Assignment 3, Artificial Intelligence 2

Vyron-Georgios Anemogiannis 1115202000008

1 Introduction

In this assignment we are tasked with creating a recurrent neural network model in order to do sentiment analysis on IMDB movie reviews. We are using bidirectional stacked RNNs with LSTM and GRU cells. For this assignment we have experimented with various architecture combinations and in the end we tried to implement an attention mechanism for the model.

2 Data Processing

Immediately after getting the data from the tab separated file we start the regularisation process.

- We remove all the apostrophes.
  By doing this step, sentences like ”I didn’t like” become ”I didnt like” helping the model distinguish positive sentences ”I did like the movie” from negative ones ”I didnt like the movie”.

- We make all the characters lowercase.
  Since the program distinguishes upper case from lower case characters, it is preferable to only keep one of them. So in the word vectors now, we have the ascii codes of only lowercase characters, decreasing the possible combinations.
- We remove

  Scrolling through the reviews, we often find the above mentioned sequence. We simply remove it since it has nothing to do with the actual review.
- We keep only alphabetic characters
  Keeping only the alphabetic characters, gives us word vectors of 32 possible ascii codes.
- We remove the stopwords
  Those are words that appear often in texts, removing them will help the model distinguish words that acrually make an impact on the sentiment of the review.

Since we are using Glove embeddings we deemed that stemming and lemmatization could hinder the process making words not recognisable in the Glove dictionary. This is because we noticed that in the assignment 1 where we used this technique, the lemmatization process didn’t manage to ”fix” the words, leaving them in a state without endings.

3 Data Vectorisation Using GloVe

As requested by the assignment instructions, we use GloVe word embeddings in order to vectorize our reviews. For each word, if it is in the dictionary, we match it to a 300D word embedding. We believe that the greater the vector dimension, the better will be the representation for each word. Afterwards we take their mean in order to reduce the size of the data we have to pass to the model. More specifically we observed that with a dictionary of only 50D there is no way we achieve a 80

4 Creating the RNN model

The model consists of an initialisation and forward method.
For the initialisation we get various hyper parameters as arguments. More specifically:

- Cell Type: either LSTM or GRU depending on the architecture we want to use

- input size: dimension of the input
- hidden size: The hidden dimension of each RNN cell
- number of layers: How many RNNs do we want to stack
- dropout: dropout probability
- device: the device to run the model. Hopefully cuda
- Output size: Here it is set to 2, the probability of each review belonging in the positive or negative class.
- attention: bool variable, if we want to have attention in the model
- number of heads: Amount of heads to be passed as argument in the multi head attention
- skipping: bool if we want to implement connection skipping

Inside the method, we start by creating the recursive NN with the type and the various hyper parameters passed as arguments. Unfortunately due to pytorch limitations skip connections cannot be implemented easily. For this reason, when connection skipping is enabled, we stack the RNN cells manually. Afterwards, if we have attention enabled, we initialise 3 linear fields, one as key, one as value and one as query vectors. Their input and output size is two times the hidden layer since we are using a bidirectional RNN. Then we initialise the multi-head attention module with the number of heads given as an argument. Last, we have a final linear layer in order to get the 2*hidden size vector and convert it to the 2sized vector needed as output of the model.
As for the forward part, we pass the input through the RNN.The output is only 2 dimensional in contrast with the tutorial for the fashion MST that was 3 dimensional. We cannot explain why we got this output, but since it works, we will just move on. If we have attention enabled, we then pass the RNN output through the key, value and query layers and after pass those outputs as inputs to the multi-head attention. The result is summed with the RNN output to give us the final output that is then passed to the final linear layer in order to become the 2 class classification. Note that we tried to use the haddamand product instead of addition but we got worse results. The effectiveness of attention will be discussed in a later section.

5 Train and Test functions

For the training function we get as arguments:

- model: the model we want to train
- epochs: maximum amount of epochs, the model will be trained for. Due to early stopping, we rarely pass 10 epochs.
- data loader: the training set
- learning rate: the learning rate for the adam optimiser
- X and Y validation: a validation data set to cut of early the training. The validation data set is different from the training one.
- Printing: bool variable to print each epoch
- clip: gradient clipping to be passed as an argument in clip grad norm

This is a simple training function. We use the adam optimiser and the cross entropy loss function as instructed. For each epoch if the validation set yields worse results, we end the training early and return the previous epoch model.
The test function simply prints SKLEARN's classification report for a given test set. We also return the f1 score (used for Optuna).

