Project Report : CS 7643

Nameer Rehman
nrehman9

Jun Ting Chen
jchen3191

Shan Tie
stie3

Abstract

This project uses the task of automated hateful tweets classification in English to evaluate the performance of a few commonly used natural language processing architectures and fine-tuning approaches. The baseline architecture is a naïve recurrent neural network, specifically using a LSTM as the gated recurrent unit [3], [5]. The performance of more advanced Large Language Models (LLM), RoBERT [4], was evaluated without any fine-tuning as well as using low-rank adaptation methods for more efficient parameter tuning [2], [1]. The performance of each architecture and fine-tuning approach was compared using classification metrics such as F1 score and accuracy as well as computational cost and runtime for training.

1. Introduction/Background/Motivation

Social Media networks such as X (formerly Twitter) provide a way for millions around the world to share and consume content, with millions of tweets being sent out a day. Scraping and analyzing this data can provide valuable insights how users are communicating, understanding public sentiment on a given topic, and flagging malicious activity. One such example is analyzing tweets to identify whether they are hateful or not, which was the focus of this project.

It is unclear whether an internal algorithm exists within X to analyze tweets that are reported by users, but the feature of automatically classifying hateful tweets can be a useful addition to the platform. If done correctly, it can be used to promote online safety and digital well-being, combat hate-speech, and protect users from discrimination. Another key aspect of being able to correctly classify tweets is we would want to minimize false positives. False positives would mean users are tweeting non-hateful content which is being flagged as hateful, which can cause frustration and a poor user experience.

Aside from this being a useful feature within the platform, classifying online hate speech can have many other uses and benefits to society. For example, it can be used by law-enforcement to track criminal activity, identify key societal issues, and ensure actions are taken to protect discriminated groups.

Looking at existing approaches, popular machine learning algorithms for text sentiment classification include logistic regression, support vector machines (SVM), and random forest. However, hate speech detection often involves nuanced language and subtle cues that may be challenging for traditional machine learning algorithms to capture accurately. There are a multitude of deep learning methods that are used for sentiment analysis such as recurrent neural networks (RNNs) or transformers, but their success is highly dependant on model architecture and hyper-parameter tuning.

As part of this project, we aimed to compare and contrast various deep learning architectures such as RNNs and RoBERTa, as well as fine tuning methods to identify the most effective way to classify hate speech. Effectiveness was measured using metrics such as accuracy and loss on train, validation, and test sets, as well as F1 score, a combined precision and recall metric. We also wanted to be mindful of the trade-off between increases in accuracy versus train time to ensure we are making efficient incremental improvements to our model that are worth an increased model runtime.

The dataset used for this task contains 10,000 individual tweets scraped from X, along with a label classifying each tweet as hateful (1) or not hateful (0).

2. Approach

2.1. Base RNN

Initially some field research was conducted to assess existing deep learning methods for classifying and extracting sentiment from sequential text [5]. A simple RNN model was chosen as a base architecture to use as a comparison point for other methods. RNN is a popular neural network for processing sequential text as it maintains a hidden state that captures information about previous inputs. The recurrent connections also enable the network to use information from previous time steps to influence the computation at the current time step. This is ideal as a base model as it can be built from scratch and compared against other pre-trained models.

The RNN architecture consists of an embedding layer to help the model learn better representations of the input data, LSTM layer for modeling sequential data where context and dependencies between words are crucial for identifying hateful tweets, and a linear layer for classification.

2.2. RoBERTa

Another popular architecture for sequential processing tasks is BERT (Bidirectional Encoder Representations from Transformers) and RoBERTa, which is a transformer-based architecture that uses attention mechanisms. RoBERTa is a variant of the BERT model which uses improved training strategies and larger batch sizes, leading to better performance on various NLP tasks. RoBERTa is pre-trained using unsupervised learning objectives such as masked language modeling and next sentence prediction and can be fine-tuned on specific downstream tasks. RoBERTa was chosen for the task of classifying hateful tweets for following reasons:

- The bidirectional nature of RoBERTa means it considers the entire context of a tweet when generating representations. This allows it to capture complex relationships and understand nuanced meaning of text, which is crucial for accurately classifying hateful tweets.
- RoBERTa being pre-trained on massive amounts of text data from diverse sources allows it to learn rich and generalizable representations of language. This helps in capturing a wide range of linguistic patterns, including those relevant to hateful language.
- The self-attention mechanism allows it to focus on different parts of the input text while making predictions. This is beneficial for identifying relevant phrases indicative of hateful language within tweets.

