Sentiment Analysis using Reccurent Neural Networks

Mirna Baksa
FIB - Barcelona School of Informatics
Polytechnic University of Catalonia
Email: mirna.baksa@est.fib.upc.edu

Abstract *- This report presents results obtained from experiments with Sentiment Analysis using Reccurent Neural Networks. The experiments were modeled in Keras. The objective was to see how different values of different parameters effect network performance.*

1 Introduction

Reccurent neural networks are a class of feed forward neural networks with edges that span adjacent time steps (or reccurent edges). At each time step the nodes receive input from the current data and from the previous state, which means the networks are able to remember the input they received before - therefore making them precise in predicting what input is coming next. The structure of the RNNs make them very successful in working with sequential data such as time series, financial data, audio, video, text, speech etc. This kind of data relies on deeper understanding of a sequence and its context.

With RNNs, a lot of interesting experiments can be done: text generation, sequence to sequence prediction, time series prediction as both regression and classification, translation, sentiment analysis etc.

For this report, sentiment analysis was chosen as a problem. Sentiment analysis is a process of understanding an opinion about a given subject from written or spoken language. With the always rising amount of data, sentiment analysis has become a tool for making sense of that data. This allows companies and researchers to get insights from their user feedback and automate all kinds of processes.

The experiments in this report will focus on the effect of different network parameters on its performance. We will not be trying to achieve perfect accuracy (although it is desirable and always welcome), but rather trying to get a better insight and experience with tuning network hyperparameters.

2 Dataset

The dataset chosen for the experiments in this report is the Amazon reviews: Kindle Store Category [1].

The dataset consists of a small subset of product reviews from Amazon Kindle Store category from May 1996 to July 2014. The dataset consists of a total of 982 619 entries. Each reviewer has at least 5 reviews and each product has at least 5 reviews in this dataset. The dataset comes in a .csv format consisting of these columns:

asin - ID of the product, like B000FA64PK
helpful - helpfulness rating of the review - example: 2/3.
overall - rating of the product.
reviewText - text of the review (heading).
reviewTime - time of the review (raw).
reviewerID - ID of the reviewer, like A3SPTOKDG7WBLN
reviewerName - name of the reviewer.
summary - summary of the review (description).
unixReviewTime - unix timestamp.

2.1 Preprocessing

For sentiment analysis not all columns from the dataset are needed, so for further analysis only columns **reviewText**, **overall** and **helpful** will be used. The **overall** column is the users' rating from 1 to 5, so that will be used as to map the user's sentiment.

At first the reviews are cleaned and only ASCII characters in the text are kept. Also, english stopwords were removed from the reviews. Stopwords are com-

monly used words in a language, e.g. for english some of the stopwords would be: *the, a, I, it, what, that, but* etc. Those are the words that are used to form sentences, but do not carry any special meaning context-wise. The list of stopwords was taken from the NLTK platform - the link can be found in the bibliography [2]. The cleaned review text is stored to a separate column.

Next, the users overall rating is transformed to sentiment. Since the reviews are rated on a 1 to 5 scale, the ratings were mapped to positive/negative/neutral as follows: review grades 1 and 2 are mapped to *negative*, grade 3 is mapped to *neutral* and grades 4 and 5 are mapped to *positive*.

Then, a corpus is formed from all the words in all the reviews. Only words that occur more than 200 times in the whole corpus of words are considered relevant and included in the training. Empty reviews (of length 0) are removed from the dataset. In order to save some memory and time, the review lengths will be cut - the mean of all review lenghts is calculated and all the reviews cut to that length, keeping the most frequent words in the review representation. This cut was done in order to eliminate outliers. Imagine we took the maximum length of the review as the cut size and one review was of length 1500 words, while the others were of sizes 200 words or smaller. If we represented each review with length 1500, we would have a lot of meaningless data which would slow down the training without being useful. This is the reason of taking the mean of lengths, hoping the reviews would be more representative. One other statistic that could have been used is the median which is more robust to outliers, but in our case the mean was greater than the median (the difference was about 25 words in favour to the mean) so we used the mean in order to include more words in our analysis.

