Pay Attention to Your Tickets!

Chris Caballero

February 7, 2020

1 Introduction

In NLP, the classification of text documents and sentences encompasses a myriad of topics, each with their own fine-tuned state-of-the-art (SOTA) models. While in recent months it has become clear that LLMs can easily generalize to the task of text classification, there is still a need for small and effective models. For this project, the paper I improved upon focuses on the automatic classification of support tickets [1]. Support tickets are essentially small blocks of natural language, akin to sentences, designed to provide a succinct representation of a problem at hand. The paper addresses two dimensions of this problem. The first is the model architecture for classifying support tickets. They implement both a deep convolutional neural network and a support vector machine to classify based on Word2Vec representations of the support tickets. The second dimension, which is more of a classical algorithm, aims to improve overall performance by efficiently searching the space of possible hyperparameter configurations. Of these two dimensions, I will be strictly focusing on the former.

2 Overview

The paper I will attempt to improve is "Hyper-parameter Black-Box Optimization to Improve the Automatic Classification of Support Tickets". Upon my first read, it became apparent that their model architecture for a text classifier seemed a bit dated. They used a CNN architecture to classify the support tickets based on Word2Vec embeddings. While this is not necessarily a bad practice, it is clear that there are now better ways to classify text.

Around the time that [1] was published, one of the most revolutionary papers in the recent history of machine learning, "Attention Is All You Need" [8], was released. This paper introduced the transformer architecture, which has since been adapted and applied to a variety of use-cases. Consequently, I will design a new model to improve upon the raw performance of the CNN outlined in the paper.

Related Work

The state-of-the-art in sequence transduction largely utilized convolutional neural networks and recurrent neural networks in conjunction with encoder-decoder blocks [6, 7, 9]. This practice has largely shifted to attention mechanisms since the introduction of the Transformer [5]. Using a transformer would likely be beneficial to this problem. We can find the attention-based representation of both the input and output (ticket and class) and compare them to try to solve this problem. This would not only be more efficient at scale but also potentially more effective.

Problem Statement

The classification of support tickets is an NLP multi-class classification problem. The input to the model is a ticket, which is a sequence of tokens (words) summarizing the nature of the ticket. This is equivalent to the usage of a typical 'sentence' in NLP. The data is then embedded, forming a feature matrix $\mathbf{X}$. The output of the model is the class of the ticket, which is a specific topic the ticket is addressing. Across $\mathbf{X}$, we can form this into a label vector $\mathbf{y}$. It is essential that the class is unambiguous and should not have multiple meanings in natural language. In any case, the problem at hand is to encode the input into a vector which can be easily separated within its vector space into the appropriate number of classes with accuracy.

Some additional challenges include data availability and the exact model architecture used. The data they collected was private. I tried to reach out but, unfortunately, received no response. While they outlined most of the architecture, some parts still had to be guessed, and I constructed the model to the best of my ability. I will discuss how I overcame the first challenge in the next section, 'Data,' detailing the data used, and the second challenge in the section on 'Model Architecture.'

3 Data

Although the data used in [1] is not readily available, it is not very difficult to find many equivalent datasets covering support ticket classification on some open-source platforms. I personally have experience using Kaggle, and so I pulled the data directly from there. The user who uploaded this dataset also provided a lot of code for processing, analyzing, and labeling the data. I cite them in my README.md within the code folders.

Data Processing

Before beginning on the models, I decided to re-implement the code provided. I wanted to keep almost all of the steps the same, but write the code in a more efficient and modular form for my own practice. While the details are not necessarily relevant, I will mention a couple of aspects I changed in my code implementation. The first is the creation of a Pipeline object. This takes a dictionary-style sequence of fit/transform methods and combines them to form a modular data transformation component. This allows me to better consolidate the data processing steps. The steps are: CountVectorizer $\rightarrow$ TFIDF $\rightarrow$ NMF. CountVectorizer transforms a document or set of documents into a matrix of token counts. TFIDF is an abbreviation for term-frequency inverse document-frequency, a feature extraction method that creates features based on the relative frequency of terms across documents. The last step in the pipeline is NMF, which refers to non-negative matrix factorization. The resulting matrix can be used to separate the data based on, in this case, tfidf vectors. Since the dataset does not come with hand-labeled data, and there are thousands of support tickets, this method seemed fine for trying to create a simple labeling scheme. The second set of modifications to the original code centered around adapting the code to fit this pipeline. Data analysis steps were the most affected by this change, but it was easily modified nonetheless.