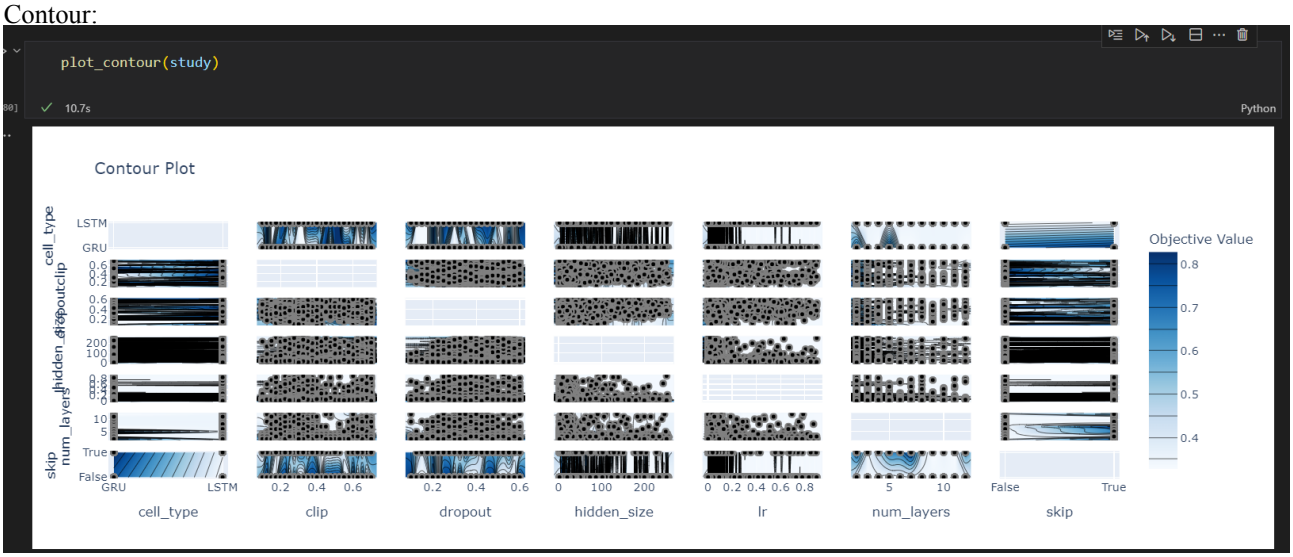
6 Neural Network Parameters Tuning

In this section we tried various hyper parameter combinations to see what works best. In order to do this, we first tried to use grid search since optuna refused to help...We tried our own function that tries 6 different hyper parameter combinations. After about 20 hours we got our results (fairly disappointing results we might add). The results were printed in a txt file and with another function and simple and reliable control find, we saw about 113 models that managed to achieve the maximum accuracy witch was 0.83.
Some things that we can say by analyzing the results from the output file are that even a simple 2 layer 2 hidden size model can achieve the maximum result.
Let's go over each parameter we tested with:

- stacked RNNs: All but onemodels that achieved the highest score, had 2 or 4 stacked RNNs. So it is safe to assume that for this classification problem there is no big need for much stacking. Only one model with 6 stacked RNNs managed the 0.83 score.
- hidden states: We managed to get the best score with all the hidden state amounts we tried. Unfortunately bigger hidden states didn't manage to surpass the score of the small one (2).
- type of cells: Both GRU and LSTM cells managed the best score. In addition, neither stood out, by outnumbering the other type of cell. This is most likely because the main difference is the time it takes to converge.
- skip connections: We could not implement those with the model we used unfortunately. We tried everything else tho :)
- gradient clipping: All clip sizes we tried (0.2,0.4,0.5,0.7) gave the best result. There is a tendency for bigger clip sizes to yield more best scores.
- dropout probability: The expected behavior would be to see that a bigger dropout probability would work better with more stacked RNNs. A dropout between 30 and 50
- learning rate: We tried .1 .2 .01 .02 .001 .002 values to get a better understanding of what order of magnitude learning rates do to the model. We see that as the model becomes more complex, it prefers a learning rate with 3 decimals as compared to simpler ones, that tend to work best with 2 decimal learning rates.
- batch size: From the lectures, we gathered that we should have a big batch size. We decided that a batch size of 6001 gives us 6 batches and works decently well.
- num of epochs: There is no reason to experiment with the number of epochs, since the early stopping, will never let the model reach a maximum. Each train phase does on average 10 epochs.

