Lesson: Artificial Intelligence II

Student's name: Christina Christodoulou

I.D Number: LT1200027

Field of Studies: Language Technology

Homework 3

REPORT

In this assignment, three multi-class vaccine sentiment classifiers are developed in Python language using stacked bidirectional Recurrent Neural Networks (RNN) with LSTM and GRU cells. My solution was implemented using the Pytorch library in Google Colab. The training and validation datasets were provided in csv form containing tweets and the labels 0 for neutral, 1 for anti-vax and 2 for pro-vax. The Glove pre-trained word embeddings from Twitter were utilized, which I found here: https://github.com/stanfordnlp/GloVe. I downloaded and unzipped the *glove.twitter.27B.zip* in Colab and used the text with the 25-dimension vectors (*glove.twitter.27B.25d.txt*) as the embeddings of the inputs of my models. For the development of the models, I experimented with:

- the number of stacked RNNs
- the number of hidden layers
- the type of cells
- the gradient clipping
- the dropout probability
- the activation functions
- the learning rate
- the batch size
- the number of epochs

The precision, recall and F1 score for each class as well as the accuracy of the models were calculated. I plotted the loss vs epochs and accuracy vs epochs as well as the ROC curve to check the models' performance. In the following paragraphs, I will explain in detail the steps followed in order to develop the models in addition to my observations from the best model's plots. What is more, I compare the best model in this assignment with the best model in the two previous assignments (Homework 1 and 2). Finally, I added attention mechanism to the best model and noted down my observations.

A. Code description - Steps

First of all, the necessary libraries like torch, tqdm, nltk, sklearn, numpy, pandas and matplotlib were imported as well as the necessary packages ('stopwords', 'punkt', 'wordnet') were downloaded from nltk. Then, the English set of stopwords was set to use. To run this notebook, the following files were uploaded: *vaccine_train_set.csv, vaccine_validation_set.csv*. The available device used for training was the GPU. The script contains many functions from the previous assignments that were used for data preparation and metrics' calculation as well as new functions and classes prepared for this assignment.

First of all, I used the *torchtext.data.Field* class to create the necessary fields for the TEXT and the LABEL. According to the Pytorch official documentation, it defines common datatypes that maintain a vocab object, which defines a set of possible values for elements of the field and their corresponding numerical representations (tensors). It also includes various parameters that can be set to represent the datatype in different ways. For the TEXT field, I set the *sequential* parameter to True to apply tokenization and tokenized the strings using the SpaCy tokenizer. I also lowercased the examples and selected to get a list containing their lengths and a tuple of padded minibatches with the *include_lengths* set to True parameter. For the LABEL field, I set the torch type as float and the sequential parameters as False so that it does not apply

tokenization. I used the *data.TabularDataset.splits* function to load the new cleaned dataframes (train,validation) in csv form and create the dataset with the fields. The vocabulary was built using the train dataset in order to index all the tokens. For the indexing, I used the Twitter Glove file of 25-dimension embeddings, with no minimum frequency. By default, any words that appear in the vocabulary, but do not appear in the chosen embeddings get initialized to a vector of all zeros. This could slow down the speed at which I converged to a decent solution. To avoid this, I used the unit_init argument to set any such words to a vector with its entries chosen from a normal distribution.

Functions for data preparation and preprocessing:

