Text Mining (732A92)

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

Maximilian Pfundstein (maxpf364)

January 12, 2020

This project investigates the capabilities of contextualised word embeddings provided by DistilBERT as a bidirectional transformer encoder compared to non-contextualised word embeddings provided by Word2Vec. As DistilBERT is based on encoders only, a feed-forward network (FFN) and an Long Short-Term Memory (LSTM) are used as models for the downstream task of text classification. The dataset used, are Amazon reviews ranging from one to five stars and the performance of all four combinations of word embeddings and classifiers are being tested with the reviews. 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

1 Introduction 4

2 Theory 5

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 10

2.2.3 DistilBERT 11

3 Data 12

3.1 Presentation 12

3.2 Preprocessing 13

4 Method 15

5 Results 16

6 Discussion 17

7 Conclusion 19

8 Appendix 22

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 approximately 3911.

1 Introduction

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). Prior to that, foremost 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.

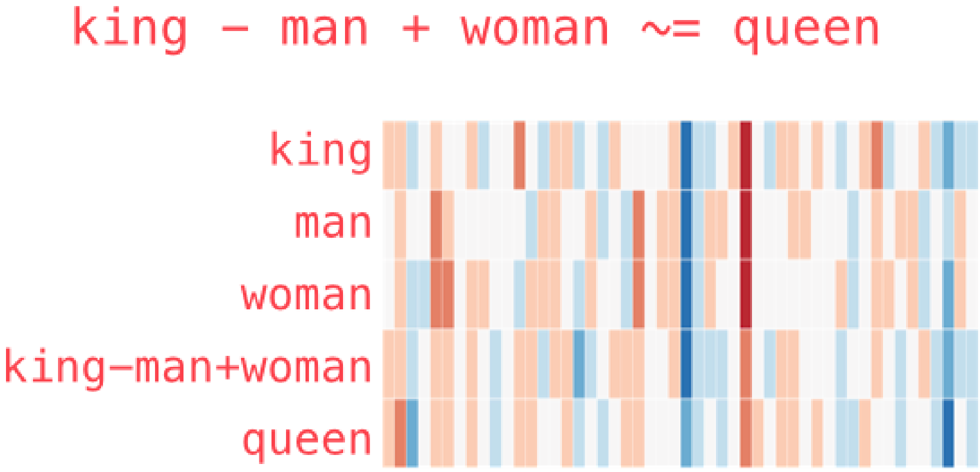


Figure 1: Subtracting the vectors man from king and adding woman almost results in queen.
Source: https://jalammar.github.io/illustrated-word2vec/

Two known methods 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 learning. GloVe embeddings in contrast are based on matrix factorisation techniques which 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, illustrating the embedded information and it's similarity.

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 its context might not always be adequately embedded (Young et al. 2017, p. 5). Therefore the next step was to use embeddings which take the context into consideration. 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 such embeddings *contextualised* embeddings.

OpenAI[1] released the *OpenAI Transformer* (Radford 2018) which also creates contextualised embeddings and provides a pre-trained model to work with. The OpenAI Transformer consists only 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 such as question answering (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 thereby 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, by comparing non-contextualised and contextualised word embeddings.

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

2 Theory

For investigating the influence of non-contextualised and contextualised word embeddings, a simple FFN and an LSTM are being used. Additionally, 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 FNN 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)
- LSTM (DistilBERT)