2.2.1 RoBERTa Implementation

RoBERTa was initialized by instantiating the pre-trained model, which is loaded with weights trained on a large text corpus. A hidden linear layer was defined to transform the model's output to the desired hidden size, followed by a linear classification layer to map the hidden representation to hateful tweet classification logits. The layers are initialized using Xavier uniform initialization.

In the forward pass, input_ids and attention_mask tensors are passed. Attention masks ensure that the model pays attention only to relevant tokens, and ignores irrelevant ones such as padded tokens. A pooled output is calculated by taking the mean of the last hidden state over the token dimension and is then passed through a hidden linear layer, followed by dropout and ReLU activation. Finally, the output is passed through the classifier linear layer to obtain the logits for classification of hateful tweets.

2.3. Low-Rank Adaptations

In order to further investigate modifications to base RoBERTa, two other advanced fine tuning techniques were selected - Low Rank Adaptation (LoRA) and Quantized Dynamic Low-Rank Adaptation (QLoRA). Low-rank Adaptation techniques can help reduce computational complexity and memory usage of neural network models while maintaining or even improving their performance. Some proposed advantages of LoRA include:

- Saved models are portable because they only contain decomposed matrices
- Simplifed model training because all parameters from the pre-trained model are frozen during the fine tuning process
- De-ranked parameters can be viewed as reduction in complexity which should improve the ability to generalize.

QLoRA has additional advantages compared to LoRA that include being more GPU memory efficient due to the quantization and ability to handle larger max sentence lengths.

2.3.1 LoRA Implementation

A wrapper class LoraRobertaSelfAttention was built on top of the RobertaSelfAttention module from the transformers module. The LoraRobertaSelfAttention class applies trainable LoRA matrices A and B to query and value matrices by adding their dot product to pre-trained queries and values. When the model RobertaLoraClassifier is instantiated, all attention heads are swapped with the LoraRobertaSelfAttention recursively. All pre-trained parameters are frozen besides the LoRA matrices A and B. The blog article by M. Dittgen [1] was referenced for implementing LoRA from scratch. Changes made to this implementation include:

- Added another linear layer to the classification head. This was done because the RoBERTa classifer we want to compare with has two linear layers
- Modified LoRA rank to understand performance loss against training and inferencing efficiency gain

2.4. Anticipated Challenges

One of the problems we anticipated was on the data itself. We could not track the source of the data to understand how it was collected. The best we can do is to perform spot check on randomly selected data. Despite "hateful" being potentially subjective, spot checks on data revealed the hateful flag to be accurate given tweet context. Additionally, we did check that there was no redundant tweets in the various data splits.

There are two versions of the data. One is the raw text which is referred to as *text*. Another version of the data was cleaned to remove non-alphanumeric characters such as hashtags and symbols such as '@' which are prevalent in tweets. Additionally, 'stopwords' characterized by the NLTK (Natural Language Toolkit) were also removed. This version of the cleaned dataset was referred to as *clean text*. Ultimately, through testing of model performance, text resulted in better model performance and was used for model tuning and comparisons.

Another anticipated issue is that large language models typically tend to overfit on the training set, although the additional classification head was in its simplest form. Attempts such as adding a dropout layer and increasing dropout rate did not improve the overfitting. Fortunately, accuracy on the validation set was acceptable (80%) even though the model overfit the training set.

3. Experiments and Results

The same loss function and optimizer was used to train across the different models. Specifically, a cross-entropy loss function was used because this was a classification task and the ADAM optimizer was used with all default parameters with the exception of the learning rate.

3.1. RNN Tuning

The RNN model was initially tested on base hyperparameters as shown in Table 1. The base model version resulted in a train accuracy of 0.999 and 0.992; validation accuracy of 0.776 and 0.611; and test accuracy of 0.793 and 0.507, for class 0 and 1 respectively. Hyperparameters were then tuned by varying one hyperparameter at a time and freezing all others. The base RNN did reasonably well in both validation and test data performance; however there was clear overfitting against the training data since training accuracy was nearly 100% whereas validation and training was much lower.

Fine-tuning the RNN model was mainly focused on reducing the overfitting of the model while still trying to maintain good model performance. The fine-tuned hyperparameters are shown in Table1. Specifically, increasing the dropout, reducing the embedding and hidden sizes contributed the most to reducing the overfitting of the model. Changing these parameters tended to decrease the model capacity and flexibility to fit to only the train data and therefore steer the model away from just memorizing the train data. Decreasing the learning rate and increasing the epochs allowed the model to take reasonably sized steps and update the gradients more slowly (Figure 1). With the tuned hyperparameters, the RNN model had a train accuracy of 0.899 and 0.797; validation accuracy of 0.710 and 0.598; and test accuracy of 0.793 and 0.433, for class 0 and 1 respectively.

