

Text Mining (732A92)

# **Impact of Contextualised and Non-Contextualised Word Embeddings on Classification Performance**

Maximilian Pfundstein (maxpf364)

January 9, 2020

This project investigates the capabilities of contextualised word embeddings provided by bidirectional transformer encoders with DistilBERT as a representative compared to non-contextualised word embeddings with Word2Vec as a representative. As DistilBERT is based on encoders only, a feed-forward network (FFN) and an Long Short-Term Memory (LSTM) are used for the downstream task of text classification. The dataset used are Amazon reviews ranging from one to five stars and all four combinations of word embeddings and classifiers are being tested. The accuracies on the test data set and the models corresponding training times are: The FFN with Word2Vec achieved an accuracy of 43.10 percent, training for 2.3 hours; the FFN with DistilBERT achieved an accuracy of 49.98 percent, training for 11.65 hours; the LSTM with Word2Vec achieved an accuracy of 50.49 percent, training for 6.35 hours; the LSTM with DistilBERT achieved an accuracy of 56.43 percent training for 12.1 hours. Concluding, contextualised word embeddings help for the task of text classification at the cost of a higher computational demand.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	Models . . . . .	6
2.1.1	Feed-Forward Network . . . . .	6
2.1.2	Long Short-Term Memory . . . . .	7
2.2	Word Embeddings . . . . .	9
2.2.1	Word2Vec . . . . .	9
2.2.2	BERT . . . . .	9
2.2.3	DistilBERT . . . . .	11
<b>3</b>	<b>Data</b>	<b>12</b>
3.1	Presentation . . . . .	12
3.2	Preprocessing . . . . .	13
<b>4</b>	<b>Method</b>	<b>15</b>
<b>5</b>	<b>Results</b>	<b>16</b>
<b>6</b>	<b>Discussion</b>	<b>18</b>
<b>7</b>	<b>Conclusion</b>	<b>19</b>
<b>8</b>	<b>Appendix</b>	<b>22</b>
8.1	Model Settings . . . . .	22
8.2	Model Settings DistilBERT . . . . .	23

**Preamble**

This project was carried out as a part of the course Text Mining (732A92) which is part of the Statistics and Machine Learning Master programme at Linköpings University in 2019/2020. The course web page can be found at <https://www.ida.liu.se/~732A92/>.

This report, the source code and all conducted analysis can be found on GitHub at <https://github.com/flennic/text-mining-project/>.

The total amount of words is around 3787.

# 1 Introduction

## 478 words

*Introduce the task or research question that you have addressed in your project. What were you trying to do? Why did you choose this project?*

Within the last years, significant progress based on deep learning has been made in the field of Natural Language Processing (NLP) (Young et al. 2017). Before that, primarily statistical methods have been used (ibid., p. 2). As these methods were prone to the *curse of dimensionality* (ibid., p. 2), the next step that followed was to represent words in such a way, that familiar words were assigned a so called *word embedding* such that related words appear in a familiar context.

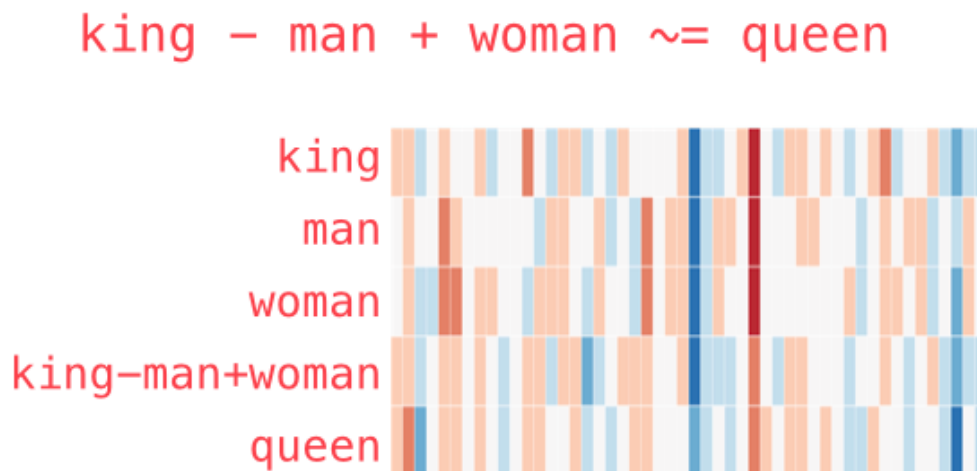


Figure 1: Subtracting the vectors man from king and adding woman almost results in queen.

**Source:** <https://jalammar.github.io/illustrated-word2vec/>

Two known representatives for word embeddings are *Word2Vec* (Mikolov, Chen, et al. 2013; Mikolov, Sutskever, et al. 2013) and *GloVe* (Pennington, Socher, and Manning 2014). *Word2Vec* is either trained based on a *continuous bag of words* model (Mikolov, Chen, et al. 2013, p. 4), predicting a missing word within a sequence, or based on a *skip-gram* model (Mikolov, Sutskever, et al. 2013, p. 2), using a sliding window over a sequence of words. Both approaches for *Word2Vec* are based on neural networks for prediction. *GloVe* embeddings by contrast are based on matrix factorization techniques that create embeddings in such a way that the dot product between two embeddings equals the log of their occurrences within a given context (Pennington, Socher, and Manning 2014, p. 2; ibid., equation 7). An example of embeddings can be seen in figure 1.

These embeddings will have one fixed embedding for each (known) word, independently from their context. We will therefore call these *non-contextualised embeddings*. This implies that for example a word with a meaning dependent on the context will not

always be adequately embedded (Young et al. 2017, p. 5). Therefore the next step was to use embeddings which are aware of their context. One of these models is Embedding from Language Model (ELMo) (Peters et al. 2018), which utilises a bidirectional language model and thereof being aware of words preceding and following the word to embed. We will call these embeddings *contextualised embeddings*.

OpenAI<sup>1</sup> released their *OpenAI Transformer* (Radford 2018) which also creates contextualised embeddings and provides a pre-trained model to work with. The OpenAI Transformer consists just of the decoder part of the originally released *Transformer* (Vaswani et al. 2017). For downstream tasks such as classification a simple classifier, like a feed-forward network (FFN), can then be used. While transformers achieved remarkable results on previous tasks (Radford 2018, p.8), they lost the ability being bi-directional, as ELMo was before.

Bidirectional Encoder Representations from Transformers (BERT), released by Google in May 2019, resolved this problem (Devlin et al. 2018), making transformers bidirectional again, using a technique called masking to prevent peeking at the solution.

While all of these models achieve better results than simpler models, they all rely on deep learning and thereof require a lot of computational resources. This report will investigate how much these achievements over the past years can actually be used in a straight-forward task such as sentence classification.

As the computational demand, even for a pre-trained BERT model, are too demanding for the given resources<sup>2</sup>, DistilBERT (Sanh et al. 2019) by *Hugging Face*<sup>3</sup> is being used.