- *read_explore_dataframe(csv_file)* function: takes as parameter a file in csv form. It opens and reads the file as a dataframe using the pandas library and utf-8 encoding, gets the values and number of its values. It also checks whether there are empty values and duplicates, and then fills these missing values.
- *text_preprocessing(text)* function: takes as parameter a text and applies some pre-processing steps like removing unusual characters, urls, punctuation, emoticons, numbers and applying lowercasing. Apart from this, it tokenizes the text using the *word_tokenize* function and gets the lemmas of the tokens using the *WordNetLemmatizer* offered by nltk and returns the re-created sentences with their lemmas. These two last steps were added for this assignment aiming to optimize the training process.
- *get_columns(dataframe, feature, label)* function: takes as parameters a dataframe, the name of the feature's column and the name of the label's column. More specifically, it selects the columns of interest and puts them in a new dataframe using pandas. It checks the distribution of each class and then separates the values of the label's column into 3 classes, 0 for neutral, 1 for antivax and 2 for provax. Here, I apply resampling of the classes due to the imbalance data problem (mainly in class 1) I highlighted in the previous assignments, using the *resample* function offered by sklearn. Instead, I created a new dataframe with the equal number of samples so as to achieve data balance and fair predictions of the model for each class. What is more, the previous function is applied to the text (feature) column for pre-processing. After pre-processing, the feature column is assigned as the x input value and the label column as the y target value. The x and y values are concatenated in a new cleaned dataframe which is returned by the function.
- *create_iterators(train_dataset, validation_dataset, batch_size)* function: takes as parameters two datasets and a batch size number. It uses the *torchtext.data.BucketIterator.splits* to create two iterators for the train and validation data. In the iterators, each index represents a token and each column represents a sentence. The number of columns is equal to the number of the batch size. The data are sorted in each batch according to the lambda function. This function instructs the iterator to try and find sentences of similar length. The examples are also shuffled in each epoch run during training.

Function for model training:

- *train(model, iterator, optimizer, criterion, clip)* function: takes as parameters a defined model, an iterator, a defined optimizer and loss function and a number for gradient clipping. It performs the training phase. More particularly, it computes the predictions using the tuple of the Tweet, which includes tensors and the lengths of the tweets. It also uses gradient clipping to avoid exploding gradients, which is a common problem in recurrent neural networks, especially LSTM. By "exploding gradients" it means numerical overflow or underflow due to large updates to weights during training.[1] Finally, it calculates and prints the loss as well as the accuracy of each epoch using the predictions and the true labels.

Functions for model evaluation:

[1] https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/

- *evaluate(model, iterator, criterion) function:* takes as parameters a defined model, an iterator, and a defined loss function. It performs the evaluation phase. More particularly, it computes and prints the loss and the accuracy of each epoch using the predictions and the true labels.
- testing(model, iterator) function: takes as parameters a defined model and an iterator. It makes predictions and returns a list of the predicted labels and the true labels.

Functions used for metric calculation:

- *accuracy(preds, y_true)* functon: takes as parameters the predictions of labels of the model and the true labels. It changes the predictions into one dimensional tensors and calculates the accuracy. This function is specifically used in the *train* and *evaluate* functions to calculate and print the accuracy of each epoch.
- *calculate_metrics(y_true, preds)* function: takes as parameters the y true values and the predicted values. It calculates all the metrics asked for this assignment, namely the precision, F1 score and recall. It also returns the estimated the accuracy of the models and prints the classification matrix as a general summary of all the metrics.

Other functions used:

- *count_parameters(model)* function: takes as parameter a defined machine learning model and returns the summation of its parameters.
- *epoch_time(start_time, end_time)* function: calculates the time in seconds during training and evaluation in each epoch
- *set_seed(seed_value)* function: sets the seed for reproducibility of the same results.

Functions that include all the other functions for each model:

- *train_evaluate_bidirectional_RNN_model_LSTM(text_field, data_train, data_val):* for the RNN with LSTM model
- *train_evaluate_bidirectional_RNN_model_GRU(text_field, data_train, data_val):* for the RNN with GRU model
- *train_evaluate_bidirectional_RNN_model_ATTENTION(text_field, data_train, data_val):* for the best model from the previous ones with attention mechanism

These functions take as parameters the TEXT field, the train and the validation data created by the TabularDataset.splits function. They construct a Bidirectional RNN with LSTM and GRU cells, set the model's parameters, train and evaluate the model. They compute and print the evaluation metrics as well as the loss, accuracy vs epochs plots and the roc curve used for multi-class classification[2].

