was not easily accounted for with linear modeling approaches, but the neural net can implement modeling for this naturally. This functionality can be extended further for learning about completely new and unannotated data efficiently, such as with the transfer learning capabilities of semi-supervised learning as well.

Still, their capabilities require taming their implementations much consideration, and very successful applications for certain NLP tasks still remain elusive. A particular challenge has been the use of neural networks for sentiment analysis, and some reasons for this are illustrative of NLP challenges more broadly:

Sometimes, language is modeled at the word level, irrespective of sequence or relationship between the words (“Bag of Words” model). Order and relationships are important because some words modify completely the meaning of others. Sentiment analysis is hard because “I’m happy, not sad” and “I’m not happy” both contain similar words and may be scored similarly under this model, yet the meanings are clearly different because of negation and order. Neural networks can be trained to account for sequences and word modification, whereas unsupervised natural language approaches (such as Latent Semantic Analysis), tend to be computationally intractable or inefficient at that level of correctness.

Context is important as well. Often, a classification problem asks for an output based on knowledge that may not have been in the training set (e.g. a famous politician not yet seen). Some of this can be offset by pre-training a model, but there will likely always be an amount of textual inaccessibility for the field of NLP. Similarly, the meaning in language can depend on not only the textual context but also the physical context of its emplacement (e.g. the difference between a sentence in a history textbook and a Facebook post). I may not use satire within a textbook, but it is much more likely that I do on a social network.

These are just some of the problems demonstrated by thinking of how challenging it is to classify text as positive or negative in meaning. As of 2020, some sentiment analysis tasks are yet still beyond the abilities of natural language processing to wrangle: the Stanford Sentiment Treebank, for example, still only yields a 56.2% level of accuracy for fine-grained classifications (nlpprogress.com).

Training a neural net to detect positive and negative sentiment is not the goal, here, however. It is often ineffective and computationally very complex and expensive, irrespective of neural net models, and cutting-edge technology is at the forefront there. Instead, I want to explore a useful low-level introductory goal which may elucidate considerations for higher-level applications. Many social networking platforms employ algorithms to pre-emptively detect language that will be classified as objectionable, and so that is an example of what I have trained the net to accomplish.

# Neural Net: Theory

Problem to Solve: A neural net as a computing system implements a form of deep representational learning. Loosely, suppose we have a dataset of *k* inputs and *k* labels . In this case, the inputs will be strings (Tweets), and the outputs will be labels (0 for not objectionable language, 1 for objectionable). A neural net will model an approximation of a function which correctly matches the inputs to their correct labels

This is usually accomplished by minimizing some error function across the entire set through a series of trials in which data is processed, checked, and the network is re-weighted to estimate the real function more closely.

Neural Net’s Innovation: The crux of neural network’s use in supervised learning is the ability to train neurons through weighted training using back-propagation. Inputs are represented numerically, are transformed through some layers of “neurons” which become “trained” with more and more input, and are and are mapped to an output layer, and once trained, the neural network can be used for generative or classificatory tasks.

A traditional full-connected neural net layer is composed of many nodes in what is known as a multilayer perceptron composed of single perceptrons. Each individual perceptron obeys this formula yielding an output, where *w* is the weight of an input and *x* is the input itself, with *b* some bias:

The goal of the neural net it to minimize the loss function and arrive at a global minimum of error, so to do this it uses an optimization function, with each optimization function leveraging a form of gradient descen.

During each training iteration (epoch), the neural net considers some sample set of inputs, reweights its nodes according to the expected output and a given learning rate, and is tested on data to ensure it is being reweighted appropriately (validation data). Evaluation occurs as part of the trial-run process at each epoch, and then once again on an entirely new sample set after the entire network is trained. By training on data that is distinct from test data, we guarantee the neural net will be agnostic to the training data, and therefore the performance we see on the new data will be authentic. This process is generally balanced to avoid an over-reliance on the initial dataset, which will increase accuracy on the very specificities of that trial data while limiting the net’s effectiveness on new structures. This is called overfitting. In practice, this separation and tuning step will look like reserving a portion of a labeled dataset specifically for validation during network training, and a third set for the final test of the net, and incrementally testing all of the data on new instances of the neural net until optimal performance is met.