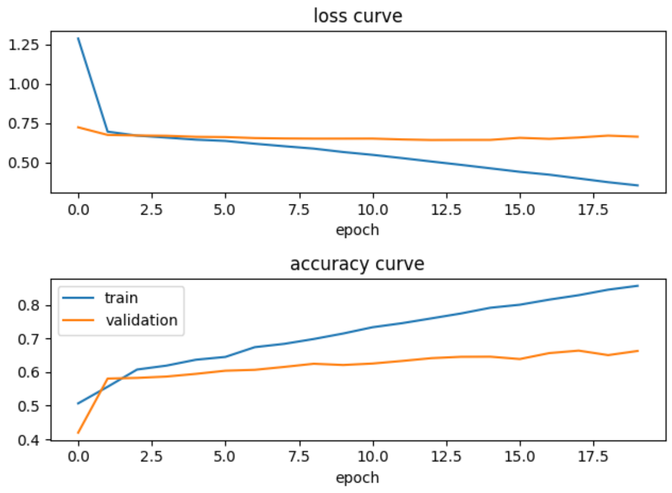

Figure 1. Best model RNN model

3.2. RoBERTa

After a few rounds of debugging, the first successful implementation of a RoBERTa based classifier demonstrated a slight advantage in accuracy prediction over the RNN model. Shown in Figure 2, training set and validation set has accuracy of 100% and 80%. Validation set class 0 and class 1 accuracy are 0.836 and 0.772; test set class 0 and class 1 accuracy are 0.811 and 0.78. Despite the model overfits the training set, performance on the validation and test sets are higher than RNN. Attempts has been made such as increasing dropout rate, reduce linear layer on the classification head, no improvement has been observed.

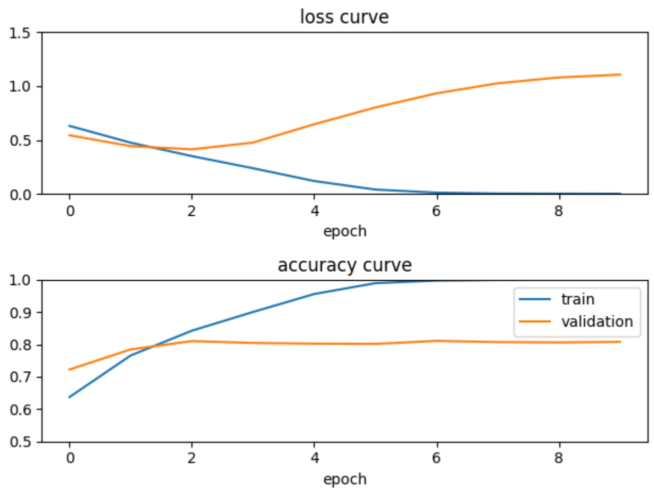

Figure 2. Best model RoBERTa

Model Version	Learning rate	Batch size	Embbeding Size	Hidden size	Layers	Epochs	Dropout
Base	0.001	128	500	28	1	10	None
Fine-tuned	0.0005	250	24	64	1	15	0.1

Table 1. Hyperparameters for base vs fine-tuned RNN

3.3. RoBERTa with LoRA

Figure 3 shows the loss curve and accuracy curve for the rank 16 LoRA model. The training set accuracy is 0.83 for class 0 and 0.762 for class 1; the validation set accuracy is 0.752 and 0.795. Compared to the base RoBERTa model, the LoRA matrices make the base RoBERTa less favorable towards the training set. Test set accuracies are 0.82 and 0.76 for class 0 and class 1, respectively, which is similar to the base model's performance.

Table 2 compares the performance of each RoBERTa classifier with LoRA at different ranks. The training accuracy increases to 0.914 and 0.884 for class 0 and class 1 at rank 32, while training accuracy drops to 0.83 and 0.762 at rank 4. This trend agrees with the intuition of the LoRA fine-tuning approach - a higher rank of matrices A and B increases the number of trainable parameters and the complexity of the model. At rank 32, the model tends to overfit the training set similar to the base RoBERTa model.

As shown in Table 3, the fine-tuned RoBERTa model has 0.12B trainable parameters. LoRA fine-tuning models studied in the current work only have between 0.8M to 1.8M parameters. The saved LoRA model is approximately 1% of the size of the saved RoBERTa classifier model without sacrificing model performance. This is valuable for users who need to store multiple versions of fine-tuned model parameters for different downstream tasks.

