EÖTVÖS LORÁND UNIVERSITY

NLP: Sentiment analysis on product reviews at Amazon

Thesis

Daniel Boda
Mathematics in Data analytics and Machine Learning

Supervisors
Dr. András Lukács
Judit Ács
Bálint Csanády

ELTE AI

Budapest
2022

1 Introduction

Sentiment analysis (SA) which is often called opinion mining is a technique in natural language processing (NLP) used to quantify the subjective content of language data. For example, we might want to decide whether a piece of text entails a positive, negative or neutral sentiment. This type of analysis is performed on textual data many times to help businesses to have an overview of the brand or the products, coming from the customer feedbacks and monitor customer demands.

Sentiment analysis helps data analysts or scientists examine product reputation, and understand customer experiences. Typically, companies develop sentiment analysis systems for customer experience management or social media monitoring.

What makes sentiment analysis important in a nutshell? Nowadays people share their comments or opinions and even their emotions in different ratings on social media sites, e-commerce sites. On these, tremendous amount of data is generated in the long run. Successful players in the market often put great efforts to identify their customers' expectations and provide them with the right service faster than their competitors. It allows them to learn what makes their clients happy or disappointed so they can be ready with tailor-made products and services to relevant needs. With regards to the marketing strategy, brands are keen on to observe the impact and the efficiency of their advertisements on the clientele.

There are many different types of sentiment analysis, in this thesis a binary classification is conducted with two categories, negative (neg) and positive (pos). The process itself is a basic NLP task applied on a wide range of data. For example, in Stanford Sentiment Treebank an accuracy ratio can go up to almost 98% with application of different form of transformers or recurrent neural networks. State-of-the-art results are reached by Lan et al., 2020 with an accuracy ratio of 97.1% using ALBERT and by Liu et al., 2019 with an accuracy ratio of 96.7% using RoBERTa. The goal of this thesis is to train models with similar performance on Amazon product reviews derived from the automotive segment.

2 Neural networks

2.1 Inspiration

Artificial Neural Networks (ANNs) are inspired by biological neural networks (Tan et al., 2019). When it is not confusing, we also use term Neural Network (NN) for ANNs. On the other hand, human neural networks are more complex than artificial neural networks, so it is better not to directly draw too many parallels between the two.

2.2 Perceptron

A neuron is often taken as a generic computational unit that takes certain inputs and gives back a single output which is a linear combination of the inputs. Almost always a bias term is also added, and the result is transformed by a non-linear function. In case of multiple neurons, the output is in Eq. 1, where individual neurons are different in their parameters or weights:

$$\begin{bmatrix} f\left(w^{(1)T}x + b^{(1)}\right) \\ \vdots \\ f\left(w^{(m)T}x + b^{(m)}\right) \end{bmatrix} \in \mathbb{R}^m \qquad \text{Eq. 1}$$

where $b = \left[b^{(1)}, \dots, b^{(m)}\right]^{\mathrm{T}} \in \mathbb{R}^m$ is the vector of biases, $W = \left[w^{(1)}, \dots, w^{(m)}\right]^{\mathrm{T}} \in \mathbb{R}^{m \times n}$ is the weight matrix and $f(.)$ is the activation function.

The output of scaling and biases can be written in the form such as:

$$z = Wx + b \in \mathbb{R}^m \qquad \text{Eq.2}$$

The activation of non-linear function is as follows:

$$\tilde{f}(z) = \tilde{f}(Wx + b) \in \mathbb{R}^m \qquad \text{Eq. 3}$$

where $\tilde{f}$ is the multivariate form of f applied coordinate-wise.

Modern neural networks typically have more than one hidden layer. In theory a single hidden layer has the ability to approximate most functions. However, more complex non-linear patterns may be captured with multiple layers achieving better performance. A multilayer perceptron (MLP) is introduced in Eq. 4:

$$\tilde{f}(z)^{(L)} = \tilde{f}(W^{(L)}\tilde{f}(z)^{(L\text{-}1)} + b^{(L)}) \in \mathbb{R}^m \qquad \text{Eq. 4}$$

where L refers to the number of hidden layers in the NN and the last hidden layer.

2.3 Activation functions

The linear combinations of input features and weights translated by the biases are put through a function to make it able to capture non-linear patterns in the data. The used activation functions are detailed below.

The sigmoid is a mathematical function having a characteristic "S"-shaped curve, regularly cited as logistic function which is widely applied in neural nets. A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point. The formula is shown in Eq. 5:

$$\sigma(\gamma) = \frac{1}{1+\exp(-\gamma)} \quad \text{Eq. 5}$$

where $\sigma(\gamma) \in (0,1)$ and $\gamma \in \mathbb{R}$.

The gradient of $\sigma(\gamma)$ can be seen in the Eq. 6:

$$\begin{aligned} \sigma'(\gamma) &= \frac{-\exp(-\gamma)}{1+\exp(-\gamma)} \\ &= \sigma(\gamma)(1-\sigma(\gamma)) \end{aligned} \quad \text{Eq. 6}$$

The tanh function is an alternative to the sigmoid function that is often found to converge faster in practice. The primary difference between tanh and sigmoid is that tanh output ranges from -1 to 1 while the sigmoid ranges from 0 to 1.

$$\tanh(\gamma) = \frac{\exp(\gamma) - \exp(-\gamma)}{\exp(\gamma) + \exp(-\gamma)} = 2\sigma(2\gamma) - 1 \quad \text{Eq. 7}$$

where $\tanh(\gamma) \in (-1,1)$.

The gradient of tanh function is shown in Eq. 8:

$$\tanh'(\gamma) = 1 - \left(\frac{\exp(\gamma) - \exp(-\gamma)}{\exp(\gamma) + \exp(-\gamma)}\right)^2 = 1 - \tanh^2(\gamma) \quad \text{Eq. 8}$$

The ReLU (Rectified Linear Unit) is piecewise linear activation function used in not just NLP, but also in computer vision due to better gradient propagation and efficient computation (Eq. 9):

$$\mathrm{ReLU}(\gamma) = \max(\gamma, 0) \quad \text{Eq. 9}$$

The derivate can be seen in Eq. 10

$$\text{ReLU}'(\gamma) = \begin{cases} 1, & \gamma > 0 \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq. 10}$$

2.4 Optimizers

Most deep learning algorithms involve optimization of some sort. Optimization task refers to either minimizing or maximizing some function $l(\vartheta)$ by altering ϑ in its domain. The function of interest is called the objective function (Goodfellow et al., 2016). When the goal is to minimize it, it may come under the name of cost function, loss function, or error function in the literature. A wide variety of loss functions can be used. Specifically for classification, we often use the cross-entropy loss, for which the probability is derived using the SoftMax function. The SoftMax function takes a vector of m real numbers as input, and normalizes it into a probability distribution, consisting of m probabilities proportional to the exponentials of the input numbers (Eq. 11):

$$\text{SoftMax}(k_i) = \frac{e^{k_i}}{\sum_{j=1}^{m} e^{k_j}} \quad \text{Eq. 11}$$

where k_i is the i-th component of the output of the NN and m is the number of neurons in the final layer of the network, which is the number of classes.

For binary classification the cross-entropy loss can be calculated as:

$$l_{\text{ce}}(y, p) = -(y \log(p) + (1 - y) \log(1 - p)) \quad \text{Eq. 12}$$

where y is a binary indicator if the class label is the classification for an observation and p is the predicted probability that the observation is of a certain class.

For multiclass classification Eq. 13 can be generalized as follows:

$$l_{\text{ce}}(y, p) = -\sum_{c=1}^{m} y_{o,c} \log(p_{o,c}) \quad \text{Eq. 13}$$

where m denotes the number of classes, y is a binary indicator if the class label c is the correct classification for observation o, and p is the predicted probability that observation o is of class c.

