Τεχνητή Νοημοσύνη ΙΙ - Εργασία 3

Μυστακίδης Ιωάννης 1115201600113

How to Run:

- Firstly, you have to upload the datasets on colab (the vaccine datasets). We read the training, validation and test data in the first cell. The paths of the datasets are simply the names of the datasets themselves, but you can change them according to the path on which you upload the files yourself. You don't have to upload any word-embeddings because we download and unzip the ones we need inside colab.
- You can either:
  - Use 'Run all'

    Which would take a little more than 1 hour to finish, because of all the plotting (not advised).
  - Or **omit** some (code) cells and run the others:

    You can omit the cells that do the plotting for each hyperparameter in the 'Model Selection and Training'. That is:
    - Cell 14: which plots the metric values per epoch for each number of stacked RNNs
    - Cell 15: which plots the metric values per epoch for each number of hidden layers
    - Cell 16: which plots the metric values per epoch for type of cells
    - Cell 17: which plots the metric values per epoch for each gradient clipping value
    - Cell 18: which plots the metric values per epoch for each dropout probability

    These cells are merely for visualization purposes only, just to visualize how each hyperparameter affects our model's bias/variance and justify our choices for the various hyperparameters (although countless more experiments did take place that are not shown). Omitting these cells will significantly reduce the running time.
    You can't omit any of the other cells because they are important for the flow of the notebook's execution.

- Finally, in order to run the model on the test set you have to:
  - Make you provided the correct path to the dataset in the first cell of the notebook (the path to the test dataset is currently set to the path of the validation set, so that this notebook runs without errors, but you should change that to the correct path to the test set)
  - Run the cells in the last section of the notebook (the plots currently in that section refer to the predictions on the validation set, but they're going to change when you run them again -assuming you provided the right path to the test dataset in the first cell of the notebook-)

Insights from Experiments:

As required, we used the Adam optimizer and the cross-entropy loss function (actually, a logsoftmax layer at the end of the network and the NLLLoss function, which is the same thing) for all of our experiments. For the word-embeddings, we used the glove twitter.27B embeddings with 200 dimensions that we also used in the second assignment (although we did try out some experiments with the other dimensions too), in order to make a proper comparison between the models. The preprocessing pipeline also stayed the same as per the cleaning of the tweets.

We've experimented with the number of stacked RNNs, the number of hidden layers, type of cells, skip connections, gradient clipping and dropout probability and their effects are visually displayed on the notebook with inline plots. We've also experimented with different other hyperparameters like the learning rate of the optimizer, passing or not passing the 'weight' argument to the loss function (because of the imbalanced dataset), epochs, number of features in the hidden state of the rnn layer(s) and many more, like we did in the second exercise (because obviously the best model depends on the values of these hyperparameters as well). Some insights worth noting are the following:

- Stacking more RNNs results in an increase in the performance of our network up to a point where it starts decreasing again. Using more than a single (bidirectional) recurrent layer allows our model to fit better on the training data and our choices for the values of the other hyperparameters (e.g. dropout probability, learning rate etc.) guarantee that we don't overfit. However, stacking up too many layers has the opposite effect and makes it more difficult for our model to learn the intricacies of our data (which a big problem considering the small size of our dataset).

- A small number (e.g. 16) of hidden layers makes the two loss curves get very close to each other, which means that this model probably generalizes better, but when we look at the f1 subplot of this model, we can see that the scores we get are not ideal. Again, for this hyperparameter, we would like a value not too small and not too big either, for example 64, which results in a small bias and small variance. From the plots it looks like,

in general, increasing the number of hidden layers allows our model to fit the data better and leads to higher to smaller bias, but increasing it too much also leads to higher variance. The optimal value obviously depends on the values of the other hyperparameters.

- When it comes to the type of the cells used, the LSTM and GRU cells had pretty similar performance with the difference that GRU was quite faster in computations. The vanilla RNN cell on the hard was very prone to exploding/vanish gradients and rarely performed as well as the others in the experiments (gradient clipping was able to help with some of its problems).

- The problem of vanishing gradients is somewhat solved by the use of LSTM or GRU cells (the effect of vanishing gradients was more apparent in vanilla RNN cells). However, the problem of exploding gradients required the use of gradient clipping.

- In general, increasing the dropout probability reduces the ability of network to overfit. However, the optimal value for the dropout probability depends almost entirely on the values of the other hyperparameters and the architecture of network. A very small dropout probability would allow the network to overfit if we used a complex architecture and a large learning rate, and underfit if we did the opposite. A very large value on the other hand would not allow our network to learn sufficiently. Finding a balance between the two ends greatly depends on the values of the other hyperperameters. So if we wanted to increase the ability of the network to fit on the training data (given that the other hyperparameters are already set) we would decrease the dropout probability, whereas if we wanted to decrease this ability, we would increase the dropout probability. It all depends on where we are now, to choose where we want to go.

What we mentioned for the dropout probability is also true for almost all of the other hyperparameters. We can observe how the effect of increasing or decreasing the value of a hyperparameter affects the ability of the network to fit the training data (and therefore its later generalization), but the optimal value depends on the values of the hyperparameters. That means that we can't adjust the values of the hyperparameters in sequential way, because the optimal values of later hyperparameters will affect the optimal values of earlier hyperparameters. Finding the best values nowadays is usually done using some hyperparameter tuning method (we used simple grid search and randomized search in the first exercise), but we did the tuning manually in this exercise.

After a lot of hyperparameter tuning, our best model wasn't able to match, let alone surpass, the performance of the models in the previous assignments. As we can see in the plots in the notebook, the roc curves are close to the y=x axis which denotes a bad performance (it's as if our model makes predictions randomly). As we explained in the notebook, we believe that this is due to the tiny size of our training dataset (around 10k tweets) which is not enough to do

proper training. We believe that the reason we were able to do better training in the previous 2 exercises, even though we had the same dataset, is that the simpler models required less techniques to avoid overfitting. What we mean is that due to the great power of RNNs we had to do so much to avoid overfitting (and the small size of our training dataset didn't make that easy) where we ended up preventing our network from fitting the training data properly (but after all the hyperparameter tuning, we realized there was not other way to prevent overfitting)

Adding attention to this best model didn't increase the performance of our model as expected. We attribute this again to the very small size of our training dataset, attention made no difference because our model was already performing quite bad.

Conclusions:

In conclusion, even though Recurrent Neural Networks offer us great strength (because of the sequential nature of our data), we weren't able to take advantage of them. The tiny size of our training dataset made it impossible to match the performance that we achieved in the other two exercises (even oversampling the minority class of our imbalanced dataset doesn't solve the problem, because our training dataset's 10k examples is not enough for proper training), let alone surpass it. If we had to make a choice depending on this single dataset we would probably go with the simpler models instead of RNNs, because the size of this dataset is not enough to leverage the great strength of RNNs (to be honest, we also had a hard time with simple FFNNs, but RNNs were way more difficult to train with such a tiny dataset). This could be a lesson that data is probably more important than algorithms, cause of the current state of the art approaches cannot solve the problem of limited data. After all, our algorithms are only as good as our data.