Since there is a limit of memory that can be used, it was impossible to work with the whole dataset which consists of almost a million entries. A small subset of the dataset was chosen for analysis - the dataset was sorted according to helpfulness (which is a column included in the dataset) and then grouped by sentiment. Top 5000 rows from each class was taken into the dataset. This way it was enssured that we do not have disbalanced classes (in terms of dimensionality) and hopefully we get the most meaningful reviews to train the network on.

Possible problems we could have with this dataset is simply human nature - the reviews may not be concise, spelled correctly or can be simply not very useful. This is why they were ranked by helpfulness in order to get the most significant reviews for each class, but since the "helpful" column is also user feedback this should be taken with a grain of salt.

After the preprocessing the dataset is split in train, validation and test sets in ratio 80:10:10.

2.2 Examples

Some examples of reviews are shown in table 1, one review for each overall grade.

Table 1: Examples of reviews

Review text	Rating
I love read the Washington Post in my Kindle Reader because it's very good and important for everyone. I recommend it!	5
I enjoyed this one tho I'm not sure why it's called An Amy Brewster Mystery as she's not in it very much. It was clean, well written and the characters well drawn.	4
Did not impress me that much, as I have read the one written by The Yogaville founder (which was detailed and informative).	3
I give this 2 stars because I have read worse books. This book is very boring. Very "yadda, yadda" type. I don't recommend it.	2
It was too slow from the very beginning! To much information about a number of charcters at the very beginning which turned out to be quite boring!	1

2.3 Code structure

Code used in the experiments can be found at a GitHub repository [3].

The repository consists of a file in which the dataset filtering was done and a file in which the dataset is preprocessed, the model is defined and trained.

1. **filter.py**
   This file filters the dataset as described in section 2.1.
2. **rnn.py** This file loads the previously filtered dataset, preprocesses it in a way described in section 2.1, prepares the data for training, defines the model parameters, starts the training and at last calculates some metrics and draws graphs. The parameters were changed accordingly for each experiment.
   Since 95% of the code in this file is reused for each experiment, in the GitHub repo there will be only this file - with all the different parameter values commented out.

3 Experiments

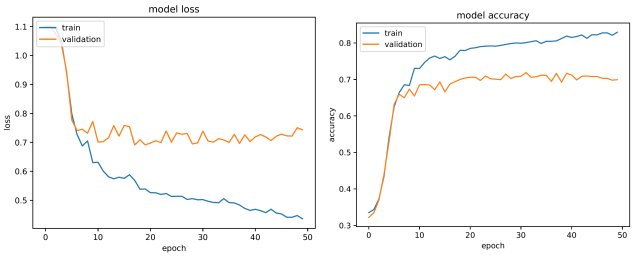
As mentioned before, the experiments will try to show the effect of changing one network parameter at a time. In each experiment one parameter will be varied, while others are fixed. Default network values are in table 2, and in each experiment it is defined which of these will change and to what values. Default values are generally important since they will also effect performance, but what matters here is the relative change in performance from one parameter value to the next.

Table 2: Default parameter values

parameter	default value
optimizer	SGD
learning rate	0.01
number of layers	1
number of neurons	128
learning rate	0.01
activation	softmax
dropout	0
batch size	64
network type	LSTM

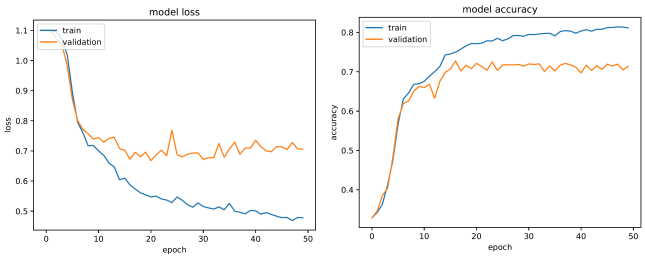
3.1 Experiment #1 - The number of network layers

In the first experiment the code was ran three times, each with different number of network layers - 1, 2 or 3 layers. Results can be seen in figures 1, 2 and 3.