An optimal solution is $\varphi^* = \operatorname{argmin} l(\vartheta)$. φ^* is called a global minimum which is not guaranteed to be obtained during the optimization. For functions with multiple inputs: $\mathbb{R}^n \mapsto \mathbb{R}$, the concept of partial derivatives should be used to find the direction in some sense. In terms of the definition, the partial derivative $\frac{\partial}{\partial \vartheta_i} l(\vartheta)$ measures how the function l changes as ϑ varies only

at the *i*-th coordinate. The gradient is helpful by generalizing the notion of derivative with respect to a vector. The directional derivative in direction u (a unit vector) is the slope of the function l in direction u. To minimize l, the direction can be found in which the function l decreases the fastest i.e., vector u points in the opposite direction as the gradient. It is called as negative gradient. The function l may be decreased by taking steps in the direction of the negative gradient. This method is widely used under the name of steepest descent or gradient descent. The gradient can be calculated efficiently by backpropagation algorithm using the chain rule. The new point is determined according to Eq. 14:

$$\vartheta^{t+1} = \vartheta^{t} - \alpha \nabla_{\vartheta} l(\vartheta^{t}) \qquad \text{Eq. 14}$$

where α is the learning rate or step-size, a positive scalar which can be determined in many different ways. A popular approach is to set α to a small constant. Another approach, called line search, is to evaluate $l\left(\vartheta - \alpha \nabla \vartheta l(\vartheta)\right)$ for several values of α and choose that one which gives the smallest objective function value. Steepest descent iteration stops if every element of the gradient is approximately zero.

The update step can be computationally expensive which can be handled by stochastic gradient-based optimization technique. For example, Adam is a stochastic optimization algorithm that requires first-order gradients. It computes individual adaptive learning rates for different parameters from the estimation of first and second moments of the gradients. This is where the name Adam is derived from. The simplified description of it is in algorithm 1:

Algorithm 1: Adam

1 $\alpha := 0.001$ (default parameter)
2 $\beta_1 := 0.9, \beta_2 = 0.999$ (default parameters)
3 $\varepsilon := 10^{-8}$(default parameter)
4 $m_0 := 0$ (initial parameter)
5 $v_0 := 0$ (initial parameter)
6 $t := 0$ (initial parameter)

7 **while** ϑ_t not converged **do**
8 $t \leftarrow t + 1$
9 $g_t \leftarrow \nabla_{\vartheta} l_t(\vartheta_{t-1})$
10 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
11 $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
12 $m'_t \leftarrow m_t / (1 - \beta_1^t)$
13 $v'_t \leftarrow v_t / (1 - \beta_2^t)$
14 $\vartheta_t \leftarrow \vartheta_{t-1} - \alpha \cdot m'_t / \left(\sqrt{v'_t} + \varepsilon\right)$
15 **end while**
16 **return** ϑ_t

where $l(\vartheta)$ is a stochastic objective function with parameters ϑ, g_t is a gradient vector at time-step t, α step-size, β_1, β_2 are exponential decay rates for moment estimates, ϑ_0 is an initial parameter vector, g_t^2 indicates element-wise square: $g_t \circ g_t$.

Line 9 refers to the calculation of gradients with respect to the stochastic objective at t, line 10 and 11 apply the update of the biased first moment estimation and second raw moment estimate, line 12 and 13 corresponds to the computation of bias-corrected first moment estimation and second moment estimation respectively, and line 14 refers to the update of the parameter vector at time t.

2.5 Regularization

Neural nets are highly prone to overfitting, as is the case with many machine learning models where a model approaches near-perfect performance on the training set, but the generalization ability is lost to predict accurately on test data. A widely applied technique used to avoid overfitting is the incorporation of an $l2$-norm penalty term. The goal is to extend the original loss function l_{ce} with an extra term so that the overall cost is calculated as in Eq. 15:

$$l_{\text{R}} = l_{\text{ce}} + \lambda \sum_{i=1}^{L} \|W^{(L)}\|_F \qquad \text{Eq. 15}$$

In the above formulation, $\|W^{(L)}\|$ is the Frobenius norm of the L-th matrix $W^{(L)}$ in the network and λ is the hyper-parameter which controls how much weight is put on the regularization term in relation to the original cost function. The Frobenius norm of the weight matrix can be seen in Eq. 16:

$$\|W^{(L)}\| = \sqrt{\sum_j \sum_k W_{jk}{}^2} \qquad \text{Eq. 16}$$

The parameter λ is chosen via hyperparameter-tuning, if the value of λ is high, the weights will have a tendency to be close to 0, and the model may not learn anything relevant from the training set. Too low a value may cause overfitting with no ability for generalization. The bias terms are not regularized, so it has no contribution to the cost term.

The other type of powerful regularization technique is dropout, which was first introduced by Srivastava et al. in “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. The idea is that in the training phase a random subset of neurons is temporarily disabled (set to 0) with some probability p during each training step (forward and backward pass). At inference, the

full network is used to compute the predictions. As a result of this process, the network cannot rely as much on the activation of individual neurons, this inhibits its ability to memorize the training data, which combats overfitting as the network is forced to learn more general patterns.

Another effective technique to reduce overfitting is to increase the size of the training data. In machine learning the size of the labeled data would be costly to be increased. Although, in image recognition for example, there is an effective way to do so by rotating, flipping, scaling, shifting the images. This is known as data augmentation. Applying some form of it may cause a big leap in the accuracy of a model.

Training Deep Neural Networks can be complicated due to the fact that the distribution of each layer's inputs may change over training, as the parameters of the previous layers are updated. As a result of that, the training is slower by requiring lower learning rates and careful parameter initialization. The solution is to normalize layer inputs in each mini-batch. It can be interpreted as a regularizer, in some cases, with no need for Dropout. Batch Normalization applied in a state-of-the-art image classification model, achieves the same accuracy with 14 times fewer training steps, and overperform the original model significantly (Ioffe, Szegedy, 2015).

Last but not least, another method worth mentioning is the early stopping. This is a sort of cross-validation technique where a random subset of the training set is filtered as validation set. If the performance on the validation set starts deteriorating, the training of the model is stop immediately. That is where the name, early stopping comes from.

2.6 Epoch, batch, iteration

One epoch is defined as a pass of the entire dataset forward and backward through the neural network exactly once. In many applications the dataset is divided into several smaller batches determined by batch size which allows faster convergence in the training phase. As the number of epochs increases, the more times the weights are changed in the neural network and the model goes from underfitting to optimal to overfitting (Figure 1). As there is no right answer for the question regarding the sufficient number of epochs, the maximum point of the accuracy curve is aimed in a classification task taking into consideration the time limitation.

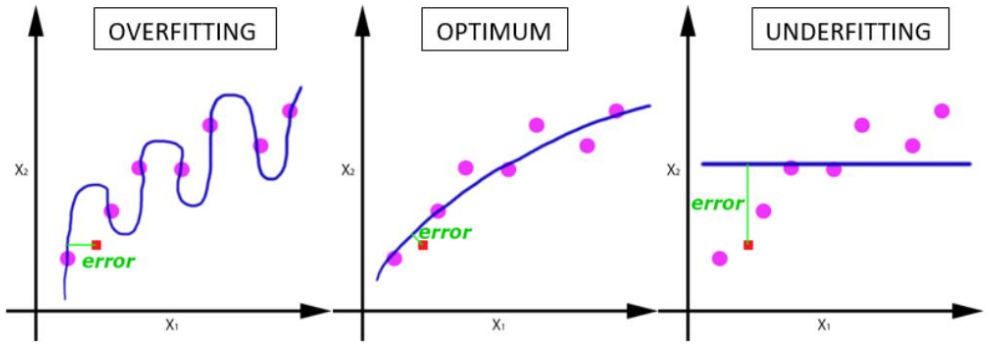

Figure 1: Type of model fitting (source: towardsdatascience.com)

3 LSTM

Long-Short-Term Memory(LSTM) models are a type of Recurrent Neural Networks (RNNs). The specialty comes from the fact that it is able to learn and remember over long sequences of input data using certain gates which are responsible for controlling the flow of the information in the network. The LSTM was introduced by Hochreiter and Schmidhuber in 1997. The main difference between traditional neural networks and RNN is that RNNs are networks with loops in them, allow information to persist. LSTMs address the following limitations:

- Short-term memory: it means information from earlier time steps is discarded when moving ahead in time, which may be resulted in the loss of important information.