## 2 Theory

### 1599 words

*Present relevant theoretical background, and in particular the models that you have used. Where appropriate, use mathematical formulas.*

For investigating the influence of non-contextualised and contextualised word embeddings, a simple FFN and an LSTM are being used. In addition to that, both will be trained once based on non-contextualised word embeddings using Word2Vec and once based on contextualised word embeddings from DistilBERT. For simplicity we will refer to the FFN and LSTM as models, to Word2Vec as non-contextualised and to (Distil)BERT as contextualised word embeddings interchangeably. Following from that, we will investigate the following four combinations:

- FFN (Word2Vec)
- LSTM (Word2Vec)
- FFN (DistilBERT)

---

<sup>1</sup><https://openai.com/>

<sup>2</sup>Mainly constrained by 16GiBs of memory and 8GiB of video memory.

<sup>3</sup><https://huggingface.co/>

- LSTM (DistilBERT)

## 2.1 Models

### 2.1.1 Feed-Forward Network

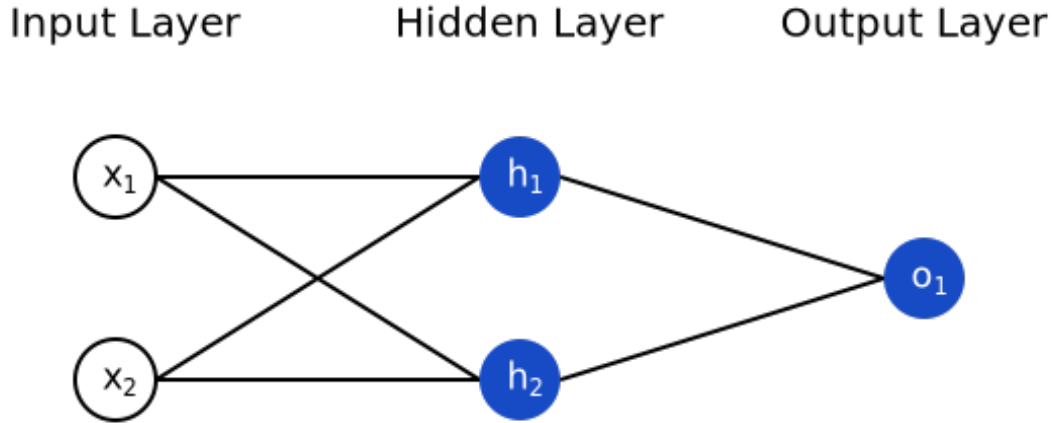


Figure 2: Illustrative feed forward network.

**Source:** <https://victorzhou.com/blog/intro-to-neural-networks/>

A FFN consists of multiple layers where the information only flows in one direction. Usually the input is fed to a first layer, which is a linear combinations of the inputs and the weights of the first layer.  $X$  is the input and  $W_i$  are the weights of the  $i$ th layer including the bias. Then an activation function is applied to map the output back to a specific range, usually the Sigmoid activation function or a rectified linear (ReLU) function:

$$\hat{Y}_i = \sigma(X^T * W_i) \quad (1)$$

This is usually called a dense layer which are stacked and thereby build the model architecture. The intermediate layers are usually called *hidden* layers as they cannot be seen directly from the outside. Figure 2 shows an illustrative FFN. Depending on the problem, the last activation layer might be skipped. Afterwards, an error function to optimise later on is being applied to the output. This whole process is called *forward pass*. To update the weights, a *backward pass* is being applied to update the weights. The error depends on the weights and can be written as:

$$\nabla E = \left( \frac{\delta E}{\delta W_n}, \frac{\delta E}{\delta W_{n-1}}, \dots, \frac{\delta E}{\delta W_1} \right) \quad (2)$$

For updating one layer of weights, the partial derivative is being calculated. For the last layer, here denoted as  $W_n$ , the gradients and this the update would look like the following:

$$\frac{\delta E}{\delta W_n} = \frac{\delta E}{\delta \hat{Y}} \frac{\delta \hat{Y}}{\delta A_n} \frac{\delta A_n}{\delta W_n} \quad (3)$$

$A_n$  is the intermediate value between the linear combination and the applied activation function. The weights for this layer can then be updated with the following formula:

$$W_n^{updated} = W_n - \alpha \odot \frac{\delta E}{\delta W_n} \quad (4)$$

The remaining layers are updated accordingly, the chain rule just yields in longer equations. Deep neural networks are prone to the vanishing gradient problem (Pascanu, Mikolov, and Bengio 2012).

### 2.1.2 Long Short-Term Memory

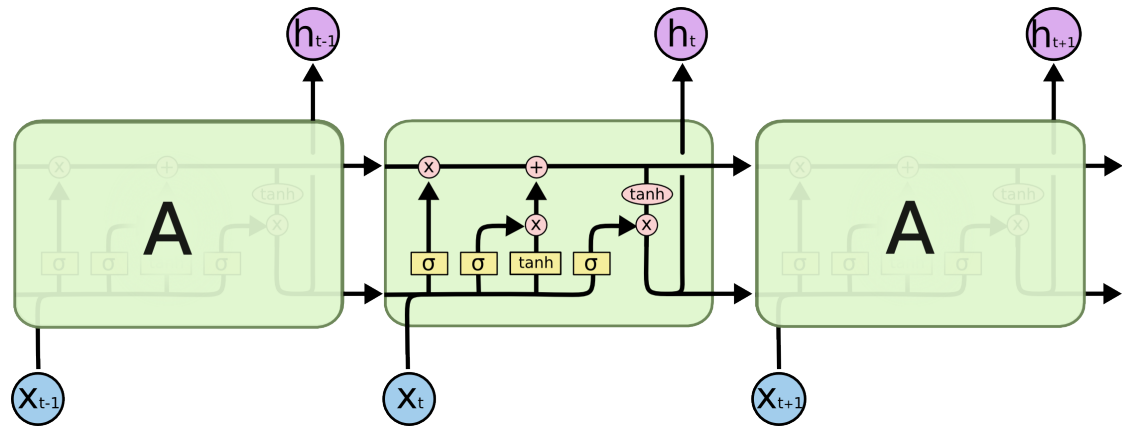


Figure 3: Unfolded LSTM showing the information flow.

**Source:** <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

An LSTM belongs to the field of recurrent neural networks (RNNs) and handles long term relationships better than a classical RNN (Hochreiter and Schmidhuber 1997, abstract). As text can be seen as a sequence with long term dependencies, LSTMs are suitable for catching those long term dependencies. LSTMs, as well as RNNs, look at one element of the sequence (here a word) and in addition to the normal output vector the old cell state is passed on to the cell handling the second element of the sequence. Utilising this hidden state, information can be preserved and pass on. An unfolded LSTM can be seen in figure 3.

In each time step, an LSTM cell sees the Long Term Memory (LTM), denoted as  $C_{t-1}$ , and the Short Term Memory (STM), denoted as  $h_{t-1}$ , thus the output from the previous cell (previous time step). Additionally it sees the current element of the sequence, denoted  $x_t$ . It then produces an output  $h_t$ , the LTM  $C_t$  and STM  $h_t$  for the next cell.