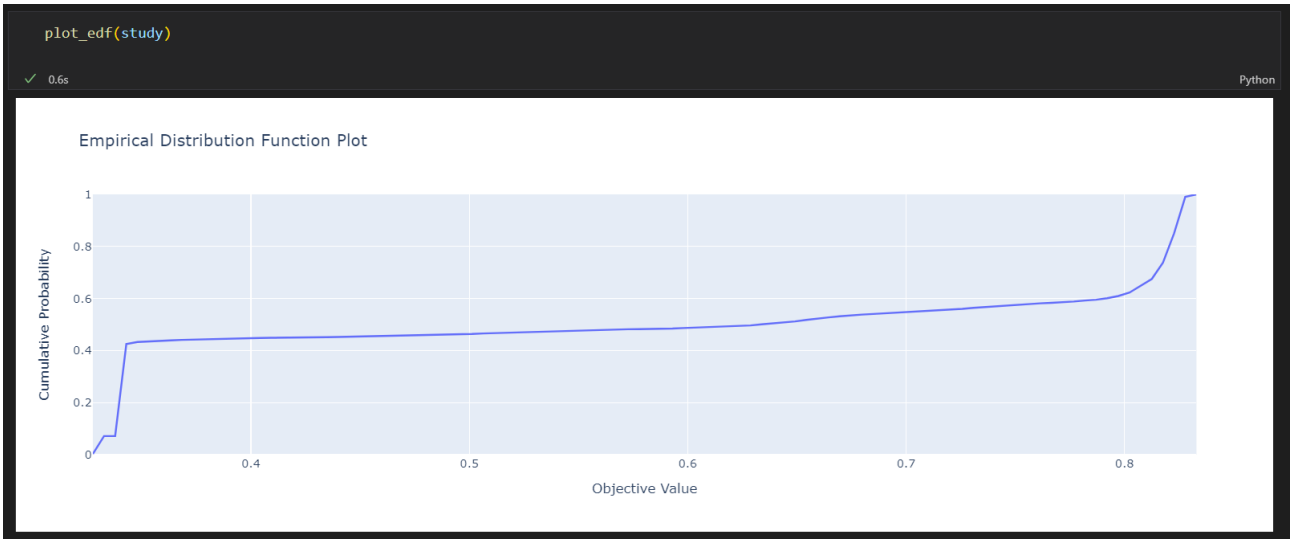
After a bit of swearing and lots of coffee we also got optuna to cooperate, yey!
We tried random search for 5000 trials with added connection skipping, since we hadn't implemented it in the previous run.
The best model came out to have the following hyper parameters: 'hidden size': 16, 'num layers': 3, 'cell type': 'LSTM', 'dropout': 0.375, 'lr': 0.042, 'clip': 0.325, 'skip': False
Even tho we used random search, we still didn't surpass the grid search best results. This time tho, we have some diagrams to show! (note those plots are not in the HW3.ipynb file since in order to visualize them, VS code becomes unresponsive)

- Contour:



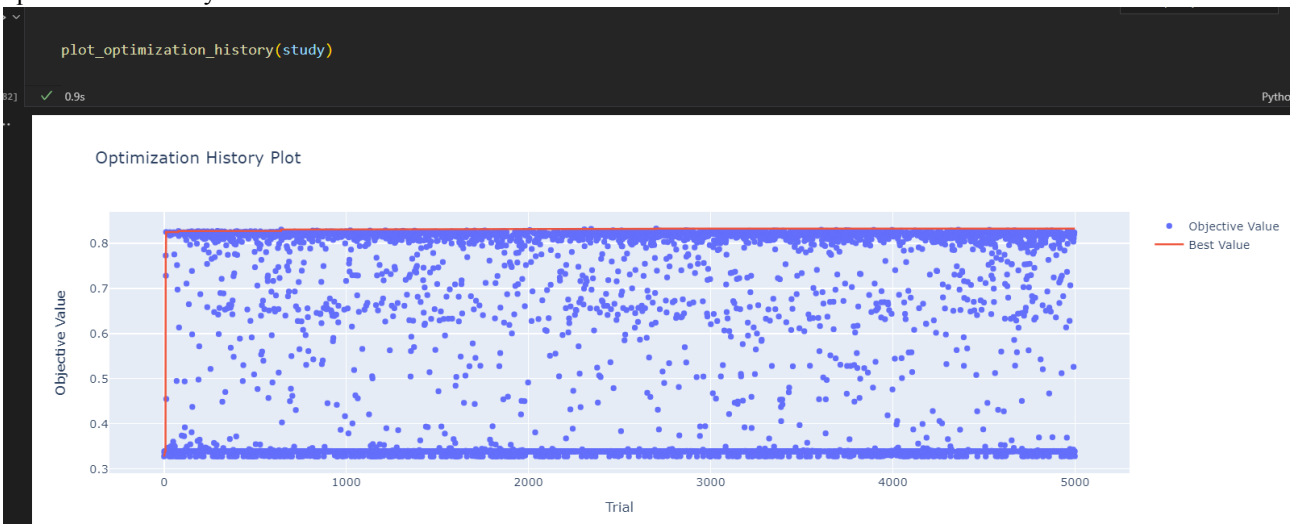
Contour plots how a Z value changes as a function of inputs X and Y.
We see a preference for GRU cell types, which is reasonable since they converge quicker with their extra parameters. We also see that the number of layers have the biggest impact on the score, with a preference for small amounts of layers. The learning rate has denser (thus better) results, the lower it is.