- Vanishing gradient: the main problem is that if a gradient value becomes extremely small, it has no contribution in the learning phase. In case of vanishing gradient, gradient shrinks as backpropagation through time is executed.

- Exploding gradient: the exploding occurs when large error gradients accumulate, resulting in extremely large updates to neural network model weights in the training phase. Therefore, the model is unstable and incapable of learning from the training data.

In this work a bidirectional, multi-layer LSTM is developed with Pytorch, in which the number of features in the hidden state was set to 100, the number of recurrent layers to 2 (stacked LSTM) and the dropout probability equal to 50%.

The structure of an LSTM is built up from the following stages as shown in Figure 2:

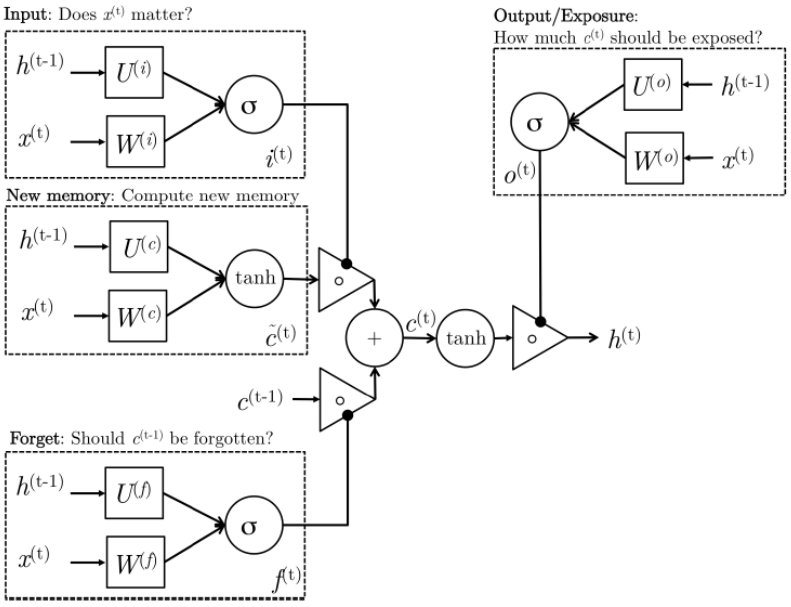

Figure 2: LSTM architecture (source: Stanford CS224n)

The corresponding equations for the Figure 2 are the following:

(Input gate)	$i_t = \sigma(W^{(i)} x_t + U^{(i)} h_{t-1})$	Eq. 17
(Forget gate)	$f_t = \sigma(W^{(f)} x_t + U^{(f)} h_{t-1})$	Eq. 18
(Output/Exposure gate)	$o_t = \sigma(W^{(o)} x_t + U^{(o)} h_{t-1})$	Eq. 19
(New memory cell)	$\check{c}_t = \tanh(W^{(c)} x_t + U^{(c)} h_{t-1})$	Eq. 20
(Final memory cell)	$c_t = f_t \circ c_{t-1} + i_t \circ \check{c}_t$	Eq. 21
	$h_t = o_t \circ \tanh(c_t)$	Eq. 22

where

- $x_1, x_2, \ldots, x_T$: the word vectors corresponding to a corpus with T words
- $x_t \in \mathbb{R}^d$: input word vector at time t
- $W^{(i)}, W^{(f)}, W^{(o)} \in \mathbb{R}^{m \times d}$: corresponding weight matrix for a certain gate multiplied with the input word vector x_t
- $U^{(i)}, U^{(f)}, U^{(o)} \in \mathbb{R}^{m \times m}$: corresponding weight matrix for a certain gate multiplied with the output of the previous time-step, h_{t-1}
- $h_{t-1} \in \mathbb{R}^m$: output of the non-linear function at the previous time-step, t-1
- σ(): the coordinate-wise sigmoid function $\mathbb{R}^m \mapsto \mathbb{R}^m$
- tanh(): the coordinate-wise tanh function $\mathbb{R}^m \mapsto \mathbb{R}^m$
- ∘: Element-wise multiplication $\mathbb{R}^m \mapsto \mathbb{R}^m$

New memory generation ($\check{c}_t$): This stage is responsible for generating new memory $\check{c}_t$ which includes certain aspects of the input word x_t and the past hidden state h_{t-1}.

Input gate (i_t): This gate determines whether or not the input is worth preserving based on the input word x_t and the past hidden state h_{t-1} and is used to control the new memory.

Forget gate (f_t): This gate assesses whether the past memory cell can be considered useful or not for the computation of the current memory cell. The forget gate takes a look at the input word and the past hidden state and generates f_t.

Final memory generation (c_t): First it takes the advice of the forget gate f_t and correspondingly forgets the past memory c_{t-1}, then it takes the advice of the input gate i_t and gates the new memory $\check{c}_t$ accordingly. The final memory c_t is produced by the summation of these two components.

Output/Exposure gate (o_t): this gate separates the final memory from the hidden state and makes the assessment on what parts of the memory c_t needs to be present in the hidden state h_t, as the final memory c_t incorporates information that may not be required to be preserved. The gate produces o_t, which is in control of the point-wise tanh of the final memory.

4 GloVe

GloVe is an unsupervised learning algorithm to obtain vector representations for words. Aggregated global word-word co-occurrence statistics from a corpus in the form of occurrence matrix is served as a field for training. The output representations may show interesting linear substructures of the word vector space. Filling up this matrix is done through a single pass over the entire corpus. This step may be computationally expensive in case of a large corpus size, but it must be carried out only once. The training iterations after that are much faster, because the number of non-zero entries in the matrix is typically smaller than the total number of words in the corpus. The computational complexity of the model depends on the number of nonzero elements in the matrix. As this number is always less than the total number of entries of the matrix, the model scales no worse than $O(|V|^2)$. The starting point for word vector learning is the ratios of co-occurrence probabilities (right-hand side in Eq. 23):

$$F(w_i, w_j, w''_k) = \frac{P_{ik}}{P_{jk}} \qquad \text{Eq. 23}$$

where $w \in \mathbb{R}^d$ are word vectors, $w'' \in \mathbb{R}^d$ are separate context word vectors, P_{ik} is the probability of the word i appearing in the context of word k, P_{jk} is the probability of the word j appearing in the context of word k and the function $F(.)$ models ratio of the probabilities. The weights for the function are determined by statistical learning. Since the left-hand side of the Eq. 23 is a vector

space and all vector spaces are linear structures, the vector differences $w_i - w_j$ are used to present the ratio of $\frac{P_{ik}}{P_{jk}}$ in F. To match the scalar of the right-hand side the dot product with w'' is applied:

$$F\left((w_i - w_j)^{\mathrm{T}} w''_k\right) = \frac{P_{ik}}{P_{jk}} \qquad \text{Eq. 24}$$

The distinction between a word and a context word should be free to exchange, but Eq. 24 is not invariant of this relabeling. In order to achieve such symmetry, homomorphism is used. Loosely speaking, a homomorphism will preserve the algebraic structure between the two groups interchangeably:

$$F\left((w_i - w_j)^{\mathrm{T}} w''_k\right) = \frac{F(w_i{}^{\mathrm{T}} w''_k)}{F(w_j{}^{\mathrm{T}} w''_k)} \qquad \text{Eq. 25}$$

From Eq. 24 and Eq. 25, Eq. 26 is derived:

$$F(w_i{}^{\mathrm{T}} w''_k) \; = \; P_{ik} = \frac{X_{ik}}{X_i} \qquad \text{Eq. 26}$$

where X_{ik} is the number of times word i occurs in the context of k and X_i is the number of times i occurs in any context.

