DL 2424 PROJECT:
PREDICTION OF HELPFULNESS AND RATING IN AMAZON REVIEWS

Jon Arrizabalaga, Linus Persson, Natalija Zivanovic

1 Brief description

Amazon is one of the most popular platforms for virtual shopping where the exchange of a huge variety of products takes place. Due to the fact that the consumers are not able to physically access the product before buying it, Amazon needs to provide some kind of reliability proof. To do so, Amazon allows the users not only to rate the product, but also to write reviews. Moreover, a user can assign if a specific review has been helpful. The helpfulness of the reviews can be predicted by applying Deep Learning methods.

Different models that predict the helpfulness of Amazon reviews are exposed. A very simple RNN will be the starting point. Several improvements and trials are going to be built upon this model. Even if every method will be an attempt to improve the performance of the previous one, all models are strongly related. Except the first model, the core of all the other networks will be a LSTM. Apart from discussing different network layouts and their respective results, the relevance of pre-processing data is going to be explained. This latter analysis will expose not only filtering and balancing, but also word embedding methods, such as GloVe and Word2Vec.

Due to the computational cost required to train deep neural networks, a laptop equipped with a 4 core intel i7 processor, a NVIDIA GTX 950M GPU and 8GB of RAM has been used. Pytorch and Python have been the basic software platforms. Out of all the packages, torchtext (for data processing) and gensim (for word embedding) played a significant role. The scripts developed during this project are available in Github [1].

2 Related work

There has been a surprising amount of research related to text analysis. When it comes to review recognition, most of the work has focused on sensitivity. Due to the subjectiveness that involves evaluating helpfulness, labeling becomes less partial, which results in an increase of the task's complexity. Even if helpfulness has been studied much less, some related papers([2], [3], [4]) have been used as guidance.

It needs to be highlighted that the tutorial and explanations provided in [5], have been a great help. These tutorials have eased the implementation in Pytorch. Since the goal of these scripts is to predict the sensitivity of movie reviews, the data and the network's purpose are different. These files have been a good basis, but several modifications and upgrades have been introduced along the project. Moreover, we would like to express our gratitude to Benjamin Trevett, author of this tutorial. Not only has he helped with issues related to code development, but he has also provided some interesting recommendations regarding neural networks.

3 Data

There is a considerable amount of published data to work on. Paper [3] and several others propose to work with the following database: *https://www.kaggle.com/snap/amazon-fine-food-reviews*. Due to its popularity, this data set was chosen for the project. It consists of 568400 fine food product reviews available in Amazon. Each item has got its respective ID, rating, helpfulness, review title and review description:

1. **Identification of product and consumer:** *UserId* and *ProfileName* identifies the user, while *ProductId* recognizes the product.
2. **Helpfulness:** The helpfulness is distributed into two fields. On the one hand, *HelpfulnessDenominator* represents the total amount of people that voted for that specific review. On the other hand, *HelpfulnessNumerator* reflects the people that considered the review to be helpful.

3. **Rating:** The field named *Score* is the rating of the product given by the author of the review.
4. **Review information:** The information posted by the review author is separated into three fields. The first one is the moment when the review is published (*Time*), the second one is the review's title (*Summary*), and the last one is the review itself (*Text*).

3.1 Labeling

Since the goal of this project was to predict the helpfulness of the reviews, the *helpfulness ratio* was used as a label. In the previous section, it has already been explained that there are two fields that represent the helpfulness of the product: the total amount of votes (*HelpfulnessDenominator*) and the ones that have found the review to be helpful (*HelpfulnessNumerator*). Hence, there was not a parameter that provided the absolute helpfulness of the review. To do so, the helpful votes were divided by the total amount of votes. If the resulting number was bigger than 0.5, the review was considered to be helpful (labelled as a '1'). In the other case the review was classified as not helpful (labelled as a '0'). This is exactly what the following formula represents:

$$Votes_{Helpful}/Votes_{Total} > 0.5 = Helpful \rightarrow 1$$

$$Votes_{Helpful}/Votes_{Total} < 0.5 = Unhelpful \rightarrow 0$$

3.2 Filtering

Filtering data had a big impact on the upcoming analysis. Refering to paper [2], [3] and [4], there are different criteria for filtering and it is difficult to say which is the correct one. Taking into account that the data set contains 568400 reviews, all of them do not contribute in the same way. For example, some do not have a helpfulness rating and others only have one vote. These cases did not provide any valuable information to the network.