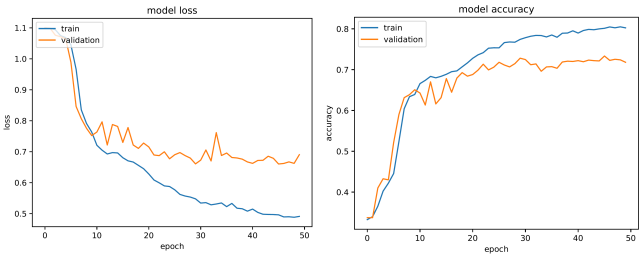
(a) Loss (b) Accuracy

Fig. 1: 1 layer



(a) Loss (b) Accuracy

Fig. 2: 2 layers



(a) Loss (b) Accuracy

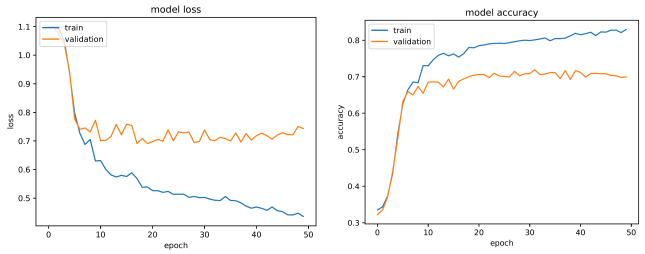
Fig. 3: 3 layers

None of the runs achieved spectacular changes. The biggest difference was the time it took for 50 epochs to finish which was 15 minutes for the 1 layer, 34 for the 2 layers and 54 minutes for 3 layers. From this we see that adding more layers (with default dimensionality) while keeping the other parameters fixed does

not result in better performance, while it adds to the time needed to finish the 50 epochs. Keep in mind that the layers are of relatively small dimensionality (128 neurons per layers), so this also contributes to the results.

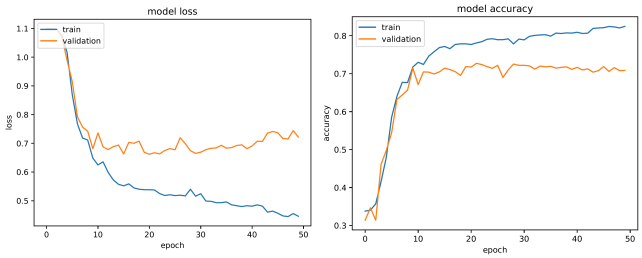
3.2 Experiment #2 - Number of neurons in a layer

This experiment was also run three times, with 128, 256 and 512 number of neurons the network with one layer. The last experiment (with 512 neurons) could be ran only for 35 epochs due to the time limit of 1 hour in the server (this is for the debug queue, seeing that my jobs were blocked for a long time in the training queue). Results can be seen in figures 4, 5 and 6.



(a) Loss (b) Accuracy

Fig. 4: 128 neurons

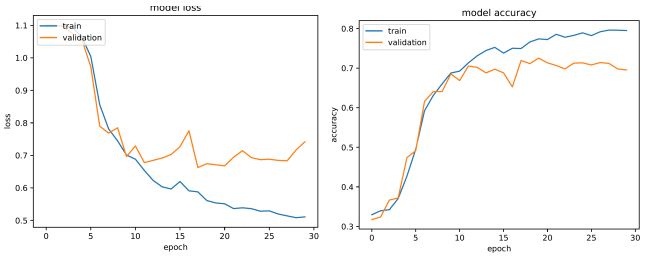


(a) Loss (b) Accuracy

Fig. 5: 256 neurons

The results are similar: the performance does not get significantly better with more neurons in the single layer. Probably the best combination would be more layers (as in experiment 1) and more neurons per layer, but the point of this experiment was to see if adding more neurons while keeping other parameters fixed would have an impact.