This process is enabled by an architecture consisting of four gates, the *forget gate*, the *learn gate*, the *remember gate* and the *use gate*.<sup>4</sup>

**Learn Gate** The learn gate first takes the new element  $x_t$  and combines it with the STM ( $h_{t-1}$ ) using concatenation, then passing it through a  $\tanh$  activation function, after taking a linear combination with its weights  $W_n$ . This output is denoted as  $N_t$ . Next it is multiplied by an ignore factor  $i_t$ , making it ignore unimportant information. The ignore factor is the Sigmoid activation function applied to the linear combination between  $W_i$  and the previously concatenated values. The output is given as  $N_t i_t$  with:

$$N_t = \tanh(W_n(h_{t-1}, x_t)) \quad (5)$$

$$i_t = \sigma(W_i(h_{t-1}, x_t)) \quad (6)$$

**Forget Gate** The forget gate looks at the previous LTM memory  $C_{t-1}$  and decides which information is kept and which to throw away. A forget factor  $f_t$  is being calculated. Therefore the output of the forget gate is given as  $C_{t-1} f_t$ . The forget factor is the linear combination of the weights  $W_f$  multiplied by the concatenated elements of the sequence  $x_t$  and the previous STM  $h_{t-1}$ . Then the Sigmoid activation function is applied:

$$f_t = \sigma(W_f(h_{t-1}, x_t)) \quad (7)$$

**Remember Gate** The remember gate looks at the previous LTM  $C_{t-1}$  and STM  $h_{t-1}$  and then calculates the new LTM state  $C_t$ , utilising the previously calculated ignore and forget factors  $f_t$  and  $i_t$ :

$$C_t = C_{t-1} f_t + N_t i_t \quad (8)$$

**Use Gate** The use gate first applies the  $\tanh$  activation function to the linear combination  $W_u$  of the output of the forget gate ( $C_{t-1} f_t$ ) to calculate  $U_t$ . It then calculates  $V_t$

---

<sup>4</sup>Explanations based on the course "Intro to Deep Learning with PyTorch" on Udacity. Biases have been omitted, as they're included in the weights, variables have been partly renamed to fit the figure. Also <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> and the official paper (Hochreiter and Schmidhuber 1997) have been used.



by taking the Sigmoid activation function of the linear combination of  $W_v$  and the concatenation of STM  $h_{t-1}$  and the current element of the sequence  $x_t$ . Then, the output of the use gate is defined as  $U_t V_t$ .

$$U_t = \tanh(W_u C_{t-1} f_t) \quad (9)$$

$$V_t = \sigma(W_v(h_{t-1}, E_t)) \quad (10)$$

The advantage of this architecture is that the previous outputs and hidden states can be treated as fixed variables and the back propagation (backward pass), does not have to be calculated for the whole sequence. In practice libraries such as PyTorch use "reverse-mode automatic differentiation" (Paszke et al. 2019). Also due to the limited scope of this report, backpropagation for LSTMs will not be further investigated.

## 2.2 Word Embeddings

### 2.2.1 Word2Vec

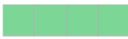

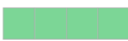



Word2Vec (Mikolov, Chen, et al. 2013; Mikolov, Sutskever, et al. 2013) is being used for the non-contextualised word embeddings. For creating the embeddings, first two matrices called the *embedding matrix* and the *context matrix* are being created. Their size also determines the dimensionality of the word embeddings later on. In each learning step a sample is taken from the embeddings and  $n$  negative samples, not occurring in the skip-gram of the original word are taken. Thus process is called *negative sampling*. The positive sample is looked up in the embeddings matrix, the negatives samples are looked up in the context matrix. Then the dot product between the positive and each negative sample is being taken (so we have  $n$  dot products). As each word embedding has the size  $1 \times \text{embedding size}$ , the dot product will result in a scalar, to which the Sigmoid activation function is being applied. The result is then compared with the desired output (e.g. 1 for the positive word, 0 for the negative words). Figure 4 illustrates the intermediate steps. This whole process is conducted by a neural network, so it's easy to calculate the error and to update the weights. The weights are in this case the embedding and the context matrices. When the whole process is done, we take the embeddings matrix as the word embeddings. It's size is number of words  $\times$  embedding size. The number of words depends on the corpora and the minimum amount of appearances to count in a word.

### 2.2.2 BERT

BERT (Devlin et al. 2018) is an encoder based on bidirectional transformers<sup>5</sup>. Transformers use attention heads (Vaswani et al. 2017) for focusing on different parts or

---

<sup>5</sup>Transformers will not be explained in this report as the number of words is quite limited. For a more in-depth understanding this blog post is recommended: <https://jalammargithub.io/illustrated-transformer/>

input word	output word	target	input • output	sigmoid()	Error
not 	thou 	1	0.2	0.55	0.45
not 	aaron 	0	-1.11	0.25	-0.25
not 	taco 	0	0.74	0.68	-0.68

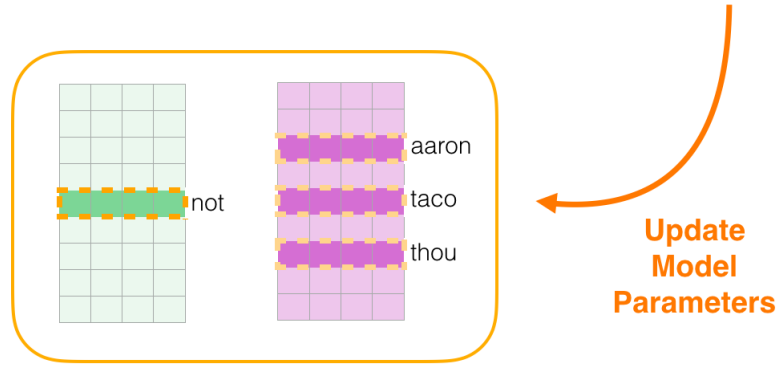


Figure 4: Training process of Word2Vec embeddings.

Source: <https://jalammar.github.io/illustrated-word2vec/>

words of a sentence. For example, if the word *it* is being processed, the attention heads will give focus to different parts of the whole sequence. This way the model can know, which entity is meant by *it*. The advantage given is that a sequence is not read from one side to the other, but the whole sequence is processed at once. This is also called *masked LM* or *masked language modeling* (Devlin et al. 2018, p. 4).

BERT can unambiguously represent single sentences and pair of two sentences (e.g. question/answer) (ibid., p. 4) and sentences are differentiated by a special [SEP] token. BERT is trained on two tasks: *MLM* (prediction of the real words behind the [MASK] tokens) and Next Sentence Prediction (NSP).

**MLM** During training of BERT, before a sequence is passed to BERT, 15 percent of it's elements are replaced by special [MASK] tokens, the input is partly shuffled and a special classification token [CLS] is being added at the beginning of the tokenised sentence (ibid., p. 4). BERT then learns to predict the masked tokens. Figure 5 illustrates the process.