Dataset & Dataloader

When developing models using PyTorch, it is common practice to use DataLoader objects to feed the data into the model. These objects are generator-based, so they load data, apply necessary processing, and pass it to the model on an as-needed basis. This is nice since trying to apply the transformations necessary to all of the data is incredibly expensive and sometimes likely intractable. First, I created the TicketDataset, which is the class I used to transform the data I processed into a Dataset object, which is the correct format for creating a DataLoader object. This also works as

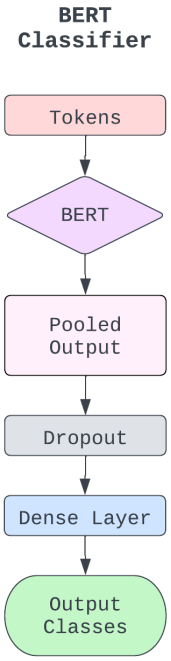
a generator which increases overall efficiency.

The data must be tokenized before sending it into the model. The easiest way to do this is to use a fitted tokenizer from a larger model, in this case, 'bert-base-uncased.' The process of dataset/dataloader creation is as follows. First, the **getitem** method is implemented to create a valid iterator/generator. This defines the mechanism for retrieving an item from the dataset. In this method, I first tokenize the data at the index provided, using a fixed padding length and allowing for truncation so that all of the data is of the same size. The result is a dictionary with tokens and labels. To create a DataLoader, I simply use the DataLoader method which constructs it directly from the dataset. I can now use this directly with whatever torch model I create (within reason, of course) in a training loop.

4 Model Architecture

Across all of the models, the main consistency is the mechanism of feeding data into the models. PyTorch makes it relatively easy, with practice, to develop efficient models that operate on tensors within the GPU. To do this, a proper DataLoader with appropriate batch sizing is necessary to reduce the overall training time. This is not relevant enough to warrant its section, but I wanted to address this since it took a good amount of time to properly shape the data so that all of the following steps work when parallelized.

Bert Classifier

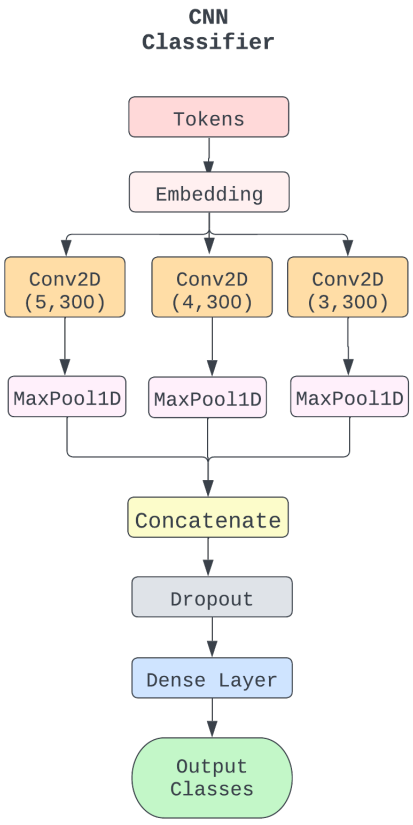


Since I was going to create an implementation of their model, along with my own model architecture, I wanted to test this all against a well-known SOTA model for sentence classification, BERT [4]. This model was simply pulled from a pre-trained version, 'bert-base-uncased.' This model fits best

with the tokenizer and had the best performance. However, as I will explain in the results section, the performance lift is marginal compared to the difference in model size over the other two models. BERT is a large language model, and while it is significantly smaller than what is normal today, it is still far larger than needed for support ticket classification.

The first two models are the main focus of the project, while the Bert model was just out of interest to see how a much larger pre-trained language model did when fine-tuned on this task. The architecture of this model can get a bit redundant since it is based on the transformer architecture, but the full details can be found in [4].