- edf:



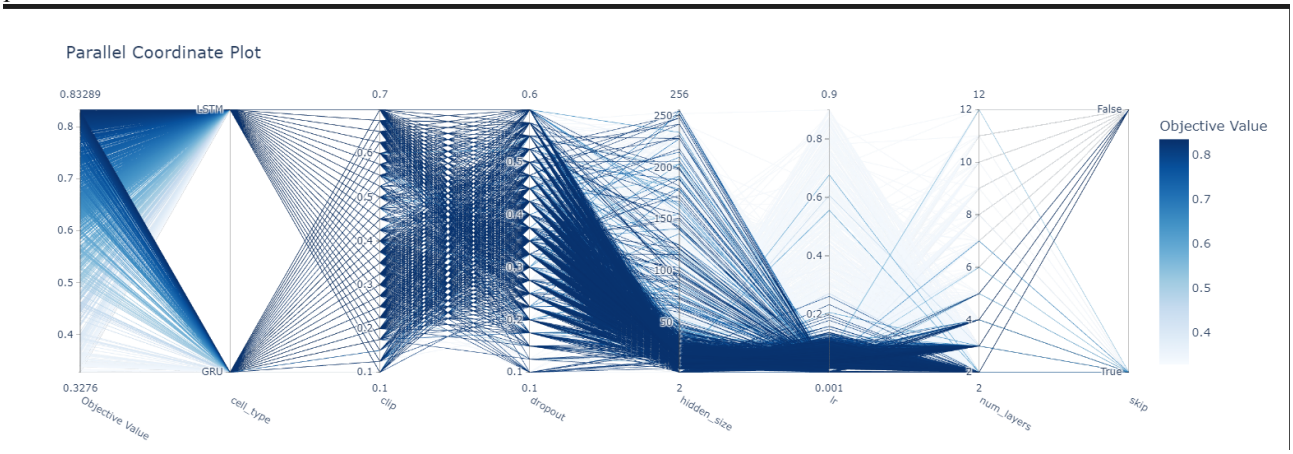
Since the probabilities are inverse, we see that we cant often get high accuracy values and we most often get accuracies closer to 0.3 to 0.4.

- optimisation history:



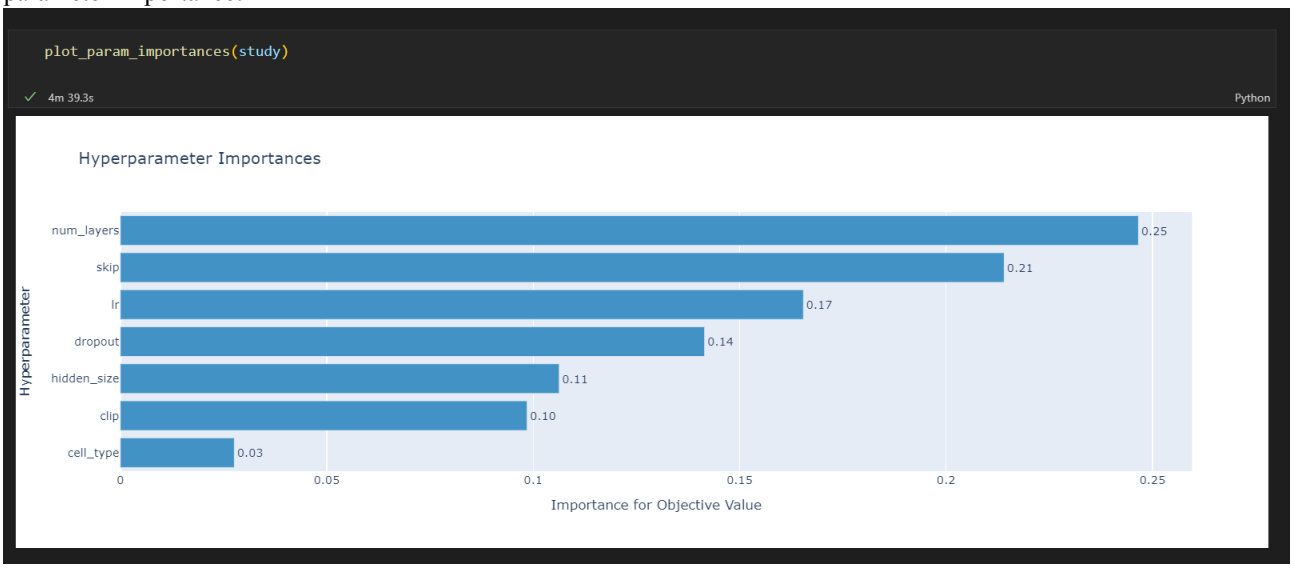
We see that in most trails we achieved either accuracy towards 0.3 to 0.4 (this happens when early stoppig intervines in the first epochs) or accuracy towards the maximum. There are not many outliers in between.

