Tweet Sentiment Extraction via Sequence Modeling

Ammar Husain

1 Introduction

Twitter is a social networking platform that poses the most interest language processing problems. The language is often very natural, colloquial, and evolving very rapidly. Twitter datasets have been released before often with the goal of sentiment analysis, however with this dataset, we are posed with a new challenge where we must extract a sub sequence. I will explore the use of sequence modeling to classify set membership, and explore some of the challenges with text representation.

2 Dataset

The data was obtained from the feature Kaggle competition "Twitter Sentiment Extraction" (https://www.kaggle.com/c/tweet-sentiment-extraction). This includes 27,482 labeled training tweets, and a test set of 3534 tweets. Each tweet comes with the raw text (censored for profanity), and a sentiment tag. A tweet can be tagged positive, negative, or neutral. The trained examples also come with a selected that represents the support for the sentiment label.

Our goal is to extract the supporting text for a given tweet and label. This is scored using a word level jaccard score. This is defined as:

$$score = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}$$

where y_i is the ground truth suppport text vector and $\hat{y}_i$ is the predicted support text. Each text vector is created by splitting raw text only on whitespace, and then converting to lowercase. Punctuation is left in the text, and is not separated from the attached words. For example:

"Twitter is really, really cool!" $\rightarrow$ ["twitter", "is", "really,", "really", "cool!"]

To optimize for this score I approached this problem as a sequence modeling problem. For given data in form:

$$\mathcal{D} = \left\{\left(\left(x^{(1)}...x^{(m)}\right)_i, \left(y^{(1)}...y^{(m)}\right)_i\right)\right\}_{i=1}^{N}$$
$$\mathcal{D} = \left\{\left(\left(x^{(1)}...x^{(m)}\right)_i, y_i^{(1)}...y_i^{(m)}\right)\right\}_{i=1}^{N}$$

Here each row has a sequence of size m, each $x_i^{(m)} \in \mathbb{R}^d$ (representing word embeddings.), and m labels, each label $y_i^{(m)} \in \{0, 1\}$, representing membership in the selection output of the support text. The goal of

I want to learn an sequence representation Γ which I can use to predict set membership y. We can do this by approximating the probability $P(Y_i|X_1...X_N)$.

2.1 Preprocessing

For this project, I wanted to learn the word embeddings from the dataset itself, instead of using for example pretrained word2vec embeddings. I wanted to compare the performance of the sequence

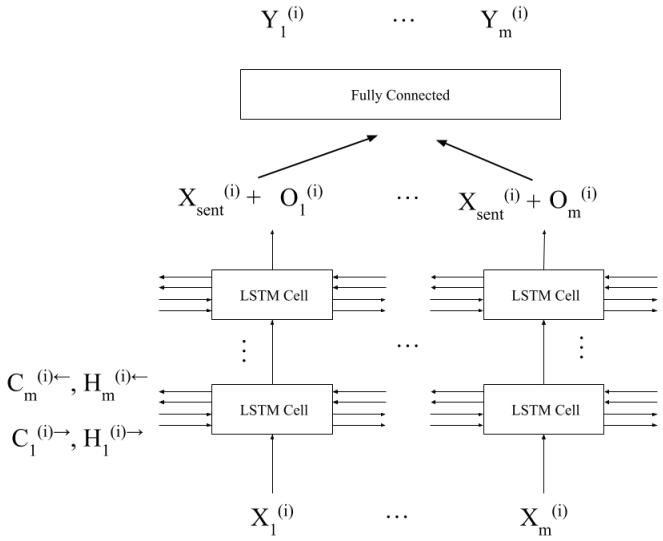
modeling architectures. However also, the language in this dataset is very natural and esoteric. There are misspelled words, words outside of any formal dictionary, emojis, and words with repeated characters for emphasis. Using any predefined vocabulary, or simply building a vocabulary off of this corpus, when validation data would be encounterd, there would be a lot of unknown, unmapped tokens.
To handle this I used a Word Piece tokenizer, with a prebuilt vocabulary from the BERT architecture. The word piece essentially creates tokens on the character level, and then iteratively adds commonly occurring segments. This way, roots, and word segments are now the base from which any current or new word can be built from. At the worst case, a completely unseen word with no similar word segments, would be mapped to its individual characters. To make training, and GPU transfer more performant, after tokenization, each sequence is padded to 150 tokens (this accounts for the longest sequence).

3 Models

Using the same preprocessing and tokenization scheme, I used both an LSTM and a Transformer Encoder layer to model the sequence. The sequence representations are then passed through a linear layer to predict set membership.