**NSP** Given two sentences A and B, BERT has to predict the likelihood of sentence B actually following sentence A (ibid., p. 4). Figure 6 illustrates the process.

As BERT is just an encoder, another classifier for downstream tasks such as classification is compulsory. The encoder outputs between the different can be used as contextualised word embeddings. Figure 7 shows their corresponding  $F_1$  scores on the

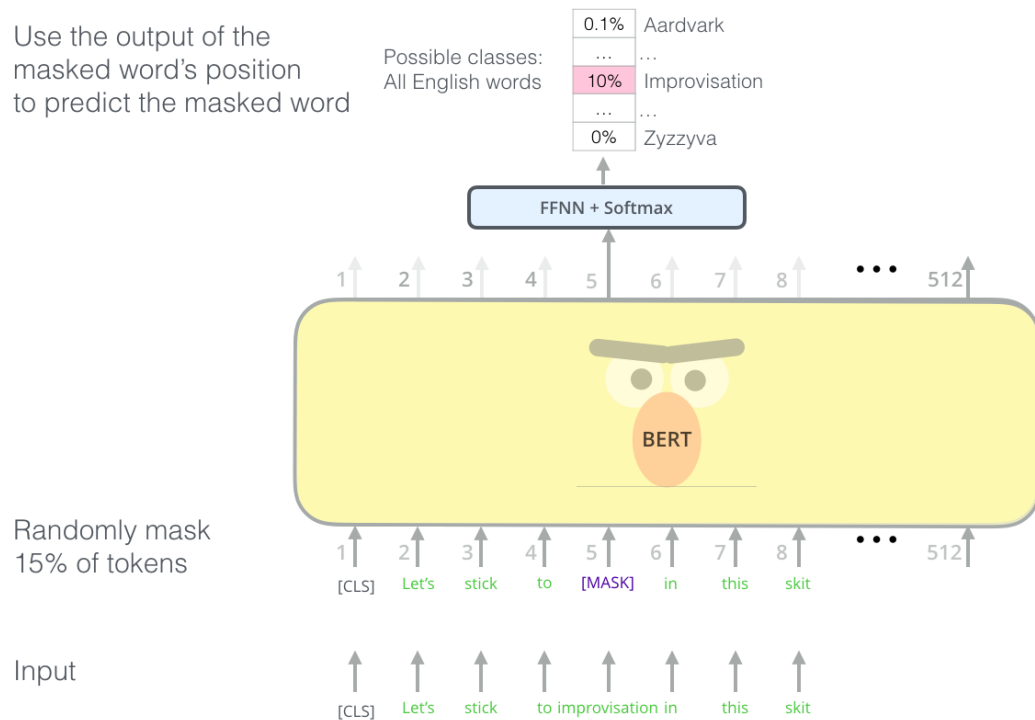


Figure 5: BERT for masked language modeling.

Source: <https://jalammar.github.io/illustrated-bert/>

development set during training of BERT. We will use the output of the last encoding layer in this project as the embedding which achieved an  $F_1$  score of 94.9 percent. The specified [CLS] token, originally thought for classification, will not be used as the aim is not to achieve a high accuracy, but rather to compare non-contextualised and contextualised embeddings.

### 2.2.3 DistilBERT

As even the smallest BERT model has high computational demands, *DistilBERT* (Sanh et al. 2019) is used for this project. Distillation is a process in which a smaller model tries to mimic the behaviour of a larger model in such a way, that it behaves almost the same. The concept was introduced by Buciluundefined, Caruana, and Niculescu-Mizil 2006 and later generalised by Hinton, Vinyals, and Dean 2015. The smaller model can also be an ensemble of models (similarly to the concepts of AdaBoost). The result is a model that is 40 percent smaller, 60 percent faster while keeping 97 percent of BERT understanding capabilities (Sanh et al. 2019, p. 5).

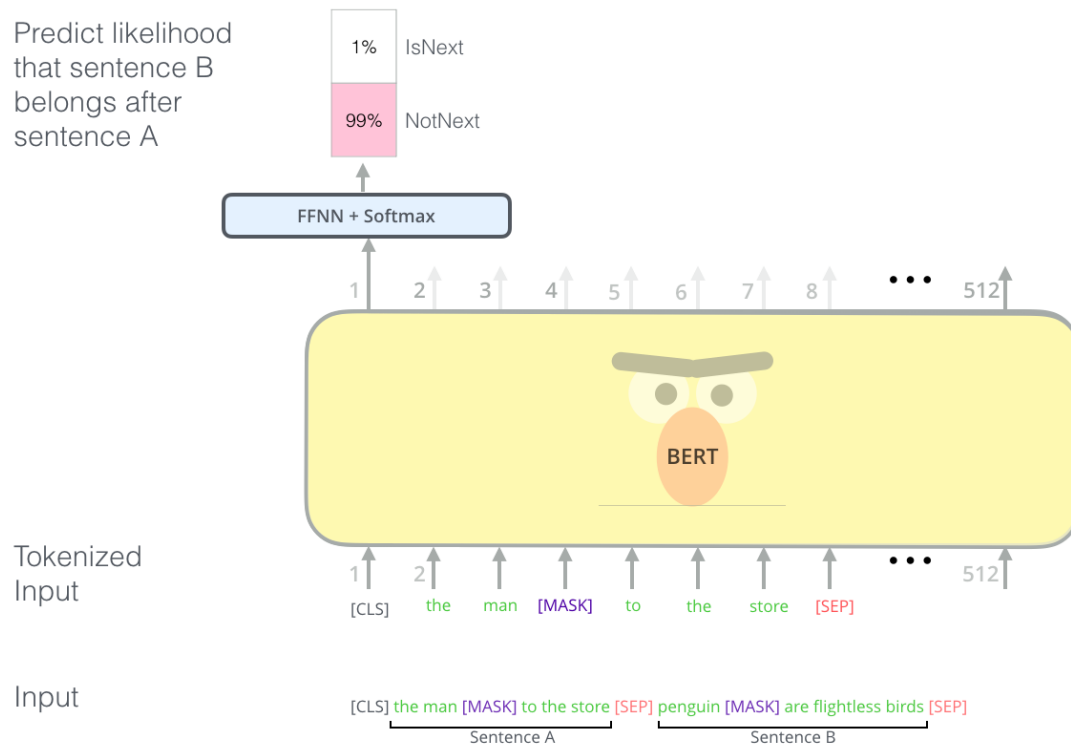


Figure 6: Bert for next sentence prediction.

Source: <https://jalammar.github.io/illustrated-bert/>

## 3 Data

334 words

### 3.1 Presentation

*Present your data. What information does it contain? Where did you get it from? What preprocessing did you do, if any?*

The data used in this project consists of Amazon reviews categorised by one to five stars and is provided by Xiang Zhang<sup>6</sup>. The data is freely available and can be downloaded on Google Drive: [amazon\\_review\\_full\\_csv.tar.gz](#).