It is impossible to classify a review without any votes. It might either be helpful or unhelpful. Some papers propose to take this into account by adding two units to the total amount of votes and one unit to the positive votes. Consequently, the helpfulness ratio of a review with zero votes would be 0.5. Since the criteria to consider a review helpful was that the ratio had to be bigger than 0.5, all the reviews that have 0 votes was considered unhelpful. This was not fair, because an unrated review can be helpful. Therefore, reviews with no votes were neglected. The main difficulty of this task was that evaluating the helpfulness of a text was not completely objective. There might be a specific review that is helpful for someone, while it might seem nonsense to another one. In order to minimize this, reviews with a single vote were not considered. The higher the threshold of minimum quantity of votes, the higher the quality of the data. Nevertheless, a high threshold requires having large amounts of data. Even if it seemed like initially there were lots of available reviews, after filtering only 185 028 reviews were left. Taken into account that this data had to be balanced, it was clear there were no spare data.

3.3 Balancing

An unbalanced data set is a data set that has too much data of one specific class. This can lead to the classifier being overwhelmed by the large classes and ignoring the small classes [6]. Out of the filtered reviews, 61% were helpful reviews. This means that 61% accuracy could be achieved by just labelling every example to class zero. Taking into account that neural networks have difficulties to handle unbalanced data, the network's learning capacity would be considerably reduced. Therefore, it was necessary to balance the data set before being fed into the classifier for training. The balancing was done by keeping all the non helpful reviews (nearly 48000) and reducing the helpful reviews, so that a 50/50 relation was reached. This resulted in a total amount of 98474 reviews. 20% of these reviews were defined as validation data. The remaining 80% was split into training and test data with relation 80/20.

4 Word embedding

Word embedding is a method to map words sentences to vectors. With this embedding, words get a context in relation to other words, capturing the meaning. This is often used as pre-processing for neural networks doing natural language processing. Two common methods are Word2Vec or its extended version GloVe (Global Vectors). [7]

For this project, word embeddings have been used when building the dictionary. This way, the weights of the embedding layer were not initialized randomly and the network did not need to learn them from scratch. A pre-trained GloVe model was compared to a self-trained Word2Vec model. Both were implemented when building the dictionary by using the torchtext package.

4.1 Pre-trained - GloVe

Torchtext package provides the option of including some pre-trained vectors that are available online ([8] and [9]). When choosing the pre-trained vector, it needs to be taken into account that the vector's embedding dimension and the embedding dimension of the network need to match. *glove.6B.50d* or *glove.6B.100d* was used in this project, depending on the embedding dimension of the network being trained.

4.2 Self-trained - Word2Vec

With the purpose of increasing the performance of the network, two word embedding vectors have been built and trained by using all the reviews (without being filtered nor balanced). To do so, Word2Vec from gensim module ([10]) was used. During this process, online literature played a key role ([11] and [12]). For the sake of simplicity, an independent script was used to develop the modules that generate the trained vectors. Before building the model, some word pre-processing had to be done. Firstly, all the non-letter characters were removed and all characters were converted to lowercase. Afterwards, all the line-breaks and stops were substituted by stop or linking words. When building the model, the following parameters were considered:

- *Size = 50 or 300:* Dimensionality of word vectors.
- *Window = 7:* Maximum words between the current and predicted word.
- *Min_count = 3:* Minimum amount of times that a word needs to appear.

Two different models (dimensionality 50 and 300) have been trained for 10 epochs. Since the dimension of the trained embedded vectors and the network's embedding size need to match, it is interesting to have two models available with different dimensions. After training both models, they have been saved to their respective files (*w2v_reviews_50.text* and *w2v_reviews_300.text*).

The integration of the self-trained Word2Vec models also takes place through torchtext. In this case, the implementation was not straight forward and extra code had to be written. Online literature ([13]) was crucial to achieve this task. In the left part of Figure 1 a poster representing the layout of the word vectors for the embedding dimensions of 300 is shown. Thanks to the amplified version at the right, it can be seen that the related words tend to be close to each other.