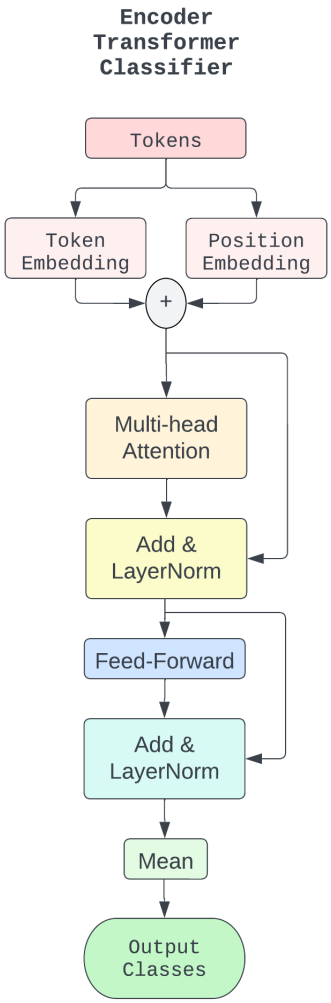
CNN



Before trying to improve upon the results of the paper, I first needed to re-implement their model and evaluate it on my dataset. The following architecture is my closest approximation of the model used in [2]. The first phase is the embedding layer. To save time, I chose to have both models learn their embeddings in an end-to-end fashion, rather than pre-training with Word2Vec. This is likely more inconsistent than the latter but should not be too big of an issue for the project overall. These embeddings are then passed to the convolutional layers. A convolutional layer is essentially a fixed-sized window that is applied across a collection of data, whether it be an image or some other feature matrix. The main mechanism is a parameter matrix that is shared by various sections in the input. For the problem of text classification, it makes sense to apply the convolution across the dimension of the embeddings, with a window corresponding to neighboring words [3]. This then is equivalent to grabbing a few adjacent words in the sentence, getting their embeddings, and combining the results according to the convolution.

The results of these convolutions are then pooled and flattened into a single vector, which is passed through a fully-connected layer and classified. Pooling the results is done with 'Max Pooling,' which takes the largest element in a window, thereby reducing the effective size for subsequent layers. This itself is similar to dropout, which is additionally used to further trim the data and reduce over-fitting. Flattening the output then allows the data to pass through a simple fully-connected layer. This is a perceptron that reduces the dimensionality of the data to the number of classes. An activation function, default: 'Sigmoid,' is used to convert the data to a set of probabilities, each representing the relative odds of a class being the correct class for the given input.

Encoder Transformer



For my implementation, I chose to take inspiration from the Transformer model [8]. Self-attention has proven incredibly useful, especially in the field of NLP. It allows the model to take into account data that would otherwise be well out of scope, especially when compared to the limited view of an LSTM or convolutional neural network. The architecture begins with two embedding matrices. The first is a token embedding, identical to the one used in the CNN. The second is a position embedding. This simply encodes the relative position within the sentence for each token. This is

added to the token embedding, resulting in an embedding with both token-level information, along with some position information [8]. This becomes relevant when the model attempts to compare tokens across the sentence.

The next stage is multi-head attention, which consists of several parallel self-attention layers [8]. A self-attention layer is best described by its three matrices: the key, query, and value matrices. These are each linear layers that apply a parameter transformation to the input and transform the input into three matrices. The key matrix learns relevant parameters to the task, as it will try to extract a set of 'keys,' 'queries,' and 'values' from the input. These 'keys' will then be tensor-multiplied with the 'queries,' resulting in a similarity matrix between the keys and queries. An index (i, j) in the similarity matrix will contain a value between (0, 1) where 1 implies perfect similarity and 0 implies no similarity at all. The hope is that the value in (i, j) provides meaningful information conveyed between the key vector for the token at index i and the query vector for the token at index j. We then have a matrix that summarizes this information, comparing each key with each query (one per token in the sequence). This output is then normalized, and softmax is applied to the matrix. This converts the output from the previous step into probability distributions. Dropout is then applied as in the CNN to reduce overfitting and model complexity. The last step in the self-attention head is to perform a tensor product between this output and our value matrix. The value matrix is intended to simply capture an encoded representation of the original input. The result is essentially the relative weights for each token in the input and for each dimension in the embedding space. This is similar to what happens in the convolution $\rightarrow$ pooling steps above but more efficiently.

After the attention output is combined across the parallel self-attention heads, we then apply a feed-forward neural network [5]. The structure of the feed-forward layer I used is a pair of linear layers with ReLU activations that first expand, then contract the data, with a final application of dropout to the output [5]. This output is then averaged across inputs and finally classified. I chose to take a simple average because all of the data has already been appropriately weighted, so averaging is just a way to condense the overall information. The multi-head attention and feed-forward layers also include skip connections, which are typical in transformers. These allow for deeper neural networks since they overcome the vanishing gradient problem by providing a gradient highway for the output directly back into the input.

Additionally, LayerNorm [1] is applied to the outputs of both the multi-head attention and feed-forward layers. Similar to BatchNorm, LayerNorm normalizes the data to reduce certain issues that may arise with the distributions as the information moves through the model. However, LayerNorm normalizes the distribution along the channel dimension, while BatchNorm normalizes along the batch dimension. Both increase model generalization by reducing overfitting and can increase efficiency by smoothing the gradient landscape [1].