Similar to the previous experiment, adding more neurons results in more time needed to run 50 epochs. Performance-wise it looks as if adding neurons does not



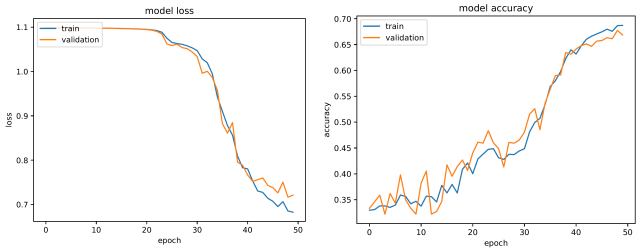
(a) Loss (b) Accuracy

Fig. 6: 512 neurons

have a positive effect, but it is hard to tell since adding any more than 256 neurons resulted in training time longer than 1 hour which is the servers time limit for the queue I was using.

3.3 Experiment #3 - Learning rate

This experiment was run with three different learning rates of 0.001, 0.01 and 0.1. Results can be seen in figures 7, 8 and 9. The results of this experiment are a bit more representative than the previous ones.



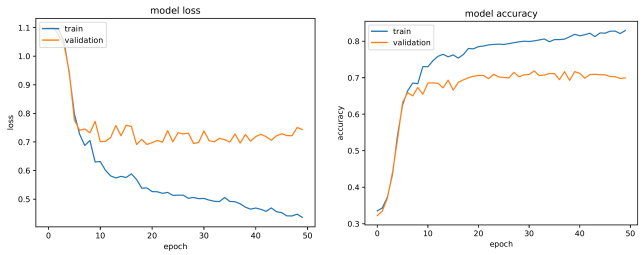
(a) Loss (b) Accuracy

Fig. 7: Learning rate 0.001

With learning rate of 0.001 the convergence is much slower, which was expected. The validation loss and accuracy curves follow the training curves, but for 50 epochs there is not enough information to draw valid conclusions about the networks final accuracy. The network would probably diverge after some more training time, but what is important is the effect of a small learning rate on performance - since the optimizer is taking small steps towards gradient, the convergence is slow.

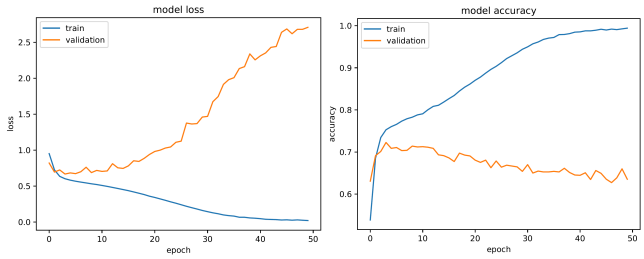
The result for learning rate of 0.01 is the same as in figure 4. This learning rate is probably right for this problem, but the network underperforms because other parameters need tuning.

The learning rate value of 0.1 was obviously too big since the network validation loss started diverging after the 10th epoch. The risk of having a too big learning



(a) Loss (b) Accuracy

Fig. 8: Learning rate 0.01



(a) Loss (b) Accuracy

Fig. 9: Learning rate 0.1

rate is that we take big steps in direction of the gradient and it may happen that we overshoot the minimum. It may fail to converge, or even start diverging.

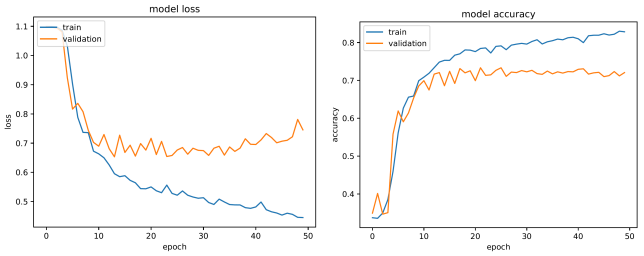
The goal of this experiment was to see what effect the choice of the learning rate value has on the network behaviour. The results show expected behaviour - extreme (too small or too big) values will have bad effects on performance so the choice should be made carefully.

Getting the learning rate right would mean less time to train a more accurate model, but this is often not an easy task. There are some methods and optimizers in which the learning rate can change during the training (adaptive learning rate), which can give us a more robust network.