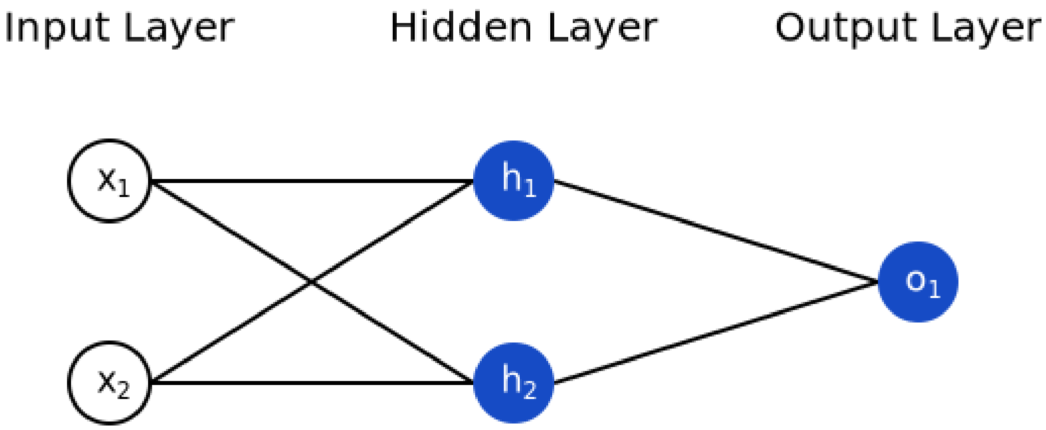


Figure 2: Illustrative feed forward network.
Source: https://victorzhou.com/blog/intro-to-neural-networks/

2.1 Models

2.1.1 Feed-Forward Network

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

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

This combination is called a dense layer and multiple of these are being stacked and represent the models architecture. The intermediate layers are usually called *hidden* layers as they cannot be directly seen from the outside. Figure 2 shows an illustrative FFN with one input layer, one hidden layer and one output layer. Depending on the desired output, the last activation layer might be skipped. Finally, an error function is being applied to the output. This whole process is called *forward pass*. To update the weights of the model, a *backward pass* is performed, which calculates the gradients of

[1] https://openai.com/
[2] Mainly constrained by 16GiBs of memory and 8GiB of video memory.
[3] https://huggingface.co/

the weights, because the error depends on the weights. This can be written as:

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

For updating one layer of weights, the partial derivative is being determined. For the last layer, here denoted as W_n, the partial derivative and thus the gradient of W_n would look like the following, where α denotes the *learning rate*:

$$\frac{\delta E}{\delta W_n} = \frac{\delta E}{\delta \hat{Y}} \frac{\delta \hat{Y}}{\delta A_n} \frac{\delta A_n}{\delta W_n} \tag{3}$$

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:

$$W_n^{updated} = W_n - \alpha \odot \frac{\delta E}{\delta W_n} \tag{4}$$

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), which can be an issue.

2.1.2 Long Short-Term Memory

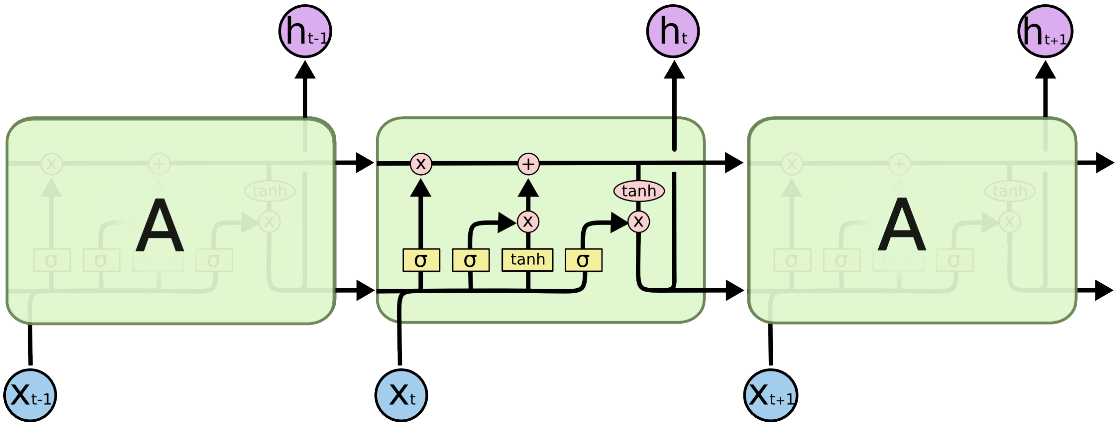


Figure 3: Unfolded LSTM showing the information flow.
Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

An LSTM belongs to the field of recurrent neural networks (RNNs) and handles long dependencies better than a classical RNN (Hochreiter and Schmidhuber 1997, abstract), since RNNs are prone to vanishing and exploding gradients (Pascanu, Mikolov, and Bengio 2012). As text can be seen as a sequence with long term dependencies, LSTMs are particularly suitable for catching such long term dependencies. LSTMs, as well as