Class that builds the models:

BIDIRECTIONAL_RNN_LSTM_GRU(nn.Module), which constructs a Bidirectional Recurrent Neural Network classifier with either LSTM or GRU cells based on selection. According to Devopedia, the Bidirectional RNNs process the sequence in both directions. In fact, two distinct RNNs are used: one for the forward and one for the reverse direction. As a result, a hidden state is the output of each RNN, which are concatenated to form a single hidden state. [3] Both RNN models have the same structure, with the only difference being the cell type (GRU/LSTM) and the initialization of the hidden state. The hidden state for the LSTM is a tuple containing both the cell state and the hidden state, whereas the GRU only has a single hidden state. To initialize such a model, the following parameters were given: the cell type, the size of the vocabulary, the number of dimensions representing each word, the hidden size, the number of classes, the embedding vectors, the numbers of layers, the dropout and the pad token index from the vocabulary. It

[2] https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html#sphx-glr-auto-examples-model-selection-plot-roc-py

[3] https://devopedia.org/bidirectional-rnn

applies two dropout layers and the RELU activation function. The *forward* function defines the forward pass of the inputs. The tweets are passed through the embedding layer added with a dropout layer to get the embeddings, then they are packed to process the non-padded elements. In this way, the computation is faster. The packed embeddings pass through the LSTM or GRU cells to learn from both directions. The packed output of the cells is unpacked to a tensor. Then, the final forward layer and the backward hidden layers are concatenated and passed through a dropout layer. Finally, the single hidden state is passed through the fully connected linear layer and then the RELU activation function to get the probability of the sequences.

For question 2, I developed another class of a Bidirectional RNN with LSTM/GRU similar to the previous one, but I incorporated attention. This class was used to train the best model from question 1. The *attention(self, output, final_state)* function computes the weights for each sequence in the RNN's output and then computes the hidden state. The new hidden state from the attention is passed through the fully connected linear layer and then the RELU activation function to get the probability of the sequences.

B. EXPERIMENTS

Firstly, I experimented during the pre-processing phase with two lemmatizers: the PorterStemmer and the WordNetLemmatizer. I noticed that the WordNetLemmatizer outputs about one thousand more stems than PorterStemmer and offered better model evaluation results.

During training of the models, I experimented with:

- The number of stacked RNNs: in order to create a stacked RNN, the minimum number of layers was 2. I used from 2 to 5 layers, but ended up selecting only 2, because I tried to create a very simple model to avoid overfitting.

- The number of hidden layers: I initially used 50,128,256,500, but ended up selecting 20 to overcome the serious overfitting problem. According to a rule of thumb presented by Java by Jeff Heaton (2017), "The number of hidden neurons should be between the size of the input layer and the size of the output layer".[4] I decided to follow this rule of thumb and limited the number of hidden layers between the input layer size, which were the embeddings' dimensions (25) and the size of the output layer, which was the number of the classes (3), namely 3,5,10,15,20 and 25. The final choice was 20 hidden neurons as it offered the best results.

- The gradient clipping: I used 1 to 5, but ended up using 2.

- The dropout probability: I used 0.2 to 0.7, but ended up using 0.5.

- The activation functions: I experimented with the Softmax and RELU activation functions, but used the RELU as it offered better evaluation results and smoother learning curves.

- The learning rate: I used 0.1, 0.01, 0.001, 0.0001, 0.0008, but ended up using 0.0001.

- The batch size: I used 10, 32, 64, 100, 128, 256 and 1048, but ended up using 256.

- The number of epochs: I used 5, 10, 20, 35, 40, 45, 50, 100, but ended up using 40 because the results did not vary with more number of epochs.

[4] https://www.heatonresearch.com/2017/06/01/hidden-layers.html

- The type of cells: I experimented with both LSTM and GRU cells with all the above hyperparameters.

  ✓ I used the Adam optimizer and the cross-entropy loss function as asked in this assignment. I only added weight decay of 1e-4 to the optimizer.