3.4 Experiment #4 - Optimizers

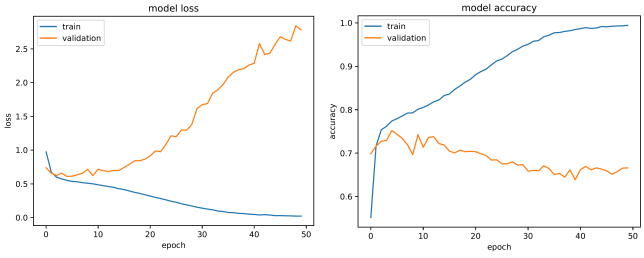
This experiment was run with all available optimizers available in Keras: SGD, RMSProp, Adagrad, Adadelta, Adam, Adamax and Nadam, but only the most representative results are shown: SGD 10, RMSProp 11, Adagrad 12 and Adadelta 13. All optimizers are ran with default values defined by Keras, since in the documentation they recommend to leave all parameters as defaults (except maybe for the learning rates which can be tuned freely).

Out of all the optimizers, SGD had the best performance, while the RMSProp diverged earliest. The reason for this could be that the default Keras parameters do not work well with RMSProp, since in the documentation it states that RMSProp usually works well with RNNs. This is a good use case for doing a hyperparameter search in order to determine which parameters work best for this problem.



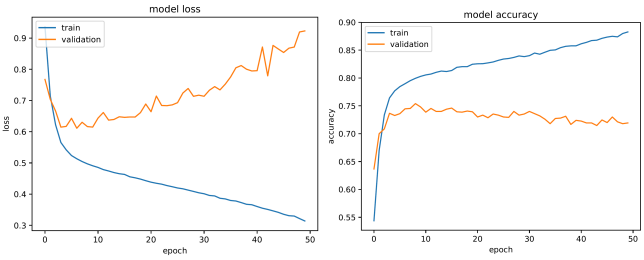
(a) Loss (b) Accuracy

Fig. 10: SGD



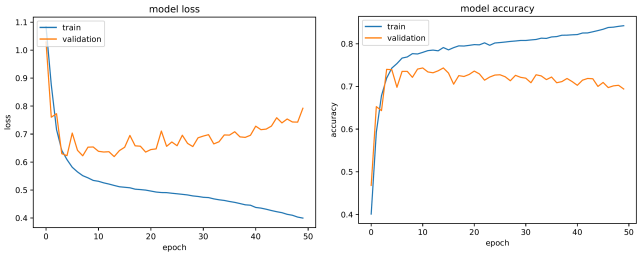
(a) Loss (b) Accuracy

Fig. 11: RMSProp



(a) Loss (b) Accuracy

Fig. 12: Adagrad



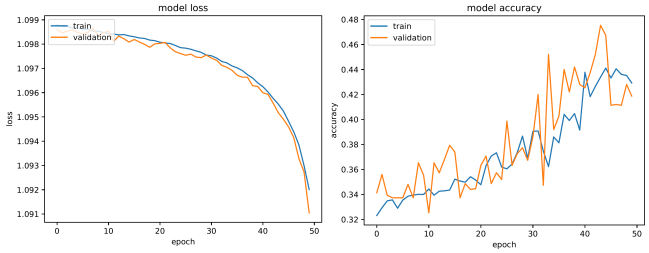
(a) Loss (b) Accuracy

Fig. 13: Adadelta

Adagrad and Adadelta are methods with parameter-specific learning lates which are adapted relative to how frequently a parameter gets updated during training, Adadelta being the more robust version of Adagrad. They both had similar performance, not diverging as early as RMSProp, but still having poor loss. Adagrad loss grew more rapidly in comparison to Adadelta. Again, this are adaptive methods but only for the learning rate - other parameters were fixed and should be tuned accordingly to our problem.

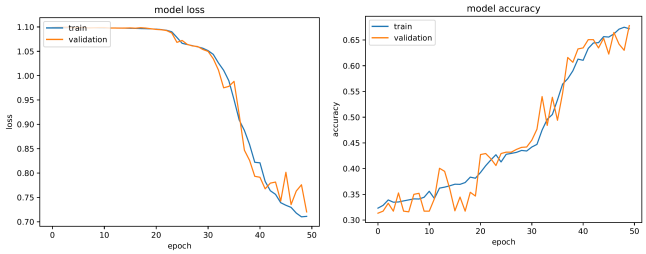
3.5 Experiment #5 - Momentum

Since SGD had the best results in the previous experiment, this last experiment tested the effect of momentum on SGD. Momentum is a very popular technique used along with SGD. It can be used to obtain faster convergence - it helps build up velocity in any direction with constant gradient descent and prevent oscillations. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go in. In this experiment we ran the training with extreme values of 0 and 1, and middle values of 0.5 and 0.95. Results can be seen in figures 14, 15, 16, 17.