Because it is often useful to see examples in action, I will describe some of the neural net components in terms of the Abstract API I used to implement it, *Keras*. The most fundamental element used in Keras is called the Dense layer. According to documentation, the layer is an implementation of the equation:

Which is identical in function to our theoretical perceptron. Essentially, the dense layer is the commonly-used fully-connected neural net layer, in which each node of the layer is connected to every node in the previous layer, receiving their input and updating the parameters of the layer (the weights) through back-propagation for forward-feeding the signals into the next layer.

To extend the functionality of a neural net, many choose to implement embeddedness as a first network layer, and this is also an option in Keras. Embeddedness solves to some extent many of the computational difficulties associated with inputting large samples by reducing the dimensionality of the input and adding meaning by representing the input as a smaller vector composed instead of abstract “features” which are thought of as relationships between concepts within a sample set.

With dense layers, an embedded layer, and the back-propagation needed to re-weight neurons, we are now left with data to be interpreted as output.

In the final layer of the neural net, we implement yet another dense layer, but instead of fully-connected to many nodes, we instead only map the output to a single node. The output of the layer is mapped with the logistic function:

Which flattens to the range [0,1].

We use a sigmoid function at the end because we want to interpret the data as binary (though that is true only in this case. More generally, we may choose a different function and we may have a different output dimensionality). Then, the output interpreter will generate a most-likely estimate for the input data and we will have our output to either reweight or provide accuracy measures with.

# Process Pipeline:

## Data

I used a dataset retrieved from the website *data.world* titled “Hate Speech and Offensive Language” which contains tweets classified as “hate speech,” “offensive language,” or “neither” by users from the crowd-sourcing platform CrowdFlower. For analysis, I distributed the data into “objectionable language” if it was rated as either Hate Speech or Offensive Language, or it was classified as “not objectionable language” if it was rated as neither. This allowed a simple binary classification task. The text was uploaded from a .csv file and used as text and binary classification labels. I split the data into a training set (to be split for training and validation) and a test set at various ratios randomly as the characteristics of the dataset did not solicit systematic distribution.

Importantly, this dataset is characterized by the closeness in character of the objectionable language sample to the non-objectionable language sample. Though they have been manually classified similarly, they contain much content overlap, so detecting minor differences in syntax or usage is key for detecting the difference. Additionally, the dataset distribution between the categories is imbalanced. We see that the objectionable content largely outweighs the non-objectionable content (83.2% objectionable vs. 16.8% non-objectionable). This is an additional hurdle which we will consider, as we do not want the accuracy to just reflect the natural distribution (we want it to be greater than 83.2%).

## Pre-Processing

Pre-processing occurs between the raw data retrieval and its input to the neural net. This involves cleaning the data so that it is more effectively interpreted by the neural net and finally converting the cleaned text to numerical input into the neural net.

To clean the data, I performed punctuation removal, stopword removal, word stemming, and later tokenization and vectorization into an integer vector format before feeding samples into the neural network. This is discussed later and its effects are seen with the “pre\_process\_demonstration()” script

## Neural Net Set-Up

The neural net consists of several baseline layers and several optional or extensible ones. The data is fed into a compiled neural network as batches of strings first into the input layer:

1. **Input Layer:** processes input strings into a format digestible for the net by specifying shape and type but does not itself contain any trainable nodes.
2. **Vectorizer Layer**: Again, this layer does not actually train the net, but instead processes strings of text and converts them into a vectorized format according to a dictionary mapping all words among the set to indices.
3. **Embedding Layer**: Optional first traininglayer. Instead of a sparse Bag of Words representation, this layer converts the vectorized input into a dense vector representation. It is necessary to convert the vectorized format of string inputs to the dimensionality expressed in the subsequent layer.
4. ***n* Hidden Layers:** These do the bulk of the training and output calculation and are the fully-connected “Dense” layers Keras implements. They have several parameters used to compile the model and dictate back-propagation:
   1. **Nodes:** This is the size of the layer. It determines the size of the outputs from this layer. This will vary for optimization.
   2. **Activation:** The activation function determines the mapping of each node’s input to their respective outputs.
   3. **Bias:** By default, this implementation adds bias rather than normalizing the output.

The model will take a specific number of hidden layers, and the intuition is that hidden layers may represent sequential knowledge, wherein each layer contains less of an effect than the previous, but allows the output to be fine-tuned sequentially by different features.