Originally the data is divided in training data with 3 000 000 data points and test data with 650 000 data points. Each data point consists of a label, indicating how many stars the review has, a title and the review text. One row looks like this:

<sup>6</sup><http://xzh.me/>

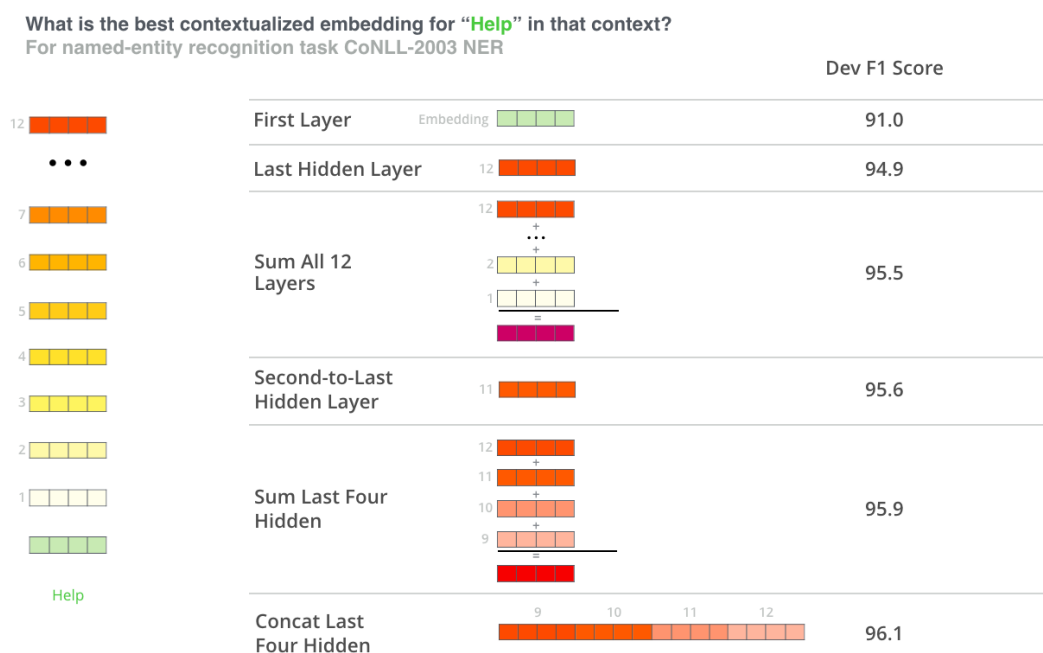


Figure 7: Using BERT for contextualised word embeddings.

Source: <https://jalammar.github.io/illustrated-bert/>

"3", "more like funchuck", "Gave this to my dad for a gag gift after directing "Nunsense, " he got a reall kick out of it!"

The class distribution can be seen in figure 8 and the distribution of the review lengths (words) can be seen in figure 9. The percentiles can be found in table 1.

Percentiles	91	92	93	94	95	96	97	98	99
Review Length	144	147	150	153	157	161	165	170	177

Table 1: Percentiles and their corresponding reviews lengths.

## 3.2 Preprocessing

The original training and test data sets are read, concatenated and shuffled. Then the column for the *title* of the review is being dropped and all labels are shifted by  $-1$  so that the labels are 0, 1, 2, 3, 4. Next, the data is split into 85 percent training, 10 percent validation and 5 percent test data. For the following sentences a padding of 200 has

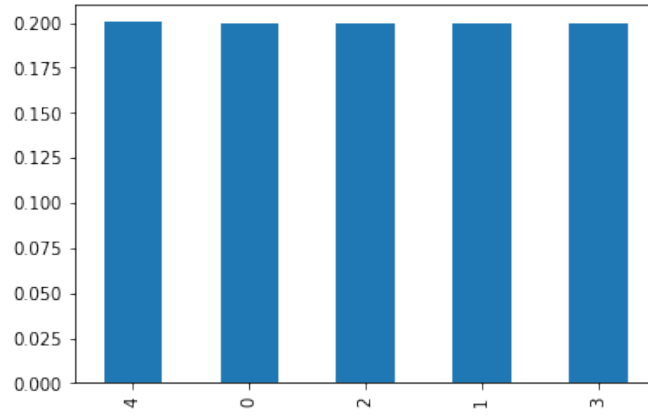


Figure 8: Class distribution.

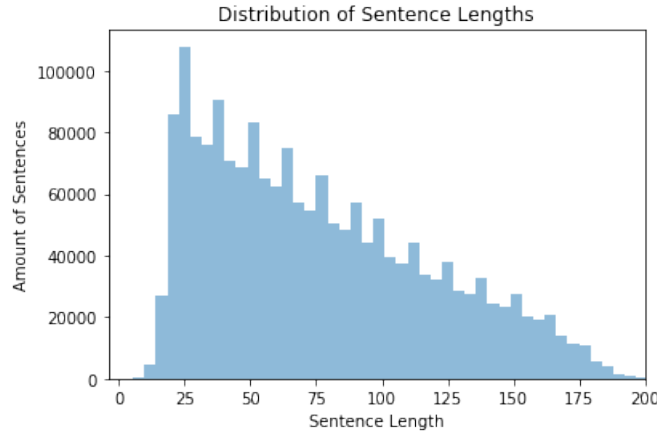


Figure 9: Distribution of review lengths.

been used<sup>7</sup>. This lies way above the 99th percentile of 177 words as it accounts for that during the tokenisation the sentences become longer due to punctuation and as well as some words becoming separated.

The padding is applied after tokenisation. The tokeniser used for both, Word2Vec and BERT, is the DistilBERT tokeniser provided by the HuggingFace library<sup>8</sup>. For Word2Vec, only the 1 000 000 most common embeddings are taken due to computational constraints. Therefore all other word are considered unknown.

After preprocessing, the three data sets are saved as `.csv` files where the first 200 columns hold the word Ids  $X$  and column 201 holds the label  $Y$ .

<sup>7</sup>Normally only the training data should be used for deciding the padding. Due to time constrains this wasn't changed. As 85 percent randomly chosen data is used for training, the result is likely to not be that much different.

<sup>8</sup><https://huggingface.co/transformers/model.doc/distilbert.html#distilberttokenizer>

## 4 Method

### 357 words

*Explain how you carried out your study. Aim to be detailed enough for others to reproduce your results.*

All four model combinations are implemented in PyTorch. The source code and detailed instructions for replicating the results can be found in the corresponding GitHub repository<sup>9</sup>.

The first step is to create the non-contextualised or contextualised embeddings for the review. In the case of Word2Vec an embedding layer<sup>10</sup> is used, mapping the token Ids to non-contextualised word-embeddings of shape  $(200 \times 300)$  where 200 is the padding length and 300 the word embedding size of Word2Vec. The embeddings are not frozen, so the embeddings can change during the training process. This is to account for the limitation of not using all embeddings. The embeddings are loaded to the embeddings layer using the Gensim library (Řehůřek and Sojka 2010). For the contextualised embeddings, the tokenised sentences are fed to DistilBERT<sup>11</sup> and the output of the last encoder is then taken as the embedding. The pretrained model *distilbert-base-uncased* is used and all weights are frozen during training.