Replacing F with exponential function:

$$e^{(w_i{}^{\mathrm{T}} w''_k)} \; = \; P_{ik} \qquad \text{Eq. 27}$$

Then applying log on both sides, the Eq. 28 can be formulated:

$$w_i{}^{\mathrm{T}} w''_k = \log(P_{ik}) = \log(X_{ik}) \text{ - } \log(X_i) \qquad \text{Eq. 28}$$

Since $log(X_i)$ is independent of k, it can be absorbed into a bias b_i. An additional bias term b''_k for w''_k is also added and Eq. 29 is derived:

$$w_i{}^{\mathrm{T}} w''_k + b_i + b''_k = \log(X_{ik}) \qquad \text{Eq. 29}$$

Eq. 29 gives equal weights to all word-word co-occurrences, no matter whether it is signal or noise, so a weighting function $\omega(x)$ is introduced to put weights on the words according to the

available information regarding the content. Casting Eq. 29 as a least-squares problem and introducing a weighting function $\omega(X_{ij})$ into the game, the following loss function l of GloVe model is formulated in Eq. 30:

$$l = \sum_{i,j=1}^{V=|\text{vocab}|} \omega(X_{ij})(w_i^{\mathrm{T}} w''_k + b_i + b''_k - log\, X_{ik})^2 \qquad \text{Eq. 30}$$

where the function ω holds the followings properties:

1. $\omega(0) = 0$ and if the function ω is continuous, it should vanish, as $x \to 0$ fast enough that $\lim_{x \to 0} \omega(x) \log^2 x < \infty$,
2. $\omega(x)$ should be non-decreasing (rare co-occurrences are not over-weighted), $\omega(x)$ should be relatively small for large values of x (frequent co-occurrences are not over-weighted).

In the original paper the proposed ω function is an extension of the clipped power law function, which is widely used in Statistical Modeling:

$$\omega(x) = \begin{cases} (x / x_{\max})^{\tau}, & \text{if } x < x_{\max} \\ 1, & \text{otherwise} \end{cases} \qquad \text{Eq. 31}$$

The weighting function ω with $\tau = ¾$ can be seen in Figure 5:

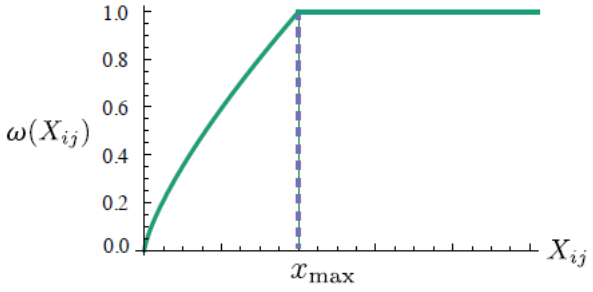

Figure 5: Weighting function values at domain of X (source: original paper)

So the main take-away comes from the problem that two words may have different distances based on the context, which is not possible to be captured by a single scalar as is the case in many metrics. For example, "man" may be similar to "woman" as they are both human beings, on the other hand, the two words can be considered opposites because they represent

different genders. The object for an enlarged set of discriminative numbers is the vector difference between the two word vectors. GloVe is designed in order that such vector differences capture the meaning as much as possible.

5 Transformers

The Transformer is a novel architecture in NLP with a purpose to solve sequence-to-sequence tasks handling long-term dependencies. The computation of representations of its input and output is based on self-attention without the usage of sequence-aligned RNNs or convolution. Attention makes it possible to focus on relevant parts of the input sequence in the prediction of the output sequence. The model architecture with the encoder-decoder structure can be seen in Figure 3:

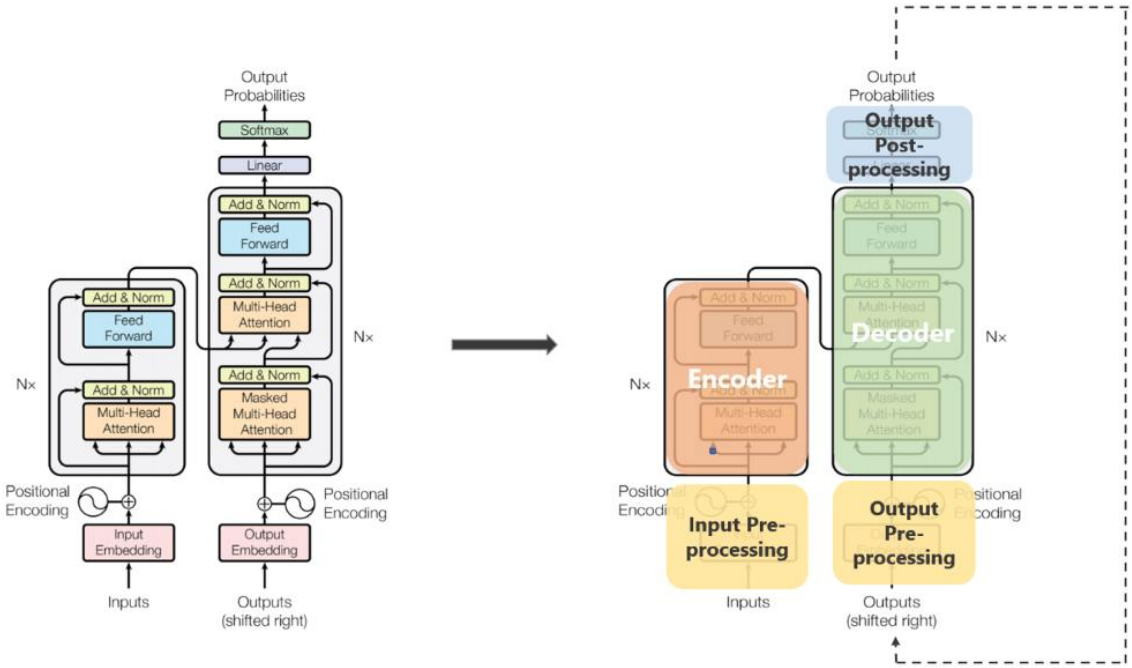

Figure 3: Transformer architecture (source: left-original paper, right-towardsdatascience.com)

The encoder maps an input sequence $x = (x_{1, ..., x_d})$ to a sequence of continuous vector representations $z = (z_{1, ..., z_d})$. The input consists of learnable embeddings with no positional information. Based on z, the decoder generates an output sequence $y = (y_{1, ..., y_e})$, one at a time. At every single step the model uses the previously generated output as additional input in generating the next. As it can be seen on Figure 3, the transformer is built from stacked self-attentions and point-wise, fully connected layers for both the encoder and the decoder.

The encoder is composed of stacking a number of identical layers together, each layer having two sub-layers. The first is a multi-head self-attention mechanism, and the second is a

point-wise fully connected FFN. Residual connection, also known as skip connection is applied around each of the two sub-layers, followed by layer normalization. This technique is widely adopted in other deep learning fields as well, such as computer vision. For example, it enabled the Resnet architecture to be able to contain deeper and deeper networks, which are harder to successfully train without relying on these techniques. These residual connections, sub-layers and the embedding layers, generates outputs of dimension $d_m = 512$.

