Statistical Language Modeling Final Project Report

Lukas Justen **Keith Pallo** **Albert Guo**
{lukas.justen, keithpallo2019, zguo}@u.northwestern.edu

Abstract

In this paper, the BERT model from Google Research is applied to a custom sentiment regression task. The work focuses on understanding what is required when implementing the model, and what potential end users should consider. Overall, it is found that the fine-tuning approach works astoundingly well overall and is able to adapt to a difficult task with a single epoch of large data training. However, the model's complexity is also it's greatest weakness - when challenges occur it is difficult to fully diagnose bottlenecks and optimize towards a given problem. In the end we conclude that users of the model should carefully consider when to apply BERT and attention based models generally, as other methods may offer a more malleable and easily understood approach.

1 Topic & Motivation

In this project, we seek to explore the application of the statistical language model recently developed by Google Research, commonly referred to as BERT. At its core, BERT makes use of the newly developed concept of attention via bidirectional transformers. The goal of this project was applying BERT to a new regression based sentiment task by fine tuning the pre-trained base model. According to the creators of BERT, fine-tuning should achieve state of the art results for a majority of NLP tasks - we desired to test this claim and explore some of the models difficulties and benefits associated with this transfer learning approach. In more detail, we aimed to find out how to effectively cast BERT into a regression task and to set critical parameters. BERT, as with many other sequence to sequence models, is trained as a classification model. The pre-training methods employed by Google were novel for this architecture, and hence, the downstream effects are currently somewhat understood. Since BERT is extremely complex under the hood, it is not exactly clear how to effectively cast the problem into regression. Additionally, setting the core parameters is challenging given the model complexity and the unique hardware constraints that it imposes. For example, using the model on a large GPU can be difficult with long sequence lengths. This often brings in the consideration of using a Google TPU for non-trivial implementations. However, as TPUs are not available at scale - users should keep this top mind. Overall, we believe that our project is of high value for future users in considering if the full BERT fine-tuning approach is right for a specific problem. This is also an excellent culmination for our Statistical Language Modeling course as it brings together large scale model training and implementation along with considerations of how attention can overcome the deficiencies of recurrent based models.

2 Dataset

The chosen sentiment analysis task is, given a comment from the Civic Comments platform, predict the toxicity of the comment (sourced from Kaggle). Ground truth results were crowdsourced from a variety of surveyors, and a higher target value indicates a more toxic comment. In total Kaggle provides approximately 1.8 million training examples. Roughly 70 percent of the comments provided have no toxicity at all resulting in a highly unbalanced dataset. In addition, about half of the toxic training examples have a target value lower or equal to 0.2 which makes the dataset even more interesting, and more imbalanced. Figure 1 is a histogram which shows the toxicity of the comments provided, excluding non-toxic comments.

Moreover, comments were not preprocessed in

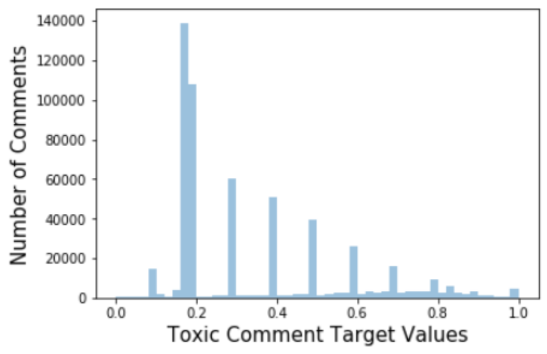

Figure 1: Toxic Comment Target Values

any way because the BERT tokenizer does not produce out of vocabulary words by design. This is the intention of the unique tokenizer - since pretraining has been completed on such a massive dataset, it is better to allow for character level tokenization instead of replacing any unknown words with an UNK token. For this particular dataset the tokenizer created a vast majority (over 99%) of training examples that have a maximum sequence length of 256. Figure 2 shows a histogram which displays the token length by number of comments.