1. **Output Layer:** This layer defines the output size. In the case of a binary classifier, the output coalesces to a single output signal, which is interpreted as 1 when high and a 0 when low. For binary classifiers, this is accomplished by setting the output activation as a sigmoid function.

Then, we consider set-up complete. Importantly, set-up is where many of the hyperparameters which affect the accuracy are contained.

## Processing

To train the network, the layers are compiled into a model with an optimizer function and a loss function:

**Optimizer**: Adam

**Loss**: Binary Cross-Entropy  
**Batch Size**: 8

We specify a loss function, optimizer, and batch size, compile the model, and run several epochs to train the data. The accuracy and loss functions are returned for each of the training data and the validation data for review. It is also noted that these parameters, specifically batch size, are simply arbitrary for the base model which demonstrates functionality and have not been optimized. There are different implementations which use different parameters, though.

An alternative to batch training is online learning, wherein samples that are processed are then discarded and the gradient descent optimization occurs stochastically. Had I implemented a sophisticated web crawler, for example, this is what I would have done. But for exploring hyperparameter optimization, batching is reproduceable and thus evaluable.

## Evaluation

Neural net performance can be measured and optimized across several metrics:

1. Accuracy: Whether the data performs sufficiently well on its own training data, validation data, and some completely new data that is within the scope of the training goals. This may be performed after each training instance or during deployment, should the neural net become deployed as a product.
2. Runtime: When a neural net takes many hours to process, we know it is not reasonable in supervised learning for light applications. Instead, heavy neural nets can be
3. Memory Usage: I am not as concerned with this metric as my entire dataset, neural net, and associated code fit comfortably in my personal system, though this will be of concern if a back-end were running many instances of different neural nets on different datasets.

# Discussion of Results

Based on the output from my baseline model, it appears the training data and neural net set-up did not lend well to increasing the overall accuracy of the model. First, I attempted to demonstrate baseline functionality by running the neural network on an entirely random dataset of random character words in a 50/50 distribution. The result was an approximately 50% level of accuracy. This was the expected outcome, and different parameters did not affect it. This model is demo’d with rand\_model().

Then, I wanted to test the net on the actual data. While the training data loss steadily decreased and its accuracy increased, the validation data does not show greater accuracy, and instead the loss increased after several epochs. We begin at an accuracy of around 82% after the average first epoch, and steadily the model achieves up to 84% accuracy, which is around 1% better than the expected distribution given in the training and validation dataset. Though it is only 1% increase in overall accuracy, at 84% the increase in accuracy demonstrates a more condensed level of improvement.

Though this is not a very significant jump in accuracy, it is not with very optimized hyperparameters, and yet it still it demonstrates the framework’s initial learning ability. Dataset information can be viewed with dataset\_info(), and baseline testing can be demo’d with baseline\_model().

Though further personal exploration did not yield an incredible increase in accuracy, implementing a grid search-style optimization or successive halving would likely yield a local optimum higher than my manual high of 84% accuracy. Similarly, selecting an alternative dataset altogether may increase the efficiency. It must be noted that, although the dataset used here seems to be labeled consistently, it is possible there is not enough structuring in the labels to be accurately classified. This may be a good explanation for the network not gaining accuracy with training, but we would want to look more deeply at the data to conclude this.

### Characterizing Data:

In one way, the data fed into the neural net is useful in illustrating the various capacities, limits, and optimums of a neural network natural language processing pipeline. However, even if the dataset had increased significantly in accuracy, it must be said the nature of the data also presents questions. Beyond questioning the validity of the dataset’s labels, it’s possible that, instead of measuring and training on sequential syntax and complex constructions, the network is instead picking up on keywords. This would amount to comparable performance with a simple word count-to-label calculation. Because objectionable language is often correlated with a particular vocabulary, the task itself may reduce the function of the neural net to simply detecting the words with high objectionability. Importantly, though, the network may still be demonstrably more efficient than alternative linear processing methods, it may be more lightweight than complex high-dimensional analysis, and it can be deployed with different datasets as well, anyway.

In any case, I decided I wanted to know more about the dataset and why its accuracy may be limited. I suspected it was because the topics all contained extremely similar language yet certain constructions made some objectionable while others were not. Upon analyzing the data with a Latent Semantic Analysis application I had developed earlier, I was able to confirm that the most salient topics shared throughout the dataset were, in fact, language I found associated with objectionability, but which can be used in non-objectionable contexts as well. Similarly, the topics contained extensive term duplication irrespective of a variety of input parameters, suggesting very duplicative inputs to the neural network. This may also be a contributing factor in the limited accuracy.