The decoder is also composed of a number of identical layers stacked together. The difference comes from the fact that the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections are in use around each of the sub-layers with layer normalization. The sub-layers and the embedding layers produce outputs of the same dimension. This is done to facilitate the residual connections. Modification of the self-attention sub-layer in the decoder stack can be seen on Figure 3. Masking ensures that the predictions for the i-th position can depend only on the known outputs at positions less than i to avoid information leak.

An attention function can be described as mapping a query and a set of key-value pairs to an output. The output is computed as a weighted sum of the values, where the weight is quantified by a compatibility function of the query with the corresponding key. The Multi-Head Attention is depicted in Figure 4:

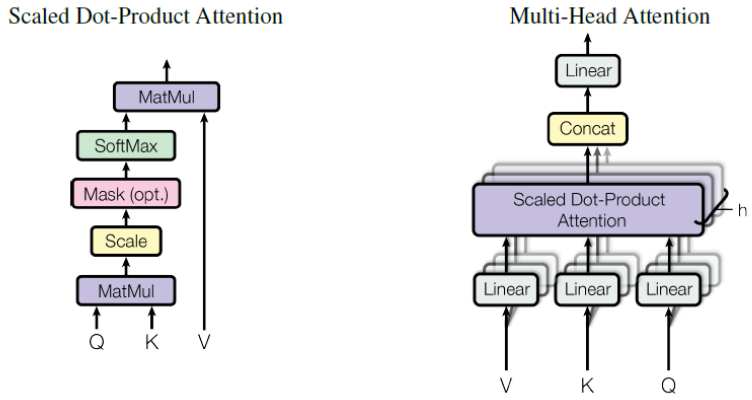

Figure 4: Multi-Head Attention (source: original paper)

The Scaled Dot-Product Attention (left part of Figure 4.) can be calculated by the following formulation:

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^{\text{T}}}{\sqrt{d_k}}\right)V \qquad \text{Eq. 32}$$

where Q is matrix of queries, K is a matrix of keys, d_k is the dimension of keys. The dot product of QK^T scaled by $\sqrt{d_k}$ is done for better gradient flow which is important if the value of the dot product in previous step is unreasonably big. Putting it through a non-linear SoftMax transformation, it represents the weights on values V. The purpose of which is to keep preserving only the values V of the input words, by multiplying them with high probability scores from a SoftMax function(∼1), and removing the others by multiplying them with the low probability scores (∼0) from a SoftMax function.

The Multi-Head Attention (right part of Figure 4.) deploys the efficiency of parallel operations by linearly projecting the queries, keys and values h times to d_k, d_k and d_v dimensions, respectively. In the original paper h equals 8 (Vaswani et al., 2017). On each of these projected versions of queries, keys and values, the attention function is performed in parallel and the results are concatenated and linearly projected. The most important thing in multi-head attention is that it makes the model able to focus on different positions, which is not possible with a single attention head. The multi-head attention function is the following:

$$\text{MultiHead}(Q, K, V) = [head_1, \dots, head_h]W^O \qquad \text{Eq. 33}$$

where $head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and [.] means concatenation. The projections matrices are $W_i^Q \in \mathbb{R}^{d_m \times d_k}$, $W_i^K \in \mathbb{R}^{d_m \times d_k}$, $W_i^V \in \mathbb{R}^{d_m \times d_v}$, while $W^O \in \mathbb{R}^{hd_v \times d_m}$. The purpose of W^O is that the size of the concatenated vector is too large to be fed to the next sub-layer, so the vector must be scaled down by multiplying with it.

The Transformer uses multi-head attention in three different ways:

- In encoder-decoder attention layers which allows every position in the decoder to attend over positions in the input sequence (attention between the input sequence and the output sequence).
- The encoder contains self-attention layers by which the encoder can attend to positions in the previous layer of the encoder.
- The decoder contains self-attention layers which allow the decoder to attend to positions in the decoder up to a certain position. This is implemented inside of scaled dot-product attention by masking out (technically by setting it to -inf) the words that occur after it for each step.

Every layer in the encoder and decoder involves a fully connected feed-forward network, which is applied position by position separately. The net consists of two linear transformations with a ReLU activation between them:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \qquad \text{Eq. 34}$$

While the linear transformations are the same over the positions, different parameters are used from layer to layer.

Since the model contains no recurrence or convolution, in order to determine the order of the sequence, some information about the position of the individual tokens in the sequence must be inserted. Positional encodings (vectors) are added to the input embeddings at the beginning of the encoder and decoder modules. The positional encodings have the same dimension as the embeddings so they can be summed up. The positional encoding vectors follow specific periodic functions, namely combinations of various sines/cosines with different frequencies. The model can rely on this additional information to determine the position of individual words with respect to each other.

6 BERT

BERT is a type of Transformer Encoder which is pre-trained on large corpora in a self-supervised way, introduced by Devlin et al., 2019. Originally it was pre-trained on English texts crawled from the Internet. The self-supervised training means that there is no need for human-annotated data, the training objective is generated automatically.

BERT is an abbreviation of Bidirectional Encoder Representations from Transformers. The feature that makes it unique is that it has a generic architecture which can be successfully applied to a wide range of NLP tasks. These may be out of the scope of the original task the model was trained on, but it gives state-of-the-art results nonetheless. The key point is that the same pre-trained model can be applied with fine-tuning adding only one additional output layer for a variety of tasks.

The framework is built up from two stages, namely pre-training and fine-tuning. In the first stage, the model is trained as a so-called masked language model (MLM) in a self-supervised fashion. In the second phase, initialization is done for the model with the pre-trained parameters, and the additional weights are fine-tuned using labeled data from the downstream tasks, which have separate fine-tuned models. In the thesis the BERT base model is applied. It has 110M total parameters, the number of layers (Transformer blocks) is 12, the hidden size is 768 and the number of self-attention heads is 16. A final linear layer projects the output of BERT into $\mathbb{R}^2$ for classifiation.

The input token sequence to BERT takes a form of WordPiece embeddings with a vocabulary size of 30 000 tokens. A special classification token ([CLS]) is used as the first token in every single sequence: the final hidden state for this token serves as basis for the aggregate sequence representation regarding to classification tasks. A pair of sentences can be concatenated together to form a single sequence, separated by a special token ([SEP]). There is also a learnable embedding to every token which indicates the reference to sentence A or sentence B. For better understanding Figure 6 can serve as help:

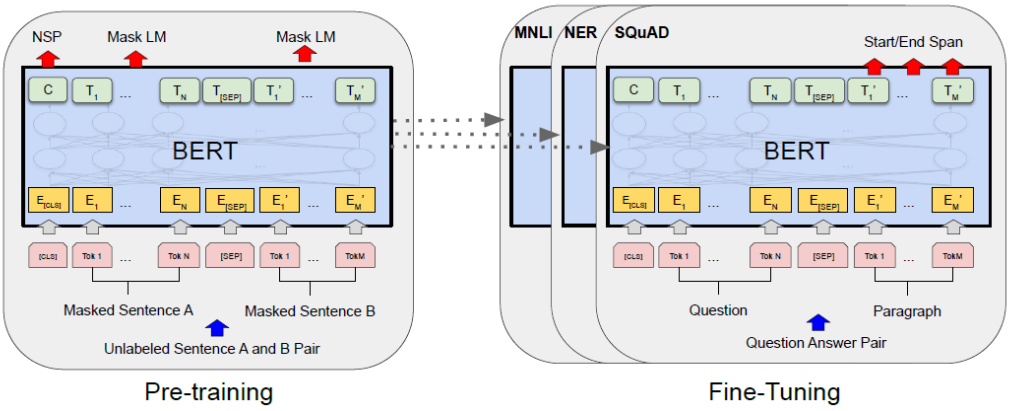

Figure 6: Two main part of Bert (source: original paper)