RNNs, look at one element of a sequence (in this context a word) and additionally to the normal output vector, the old cell state is passed on to the next cell. This additional information is also called the *hidden state*. Then, the next cells inputs are the hidden state and the second element of the sequence. Utilising this hidden state, information can be preserved and passed 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}, 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 the 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*.[4]

Learn Gate The learn gate takes the new element x_t and concatenates it with the STM (h_{t-1}). Then, after taking a linear combination with its weights W_n, it passes the results through a *tanh* activation function. This value is denoted by N_t. Finally, it is multiplied by an ignore factor i_t, making it ignoring 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 of the learn gate is thus given as $N_t i_t$ with:

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

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

Forget Gate The forget gate looks at the previous LTM memory C_{t-1} and decides which information is being kept and which is being thrown away. This information is encapsulated in a forget factor f_t. The forget factor is the linear combination of the weights W_f multiplied by the concatenated element of the sequence x_t and the previous STM h_{t-1}. Afterwards, the Sigmoid activation function is applied. Finally, the output of the forget gate is given as $C_{t-1}f_t$:

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

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:

$$C_t = C_{t-1}f_t + N_t i_t \tag{8}$$

[4] 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.

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

$$U_t = \text{tanh}(W_uC_{t-1}f_t) \tag{9}$$

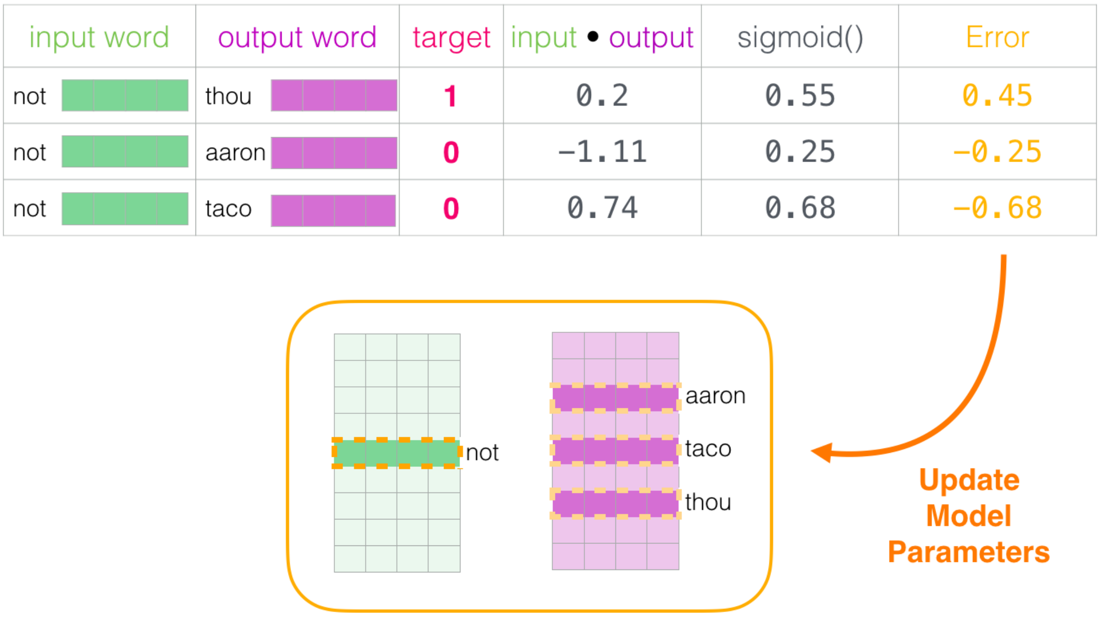
$$V_t = \sigma(W_v(h_{t-1}, E_t)) \tag{10}$$