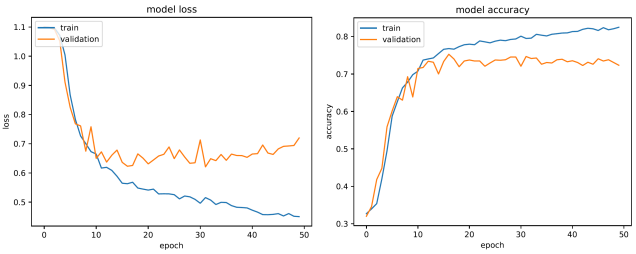
(a) Loss (b) Accuracy

Fig. 14: No momentum



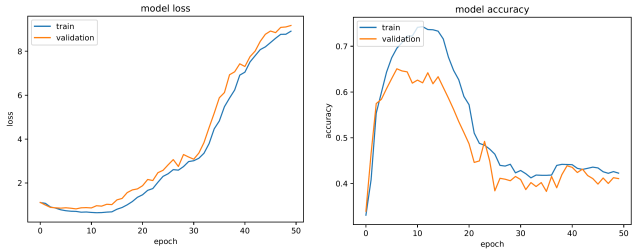
(a) Loss (b) Accuracy

Fig. 15: Momentum 0.5



(a) Loss (b) Accuracy

Fig. 16: Momentum 0.95



(a) Loss (b) Accuracy

Fig. 17: Momentum 1

In this experiment the situation is similar to the experiment #3 with values of learning rates. Extreme values of the momentum (0, 1) result in a poor performance - without momentum, the convergence is extremely slow and reaches only 10% growth in accuracy on the validation set in 50 epochs. With momentum 1, both training and validation losses diverge since the momentum probably accelerated the gradient descent in the wrong direction.

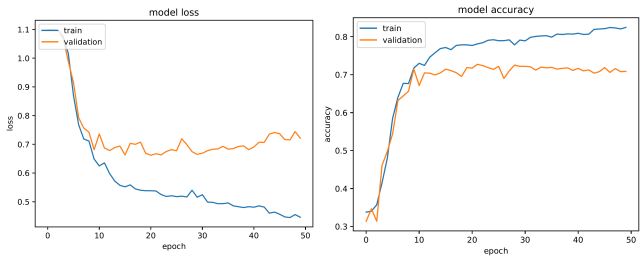
The momentum value has to be chosen carefully and, if possible, multiple values should be tried out to see which momentum works best for the given problem.

3.6 Experiment #6 - Dropout

Dropout is a regularization method which is used to reduce overfitting in neural networks. The concept is simple - given a dropout rate p (a percentage), an individual unit (neuron) will be ignored (dropped) from the network during training with the probability of $1\text{-}p$ or kept with the probability p.

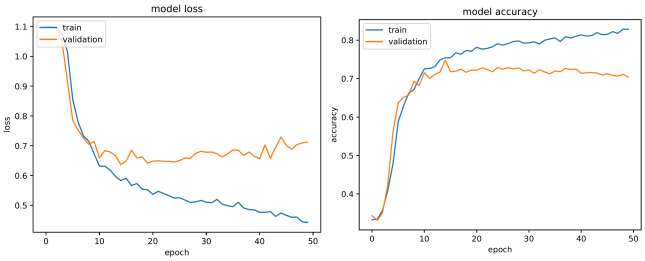
Since dropout is used to stop the network from overfitting, it would be interesting to see what effect it has on the network. This experiment was run with four different dropout rates: 0, 0.5 (most common rate), 0.7 and the extreme value of 1. This experiment was conducted with 256 neurons in one layer in hope to see the

effect better. The results are shown in figures 18, 19, 20, 21.