The representation of every token is constructed by summing up the corresponding token, segment, and position embeddings. Input embeddings are denoted as E, the final hidden vector of the special [CLS] token as $C \in \mathbb{R}^H$, and the final hidden vector for the i-th input token as $T_i \in \mathbb{R}^H$, where H refers to the hidden size. A visualization of this construction can be seen in Figure 7:

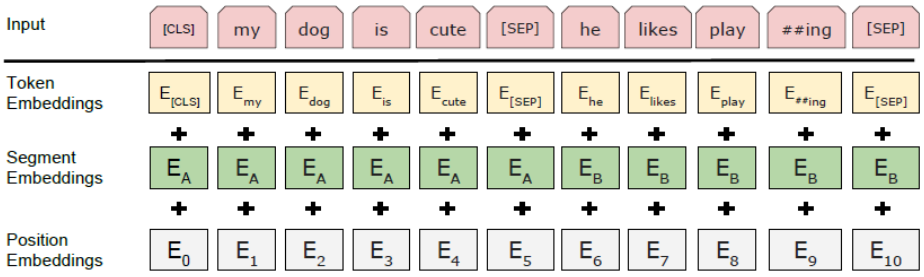

Figure 7: Different embeddings (source: original paper)

This structure solves two types of problems: the ordering due to Position Embeddings (similar to positional vectors in Chapter 5) and the differentiation of sentences (A or B) due to Segment Embeddings. The solution is to add these embeddings to the original tokens and use the result as input for BERT.

Bert is pre-trained with two unsupervised objectives which are shown by the left and right side of Figure 6 respectively:

- For the masked language modeling (MLM) task the model masks 15% of all WordPiece tokens at random[1]. After that it runs through the entire masked sentence and makes a prediction on the masked words.
- In the Next sentence prediction, the question is whether the second sentence should directly follow the first, or not (the two sentences are separated by the [SEP] token). This is the classification task on which the output [CLS] token is trained on.

For fine-tuning BERT, task-specific inputs and outputs must be plugged into it, to fine-tune all parameters end-to-end, which can be done relatively fast. If only one extra output layer is added for a specific NLP-task (such as the case in sentiment analysis), and no further training is done on the original weights, the process is even faster. This may work well in practice, meaning that the pretrained BERT representations are often so good quality, that the two classes are linearly separable in the representation space.

[1]The model replaces the *i*-th token with the [MASK] token 80% of the time, a random token 10% of the time and leaves it unchanged 10% of the time (out of the 15%).

7 Data

Originally the dataset contains more than 1.000.000 records which ended up computationally heavy for the NLP task in Google Colab. As a solution, random sampling was applied to determine the final train-validation-test sets (200.000-40.000-10.000). Preprocessing consists of two data manipulation steps:

1. select reviews with length greater than 10 and less than 4.000
2. truncate reviews back to a maximum length of 1.000 (512 for BERT)

The sentiments of the different reviews were grouped together, neg $= \{1, 2\}$, pos $= \{4, 5\}$ for a binary classification problem, filtering out the middle category.

To represent the text as a sequence of vectors, first certain types of tokenizations were applied. Tokenization is a way of separating a piece of text into smaller units called tokens based on certain predefined rules. Tokenization can be broadly classified into 3 types: word, character, and subword (n-gram characters) tokenization. Embedding helps to represent the semantics about the word and can be used for NLP tasks such as classification, because vector representation of words that are similar in meaning and context, appear closer together after training.

The quality of the output depends heavily on the quality of the input text used, so data preparation is an important factor in the analysis. Stop words and punctuation usually have no additional value to the meaning of the text and can potentially impact the outcome. To avoid this, it makes sense to remove them and clean the text of unwanted characters which can reduce the size of the corpus. These all can be handled by the widely-used tokenization packages in Python.

For the LTSM model spacy tokenization is applied. To deal with the issue of Out Of Vocabulary (OOV) words, an unknown token is defined in the encoding. As BERT model requires the corresponding BERT tokenizer to be applied, it is used for the model.

To initialize the elements of the embedding matrices, normal distribution is used with scale of 0.6, except for a word which can be found in Glove 100 or Google News 300 word vectors. For weight matrices, uniform distribution is used initially.

8 Results

8.1 LSTM with GloVe

In this section, the LSTM model is initialized with GloVe100 word-vector embeddings. The model is trained over 15 epochs. The training loss gradually decreases over the range as the model becomes over-fitted by memorizing even the noise itself, even though drop-out regularization is used with a probability of 50%. As opposed to that, the validation loss reaches its minimum around the 7-th epoch showing a U-shaped curve.

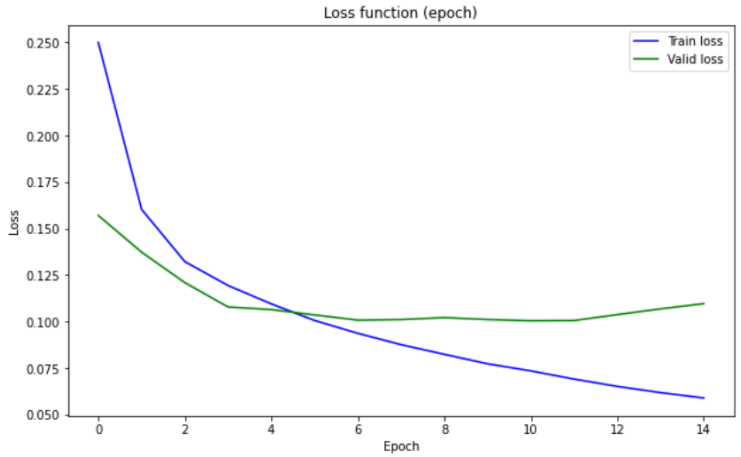

Figure 8: Loss over epochs

We can see similar over-fitting with regards to the validation accuracy. As the figure shows the accuracy peaks after the 7-th epoch somewhere.

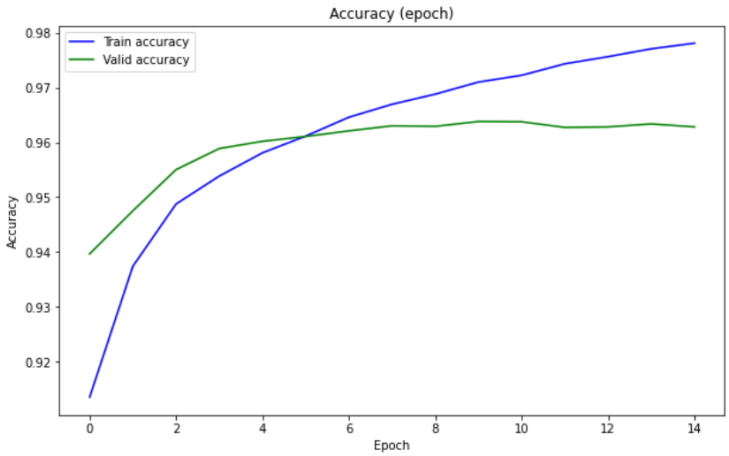

Figure 9: Accuracy over epochs

Finally, the confusion matrix is shown below in Figure 10, with 83% and 97.4% for specificity and recall respectively:

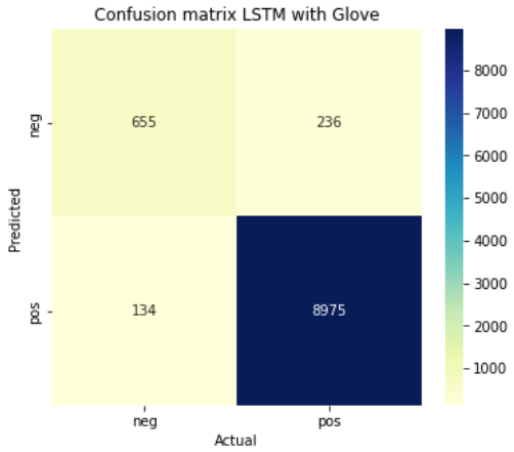

Figure 10: confusion matrix

8.2 LSTM with Google News 300 word2vec

In this section, the LSTM model is initialized with Google News 300 word2vec word-vector embeddings. The model is trained over 15 epochs. Similar things are happening here, the training loss gradually decreases over the epochs as the model becomes over-fitted. Here drop-out regularization is used as well with a probability of 50%. The validation loss reaches its minimum around the 5-th epoch as it can be seen in the curve below (Figure 11.).