- parallel coordinates:



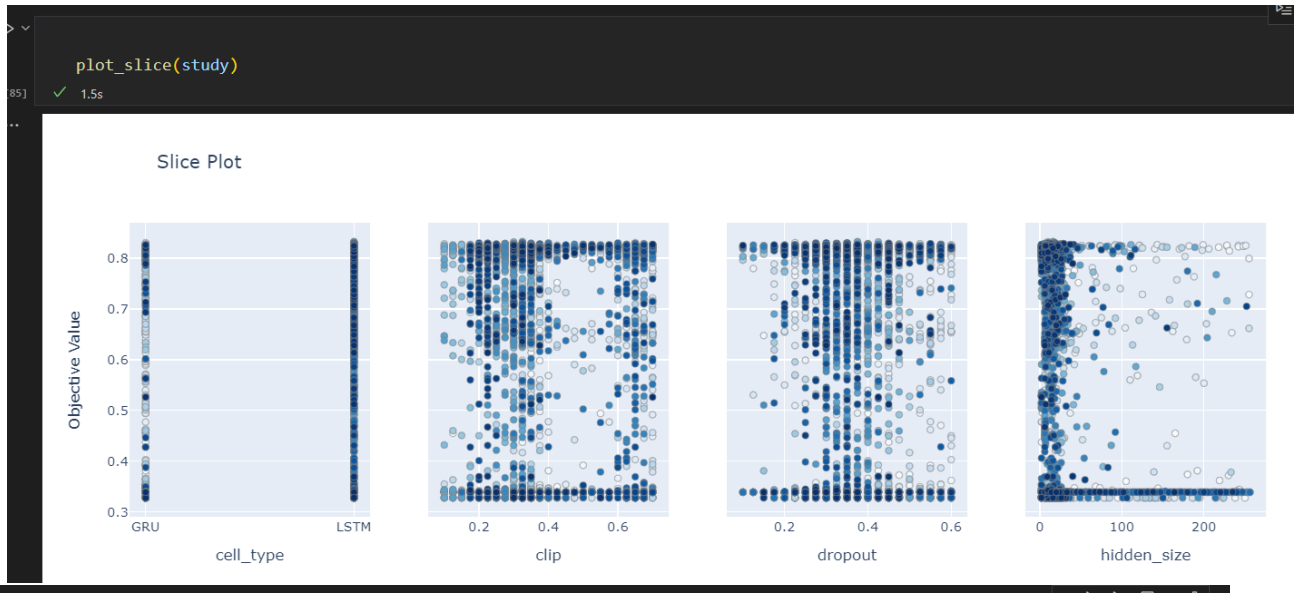
Here the darker the lines, the better the score. We see that both LSTM and GRU cells have achieved good results. Clipping is fairly uniform no matter its value. Dropout prefers higher values. For the hidden state, lower is clearly better, and so do the nymber of layers. The learning rate, clearly prefers 3 decimals. As for connection skipping, there is a preference to not being used.