C. RESULTS – OBSERVATIONS

I trained Bidirectional RNN with both type of cells (LSTM/GRU) and I observed the following:

1. The training phase took less amount of time for the RNN models with GRU cells to complete than with the LSTM cells. This is mainly due to the fact that the GRU is less complex in architecture than the LSTM because it has a smaller number of gates. The GRU has only two gates, called reset and update gate, while the LSTM has three gates, namely the input, output and forget gate. In the present assignment, the LSTM had 539,363 trainable parameters, while the GRU 535,003 trainable parameters with the hyperparameters summarized above. Since the GRU had less trainable parameters, it used less memory and was trained faster than the LSTM. [5]

2. I experimented with the same hyperparameters and both types of cells. As can be seen from the results below, the model with the LSTM cells achieved better results than the one with the GRU. More specifically:
   - Precision: Class 0 (neutral) achieved the highest ratio of correctly predicted positive observations to the total predicted positive observations comparing to the other two classes in both models. The precision in the GRU was higher than the precision in LSTM for class 0. Class 1 (anti-vax) scored lower in the GRU than the LSTM. Class 2 scored higher in GRU than in LSTM.
   - Recall: Class 1 achieved the highest ratio of correctly predicted positive observations to the number of observations comparing to the other two classes in both models. The recall in the GRU was higher than the recall in LSTM for class 1. The LSTM achieved a little higher score in class 0 than the GRU. Class 2 scored lower in the GRU compared to the LSTM.
   - F1 score: Class 0 achieved the highest weighted average of Precision and Recall, equally in both models compared to the other classes. Class 1 achieved the same score in both models. Class 2 achieved a little higher score in the LSTM than the GRU.
   - Accuracy: The Bidirectional RNN with LSTM cells achieved a little higher accuracy than the model with the GRU with no much difference. (LSTM: 6492842535787321, GRU: 6478235465965527)

Precision, Recall and F1 score for each class + Accuracy of models

MODELS	PRECISION (CLASS 0 | 1 | 2)	RECALL (CLASS 0 | 1 | 2)	F1 SCORE (CLASS 0 | 1 | 2)	ACCURACY
BI RNN WITH LSTM	0.75 | 0.64 | 0.56	0.71 | 0.73 | 0.51	0.73 | 0.68 | 0.53	0.65
BI RNN WITH GRU	0.78 | 0.60 | 0.58	0.69 | 0.79 | 0.47	0.73 | 0.68 | 0.52	0.65

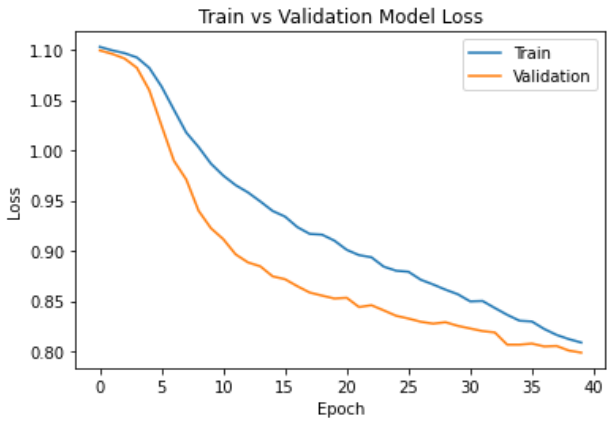
[5] https://analyticsindiamag.com/lstm-vs-gru-in-recurrent-neural-network-a-comparative-study/

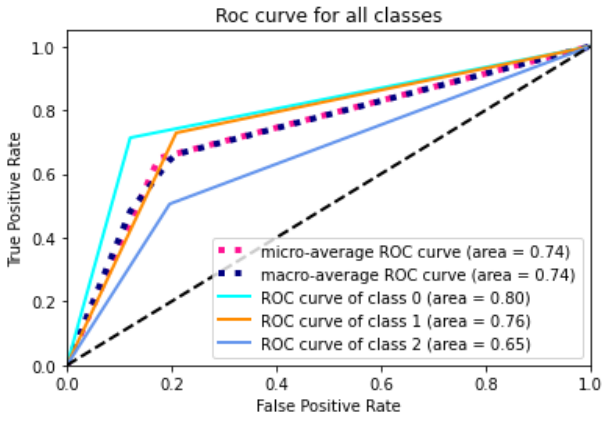
Best model: Bidirectional RNN with LSTM cells