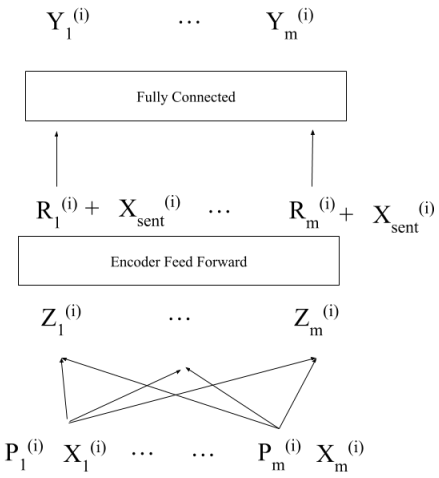
3.1 Bidirectional LSTM

LSTM's are used very commonly for sequence modeling, and is a good baseline to compare to other architectures. For this task, we are not doing sequence prediction, the output depends on the entire sequence. This is why a bidirectional network is important. The output features should represent the word in context of the sequence. The dimensions from the output of the LSTM, O_m^i, is concatenated with the sentiment of the sequence (positive, negative, neutral). The network I used consisted of 2 recurrent layers layers and a hidden layer dimension of 64 (32 for each direction).



3.2 Transformer Encoder

Transformers are used as the building blocks for the most performant NLP models. The tokenization vocabulary that I used was trained using BERT, which is a bidirectional transformer encoder module. For my network, I used the encoder module of the transformer to model my input sequences. The input has a padding mask applied to it to make sure the padded inputs are not summed in the self attention network. The word embeddings are added to positional embeddings, and passed through the encoder layer to get output representations. The sentiment features are added as dimensions to the encoded representations and then passed through the final fully connected layer. I used both 2 and 4 encoder layers, with the hidden and embedding dimension as both 64, and 4 multi-attention heads.



3.3 Model Search

3.3.1 Computing Loss

Both architectures output a set membership label for each word in the original sequence. To compute loss in accordance with the Jaccard similarity score, we can compute loss as such:

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{M_i^{'}}\sum_{j=1}^{m} y_j^{(i)} \cdot \log(\hat{y}_j^{(i)})$$

where $\hat{y}_j^{(i)}$ is the output of the set member ship for word j in sequence i. Because the sequences are being padded to a fix length, we divide each selection output loss by $M_i^{'}$ which is the number of non-padding elements in the sequence. The padding is included to account for the maximum length tweet, so majority of tweets contain an disproportional amount of padding tokens. Defining a loss function as such makes sure there are no unnecessary gradients propagated.

3.3.2 Optimization

For training, I used the Adam optimizer to search for model parameters. The learning rate was 10^{-3}, batch size was 64, and no regularization was used. The maximum number of epochs for each model is 100, however, early stopping was used. Each epoch, the best validation jaccard score is stored. If the validation loss does not decrease, and the jaccard score does not increase more than the current best score, the training is stopped.

4 Results

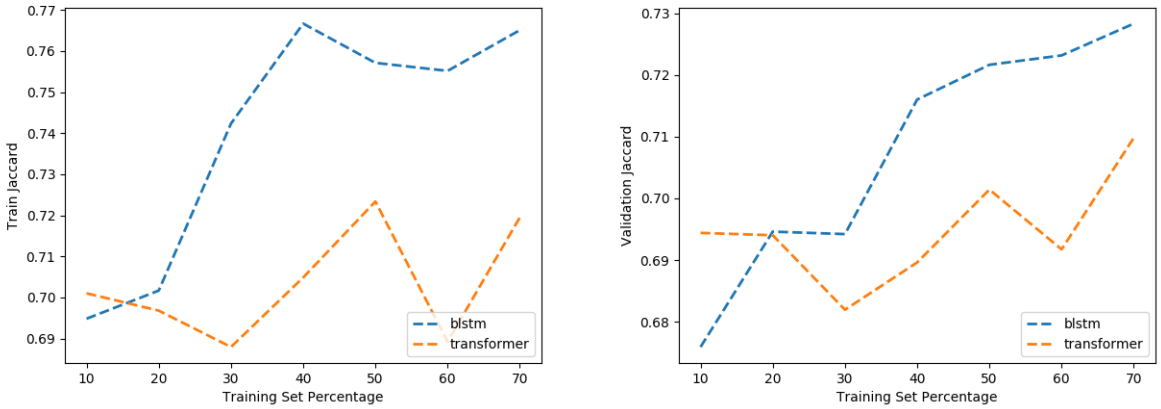
All of the models were trained under the same optimization parameters. Transformers The following table shows the parameters for the architecture:

Model	Embedding Dim	Layers	Hidden Dim	Attention Heads	Dropout
Bidirection LSTM	64	2	32 (x2)	n/a	n/a
Transformer 2 Layer	64	2	64	4	.2
Transformer 4 Layer	64	4	64	8	.2

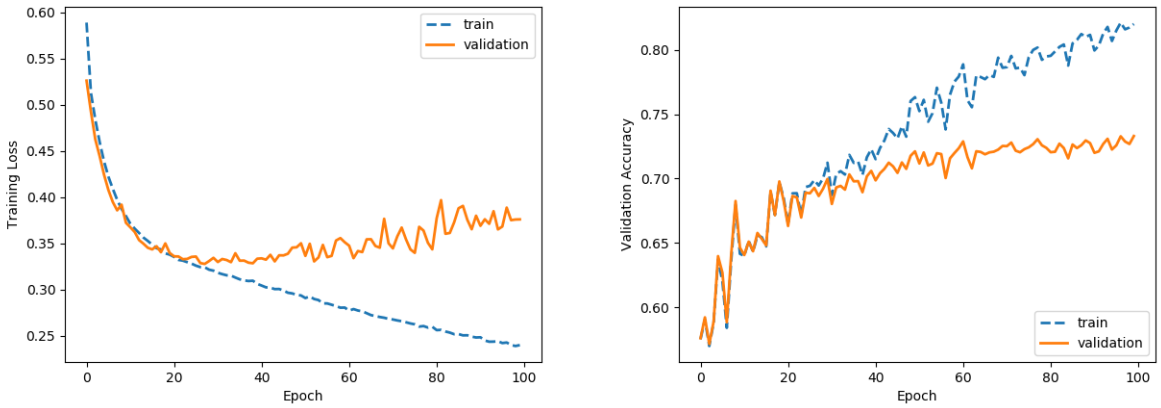
Results were calculated using k-fold cross validation (5 folds). Vocab for tokens were prebuilt on the same WordPiece vocabulary.

Model	Train Loss	Train Jaccard	Validation Loss	Validation Jaccard
Select Full Tweet	n/a	0.513	n/a	0.514
Bidirection LSTM	0.258 ± 0.016	0.772 ± 0.015	0.326 ± 0.007	$\mathbf{0.728 \pm 0.002}$
Transformer 2 Layer	0.0189 ± 0.016	0.756 ± 0.01	0.355 ± 0.014	0.717 ± 0.004
Transformer 4 Layer	0.266 ± 0.013	0.763 ± 0.006	0.332 ± 0.007	0.723 ± 0.006

These results are interesting, as the LSTM performed better than both Transformers, even though it had less parameters. Looking at the learning curves, we can see that the transformer is over-fitting with larger training sizes:



This could be because of a couple of reasons. First, the early stopping strategy we used may be cutting off the transformer too early. Currently, I am stopping the training when the loss function starts increasing. This could be problematic because the loss function we have defined does not exactly correlate with the Jaccard similarity. As you can see in the following training curve for an 8 layer transformer:



The current training pipeline would cut off as soon as the validation loss starts increasing. Another problem is the BERT WordPiece vocabulary that was used contains a lot of tokens that are not present in the Twitter dataset. This leaves a lot of random embeddings, which could make the embedding not as representative as needed.

5 Conclusions

Using sequence representations to predict set membership, we were able to obtain a comparable validation Jaccard similarity score with the Twitter dataset. The bidirectional LSTM proved to be the

most performant and easier to tame in the training process. However with future improvements in model search, and word representation, I believe I can improve this score.

6 Future Work

For future work on this, I will be tackling the word embeddings, improving model search, and redefining optimization criterion.
First, as a new sequence model, I can use a pretrained version of the BERT bidirectional transformer that already has token embeddings for the vocabulary that I am using. This would help as each sub word would already have a meaningful representation, and would work better on new unseen data.
Second, the biggest problem I ran into was model search. The transformer was constantly overfitting. This could be tackled using regularization and more finetuning of dropout.
Finally, another problem could be the loss function I'm doing. For both the LSTM and Transformer, the validation loss increases, while the validation accuracy increases as well. This means the loss function isn't as representative of our target jaccard metric. There is a statistical jaccard similarity score that treats the membership of each word as a random variable. This way we can directly optimize for the jaccard score.

References

Vaswani, Ashish, et al. "Attention Is All You Need." ArXiv:1706.03762 [Cs], Dec. 2017. arXiv.org, http://arxiv.org/abs/1706.03762.

Devlin, Jacob, et al. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." ArXiv:1810.04805 [Cs], May 2019. arXiv.org, http://arxiv.org/abs/1810.04805.