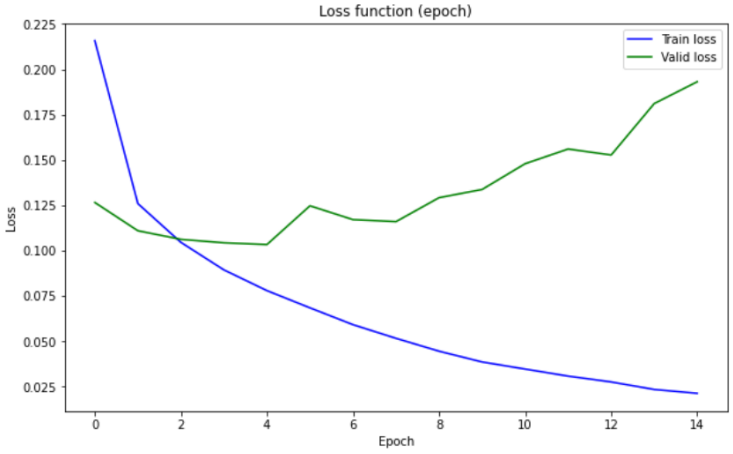

Figure 11: loss function over epochs

In terms of the accuracy ratio, nothing different is observed. The highest value of the validation accuracy can be seen after the 7-th epoch.

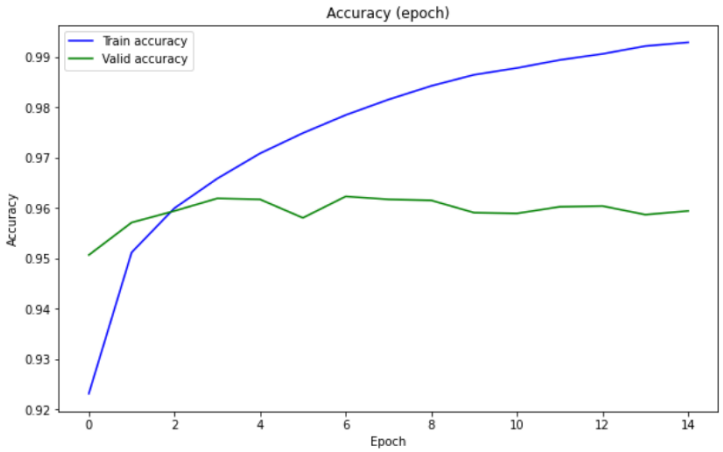

Figure 12: accuracy over epochs

In the end the confusion matrix is represented in the Figure 13 with 81% and 97.3% for specificity and recall respectively:

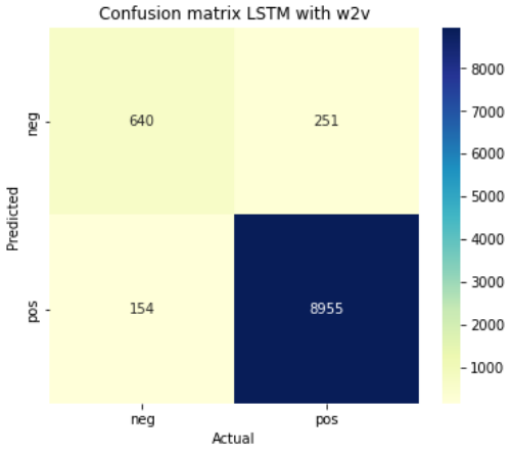

Figure 13: confusion matrix

8.3 BERT

In this section the BERT base model is fine-tuned only over 4 epochs with a batch size of 2 due to ~110M trainable parameters and Google Colab memory limitations. Very similar things are happening in this case as well. The training loss gradually decreases over the epochs as the model becomes over-fitted. Drop-out regularization is used with a probability of 50%. The validation loss reaches its minimum around the 2-nd epoch as it can be seen in the curve below.

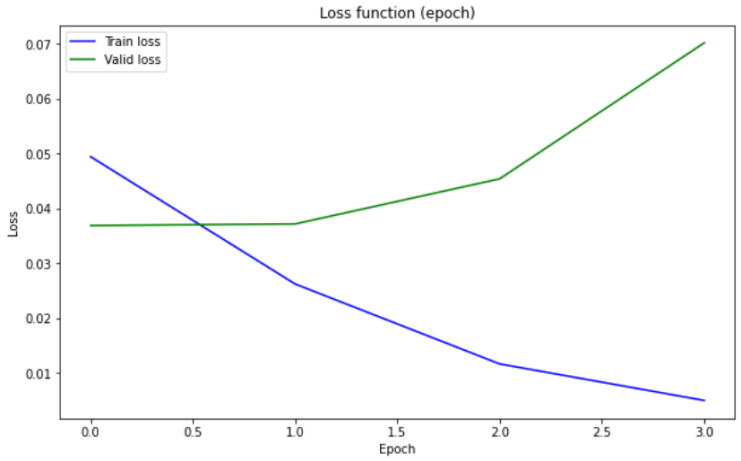

Figure 14: loss over epochs

The validation accuracy reaches its top value after the 2-nd epoch.

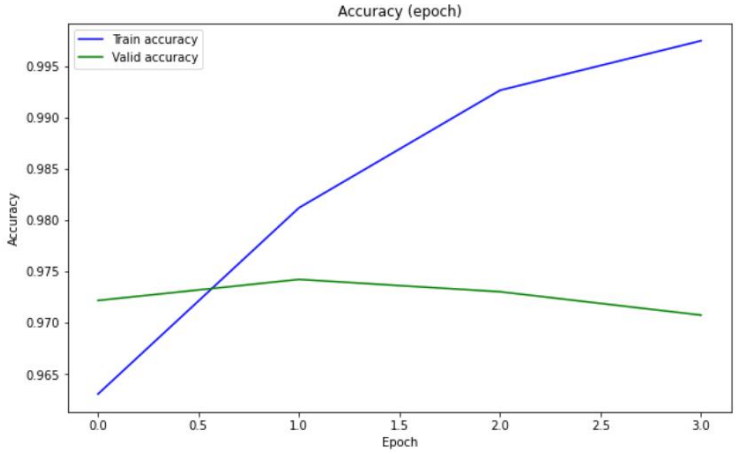

Figure 15: accuracy over epochs in case of BERT base (fine-tuned weights)

With regards to the confusion matrix, specificity and recall is 88% and 97.5% respectively. The specificity is significantly higher compared to any other model.