Best model's exact architecture:

```
BIDIRECTIONAL_RNN_LSTM_GRU(
  (word_embeddings): Embedding(20872, 25, padding_idx=1)
  (rnn): LSTM(25, 20, num_layers=2, batch_first=True, dropout=0.5,
bidirectional=True)
  (predictor): Linear(in_features=40, out_features=3, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (relu): ReLU())
```

Explaining the Loss vs Epochs plots and the ROC curve





As can be seen from the plot of loss vs epochs, the model is underfit, because the training loss decreases and continues to decrease at the end of the plot. This indicates that the model was capable of further learning and that the training process was halted prematurely. Although, I trained all models with a small and a great number of epochs, with more complex and less complex architecture, I dealt with many underfitting and overfitting issues. No matter the models' fine-tuning, the plot of training loss does not plateau and continually decreases (underfitting), while with some parameters the plot of validation decreased to a point and then started to increase again and fluctuated all over the training epochs (overfitting). It also seems that the loss learning curves of the model trained have a smaller gap between them towards the last epochs, which means lower data variance.

ROC Curve tells us how much model is capable of distinguishing between classes. It can be observed that the AUC score for class 0 is the highest, second comes the score for class 1, while the score for class 2 is the lowest. Thus, there is greater chance of distinguishing the neutral class rather than the pro-vax class. The curves are not so smooth, which might mean that the model can only provide discrete predictions instead of a continuous score.

Comparing the best model with the models of the previous assignments

At the board below, I compare the three models from the assignments 1,2 and the present assignment.

1. First model: Softmax Regression model with 21400 features from the CountVectorizer
2. Second model: First Feed-Forward Neural Network model with 25-dimension Glove pre-trained embeddings
3. Third model: Bidirectional Recurrent Neural Network model with LSTM cells and 25-dimension Glove pre-trained Glove embeddings

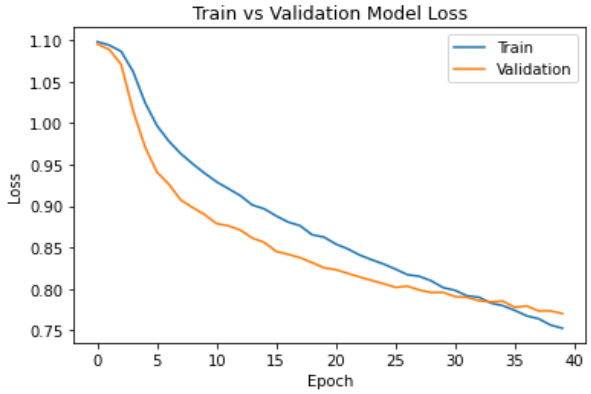
It can be estimated from the board below that the Softmax Regression model, which was the best model from the previous assignments, achieved the highest F1 scores in the three classes and the highest accuracy amongst the models. The Bidirectional RNN with LSTM, which is the best model in the present assignment, achieved lower than the Softmax Regression model due to underfitting issues. The Feed-Forward Neural Network achieved the lowest F1 scores, in class 2 even below 0.50, as well as the lowest accuracy.