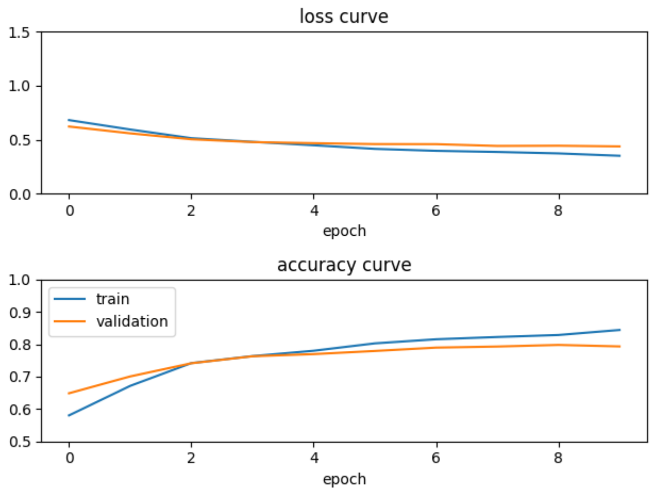

Figure 3. Best model RoBERTa fine-tuned using LoRA

3.4. RoBERTa with QLoRA

Implementation of QLoRA was focused on reducing the computational cost of fine-tuning directly on RoBERTa and squeezing even more cost savings on top of the LoRA implementation. Similar to the RNN hyperparameter tuning, QLoRA was tuned by varying one parameter at a time. The parameters that were tuned included the rank, alpha (LoRA scaling), dropout, bias, layers to adapt with QLoRA, and epoch. The base implementation of QLoRA with RoBERTa allowed for re-computing biases for all layers and adapted multiple modules that included the self-attention layer ('query', 'key', 'value') as well as intermediate and output layers. These two settings caused the model to overfit to the train data leading to very low model generalizability. Thus only the self-attention layers and LoRA-only bias terms were adapted. Dropout was increased from its default=0 to 0.4. Dropout helped regularized the model training process by randomly changing the values of certain inputs to 0 and in doing so prevents the model from being hypersensitive to any particular feature.

Four different rank parameters were tried (2, 4, 8, 16), while fixing all other parameters. These results showed that rank=4 achieved the best balance in terms of model performance and generalization (Figure 4). Rank greater than 4 resulted in severe model overfitting and rank=2 had lower performance across the board compared to rank=4. The alpha scaling term controls how much the fine-tuned model weights are weighted when combined with the pre-trained model weights. A higher value will mean the newly fine-tuned parameters will be weighted more compared to the pre-trained weights. Tuning this parameter in determines how much of the fine-tuning should influence the model.

$$W_{fine-tuned} = W_{pre-trained} + \frac{\alpha}{rank} * M_{LoRA}$$

3.5. Model Runtime and Complexity

Train time per epoch and parameter size were measured for each model to quantify runtime and complexity trade-offs that may occur (Table 3). Note, as train times can be dependant on hardware and environment, these measures were used as a comparison point between models within the scope of this project. The fine-tuned RNN had a runtime of 13.46s per epoch and 5.1MB parameter size, which was significantly lower than RoBERTa's 53.56s per epoch and 478MB parameter size. However, RoBERTa saw test accuracy for class 1 nearly double to 0.78 when compared to fine-tuned RNN, which makes it a justifiable trade-off. The

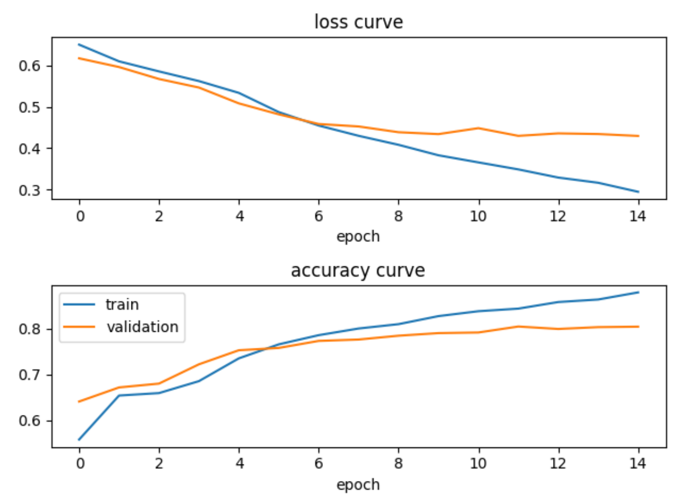

Figure 4. Best model RoBERTa fine-tuned using QLoRA

implementation of QLoRA in particular is where we see the Low Rank Adaptation technique shine, as it reduces runtime to just 1.7s per epoch while shrinking parameter size to just 0.585MB, all while maintaining similar accuracy.