### Pre-Processing Considerations:

It is also important to discuss the pre-processing suite. It is clear from my implementation that pre-processing affects the output efficiency and accuracy. By removing punctuation, stopwords, and flattening many words with similar meaning using stemming, the input corpus is reduced very significantly, allowing faster computation among fewer terms due to a reduced sample dimensionality. These effects are documented in pre\_process\_demonstration(), where we see that stopword removal reduces size of individual samples considerably (from 23 on average before stopword removal to just over 12 tokens, in one case), and stemming reduces corpus size. These results are seen by running pre\_process\_baseline(), which explores stemming, stopword-removal, and neither and both. It appears that both stemming and stopword removal decrease the overall runtime (likely due to reduced corpus size and therefore smaller input dimensions) while increasing the overall accuracy of the model.

Embedding is another factor to consider. Overall, embedding added around 50,000 training parameters, which increased about ten-fold from the simple baseline model that didn’t incorporate embedding. The results are demo’d with pre\_process\_embedded(), which also incorporates the findings from the pre-processing investigation.

Next, I considered manipulating the inclusion of hashtags in the pre-processing suite. By including the text from hashtags, we see that the accuracy of the neural net actually decreases. With hashtag text, we get a maximum accuracy of 0.8373%, whereas with no hashtag text, we boost to 0.8383%. Hashtags serve to label tweet topics, but do not necessarily modify the contents of the tweet. Therefore, it may make sense that removing tweet hashtags removes unnecessary noise in the network. Countering this, in my final neural net, we see that including hashtags increases overall accuracy. This may be due to the increased modeling power of larger neural nets which may be able to learn more subtle correlations or modifications of hashtag usage.

Finally, to yield the most optimized version of the neural network, I varied the number of hidden layers along with the size of the hidden layers. These factors, in addition to whether I included an embedded layer, seemed to contribute most to runtime fluctuation, and this is expected, as increasing the number of trainable parameters causes additional calculations for each sample, including forward calculation and backpropagation. As suspected, increasing the number of hidden layers and increasing the size of a hidden layer contributes to an increase in neural net accuracy, but at the expense of runtime for training. These results are demonstrated in my final version of the neural net, in baseline\_model(), which is discussed in the conclusion.

In a more complex dataset, or even in this dataset, it may be useful to implement by training and evaluating with, instead of individual terms, using *n-grams.* Because I suspect this network generally relies on selecting individual terms to label as objectionable, it is possible this will not increase the model’s ability to model sequence, but it will increase the diversity of input terms. However, with a high frequency of modifying words and complex language sequences, grouping words sequentially (as in n-grams) may allow the hidden layers in the network to learn some sort of linguistic structure, though this implementation is computationally heavy, so it must be balanced with increasing network depth and size.

I do not report here use of the neural network framework on datasets other than the objectionable content dataset, but it is simple to input any dataset and label output to evaluate performance of many hyperparameters.

# Conclusion:

My experiences with developing neural nets and adapting their parameters has shown that neural net optimization is a complicated field, and my research has shown that it is also still a nascent field. Though neural nets rose to prominence several years ago, NLP applications with neural nets have a higher ceiling than we see currently, as shown by benchmarks consistently being surpassed in the field of NLP each year.

With my dataset containing objectionable language, I found that I was able to reach an accuracy of 84.00% when classifying the data on a test set. This was achieved on my baseline\_model() using Stemming, Stopword removal, inclusion of hashtags, a corpus size of 7,000, a batch size of 32, hidden layer count of 3, and hidden layer size of 600. Though this outcome isn’t exactly what I had anticipated or hoped, it is certainly an improvement to random estimation, and runtime for a single evaluation is very fast once the neural net is implemented.

The most salient result of my project is the amount of improvement that is still yet possible. Better data cleaning, different neural-net layer types and function optimizations, or perhaps just a more robust topography may yield optimized results beyond my experimentation here.