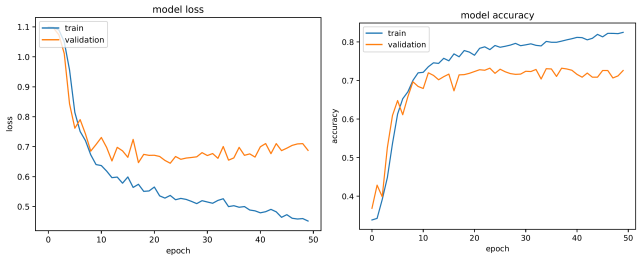
(a) Loss (b) Accuracy

Fig. 18: Dropout 0



(a) Loss (b) Accuracy

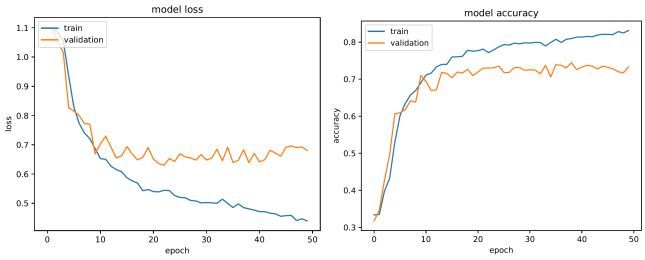
Fig. 19: Dropout 0.5



(a) Loss (b) Accuracy

Fig. 20: Dropout 0.7

Even though the results are not as visible as they would be if we trained a more complex model for more epochs, there are some subtle but interesting things to notice from the graphs. It seems that with applying more dropout the validation plots look less smooth - this is logical since we are basically ”plugging out” some neurons out of the network. Also, with the extreme value of the dropout (1) it seems that the validation loss dropped more than with other dropout rates.



(a) Loss (b) Accuracy

Fig. 21: Dropout 1

With lower rates (0, 0.5) the loss started to rise after some time (cca. 35th epoch), while with bigger rates it seems to stay stable (not dropping, but also not getting much bigger).

4 Discussion

What these experiments did is actually similar to a ”decomposed” grid search. Grid search is a brute force method to determine which parameters work best for a certain problem. We define which parameters to explore, give each parameter a range of values to choose from and grid search will take all the combinations of all the parameters and apply each of them to a model. The goal is to train each of these models and then evaluate them in some way and select which one performs best. Of course, for a large number of parameters and a large number of values to choose from there are smarter ways (but also harder to implement) to do so.

In this report, we did a similar thing by varying one parameter at a time to see which one achieves best results. Grid search was tried to be ran (using the scikit-learn library), but it exceeded the memory limits on the server even with a very small subset of parameters. This was unfortunate because it would be interesting to see which model performs best and if there is something we missed in this report.

5 Extensions

There are some other parameters that could have been analysed - e.g. batch size, the type of the Reccurent Neural Network (GRU vs LSTM), number of words used for review representation etc.

Furthermore, the most interesting extension in my opinion would be trying to tune RMSProp to this problem. The Keras documentation states that this optimizer works well with RNNs, but the results in experiment #4 did not show that, probably because the default RMSProp parameters were not appropriate for our dataset.

Lastly, a lot of data from this dataset was wasted - we used only 15 000 reviews while the dataset consisted of almost a million reviews. This is unfortunate - more data is always better and we would be able to analyse the results with more certainty if we used more reviews.

6 Conclusion

This report conducted 6 experiments, each varying one parameter at a time. Each experiment was ran with a few different parameter values. For each experiment and each parameter value the loss and accuracy graphs were drawn and results analysed.

We could see the different effects of different parameter values on performance, some more prominent than others. The most successful (in terms of changes seen with different values) were experiments with learning rates, optimizers and SGD momentum.

References

[1] Amazon Reviews: Kindle Store Category, https://www.kaggle.com/bharadwaj6/kindle-reviews

[2] NLTK Stopwords Corpus, https://www.nltk.org/nltk_data/

[3] Code repository on Github, https://github.com/mirnabaksa/Deep-Learning