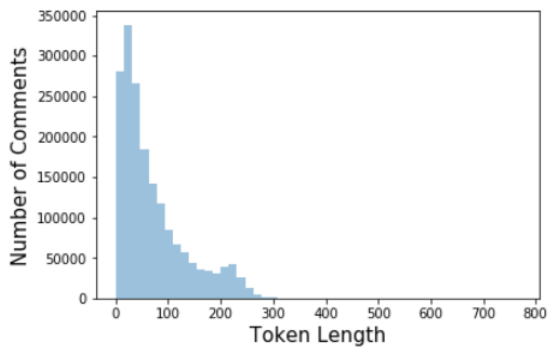

Figure 2: Tokenized Comment Length

3 Evaluation Metrics & Results

For this competition, Kaggle defined it's own custom evaluation metric, which makes a modification to the common AUROC metric (Area Under the Receiver Operating Characteristics Curve). The custom metric calculates three sub AUC scores which together are referred to as bias AUC scores. Each of the scores are calculated for a subset of comments that mention a population identity (e.g. male / female). The goal of this approach is to measure the ability of the classifier to also identify unintended bias towards specific population groups. All the bias AUC scores of the same type are then averaged across the different identity groups with (1) where m_s is the AUC score for the identity s:

$$M_p(m_s) = \left(\frac{1}{N}\sum_{s=1}^{N} m_s^P\right)^{\frac{1}{P}} \tag{1}$$

Note that p is a constant set to -5 to penalize heavily for making large errors. Kaggle only calculates the bias AUC scores for subpopulations that are mentioned more than 500 times in total. The final evaluation metric is then calculated by averaging the traditional full AUC score as well as the three bias AUC scores with equation (2) where $m_{s,a}$ is the AUC score for identity s using submetric a. Here, all four weights w are set to 0.25:

$$\text{score} = w_o AUC_{overall} + \sum_{a=1}^{A} w_a M_p(m_{s,a}) \tag{2}$$

4 Model & Implementation

Our overall implementation from the standard BERT code that we found on GitHub is largely the same. This is by design, as Googles intent was to wrap the pretrained model into one flexible API. The core of the model consists of three python wrappers called modeling, tokenization and optimization. As the names might suggest, the code within these files defines all core model information ranging from the actual tensorflow graph to the transformer optimization. Overall, one major downside to this approach is that the code can be quite restrictive with respect to having a variety of different output classes and keeping track of the loss function. Additionally, there is a fourth file called "run_classifier" which contains the functions for building the model and the associated input features regardless of the task. This file allows a TPU estimator (high level Keras API) to run the training, evaluation and prediction components on the given data and model definition.

Our main modification to the standard code was to not use the BERT predefined processors (such as COLA) so that we can tokenize and convert the inputs and outputs to features with a custom python script. In order to keep track of the loss function we added additional code to the original Tensorflow implementation which logs the loss to a Tensorboard. In terms of regression casting we employed two main architectures. The first was to perform a binary classification, and then output the toxic value as the softmax probability of class 2 for the final value. The logic behind this is simple - instead of having an extremely large number of output classes, we have our model simply predict a given comment as toxic and nontoxic, and then use its confident (probability) metric of a toxic comment from the softmax as the true toxic value. This model type is referred to as Overall Architecture Type 1 in Table 1. The second architecture is to have multiple classes, and then take the argmax as is traditional. The only modification with this approach is then converting the class to a float value - but this is done in a straightforward key-value pair method. If the value is a min or max class, we predict the extreme end value, otherwise we take the median of the bound values (for example if class 3 corresponds to training on comments between .15 and .25 toxicity, we then output .2 toxicity as the regressed value). This model type is referred to as Overall Architecture Type 2 in the Table 1.

5 Results

All models were trained on a Google Cloud Platform VM compute instance using 4 CPUs (total 16B of RAM), coupled with a Tesla P100 GPU (16GB of RAM). One thing was was very surprising at first was the fine-tuning" training time. On average, running a single epoch took over 8 hours even with a state of the art GPU. Furthermore, tokenization on the CPU can take a significant amount of time (about 30 minutes for the instance obtained). The results and main parameters for our deployed models are shown in Table 1.

6 Analysis & Interpretation

Throughout the implementation of the models shown in the results, there was significant learning - some of which was very surprising when considering the pitch of BERT. Our first approach was to get the easiest version of the model working, so a simple two class split at 0.5 toxicity was implemented. All model parameters were chosen by sourcing the most common settings online for similar tasks, and somewhat surprisingly, a very high score was achieved. However, the loss function observed during training was extremely volatile, and did not appear to converge in any meaningful way. After digging into the data more closely, we hypothesized that this was likely due to the max sequence length being far below the tokenized length for our comments, so the sequence length was increased to BERTs maximum capacity - 512. However, memory allocation resource issues occurred at every attempt, and it was determined the Tesla P100 GPU was not large enough to compute and update gradients successfully with a significant batch size. These were two problems that persisted throughout the duration of the modeling (memory pressure and volatile cost functions). Using a binary search the largest batch size of 8 for this sequence length was determined, but the volatility of the loss function increased greatly. Given our challenge and the imbalance of the dataset, we found that this batch size was simply too small. After training on only 10% of the training data our evaluation score had decreased significantly compared to those available on Kaggle. This was our first key learning - any time a long sequence length is required, there is a distinct tradeoff between batch size and sequence length for such a complex model that is allowed by even state of the art GPU's. Additionally, since training one epoch can take an extremely long time due to the complexity of one gradient update, it can be quite unwieldy to apply many different approaches radically at the commencement of a project.