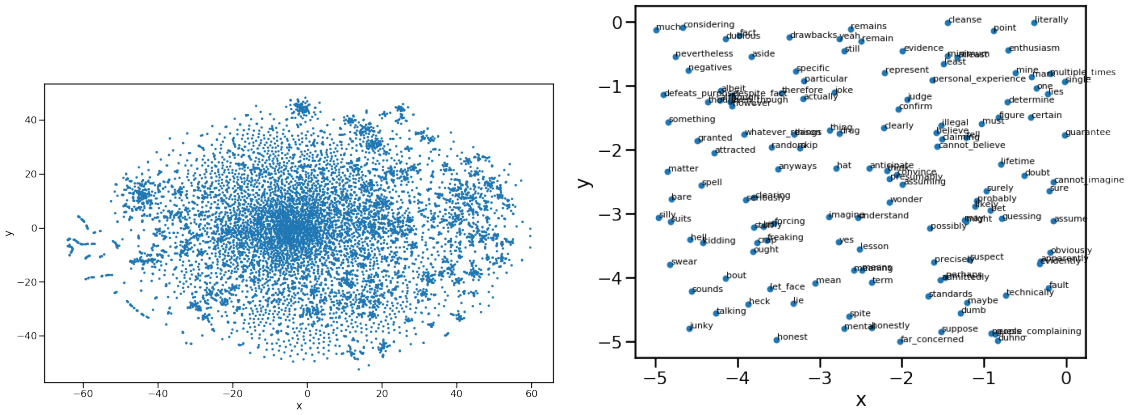

Figure 1: (Left) Poster of a self-trained word vector developed with Word2Vec and an embedding size of 300. (Right) Zoom of the word vector, where the proportion between semantic meaning and geometrical distance is visible.

5 Helpfulness prediction

The helpfulness prediction was the main task of the project. As mentioned above, different models that tackle this problem have been built. These models were a consequence of an ordered set of improvements. In this section, these models and their evolution are going to be developed and explained. An analysis of the obtained results will also take place.

5.1 Simple RNN

The first network was a simple Recurrent Neural Network (RNN). A RNN is an artificial neural network that has sequence of words as an input, in this case product reviews, and produces a hidden state, h, for each word. To produce the next hidden state, the current word is fed into the network together with the hidden state from previous word:

$$h_t = RNN(x_t, h_{t-1})$$

The last hidden state h_T is introduced in a linear function (f), so that it provides the prediction (y) as an ouput:

$$y = f(h_T)$$

When it comes to pytorch implementation, the network was designed so that it goes through three steps. Initially, words were represented through large one hot vectors. The embedding layer reduced these vector, these vectors were reduced in size. The reduction took place down to a dimension called embedded dimension (em_{dim}). The embedded word representation was introduced into the RNN network. Pytorch already offers an inbuilt function (*self.RNN*), whose input is the embedded word representation and return the last hidden layer. Beforehand, the dimension of the hidden layers had to be defined (hid_{dim}). As mentioned above, the last hidden layer was sent to the linear function, which answered with a prediction of the helpfulness. For this network, a batch-size of 32 was chosen along with hid_{dim}=100 and em_{dim}=50. With an input dimension equal to the size of the vocabulary (91 383 tokens), an output dimension of one and the previously mentioned embedded and hidden dimensions, the network had 4,584,451 trainable parameters. *Adam* was chosen as an optimizer. According to literature, this adaptative learning rate optimization algorithm is the most versatile when training deep neural networks [14]. In order to calculate the loss *BCEWithLogitsLoss* was picked up. This criterion calculates sigmoid and binary cross entropy steps. The performance of this simple network can be seen in Figure 2.