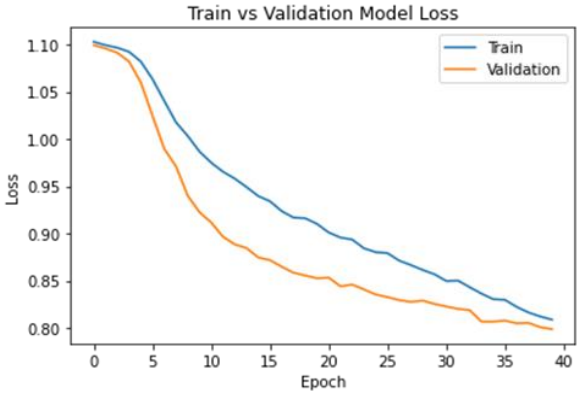
MODELS	PRECISION (CLASS 0 | 1 | 2)	RECALL (CLASS 0 | 1 | 2)	F1 SCORE (CLASS 0 | 1 | 2)	ACCURACY
SOFTMAX REGRESSION MODEL	0.93 | 0.94 | 0.91	0.89 | 0.98 | 0.91	0.91 | 0.96 | 0.91	0.92
FEED-FORWARD NEURAL NETWORK MODEL	0.58 | 0.59 | 0.58	0.73 | 0.69 | 0.32	0.64 | 0.64 | 0.41	0.58
BI RNN WITH LSTM	0.75 | 0.64 | 0.56	0.71 | 0.73 | 0.51	0.73 | 0.68 | 0.53	0.65

Evaluating the results of the model with attention

From the board below, it can be observed that there is no great improvement in the evaluation results with the introduction of the attention mechanism as expected. In the F1 score, the score for class 2 was improved, whereas the scores for the other classes were lower. The accuracy did not improve either.

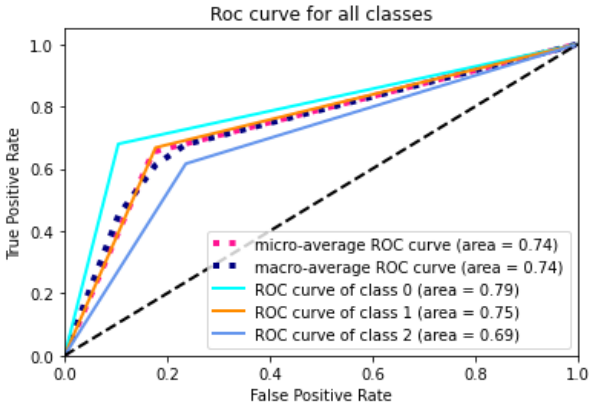
MODELS	PRECISION (CLASS 0 | 1 | 2)	RECALL (CLASS 0 | 1 | 2)	F1 SCORE (CLASS 0 | 1 | 2)	ACCURACY
BI RNN WITH LSTM	0.75 | 0.64 | 0.56	0.71 | 0.73 | 0.51	0.73 | 0.68 | 0.53	0.65
BI RNN WITH LSTM + ATTENTION	0.76 | 0.65 | 0.57	0.68 | 0.67 | 0.62	0.72 | 0.66 | 0.59	0.65

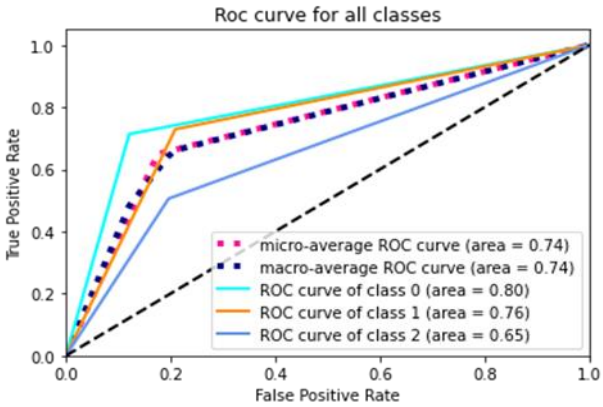




Left plot: RNN with LSTM and Attention mechanism Right plot: RNN with LSTM

From the left plot it is shown that the introduction of the attention mechanism created lower data variance as the gap between the two loss learning curves is smaller. This means that the model starts to generalize better than without the attention mechanism. However, the validation loss starts to increase after 35 epochs, which is a sign of overfitting.





Left ROC curve: RNN with LSTM and Attention mechanism Right ROC curve: RNN with LSTM

It can be observed from the left ROC curve that the AUC score for class 2 was improved, while for classes 0 and 1 was a little lower. Again, there is greater chance of distinguishing the neutral class rather than the pro-vax class. The curves remain not so smooth, which might mean that the model can only provide discrete predictions instead of a continuous score.