The LSTM directly takes the sequences as in input whereas for the FFN the sequences must first be flattened, thus resulting in 60 000 inputs. The model architectures are given by the following. The model parameters can be found in the appendix at 8.1.

**FFN (Word2Vec)**  $X \rightarrow \text{Embedding} \rightarrow \text{Flattening} \rightarrow \text{Linear} \rightarrow \text{Sigmoid} \rightarrow \text{Dropout} \rightarrow \text{Linear} \rightarrow \text{LogSoftMax}$

**LSTM (Word2Vec)**  $X \rightarrow \text{Embedding} \rightarrow \text{LSTM} \rightarrow \text{Dropout} \rightarrow \text{Linear} \rightarrow \text{Sigmoid} \rightarrow \text{LogSoftMax}$

**FFN (DistilBERT)**  $X \rightarrow \text{DistilBERT} \rightarrow \text{Flattening} \rightarrow \text{Linear} \rightarrow \text{Sigmoid} \rightarrow \text{Dropout} \rightarrow \text{Linear} \rightarrow \text{LogSoftMax}$

**LSTM (DistilBERT)**  $X \rightarrow \text{DistilBERT} \rightarrow \text{LSTM} \rightarrow \text{Dropout} \rightarrow \text{Linear} \rightarrow \text{Sigmoid} \rightarrow \text{LogSoftMax}$

The last linear layer is always used to map to the five classes of Amazon reviews. Furthermore, dropout (Srivastava et al. 2014) is being applied for regularisation and the gradients for the LSTM are clipped at a value of 5 to prevent exploding gradients (Pascanu, Mikolov, and Bengio 2012). Adam (Kingma and Ba 2014) is used as the optimiser and cross-entropy loss as the loss function.

---

<sup>9</sup><https://github.com/flennic/text-mining-project>

<sup>10</sup><https://pytorch.org/docs/stable/nn.html?highlight=embedding#torch.nn.Embedding>

<sup>11</sup>[https://huggingface.co/transformers/model\\_doc/distilbert.html](https://huggingface.co/transformers/model_doc/distilbert.html)

As the data set is balanced, accuracy is used as it provides a single value to estimate the models performance.

## 5 Results

### 395 words

*Present your results in an objective way. Use tables and charts, but do not forget to also include a summary in text form. Do not interpret your results.*

All four model combinations and their achieved accuracies can be seen in table 2. The accuracies for training have dropout enabled<sup>12</sup> and are trailing accuracies. This means that the accuracy will jump between the epochs, as the model is usually less accurate during the beginning of an epoch and the trailing training accuracy is always the average until the current batch. Figure 11 shows the accuracies for the FFN with Word2Vec embeddings for training, validation and test for different epochs, figure 12 shows the accuracies for the FFN with DistilBERT, figure 13 shows the accuracies for the LSTM with Word2Vec embeddings and figure 14 shows the accuracies for the LSTM with DistilBERT embeddings. The lowest baseline accuracy is at 20.00 percent which could be achieved by simply predicting one class at all times.

Model — Data	Training	Validation	Testing
FFN (W2V)	52.80%	52.82%	43.10%
FFN (BERT)	49.88%	49.88%	49.98%
LSTM (W2V)	50.14%	50.08%	50.49%
LSTM (BERT)	56.13%	56.14%	56.43%

Table 2: Highest accuracies for the different setups. Note that training has dropout enabled.

It can be seen that the FFN with Word2Vec embeddings has the lowest testing score with a test accuracy of 43.10 percent. The FFN with using DistilBERT embeddings achieves a higher test accuracy of 49.98 percent, closely followed by the LSTM using Word2Vec embeddings with a test accuracy of 50.49 percent. The highest accuracy is achieved by the LSTM in combination with DistilBERT embedding and is 56.43 for the test data set. For all model combinations except the FFN using Word2Vec embeddings, the test accuracy is slightly higher compared to validation, which seems a bit odd, but this behaviour can be seen for all these other models. For the FFN using Word2Vec the test accuracy is way lower than the training or validation accuracy<sup>13</sup>.

Figure 10 shows the training times for the model combinations. The models using non-contextualised word embeddings (Word2Vec) are trained for 30 epochs and the models using contextualised word embeddings (DistilBERT) are trained for 3 epochs.

<sup>12</sup>Enabling another prediction without dropout would have been too time consuming for this project.

<sup>13</sup>Due to these results the source code has been investigated but no issues have been found. Therefore these results are treated as valid.



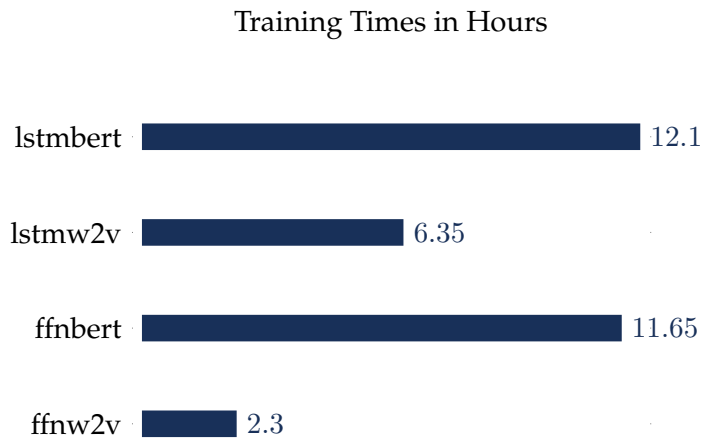


Figure 10: Training times for the different model combinations.

Both models using DistilBERT word embeddings have the highest training times with 12.1 hours for the LSTM and 11.65 hours for the FFN. Around half of the time is needed for the LSTM with Word2Vec word embeddings and a training time around 6.35 hours. The lowest training time is achieved by the FFN with Word2Vec embeddings and takes just 2.3 hours.

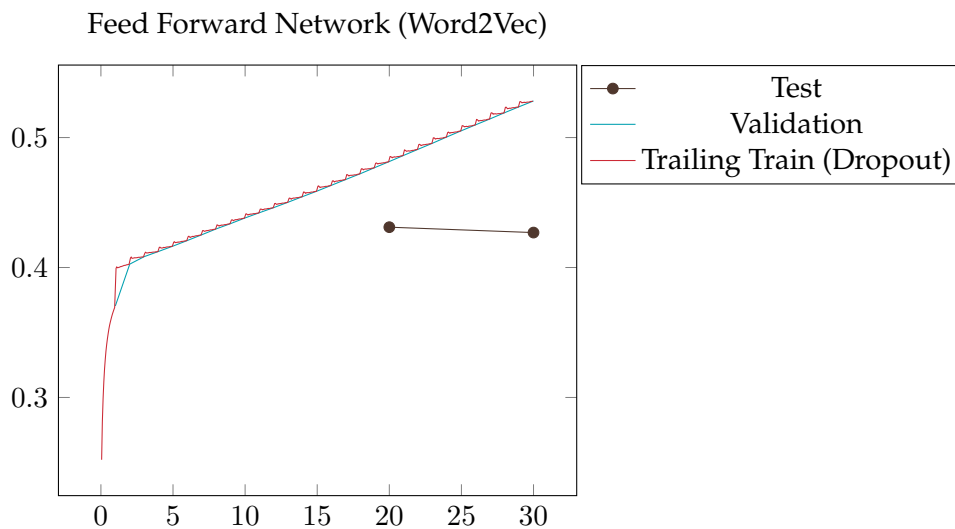


Figure 11: Feed Forward Network (Word2Vec)