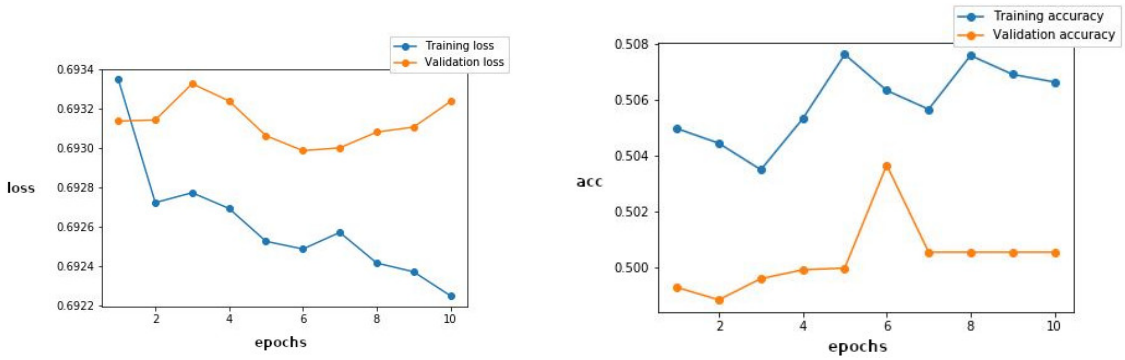

Figure 2: Loss and accuracy functions for a simple RNN with an input size of 91383, ouptut size of 1, 1 hidden layer with a dimension of 100 and an embedding size of 50.

It is visible that the network was not learning at all. On the one hand, accuracies did not increase (they oscillated around 50%). On the other hand, the loss did not have a clear trend and it did not seem to decrease . A test accuracy of 50.4 % was achieved with this network.

5.2 Simple LSTM

The first improvement consisted on converting the previous RNN network into LSTM. This upgrade was mandatory, because handling RNNs involves the risk of falling into the *vanishing gradient problem* [15]. This occurs when the gradient is so small that weights cannot be updated. The theory behind LSTM is extremely similar to the previously explained RNN, except for the fact that the output of the *self.LSTM* function is not only the hidden layer (h_t), but also the cell (c_t). This new term is an input for calculating the following hidden layer and cell:

$$(h_t, c_t) = LSTM(x_t, h_{t-1}, c_{t-1})$$

Nevertheless, the last cell is not sent to the linear function. Hence, the last hidden layer remains being the single input when predicting the helpfulness in the linear function. In other words, the expression to predict the helpfulness through the last hidden state remains the same.

Since the unique improvement was to change the type of network, there were not any new parameters. To be consistent and have a feeling of the modification's impact, the dimensions were not modified. Due to the fact that applying a LSTM network adds a parameter (c_t), the amount of trainable parameters was a little higher (4,628,701). The results can be seen in Figure 3.

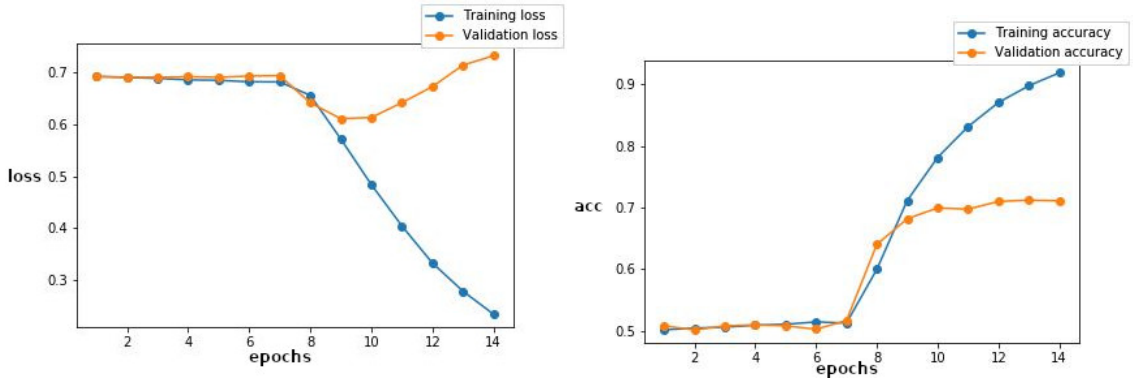

Figure 3: Loss and accuracy functions for a simple LSTM with an input size of 91383, ouptut size of 1, 1 hidden layer with a dimension of 100 and an embedding size of 50.