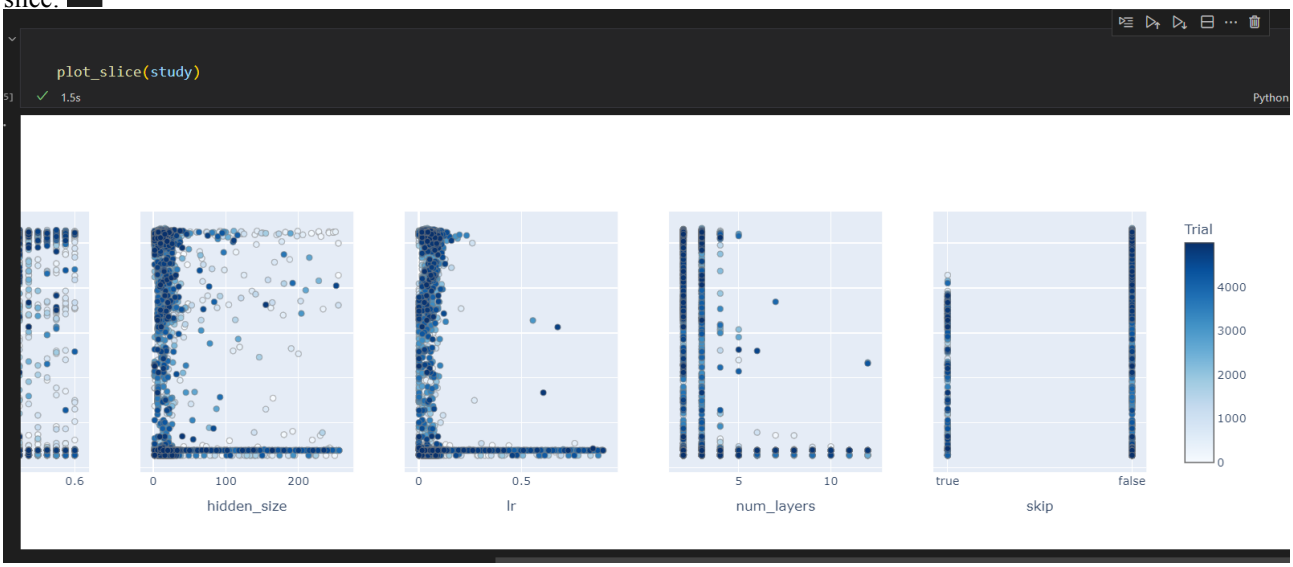
- parameter importance:



So the biggest difference to the final value, came from the amount of layers. Skipping was a close second. Learning rate and dropout did play a role but not as important as the first 2. Last the cell type didn't really make a difference. This could be attributed to the fact that both can achieve the same results, the difference is int he time it takes for them to converge.

- slice:





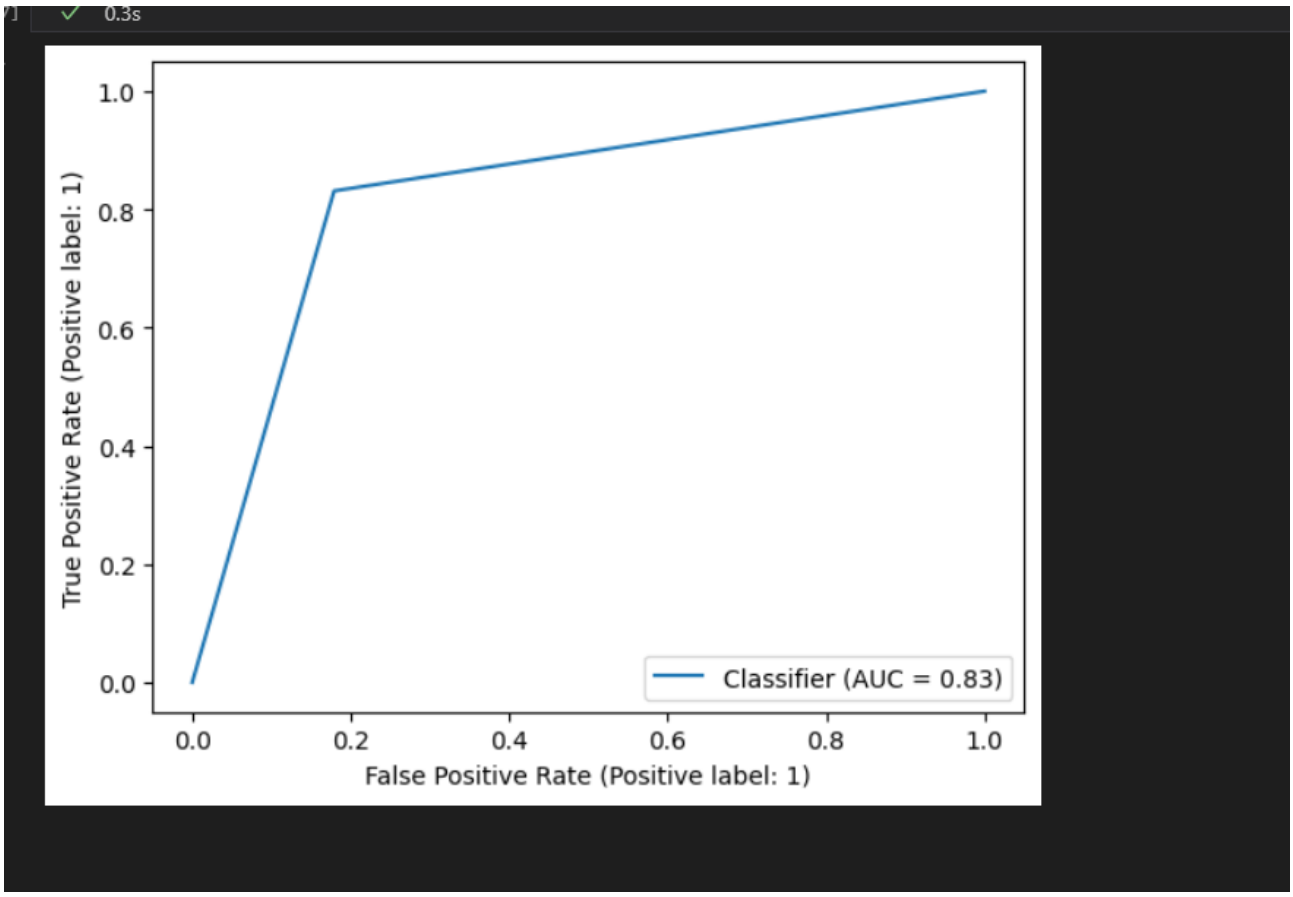
So we see that: both cell types can achieve good results, clipping works best with values from 0.2 to 0.4 but all can get good scores, dropout gets best results for values close to 0.4, hidden size is strictly small, laming rate, also must be extremely small, layers sjould be less than 5 and for connection skipping only false values give maximum results.