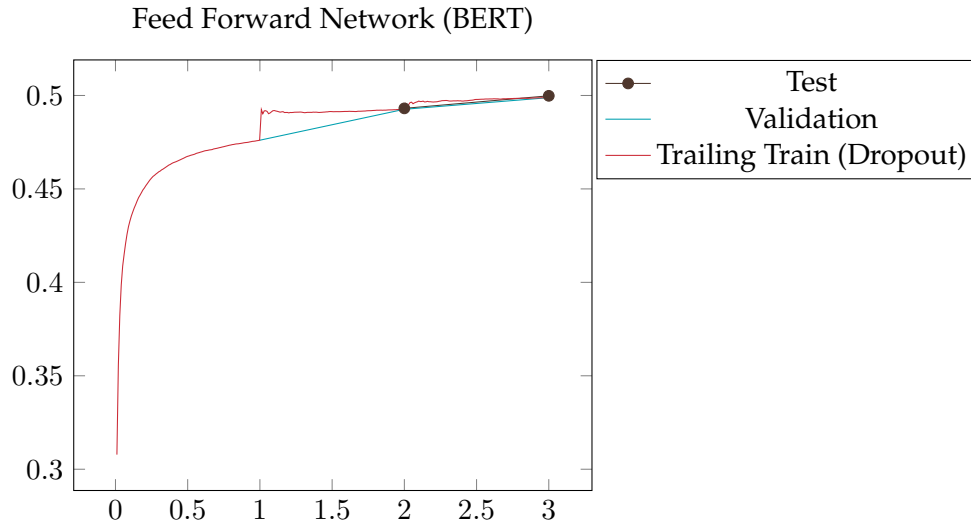


Figure 12: Feed Forward Network (BERT)

## 6 Discussion

### 424 words

*What do you make of it? Discuss your results and present your analysis. Relate your results to your task/question.*

The results show that contextualised word embeddings provide a higher accuracy, with a difference of almost 7 percent points for the FFN and almost 6 percent for the LSTM. Apart from achieving a higher accuracy also less iterations though the training data set are necessary for not only achieving the same, but better results. This makes the trained model less prone to overfit the data. Each training sample, in this case, is seen 40 times compared to the FFN using Word2Vec word embeddings and 20 times compared to the LSTM using Word2Vec word embeddings compared to their non-contextualised counterparts. Note that for this factor not only the difference in epochs must be taken in, but also the amount of data loaders used. For example, when 2 data loaders are used, the whole data set is seen twice during one epoch<sup>14</sup>. The overfit can be especially seen for the FFN using Word2Vec embeddings. Looking at figure 11 it can also be seen that the test accuracy becomes lower going from 20 to 30 training epochs.

It can also be seen that using contextualised word embeddings the training times for the models are a lot higher, the biggest difference between the LSTM using DistilBERT embeddings and the FFN using Word2Vec embeddings with almost a factor of 6. One interesting finding is that for the Word2Vec embeddings the LSTM increases the training time by a factor around 3, whereas for the DistilBERT embeddings the difference is almost negligible. The reason for that could be a parallelisation during the training process, as the implementation used here utilised multiple data loaders. This could be

<sup>14</sup>This is due to lazy loading the data from disk as the whole data does not fit into memory.

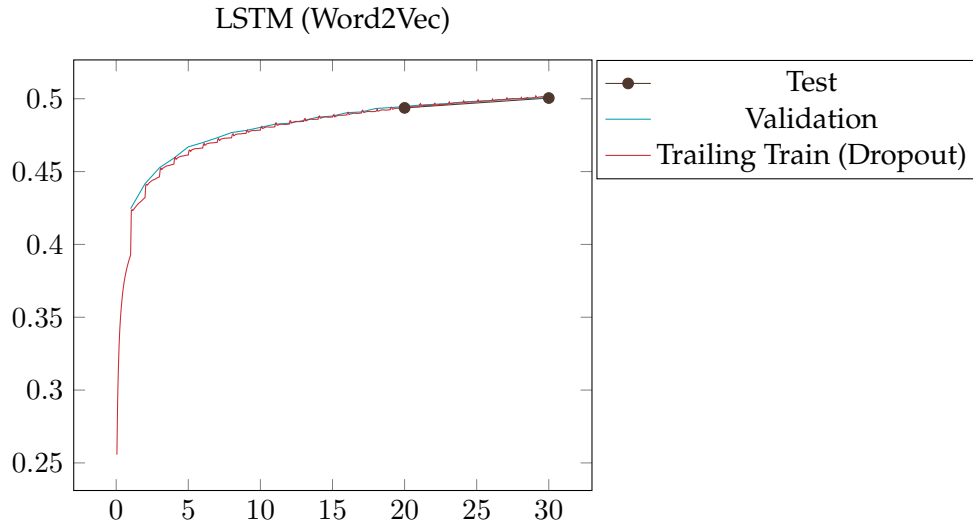


Figure 13: LSTM (Word2Vec)

taken as an advantage in the future, as parallelised hardware is easier scalable (see core counts for CPU and GPU with are basically made for parallel execution) than single thread performance. Another interesting finding is that, despite the fact that DistilBERT basically is a bidirectional encoder, still an LSTM is necessary to extract all the information from the contextualised embeddings. The FFN using the word embeddings from DistilBERT is way less accurate than the equivalent LSTM. Also note, that due to computational limitations, this is a rather simple LSTM with just two layers, a small hidden state and foremost, it is just unidirectional. Therefore one presumption is that a larger BiLSTM might achieve even higher accuracies.

For the results acquired in this project, the only hyperparameter search that was conducted, was manual tweaking the learning rate.

## 7 Conclusion

### 190 words

*Based on your results and their analysis, what new knowledge do you take away from your project?*

Contextualised word embeddings are indeed more powerful than their non-contextualised counterparts. The price for that is a high computational demand. Some of the additional demanded computational power is compensated by parallelisation, still it's high. Especially considering that DistilBERT and not BERT is being used. DistilBERT is not only 60 percent faster (Sanh et al. 2019, p. 5), but also 40 percent smaller, so the batch size can be increased which makes it even faster compared to BERT.

