ECE 364: Programming Methods for Machine Learning

Twitter Analysis Project Report

By: Sahil Jain and Rishi Vemulapalli

Introduction

This project is based on the WNUT-2020 Competition[1]. The objective of the task is to design a Binary Classifier to identify whether or not a given tweet related to COVID-19 is informative (Fig. 1). Such informative Tweets provide information about recovered, suspected, confirmed, and death cases as well as the location or travel history of the cases. The dataset can be downloaded from the competition's GitHub repository[2]. The dataset contains three sets – training (train.tsv), validation (valid.tsv), and test (test.tsv). We only used the training set to train the model. The validation set was used to tune the hyperparameters or select the best checkpoint in training. The final scores were reported on the test set in predictions.csv.

The dataset comes in the form of .tsv files. Each file contains three fields:

1. Id: ID assigned to a tweet.
2. Text: The content of the tweet.
3. Label: Either INFORMATIVE or UNINFORMATIVE.

ID	Label
1235770448966754309	UNINFORMATIVE
1235748200742416384	INFORMATIVE
1236129620363100161	UNINFORMATIVE
…	…

Fig. 1. predictions.csv

[1] https://competitions.codalab.org/competitions/25845
[2] https://github.com/VinAIResearch/COVID19Tweet

Methodology and Results

Preprocessing

For this project, we preprocessed the data before it entered the model. Specifically, we assigned a 1 or 0 to the Label column to represent INFORMATIVE or UNINFORMATIVE. We then utilized a tokenizer, padding, and a vocabulary trained on the training data at a frequency of 2. We provided a vocabulary of 9577 unique tokens to convert the string of words in the Text column to a series of numbers. By doing this, we make our data more readable to the model for training, validation, and testing. Then we converted this to a custom Pytorch dataset class object called TweetDataset to convert the dataframe into a Pytorch tensor and apply all the functions previously noted.

Model Architecture

Our binary classification model is a very simple LSTM (Long Short-Term Memory) - based model consisting of only four layers and 851,525 parameters. I calculated the number of parameters automatically using the code below (not included in the program file), where the variable model is our initialized Binary Classifier.

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Number of parameters: {total_params}")
```