7 Results

The highest result we managed to achieve is accuracy = 0.83. No mater the hyper parameter combinations, we could not surpass this.
Let's start discussing the attention mechanism. By implementing this, we managed a score 0.8329308032989502 without and 0.8343590497970581 with attention. This we would consider to be just randomness during training, and should we run both models again, we expect different results. The sure thing is that attention does play a role during training. We believe that since we have reached the maximum performance we can get out of this data, we cannot showcase very well the effectiveness of the attention mechanism.
For the final scores, we decided to use the best model the second optuna study gave us, we also use this model as the saved one, for the test set:

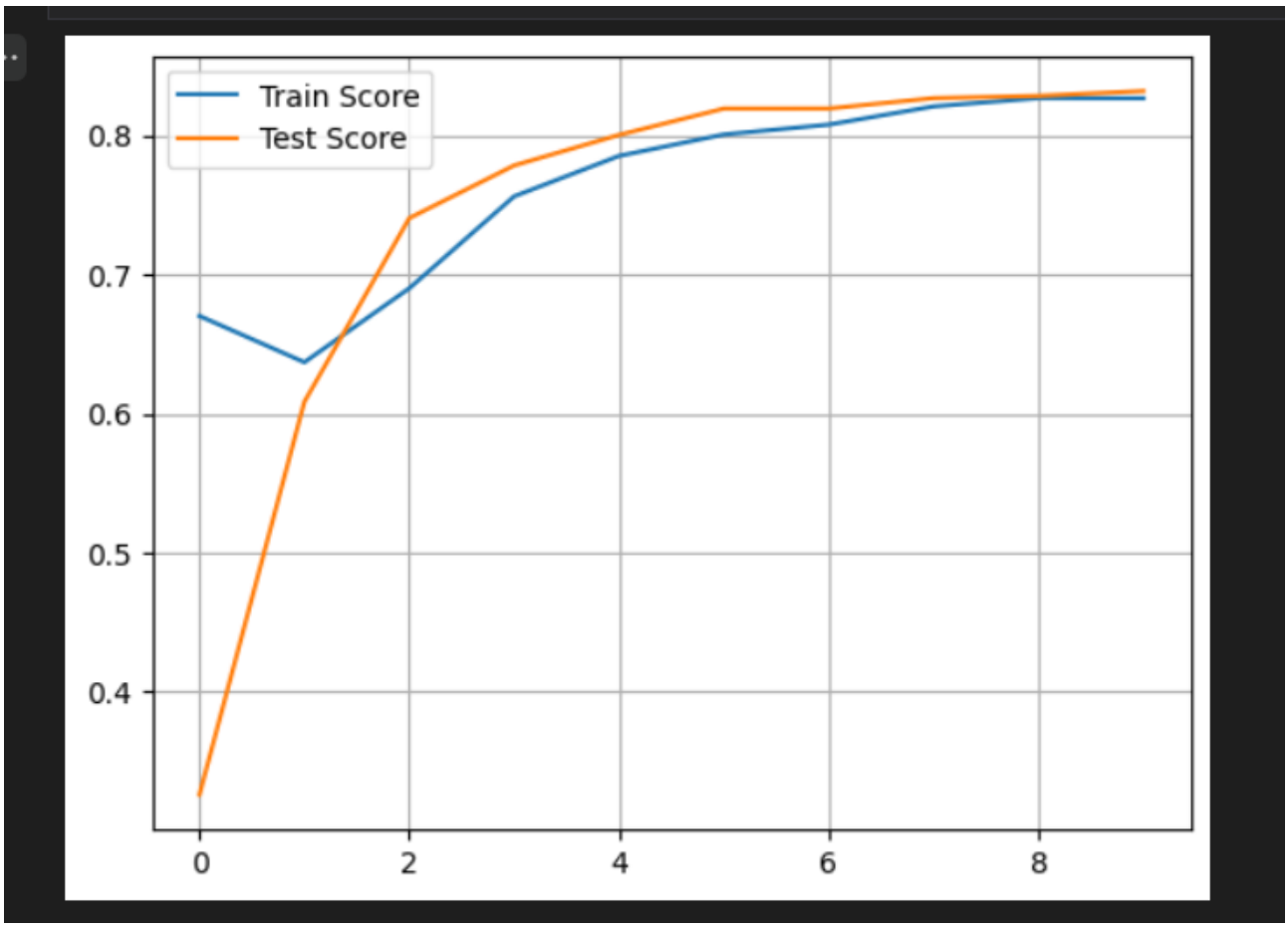
```
for epoch = 0 we got loss = 0.6932142972946167 and f1 score with
for epoch = 1 we got loss = 0.6832873821258545 and f1 score with
for epoch = 2 we got loss = 0.6324926018714905 and f1 score with
for epoch = 3 we got loss = 0.5186166763305664 and f1 score with
for epoch = 4 we got loss = 0.4668842554092407 and f1 score with
for epoch = 5 we got loss = 0.448780030012130 74 and f1 score with
for epoch = 6 we got loss = 0.4205424487590 79 and f1 score with v
for epoch = 7 we got loss = 0.3887808918952942 and f1 score with
for epoch = 8 we got loss = 0.4042562246322632 and f1 score with
for epoch = 9 we got loss = 0.385325253009796 14 and f1 score with
for epoch = 10 we got loss = 0.37200102210044 86 and f1 score with
for epoch = 11 we got loss = 0.3797561526298523 and f1 score with
              precision    recall  f1-score   support

           0       0.81      0.86      0.83      2273
           1       0.85      0.79      0.82      2228

    accuracy                           0.83      4501
   macro avg       0.83      0.83      0.83      4501
weighted avg       0.83      0.83      0.83      4501
```

The ROC curve is:



The learning curve using train set size as experience is:

The learning curve epochs as experience is:



Those learning curves tell us that not much overfitting is taking place. This is expected thanks to the early stopping mechanism.

8 Test Set - What to run

For this assignment we have handed in 2 IPYNB files, HW3.ipynb and test.ipynb.
All the work was done on HW3.ipynb locally on our PC (It was both faster and kaggle and google colab tested our patience). This file is not compatible with google colab and cannot run there (some f1 score incompatibility we cannot solve, it requires the task argument, but with this argument, it will then not work on kaggle and locally). Even tho you can't run it, all the cells have their outputs visible if you want to check them. In order to actually use the test set on google colab, we have uploaded the test.ipynb file. This file omits the model parameter tuning and by giving the test set and the trained model path and running all the cells, you can test the model.

9 Comparison with homeworks 1 and 2

Comparing the RNN results with the previous types of models (logistic regression and linear NN) yields disappointing results.
We must start with acknowledging that in the logistic regression model, the data given was processed differently than the other 2 assignments. We did stemming and lemmetization and then we used the Tfidf vectorizer. In the other 2 assignments we only used GloVe embeddings. This means that it is not an apples to apples comparison. The logistic regression managed to yield us an impressive accuracy rate of 0.89 while both NNs only managed up to 0.83. No matter what we have tried, we simply cannot surpass 0.83 with the NN models.

10 References Used or Studied

Your amazing tutorial :)
https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html
https://pytorch.org/docs/stable/generated/torch.nn.GRU.html
https://optuna.org/
https://github.com/sooftware/attentions

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.RocCurveDisplay.html
https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html
https://pytorch.org/tutorials/beginner/saving_loading_models.html
https://www.statisticshowto.com/contour-plots/
https://en.wikipedia.org/wiki/Empirical_distribution_function