With deep learning already requiring a lot of computational power, these new models taking it to the next level, which is indeed a problem. The problem is not only the

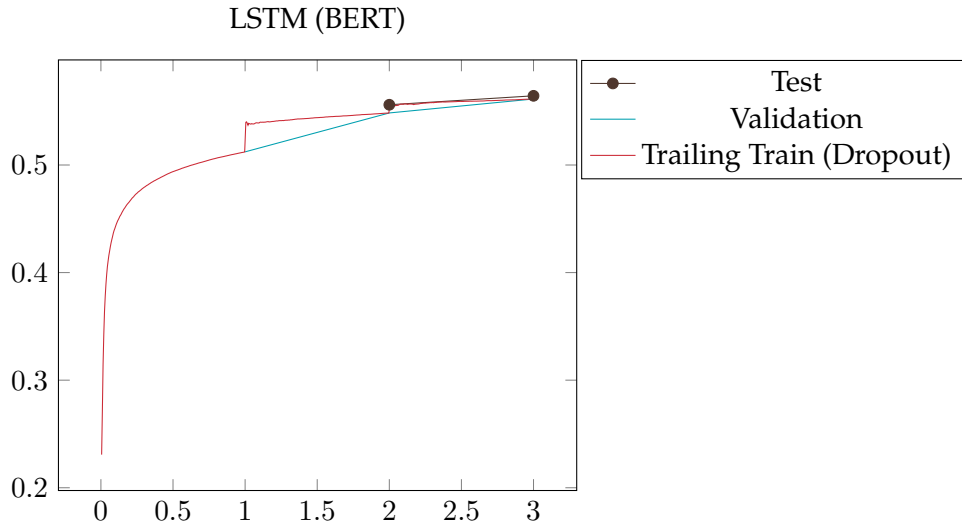


Figure 14: LSTM (BERT)

price for the required hard, but also environmental issues. However, these new transformers indeed capture more information and might push the domain of NLP forward. It can also be seen that, at least to some degree, the research society is aware of this problem and trying to reduce computation demands. Here DistilBERT is used, Google also released ALBERT (**2019albert**) which is a lighter version of BERT.

Still the question remains, if throwing hardware at our problems will be a long-term solution for the domain of NLP.

## References

- Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). “Long Short-Term Memory”. In: *Neural Comput.* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- Buciluundefined, Cristian, Rich Caruana, and Alexandru Niculescu-Mizil (2006). “Model Compression”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’06. Philadelphia, PA, USA: Association for Computing Machinery, pp. 535–541. ISBN: 1595933395. DOI: 10.1145/1150402.1150464. URL: <https://doi.org/10.1145/1150402.1150464>.
- Řehůřek, Radim and Petr Sojka (May 2010). “Software Framework for Topic Modelling with Large Corpora”. In: pp. 45–50. DOI: 10.13140/2.1.2393.1847.
- Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2012). *On the difficulty of training Recurrent Neural Networks*. arXiv: 1211.5063 [cs.LG].
- Mikolov, Tomas, Kai Chen, et al. (2013). *Efficient Estimation of Word Representations in Vector Space*. arXiv: 1301.3781 [cs.CL].
- Mikolov, Tomas, Ilya Sutskever, et al. (2013). *Distributed Representations of Words and Phrases and their Compositionality*. arXiv: 1310.4546 [cs.CL].
- Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].
- Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014). “Glove: Global Vectors for Word Representation.” In: *EMNLP*. Vol. 14, pp. 1532–1543.
- Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15, pp. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean (2015). *Distilling the Knowledge in a Neural Network*. arXiv: 1503.02531 [stat.ML].
- Vaswani, Ashish et al. (2017). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL].
- Young, Tom et al. (2017). *Recent Trends in Deep Learning Based Natural Language Processing*. arXiv: 1708.02709 [cs.CL].
- Devlin, Jacob et al. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].
- Peters, Matthew E. et al. (2018). *Deep contextualized word representations*. arXiv: 1802.05365 [cs.CL].
- Radford, Alec (2018). “Improving Language Understanding by Generative Pre-Training”. In:
- Paszke, Adam et al. (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems* 32. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: <http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- Sanh, Victor et al. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. arXiv: 1910.01108 [cs.CL].

## 8 Appendix

### 8.1 Model Settings

```
1 {
2   "orig_train_path": "data/original/train.csv",
3   "orig_test_path": "data/original/test.csv",
4   "processed_data_folder": "data/processed/",
5   "cached_model_path":
6     ↳ "checkpoints/2019-12-17_15-40_LstmWord2VecModelInteractor.pth",
7   "word2vec_path":
8     ↳ "data/embeddings/GoogleNews-vectors-negative300.bin",
9   "splits": [
10     0.85,
11     0.1,
12     0.05
13   ],
14   "padding": 200,
15   "embeddings": 1000000,
16   "categories": 5,
17   "run_model": "lstm_w2v",
18   "load_cached_model": false,
19   "models": {
20     "ffn_w2v": {
21       "data_loader_workers": 4,
22       "batch_size": 8192,
23       "learning_rate": 0.0001,
24       "epochs": 30,
25       "embedding_size": 300,
26       "dropout": 0.25,
27       "hidden": 256
28     },
29     "lstm_w2v": {
30       "data_loader_workers": 2,
31       "batch_size": 1024,
32       "learning_rate": 5e-05,
33       "epochs": 30,
34       "embedding_size": 300,
35       "dropout": 0.25,
36       "lstm_layers": 2,
37       "lstm_hidden": 128,
38       "lstm_dropout": 0.25,
39       "gradient_clip": 5
40     },
41     "ffn_bert": {
42       "data_loader_workers": 1,
43       "batch_size": 256,
44       "learning_rate": 0.0001,
```

```

43         "epochs": 3,
44         "embedding_size": 768,
45         "dropout": 0.25,
46         "hidden": 256,
47         "max_batches_per_epoch": 64
48     },
49     "lstm_bert": {
50         "data_loader_workers": 1,
51         "batch_size": 256,
52         "learning_rate": 5e-05,
53         "epochs": 3,
54         "embedding_size": 768,
55         "dropout": 0.25,
56         "lstm_layers": 2,
57         "lstm_hidden": 128,
58         "lstm_dropout": 0.25,
59         "gradient_clip": 5
60     }
61 },
62 "device": null,
63 "seed": 42,
64 "log_level": 20,
65 "cache": true
66 }

```

## 8.2 Model Settings DistilBERT

```

1  {
2      "activation": "gelu",
3      "attention_dropout": 0.1,
4      "dim": 768,
5      "dropout": 0.1,
6      "finetuning_task": null,
7      "hidden_dim": 3072,
8      "initializer_range": 0.02,
9      "is_decoder": false,
10     "max_position_embeddings": 512,
11     "n_heads": 12,
12     "n_layers": 6,
13     "num_labels": 2,
14     "output_attentions": false,
15     "output_hidden_states": false,
16     "output_past": true,
17     "pruned_heads": {},
18     "qa_dropout": 0.1,
19     "seq_classif_dropout": 0.2,
20     "sinusoidal_pos_embs": false,

```

```
21 "tie_weights_": true,  
22 "torchscript": false,  
23 "use_bfloat16": false,  
24 "vocab_size": 30522  
25 }
```