Comparing to the last network, there is no doubt that the performance improved. Even if the network needed eight epochs to start learning, both training and validation accuracy increased after the seventh epoch. The validation accuracy rose to 70% within five epochs (from 7^{th} to 12^{th} epoch). The training accuracy also started to increase after the 7^{th} epoch. However, it continued with the same trend after the 12^{th} epoch. This behaviour was also reflected on the loss function. The positive side of this network was that it seemed to have learning capability. Unfortunately, a new problem related to over-fitting arose. The test accuracy of this network was 68.56%.

5.3 LSTM with Dropout

The most common solution to overcome over-fitting is to apply a method called *dropout*. Dropout consists of randomly dropping units (along with their connections) from the neural network during training([16]). Pytorch has an inbuilt function which randomly zeroes some of the elements of the input tensor. The probability of canceling elements is the input of the function. In this network, the dropout was applied twice inside the forward function. Firstly, at the output of the embedding function and, secondly, before returning the last hidden layer. This way, the network was obliged to learn the parameters without having all the neurons available. This resulted in a more robust network, which was less prone to over-fit.

For this particular case, the dimensions were the same as above. As a starting point a dropout rate of 0.2 was chosen. A rate close to 0 resulted in a small influence of the dropout, while a rate close to 1 made the model more reliable but harder to train. Figure 4 shows the performance of this network.

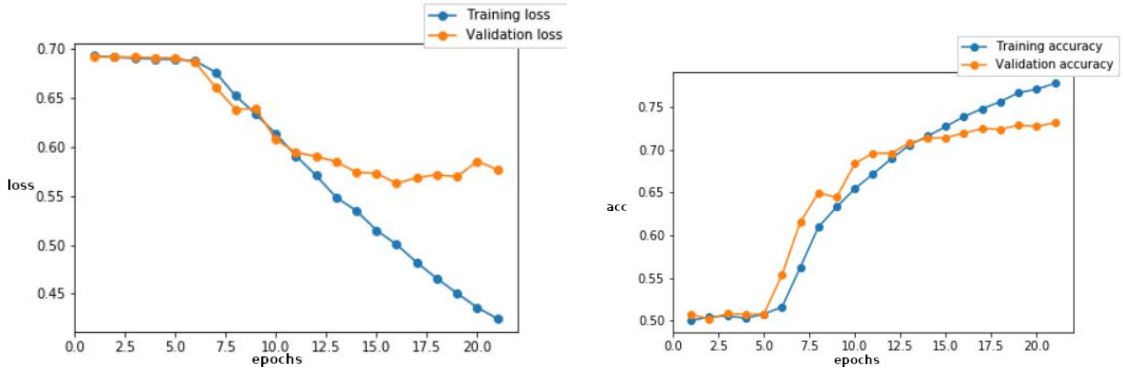

Figure 4: Loss and accuracy functions for an LSTM including dropout. Input size of 91383, ouptut size of 1, 1 hidden layer with a dimension of 100, an embedding size of 50 and a dropout rate of 0.2.

At the first glance, the results from this model were very close to results from the previous model's. The training accuracy seemed to increase, while the validation accuracy became constant after epoch 13 at an approximate value of 70%. However, it is obvious that over-fitting was much less accentuated than in the last model. For the previous model, when 14^{th} epoch was approached, the training accuracy was 90%, while in this model it was still below the validation accuracy (69%). This was a clear sign that the expected impact occurred. The test accuracy obtained with this network was 71.7%.

5.4 Upgraded LSTM

Decent results were obtained with the last model. However, over-fitting still was an issue and better results could be achieved if some improvements were introduced. Thanks to the inbuilt functions of Pytorch and many packages that can be imported, there is a sea of opportunities when it comes to network upgrades. In this section, some features used in the tutorials [5] were tested. Other improvements, such as trained word embeddings and L2 regularization are also included. All these features are explained in the following list:

- **Bidirectional:** While a unidirectional LSTM only processes previous words, a bidirectional LSTM also takes into account the upcoming ones. It consists of having two RNNs (forward and backward) operating simultaneously. One of them will provide information from the past (forward), while the other will refer to the future (backward). In terms of code implementation, this involves three modifications. Firstly, the bidirectional input variable needs to be set to "True" in the *self.LSTM*. Secondly, when specifying the dimension of the hidden layer in the linear function, it needs to be multiplied by two. Thirdly, the forward and backward hidden layers need to be concatenated before being returned by the forward function.
- **Multiple hidden layers:** In RNN networks there is a possibility to work with multiple hidden layers. In such a case, the output of the lower layers, will be the input of the upper one. The input of the linear function will be the final hidden state of the final layer. Referring to papers [2], [3] and [4] and other literature, it seems 1 to 3 layers is the most suitable amount for sentiment and helpfulness analysis in reviews. Regarding code modifications, the number of layers needs to be an input for *self.LSTM*. Apart from that, only the hidden state of the last layer needs to be returned from the forward function.
- **Packed padded sequences:** This feature does not improve the results, but it makes computation faster. It is specially interesting when the input to the network varies in size, because it accelerates the training process by omiting unnecessary computation.
- **Trained word vectors:** This upgrade consisted on the implementation of the previously explained word embeddings. As mentioned before, two different embedding methods have been considered. The first is a pre-trained GloVe vector, while the second is a self-trained Word2Vec model.
- **L2 regularization:** Regularization is a method that decreases the complexity of the model by penalizing the loss function [17]. It is a typical technique when overfitting occurs. According to [17] there are two main types of regularizations. The first one is dennoted as L1 regularization (or Lasso), while the second as L2 regularization (or Ridge). The equation of L2 regularization can be seen in the second term of the following expression:

$$L(x, y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda * \sum_{i=1}^{n} |\theta_i|$$

  Pytorch provides the opportunity to apply L2 regularization by providing a learning rate value to the *weight_decay* input. Before moving onwards it is worth clarifying that L2 regularization and weight decay are not equal terms according to paper [18]. This paper states that the similarity between both terms depends on the optimizer that is used. However, the documentation of Pytorch attributes the input *weight_decay* to the desired learning rate for applying L2 through Adam optimizer.

In this last chapter, the upgrades that were explained above are analyzed . Since we had two different trained word embedding vectors, two models are observed and compared. Firstly, the model containing *glove.6B.50d* pre-trained word vector is covered. This network had an input dimension of 91291, an output size of 1, embedded dimension of 50, hidden layer dimension of 100, 2 hidden layers, dropout rate of 0.9 and a learning rate of 0.0001. The choice of these parameters was a consequence of an extensive research and iterative trials. On the one hand, paper [3] was a very good guidance to select the amount of layers and network dimensions. On the other hand, the regularization term was chosen through an iterative process. A value of 0.00009 resulted in overfitting, while 0.0002 lowered the validation accuracy. Due to the memory limitation of the GPU, a batch-size of 16 had to be chosen. This network contained 4,927,951 trainable parameters. The performance of this network can be seen in Figure 5.

In both graphs it is visible that training and validation curves were very close to each other during the entire training. A conclusion that no overfitting takes place can be drawn. Once the training was approaching epoch 35, both accuracies and losses remained constant. This network achieved a test accuracy of 73.1%.

Finally, it was time to see how well did the self-trained Word2Vec models (*w2v_reviews_50.text* and *w2v_reviews_300.text*) perform, when combined with a completely upgraded model. Since the text has been pre-processed during the construction of the embedded word vector model, the size of the vocabulary, and, hence, the input of the layer was smaller than for the GloVe case (it achieved a size of 49131). In order to be able to compare both