```
Number of parameters: 851525
```

The first layer is an embedding layer, which will convert the input sequence of numbers into a sequence of vectors to obtain a dense representation. These tokens are then fed into the bidirectional LSTM, which processes the sequence of embedded vectors to capture temporal dependencies and contextual information. After this, the model will select the output corresponding to the last time step, which now contains information aggregated from the entire input sequence. Following this, a dropout is applied to this final output, ensuring regularization during training. This also acts as a regularizer that helps prevent overfitting by randomly dropping a fraction of the neurons during training. Finally, the

result is passed through a fully connected layer, which maps the condensed feature representation to the final binary classification decision.

Layer (type)	Output Shape
Input	[Batch size, sequence length]
Embedding	[Batch size, sequence length, 75]
LSTM (bidirectional)	[Batch size, sequence length, 192]
Slice Last Time Step	[Batch size, 192]
Dropout	[Batch size, 192]
Linear	[Batch size, 2]

Training and Validation

For this part of the project, we validated the training loop. We observed a minimum validation loss of 44.68% at 8 epochs with a final validation loss of 48.66%. As for our hyperparameters, I used a learning rate of 0.001, weight decay of 1e-4, and Adam optimizer for 10 epochs. The train time was approximately 8 - 9 minutes, and the loss plot is shown in the following figure.

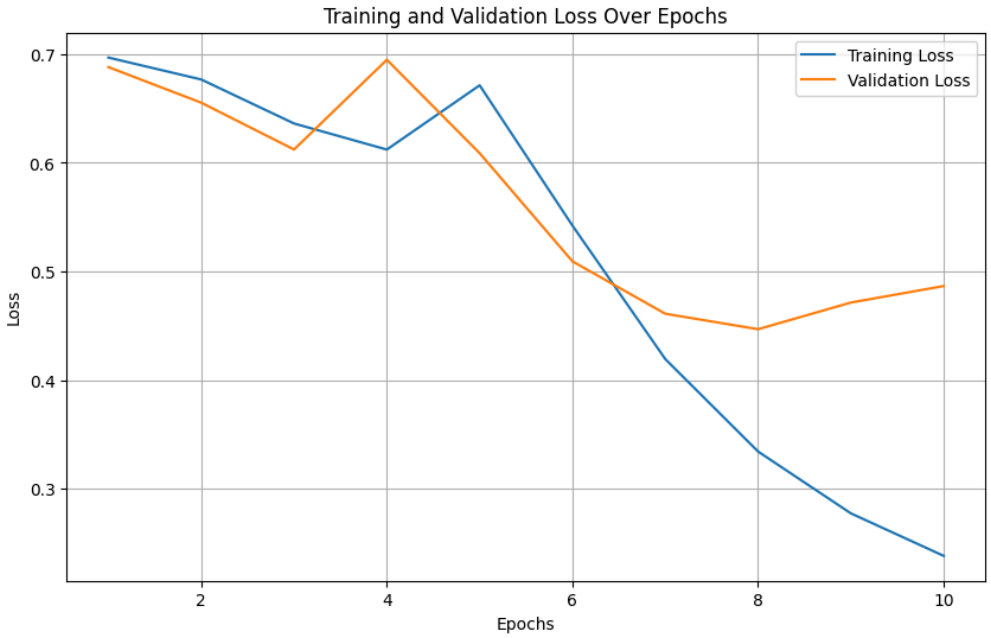

Fig 2. Loss plot during training

Testing

Using the test data and our trained model, we generated our predictions. We then reverse-map the predicted labels to the original, INFORMATIVE or UNINFORMATIVE, string labels to form the final Pandas DataFrame of the predictions on the test data (Fig. 3).

	Id	Text	Label	predicted Label
0	1235770448966754309	@USER PA hospitals don't have the capacity. La...	UNINFORMATIVE	UNINFORMATIVE
1	1235748200742416384	Coronavirus outbreak: Intel employee in Bengal...	INFORMATIVE	INFORMATIVE
2	1236129620363100161	Trump is trying to BS his way through the coro...	UNINFORMATIVE	UNINFORMATIVE
3	1241191765195001857	@USER How dramatically could we slow down this...	UNINFORMATIVE	UNINFORMATIVE
4	1245182515213676546	An Instagram post from comedienne Amy Schumer ...	UNINFORMATIVE	UNINFORMATIVE
...	...	...	...	...

Fig 3. test.tsv predictions

Accuracy Calculation and Export of Results

By formatting the results as a Pandas DataFrame, we can easily calculate the accuracy of the predictions across the actual labels and export the results to a CSV. To calculate the accuracy, I simply counted the number of rows in the DataFrame where the columns, Label and predicted Label, are different, divided by the total number of rows in the DataFrame, and subtracted from 1. For the test.tsv predictions shown in Figure 3, we had 464 wrongly predicted Labels out of a total of 2000 rows for an accuracy of 76.8%. The export of the data was done automatically by consolidating just the predicted Label and ID columns, renaming predicted Label to Label, and finally using the appropriate Pandas command to export to a CSV file.

Attempted strategies

Considering the number of layers (4) and number of parameters (851,525), it can be said that our model is very lightweight. Throughout the many iterations of the model design, this one performed the best. We tried simply tuning the parameters of the mode,l like hidden dimension and embed dimension, and hyperparameters such as the number of epochs, learning rate, and weight decay. If the values were set wrongly, then the loss plot showed signs of underfitting, as shown by an early low followed by a rapidly rising loss in the validation loss. Another route we explored was the use of light open-source models provided by Hugging Face, such as prajjwal1/bert-tiny. These provide the benefit of being pre-trained on large datasets, which can improve accuracy greatly, but the integration of the vocab and/or the model itself caused the model training time to grow significantly. Finally, we explored the idea of changing the LSTM (Long Short-Term Memory) layer to a GRU (Gated Recurrent unit) layer which could help for faster convergence with similar performance along with some pooling layers to help reduce the loss, but the loss did not change by much and the same problem as before existed where the loss plot showed signs of underfitting as shown by an early low followed by rapidly rising loss in the validation loss.

Future plans

Overall, it could be said that our model performs fairly well, as seen by the 76.8% accuracy despite being very simple in architecture. Something I want to explore in the future is possibly making the training dataset larger so that the trained vocabulary is more diverse, but still limited in scope compared to the vocab provided by pre-trained models such as BERT. By expanding the vocabulary just enough, we might be able to provide the model with extra data to form more accurate predictions. Another possible route to explore is making the model more complex by adding more layers. The model I have created right now only has four convolutional layers, so the depth of the model is not much. By adding more pooling, dropout, and LSTM layers, we can capture more features and patterns in the data to create a more accurate prediction at the expense of a larger training time.