# Further Extensions

There was a lot still that I wanted to do with this neural net framework that goes beyond personal evaluation of neural net efficiency on detecting objectionable language in Tweets, but there is also a lot yet to be done with this dataset. First, I would like to explore visually and statistically the content of this dataset. It was challenging to identify much in the dataset that contributed to its difficulty in wrangling with the neural net. Perhaps there is some feature I have not accounted for that is tripping up the training, and I would like to adjust for that in some way. Additionally, it is possible the neural net has simply not been trained on enough data, though this problem is not rectifiable without additional data, and the purpose of this exercise was to optimize given the data at hand.

One way I may have adjusted the neural net is by adding support for dropout layers or probabilistic dropouts within the hidden layers already instantiated. This way, I will avoid the problem of overfitting too early, which was an issue when I set the learning rate at its standard rate, since the loss function only increased, stochastically, in some model configurations.

Additionally, more pre-processing options may have allowed me to adapt to some structural issues with the input. As much as optimizing a net itself is crucial to lightweight efficiency, pre-processing can mean the difference between extremely accurate and not accurate at all as well as memory-intensive with little return or computationally shallow. I included stopword removal, stemming, punctuation removal, and Twitter hashtag processing, but ideally I would have liked to have optional parameters to process emojis as terms, keep twitter handles, and allow for multiple word groupings and sequential learning using n-grams. I would have also added different options for the pre-processing goals, like different stemming algorithms and perhaps a dense vectorizer rather than integer one.

Distributed Computing: One important use of Keras for implementing TensorFlow networks is distributed or pre-emptive computing. A few simple changes to this application would allow the use of GPU processors and a more robust use of multiple threads to enhance the execution speed of neural net training. Keras is advertised as having powerful distributed computing options that are used by cutting-edge programs that may incorporate many GPUs in parallel.

Punctuated Language: A pre-processing goal which is particularly topical now is the conversion of punctuation-interrupted language, which is used to intentionally disrupt both human and computer interpreters. This pre-processing technique would leverage some form of AI as well to parse the word “apple” from “@pple\*”, or “Cornell” from “C\*rn\*ll,” though this is dangerous technology which removes the option to maintain agentively unincorporated language in a social network. Incorporating this would flatten these terms to their original words to be interpreted as objectionable or not.

Neural Net Layer Alternatives: Keras’ suite includes many different neural net layer options. I used dense layers and an embedding layer, but it is certainly possible to achieve different performance goals with, for example, Convolutional Neural Net layers (CNN). These can account for word sequencing and relationships irrespective of input order by considering feature locality, perhaps interpreting negation semantics more consistently, for example.

Optimization Suites: As it stands, my neural net implementation serves as a playground for exploring various neural net set-ups. Ideally, with this many parameters, you might see code atop the neural net generation and evaluation which minimizes some error function in its own way. Input parameters, pre-processing parameters, sizing, and dataset selection combine into a multivariate distribution of performance in themself, so they can also be optimized in an intelligent way. This may include sequential halving algorithms or grid-walking sequences.

Different Network Outputs: The data explored here originally consisted of counts corresponding to votes for three categories: hate speech, abusive language, and neither. Although for this project I decided to fold both hate speech and abusive language into the same category, it has been shown in research that they, in fact, contain different meanings. A multi-class output is viable by implementing the softmax activation function for the output layer and a categorical cross-entropy loss function rather than a binary one.

Deployment: The end-goal of a binary classifier probably involves deployment. In the case here, I imagine the neural net deployed as a filter for outgoing content on a social network (e.g. Twitter) which may be trained in the online mode at some slow rate to account for new types of objectionable language.

# Implementation Notes

All of the code was implemented and run on my personal laptop:

**64-bit Windows**

**2.30 GHz Intel Core i5-6200U CPU**

**8.00 GB of RAM**.

**Python 3.8**

I used the Python libraries *TensorFlow* run with *Keras*, *matplotlib* to generate plots, and a 3rd party natural language toolkit, *nltk,* for my pre-processing needs.

For pre-processing, from *nltk* I used a Snowball-algorithm stemmer, a tokenizer, and *nltk*’s stopword corpuses in English, Spanish, and French, and additional stopwords ‘rt’, and ‘u,’ which I found useful to remove in NLP projects involving tweets. I used *TensorFlow’s* vectorizer for the final vector representation for sample input.

My implementation of the pre-processor and my use of the latent semantic indexing evaluation technique was borrowed in small part from a separate project I developed for Latent Semantic Indexing.