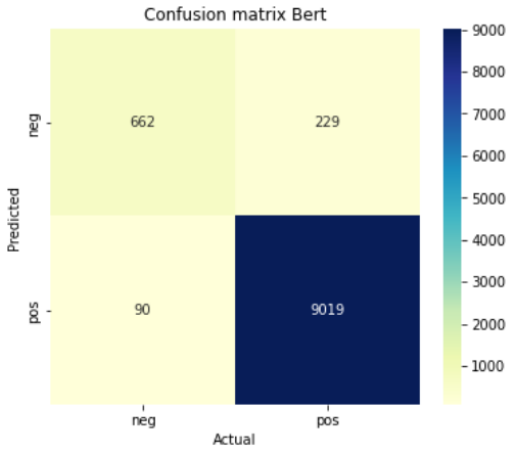

Figure 17: confusion matrix

Figure 16 is shown below with a purpose to highlight the performance of the original BERT base model as a benchmark. Here all the weights are frozen during fine-tuning, except for the last linear layer. This model still can reach an accuracy over 90%.

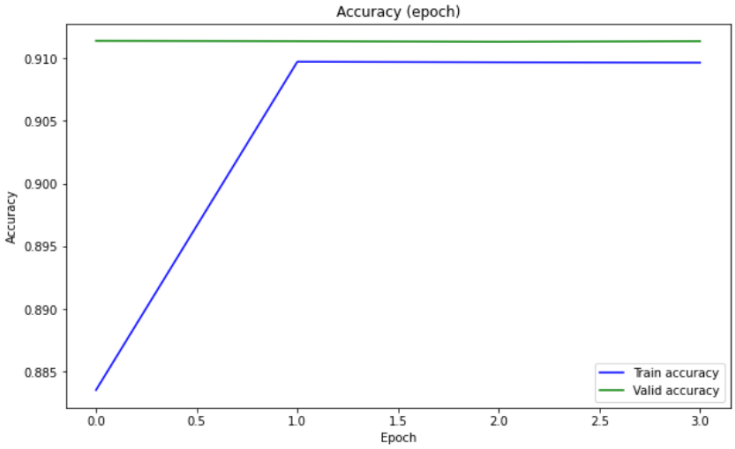

Figure 16: accuracy over epochs in case of BERT base (original weights)

8.4 Summary table

Finally, each of the models is evaluated on the test sample with a size of 10.000 observations where the original distribution of the classes is hold. No significant difference can be highlighted in the test accuracy of the last three models. The BERT base model (fined-tuned weights) with a final linear layer finishes in the first place:

Model	Test accuracy
BERT base (original weights)	91.11%
LSTM with w2v	96.00%
LSTM with Glove	96.30%
BERT base (fined-tuned weights)	96.80%

Table 1: test accuracy by models

8.5 Model comparison

In this section, a few more-or-less interesting examples are filtered out from the test dataset to see the difference in the predictions from the models compared to the original sentiment labels.

Some positive sentiments are collected in Table 2:

Text	Sentiment	Prediction LSTM w word2vec	Prediction LSTM w Glove	Prediction BERT base
It was a very good quality product, however sizing was small and I returned them and did not reorder.	Positive	Negative	Negative	Positive
First off, this is *THE ONLY* choice for transmissions asking specifically for Mercon SP (such as my 2006 F250). Do *NOT* get tricked into using Mercon LV in place of Mercon SP - IT IS NOT THE SAME AND WILL NOT WORK THE SAME no matter who tells you it will....	Positive	Positive	Negative	Negative
This is obsolete now. WOTLK was the best expansion. It has gone downhill ever since.	Positive	Negative	Positive	Negative
the one bracket came back broken after some transmission work. Figured after 17 years it was due. Also ordered a new boot at the same time. Both installed without a problem. 2000 Jeep TJ with 5/speed manual transmission.	Positive	Negative	Negative	Negative
came fast and just what I needed..	Positive	Positive	Positive	Positive

Table 2: Positive sentiments

Originally negative sentiments are selected in Table 3:

Text	Sentiment	Prediction LSTM w word2vec	Prediction LSTM w Glove	Prediction BERT base
Omg - I do not recommend these s***ty aftermarket parts. After spending hours on what should have been a routine ape hanger extension, I am thoroughly brassed off! ...	Negative	Positive	Positive	Negative
Call me harsh, but I like to receive what's pictured. Instead of the proper, perfect-fit-every-time crimp style clamps pictured, I received some garbage clamps that you have to fold over and secure by bending tabs...	Negative	Negative	Positive	Positive
If your looking to just use to remove light brake dust it works. For one it to long. Second the bristles are too light and it's flimsy. Would buy again.	Negative	Positive	Negative	Positive
Much smaller than anticipated. Was going to use for our travel trailer but my husband wanted a larger one.	Negative	Positive	Positive	Positive
At first these were amazing bright after a few months they began burning out, over time they completely failed. Even when it is 45 Degrees Fahrenheit outside they would get to hot to touch...	Negative	Negative	Negative	Negative

Table 3: Negative sentiments

9 Summary

This thesis is about a sentiment analysis on product reviews at Amazon which are tokenized according to the applied models. Two models, namely LSTM with two different initializations and BERT base model are introduced to predict the sentiment of the reviews. As the results of predictions suggest, there is no significant difference between the performance of the models. The first place is reached by BERT model with an accuracy ratio of 96.8% closely followed by LSTM with Glove100 initialization for embeddings.

10 References

Books

1. Ian J. Goodfellow and Yoshua Bengio and Aaron Courville: Deep Learning, MIT Press (2016), Cambridge, MA, USA

2. Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, Vipin Kumar: Introduction to data mining, Pearson Education, Inc. (2019), New York

Scientific papers

3. Lan, Zhenzhong and Chen, Mingda and Goodman, Sebastian and Gimpel, Kevin and Sharma, Piyush and Soricut, Radu: ALBERT: A Lite BERT for Self-supervised Learning of Language Representations, arxiv.1909.11942 (2019)(available at: https://arxiv.org/abs/1909.11942)

4. Liu, Yinhan and Ott, Myle and Goyal, Naman and Du, Jingfei and Joshi, Mandar and Chen, Danqi and Levy, Omer and Lewis, Mike and Zettlemoyer, Luke and Stoyanov, Veselin: RoBERTa: A Robustly Optimized BERT Pretraining Approach, arxiv.1907.11692 (2019) (available at https://arxiv.org/abs/1907.11692)

5. Kingma, Diederik P. and Ba, Jimmy: Adam: A Method for Stochastic Optimization, arxiv.1412.6980 (2014)(available at: https://arxiv.org/abs/1412.6980)

6. Sepp Hochreiter, Jurgen Schmidhuber: LONG SHORTTERM MEMORY, Neural Computation 9 (8): 1735-1780, 1997 (available at: http://www.bioinf.jku.at/publications/older/2604.pdf)

7. Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia: Attention Is All You Need, arxiv.1706.03762 (2017)(available at: https://arxiv.org/abs/1706.03762)

8. Pennington, Jeffrey, Socher, Richard, Manning, Christopher: {G}lo{V}e: Global Vectors for Word Representation, In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1532–1543 (2014)(available at: https://aclanthology.org/D14-1162)

9. Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, arxiv.1810.04805 (2018)(available at: https://arxiv.org/abs/1810.04805)

10. Nitish Srivastava, Geoffrey Hinton, AlexKrizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research (2014)(available at: http://jmlr.org/papers/v15/srivastava14a.html)

11. Ioffe, Sergey and Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arxiv.1502.03167 (2015)(available at: https://arxiv.org/abs/1502.03167)

Articles

12. https://paperswithcode.com/sota/sentiment-analysis-on-sst-2-binary

13. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

14. https://towardsdatascience.com/transformers-89034557de14

15. https://towardsdatascience.com/keeping-up-with-the-berts-5b7beb92766

The codes are available at the following Github link:

https://github.com/bodadaniel/Sentiment-analysis