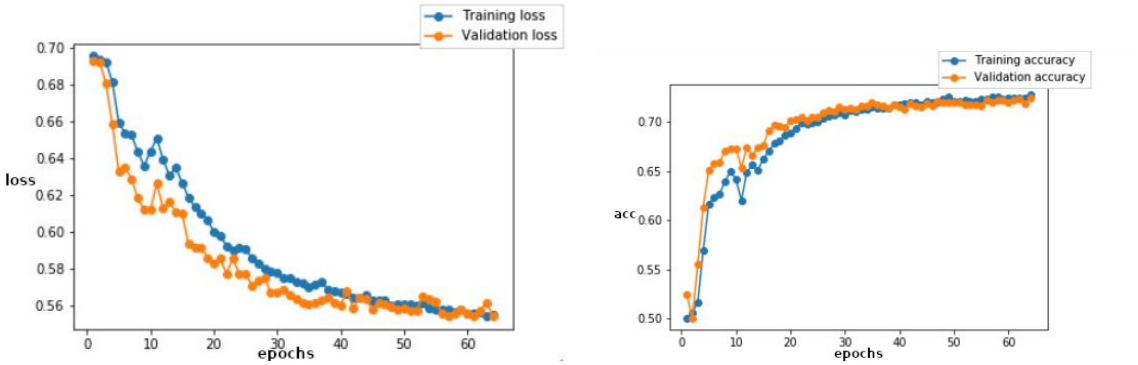

Figure 5: Loss and accuracy functions for an upgraded LSTM with GloVe pre-trained word vector (*glove.6B.50d*). Input dimension of 91291, an output size of 1, embedded dimension of 50, 2 hidden layers with a dimension of 100, dropout rate of 0.9 and a learning rate of 0.0001

self-trained Word2Vec vectors with a pre- trained GloVe vector, the rest of the parameters were kept constant (except for the batch-size which the GPU allowed to increase to 32 and the embedding size which in the second case increases from 50 to 300 because it needed to adapt to the Word2Vec model's dimension). The accuracy graphs of the upgraded models with Word2Vec vectors implemented can be seen in Figure 6.

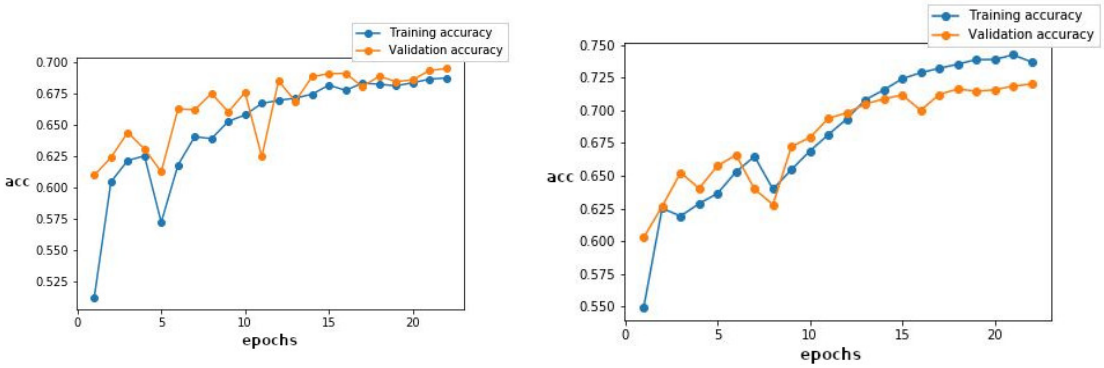

Figure 6: Accuracy for an upgraded LSTM with Word2Vec with 50 and 300 dimensions respectively.

From the model with an embedded size of 50, it can be concluded that it behaved in very similar way to the one with GloVe pre- processing. However, the validation and test accuracy (68.91%) were slightly below. Considering that the input size of the network was lower, playing around with the embedding size and hidden layer size would be necessary, so that the same or a better performance than the one with GloVe can be achieved. The other model, which has an embedded size of 300, seemed to have a small over-fitting tendency. This was an expected issue, since bigger models have more parameters to train, which increases the chance that overfitting occures. Otherwise, the network achieved higher validation and test accuracies. The test accuracy achieved 71.97%, three points above the model that contained the 50 dimensioned word2vec model.

6 Discussion and Conclusions

The purpose of this project was to develop an algorithm capable of predicting the helpfulness of Amazon reviews. To do so, the labeling criteria and raw data processing were covered. The resultant network model is a consequence of a four step process, where improvements and new features have been added. A simple RNN was taken as a starting point. The first step consisted of converting it to a LSTM model. This step has been the key, since the network has proved its learning capability. Unfortunately, it had an excessive over-fitting behaviour.