# User Interface

To demo the software, please ensure proper set-up:

1. Python 3.8 environment (TensorFlow is not supported in newer versions of Python)
2. 3rd party package installation: tensorflow.keras, matplotlib, nltk, and tensorflow.
3. ‘languageclass.csv’ and ‘test.csv’ files are present in the same directory as the langnet4701

From the python console:

import langnet4701

Then, you may either run all demos:

run\_all()

Or, there is a list of functions available to individually run:

help()

Each function returned is inspectable with DocStrings that explain how they demonstrate use of neural network optimization steps.

Extended Use:

Aside from the built-in protocols, this module is extensible for your text classification neural net evaluation uses. Input into the pre-processor is simply a string, the model function is configurable, and it simply takes in a list-like item of strings as its input and a binary classifier (one-hot encoding of a single bit) as it’s output (see docstrings). The model can be configured, tested, and recompiled using standard Keras methods.

### Step 1. Retrieve Data:

Separate Data into one list of raw strings and one list of one-hot encoded binary.

### Step 2. Pre-Process Data:

Feed a single line into the pre\_process function, which returns a processed string according to your input parameters.

pre\_process(line, stops=True, stemming=True, hashtag=True)

### Step 3. Build a Model:

Set a variable to make\_model() to pass parameters into your model. Defaults for binary classifiers are set, but you can adapt them to test different functions: where

Make\_model(vector\_train, max\_tokens, output\_seq\_len, num\_hidden,

size\_hidden, hidden\_activ=’relu’, output\_activ=’sigmoid’, loss=’binary\_crossentropy’, optimizer=’adam’, embed=True)

where: **vector\_train** is the training input data (to adapt the built-in vectorizer)

**max\_tokens** is the maximum corpus size for the vectorizer

**output\_seq\_len** is the maximum length of input to pad training data to, once vectorized

**num\_hidden** is the number of hidden layers

**size\_hidden** is the number of nodes in each hidden layer

**hidden\_activ** is the activation for hidden layers

**output\_activ** is the activation function for the output layer

**loss** is the loss function for re-weighting

**optimizer** is the optimizer algorithm for gradient descent to find minimum error

**embed** is whether to include an embedded layer

Then, the model may be trained and evaluated using model.fit(), inputting the input and output data, batch size, and number of epochs to train.

# References

Bai, Qiyu et. al. “A Systematic Review of Emoji: Current Research and Future Perspectives. Front

Psychol, 2019.

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer Science+Business Media,

2006.

Keras API Reference: <https://keras.io/api/>

Goldberg, Yoav. Neural Network Methods for Natural Language Processing. Morgan & Claypool

Publishers, 2017.

Osbert, Bastani, et. al. “Measuring Neural Net Robustness with Constraints.” arXiv:1605.07262.

“Sentiment Analysis.” NLP-Progress. nlpprogress.com, 2020.

# Appendix

LSA Results: top 5 topics and their terms given 20,000 tweets and 50 topic reduction:

topic #0(15.673): 0.621\*"bitch" + 0.347\*"hoe" + 0.279\*"fuck" + 0.228\*"nigga" + 0.205\*"pussi" + 0.198\*"like" + 0.147\*"ass" + 0.130\*"get" + 0.123\*"got" + 0.121\*"shit"

topic #1(12.931): -0.766\*"hoe" + 0.597\*"bitch" + -0.121\*"pussi" + -0.065\*"loyal" + -0.065\*"nigga" + -0.061\*"got" + -0.058\*"love" + -0.045\*"girl" + -0.041\*"shit" + 0.040\*"fuck"

topic #2(12.209): 0.886\*"pussi" + -0.338\*"hoe" + -0.237\*"bitch" + 0.095\*"eat" + 0.080\*"fuck" + 0.059\*"nigga" + 0.057\*"get" + 0.054\*"right" + 0.047\*"like" + 0.040\*"good"

topic #3(10.282): 0.800\*"fuck" + -0.335\*"bitch" + -0.271\*"pussi" + -0.208\*"hoe" + 0.169\*"trash" + 0.158\*"nigga" + 0.150\*"faggot" + 0.099\*"ass" + 0.068\*"retard" + 0.062\*"shit"

topic #4(9.716): -0.505\*"trash" + 0.441\*"fuck" + -0.371\*"like" + 0.243\*"hoe" + -0.233\*"nigga" + 0.187\*"bitch" + 0.184\*"pussi" + -0.158\*"shit" + -0.152\*"look" + -0.116\*"lol"