Our next, and best performing, model was in the middle of these two approaches - with a 256 maximum sequence length, 28 batch size and a decreased learning rate. This approach resulted in a slightly less volatile loss function. After receiving our prediction scores, we then raced to train the model further. However, during the second epoch our loss function did not really appear to go anywhere, and the Kaggle results score mirrored this, giving us a significantly lower score. This gave us another key learning - despite BERT being as a model that uses attention to deeply understand language, overfitting can still occur quite easily. Also, reflecting on our first model submitted we recognized that the toxicity of a comment could likely be obtained fairly well by observing only part of the sequence - this is something to consider heavily depending on the purpose of the model, and is potentially a unique feature of attention.

For our last implementation of the two class model, we wanted to examine BERTs ability to discern comments that were toxic or not toxic as opposed to the previous low toxicity and high toxicity approach (binary split at 0.0 vs 0.5 target value). Originally, we thought this would result in a much better overall model, as the regression cast

Table 1: Model Configurations & Results

Model ID	Kaggle Eval	Num of Epochs	Seq Length	Batch Size	Learning Rate	Overall Model Architecture
0	0.5	-	-	-	-	-
1	0.92106	1	128	64	2e-5	Type 1
2	0.91179	0.1	512	8	2e-5	Type 1
3	**0.93241**	**1**	**256**	**28**	**1e-5**	**Type 1**
4	0.92825	2	256	28	1e-5	Type 1
5	0.90988	1	256	28	1e-5	Type 1
6	0.81338	0.1	256	28	1e-7	Type 2
7	0.5 (all 0s)	1	256	28	1e-7	Type 2

would intuitively make more sense (outputting class 2 probability). However, we faced significant challenges with this model. First, we trained the model with the same parameters as our previous best implementation. However, in deployment our model only predicted purely non-toxic values. This indicated heavy overfitting. Initially we reduced the learning rate, and even decreasing the rate to 1e-11 did not alleviate this issue. Gradient clipping was also explored, but after digging into the model it was discovered BERT automatically clips gradients, and applies a large amount of standard regularization techniques. However, we were able to overcome this challenge by subsampling from the non-toxic data to match the size of the toxic examples, and reducing the learning rate even further. Although we were not able to match our previous implementation scores, this presented another unique learning. Since BERT applies nearly every standard optimization technique under the hood, these techniques are difficult to change (modifying them deviates from the pre-training procedure and may cause adverse results). The quality of the data fed into the fine-tuning task is critically important since is the main point of control that an end user effectively has.

Our final two implementations of BERT employed a more traditional 10-class classification as opposed to our previous binary classification coupled with a unique regression cast. Initially our hopes were very high for this sort of approach, but after our previous model there was a concern that many of the classes would not have sufficient data. This was confirmed, as training on only 10% of the data with a very small learning rate produced our lowest evaluation score, and a complete epoch resulting in a model whose argmax would only produce all non-toxic comment predictions (even with heavy subsampling prior to training). We also attempted a simple regression on top of the model trained on 10% of the data to attempt to yield a relationship between the probabilities and the underlying toxic value - but the results were highly suspect. This further confirmed our previous learning's, and also presented one of BERTs keys weaknesses - intuitive regression techniques still impose a very difficult challenge for the advanced attention based classification models.

7 Conclusion

Overall, it was amazing to see how powerful BERT can be in an NLP task that is far removed from the original training methods of the model. This should not be understated. However, this statistical modeling prowess brings with it substantial challenges for anyone looking to implement the model with a practitioner's mindset. Since BERT is implemented in a very specific way, model control is restrictive. A good way to take advantage of BERT itself while overcoming these challenges may be to utilize the embeddings produced, without the full model itself. Additionally, since large sequence lengths limit batch size of the model even with state of the art GPU's, one should think very hard about the requirements of the underlying task at hand. It should be noted that our team put significant efforts into TPU utilization on the Google Cloud Platform, but we were unable to do so. The problems we faced seemed widespread on various online forums, and due to the novelty of the model, still have yet to be answered. From our high quality results it is clear that attention based models are here to stay, but although BERT tries very hard to be the best approach for any task, this still may not be the case.