The main difference between this model and the original Transformer architecture is the lack of cross-attention, along with a decoder block. The original Transformer was designed for sequence-to-sequence translation, which implies the output should be a sequence, requiring a decoder block. When the decoder produces its outputs, cross-attention is performed between the encoder block's output and the current stage of the decoder block. As I do not require this added complexity, I chose to use only an encoder block, which is more similar to BERT and text classification. Therefore, I call this model the 'Encoder Transformer' to differentiate it from the original Transformer architecture. In the following section, I will discuss the results of my implementation, along with its comparison to the architecture described in [2] and the BERT classifier derived from [4].

5 Results

Model Accuracy

Epochs 5	Final Loss	Training Accuracy	Validation Accuracy
BERT	5.69	94.84	16.29
CNN	0.83	**97.91**	91.10
Encoder Transformer	**0.04**	96.93	**93.36**

Table 1: Comparison between the accuracy of the 3 models on the TicketDataset.

We first inspect the simple accuracy metric for each model across our dataset. The results can be found in Table 1 above. The transformer model I implemented seems to outperform the CNN model in both final loss and validation accuracy. There is a performance lift of +2% accuracy and -0.79 loss. The training accuracy is higher for the CNN model but this does not mean much. A properly designed model can arbitrarily match the data, leading to overfitting and near-perfect training accuracy. The lower training accuracy in the transformer is likely caused by the normalization such as LayerNorm and multiple dropout layers. This inherently makes it difficult for the model to memorize the data, and further improves generalization performance, as we find when checking the validation accuracy. Since each model was only trained over 5 epochs, both converged rather quickly and neither had much of a chance to overfit. The transformer model would likely deal with longer training better since it has many 'tools' to combat overfitting.

Cross Validation

Epochs 5	10-Fold Cross Validation
BERT	92.45
CNN	76.50
Encoder Transformer	**93.38**

Table 2: 10-Fold Cross Validation Results for each of the 3 models.

To better understand how each of these models performs, I employ 10-Fold Cross-Validation. This allows one to ensure that the model is not benefiting from a lucky training/testing split. The results of this metric are much more telling of the overall model performance. The results can be found in Table 2 above. I was pleasantly surprised to see that my model performed better than even the BERT model, which at one point was an out-of-the-box SOTA model. My implementation achieved a +0.925% lift over the BERT model and 15% lift over the CNN model. The performance lift over BERT is likely because I did not spend enough time aligning the model to this problem statement, nor did I account for model capacity. My implementation has the benefit of being a bit better tuned for the problem at hand.

Surprisingly, the 'lucky' training/testing split is exactly what happened with the CNN model. It had various runs which measured over 92% accuracy, but many runs which were not able to learn at all, stagnating at 25% accuracy. I am fairly certain that this is less an issue of model architecture and more an issue with the end-to-end training framework for the embeddings.

End-to-end training is useful when the model wants to update all parameters at once, but there are training schemes that are a better fit for some models. The random embeddings were never able to extract meaningful features from the data. To overcome this, using Word2Vec embeddings as

was done in [1] may prove effective. This is because, at the very least, the embeddings carry some valuable information. However, I would argue that the Transformer's ability to consistently learn good embeddings shows the power of the architecture, and makes it a much better candidate for anyone looking to have a fast, easy-to-train model.

Training Time

Both the CNN and transformer models train fairly quickly. For 5 epochs, the total training time on GPU was about 1 minute and 10 seconds for both. In comparison, the BERT model took about 24 minutes. This is caused by the model size being around 10x larger than the other two. For reference, on disk, the BERT model is 427MB while the CNN and transformer models are 43MB and 37MB respectively. This likely had the biggest impact in terms of overall training time.

6 Conclusion

The study demonstrates that the Encoder Transformer model I implemented is a superior choice for support ticket classification in NLP, as it consistently achieves higher performance compared to the CNN and BERT models. The transformer's ability to learn good embeddings showcases its power as an efficient, easy-to-train model. Although end-to-end training can be useful for updating all parameters at once, it may not always be the best fit for certain models. Future work could explore using Word2Vec embeddings for the CNN model to further improve its performance and get a better estimate for the relative gain of switching to a transformer style architecture.

References

1. Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." arXiv preprint arXiv:1607.06450 (2016).
2. Bruni, R., Bianchi, G., Papa, P. (2023). Hyperparameter Black-Box Optimization to Improve the Automatic Classification of Support Tickets. Algorithms, 16(1), 46.
3. Chen, Yahui. Convolutional neural network for sentence classification. MS thesis. University of Waterloo, 2015.
4. Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).
5. G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," in IEEE Potentials, vol. 13, no. 4, pp. 27-31, Oct.-Nov. 1994, doi: 10.1109/45.329294.
6. Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., Wu, Y. (2016). Exploring the limits of language modeling. arXiv preprint arXiv:1602.02410.
7. Luong, M. T., Pham, H., Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025.
8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
9. Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144.