4. Future Work

There are several research direction that can be continued for this project. Another way to assess the trained models is through zero-shot and few-shot learning scenarios. There were additional columns in the Hateful Tweet dataset that included classifying if the hateful language was targeted towards a specific group and if the tweet was aggressive. We could test these fine-tuned models on these new tasks to see if any of these models can generalize to this new task. These tasks can provide additional context to the tweets, which can be valuable in determining how to action the hate speech in the case of social media moderators or policy makers.

5. Data and Code access

Data and code for this project can be found here: https://github.com/chen112p/cs7643-project

6. Appendix A: Full Results Output by Model

7. Work Division

The breakdown of member contributions can be found in Table 4. Nameer was involved with literature survey, RoBERTa model training, and report writing. Junting was involved with literature survey, and implemented RoBERTa classifier and LoRA fine-tuning. Shan was involved with sourcing and processing the tweets, implementing RNN from scratch, fine-tuning the RoBERTa model using QLoRA, assessing model performances and writing.

References

[1] Martin Dittgen. Implementing lora from scratch. 2023. 1, 2

[2] Yelong Shen Phillip Wallis Zeyuan Allen-Zhu Yuanzhi Li Shean Wang Lu Wang Hu, Edward J. and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv*, 2021. 1

[3] Denise Löfflad Adham Nasser Mohamed Saber Manolescu, Mihai and Masoumeh Moradipour Tari. Tueval at semeval-2019 task 5: Lstm approach to hate speech detection in english and spanish., 2019. 1

[4] Chris McCormick and Nick Ryan. Bert fine-tuning tutorial with pytorch, 2014. 1

[5] Veerappampalayam Easwaramoorthy Sathiskumar G. Deepalakshmi Jaehyuk Cho Subramanian, Malliga and G. Manikandan. A survey on hate speech detection and sentiment analysis using machine learning and deep learning models. *Alexandria Engineering Journal*, 80:110—-121, 2023. 1

Model	Train (0)	Train (1)	Validation (0)	Validation (1)	Test (0)	Test(1)	Test F score
RNN Base	0.9989	0.9925	0.7764	0.6111	0.793	0.507	0.710
RNN Finetuned	0.8989	0.7972	0.7098	0.5976	0.793	0.4326	0.670
RoBERTa	1	1	0.836	0.772	0.811	0.78	0.760
RoBERTa+LoRA (R4)	0.8302	0.7615	0.7523	0.7952	0.82	0.688	0.773
RoBERTa+LoRA (R8)	0.87	0.829	0.836	0.748	0.77	0.793	0.756
RoBERTa+LoRA (R16)	0.83	0.762	0.752	0.795	0.82	0.762	0.768
RoBERTa+LoRA (R32)	0.9144	0.8841	0.792	0.7952	0.77	0.7627	0.737
RoBERTa+QLoRA (R4)	0.8851	0.8806	0.8224	0.7619	0.8228	0.7488	0.795
RoBERTa+QLoRA (R8)	0.9658	0.971	0.8057	0.8167	0.7947	0.793	0.793
RoBERTa+QLoRA (R16)	0.9882	0.9917	0.8207	0.7762	0.8175	0.7488	0.791

Table 2. Inference measures for all models

Model	Train runtime per epoch (s)	of parameters	Parameter size (MB)
RNN base	3.04	8344452	31.8
RNN Finetuned	13.46	1329784	5.1
RoBERTa	52.56	125237762	478
RoBERTa+LoRA (R4)	49.76	861698	3.4
RoBERTa+LoRA (R8)	49.76	1009154	4.0
RoBERTa+LoRA (R16)	49.76	1286671	5.1
RoBERTa+LoRA (R32)	49.76	1893890	7.3
RoBERTa+QLoRA (R4)	1.725	138240	0.585
RoBERTa+QLoRA (R8)	1.74	138240	0.585
RoBERTa+QLoRA (R16)	1.755	138240	0.585

Table 3. Runtime and complexity for all models

Student Name	Contributed Aspects
Shan Tie	Data Creation, Code Implementation, Model Training and Evaluation, Report Writing
Jun Ting Chen	Literature Survey, Code Implementation, Model Training and Evaluation, Report Writing
Nameer Rehman	Literature Survey, Model Training, Report Writing

Table 4. Contributions of team members.