In order to solve overfitting, dropout was included. Even if this upgrade resulted in a reduction of the over-fitting effect, it has not been enough. This issue was fixed by a package of improvements. The improvement included new features, such as bidirectionality, multiple RNN layers, packed padded sequences, L2 regularization and trained word embedding vectors. This last step enabled a comparison of the performance of the same network

when different embedding vectors were used (pre-trained GloVe or self-trained Word2Vec). Since the same parameters and dimensions were used for both methods and these parameters were optimized for GloVe, the pre-trained GloVe vectors performed slightly better. When augmenting the embedded size of word2vec to 300, the testing and validation accuracies increased a little. This improvement might not be related to a better performance of the word2vec model, but to the increase of the network's size. After fulfilling these four steps,the testing accuracy rose from 50.07% to 73.1%.

Even if the increase in accuracy is considerable, the results were below the ones in the papers [3] and [4]. It has to be considered that paper [4] uses a more extensive data-set (8.19 million reviews, 2.99 million after filtering), which ensures the quality of the reviews and their respective labels. At the same time, it needs to be highlighted that the results obtained in this project are above the results of paper [2].

However, there are many things that could to be improved. Due to the space/resource/time limitation, it was not possible to get deep in the different areas of this extensive project. Firslty, more valuable data can be extracted from the data set. Other fields, such as review titles, posting times or user identification might be helpful to build a more complex network, capable of mixing different approaches. Secondly, if a bigger data set would be available, a more exhaustive filtering (such as the one done in [4]) would definitely increase the performance of the network. The last interesting point would be to study the influence of building the models of word2vec with different parameters.

References

[1] N. Z. Jon Arrizabalaga Linus Persson. (2019). Prediction of helpfulness and rating in amazon reviews, [Online]. Available: `https://github.com/arrijon96/Amazon-review-helpfulness-analysis`.

[2] B. Nguy, "Evaluate helpfulness in amazon reviews using deep learning," *Stanford University*,

[3] Z. Zhou and L. Xu, "Amazon food review classification using deep learning and recommender system," *Department of Statistics, Stanford University*,

[4] J. Wei, J. Ko, and J. Patel, "Predicting amazon product review helpfulness," *Department of Electrical Engineering and Computer Sciences, University of California.*,

[5] B. Trevett. (2019). Pytorch sentiment analysis, [Online]. Available: `https://github.com/bentrevett/pytorch-sentiment-analysis/` (visited on 05/07/2019).

[6] A. K. Nitesh V. Chawla Nathalie Japkowicz, "Editorial: Special issue on learning from imbalanced data sets," -,

[7] J. Brownlee. (2017). What are word embeddings, [Online]. Available: `https://machinelearningmastery.com/what-are-word-embeddings/` (visited on 05/18/2019).

[8] C. D. M. Jeffrey Pennington Richard Socher. (2014). Glove mainpage, [Online]. Available: `https://nlp.stanford.edu/projects/glove/`.

[9] (2019). Glove github, [Online]. Available: `https://github.com/pytorch/text/blob/master/torchtext/vocab.py#L113`.

[10] R. Rehurek. (2019). Gensim, [Online]. Available: `https://radimrehurek.com/gensim/models/word2vec.html`.

[11] B. Zhang. (2018). Build and visualize word2vec model on amazon reviews, [Online]. Available: `https://migsena.com/build-and-visualize-word2vec-model-on-amazon-reviews/`.

[12] A. Morgan. (2018). Basic word2vec using gensim, [Online]. Available: `https://www.kaggle.com/twistedtensor/basic-word2vec-using-gensim`.

[13] B. Zhang. (2019). Aligning torchtext vocab index to loaded embedding pre-trained weights, [Online]. Available: `https://discuss.pytorch.org/t/aligning-torchtext-vocab-index-to-loaded-embedding-pre-trained-weights/20878/2`.

[14] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *University of Amsterdam, OpenAI and University of Toronto*, 2015.

[15] S. S. Team. (2018). The vanishing gradient problem, [Online]. Available: `https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-the-vanishing-gradient-problem/`.

[16] N. Srivastava, "Dropout: A simple way to prevent neural networks from overfitting," *Department of Computer Science, University of Toronto.*,

[17] R. Khandelwal. (2018). L1 and l2 regularization, [Online]. Available: `https://medium.com/datadriveninvestor/l1-l2-regularization-7f1b4fe948f2`.

[18] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *University of Freiburg*, 2019.