The advantage of this architecture is that the previous outputs and hidden states can be treated as fixed variables and the gradient (backward pass) does not depend on the whole sequence, mitigating the vanishing gradient problem to some extend. In practice, libraries such as PyTorch, use "reverse-mode automatic differentiation" (Paszke et al. 2019). Therefore, the calculation of the gradients for an LSTM 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, 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 from the embeddings and n negative samples, not occurring in the skip-gram of the original word, are being taken. This 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 $1 \times$ 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 these steps. This whole process is conducted by a neural network, so it is relatively easy to calculate the gradients and to update the weights. The weights, in this case, are the embedding and the context matrices. Once the whole process is completed, the embeddings matrix is being taken as the desired word embeddings. Its size is number of words $\times$ embedding size. The number of words depends on the corpora and the minimum amount of appearances a word must have.



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/

2.2.2 BERT

BERT (Devlin et al. 2018) is an encoder based on bidirectional transformer[5]. Transformers use attention heads (Vaswani et al. 2017) for focusing on different parts or 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 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 a pair of two sentences, for example question and answer (ibid., p. 4). 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 the training of BERT, before a sequence is passed to BERT, 15 percent of its 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

[5] 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://jalammar.github.io/illustrated-transformer/

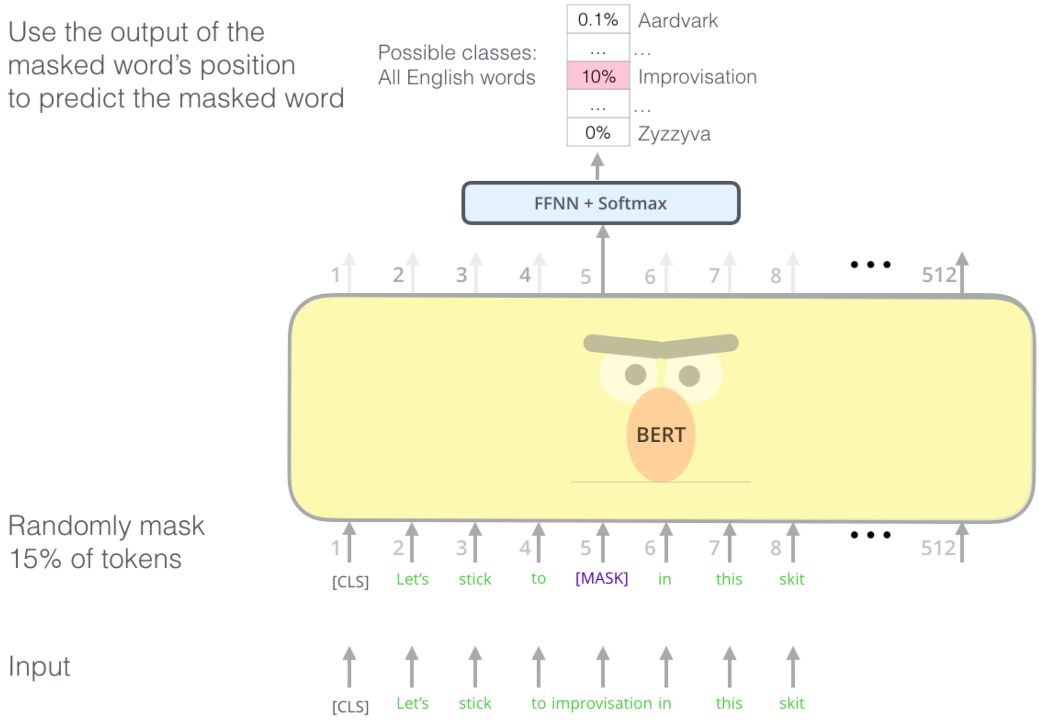


Figure 5: BERT for masked language modeling.
Source: https://jalammar.github.io/illustrated-bert/

the process.

NSP Given two sentences A and B, BERT has to predict the likelihood of sentence B actually following sentence A (Devlin et al. 2018, p. 4). Figure 6 shows this process.

As BERT is just an encoder, another classifier for downstream tasks such as classification is compulsory. The encoder outputs between the different layers can be used as contextualised word embeddings. Figure 7 shows their corresponding F_1 scores on the development set during training of BERT. We will use the output of the last encoding layer in this project as embeddings which achieved an F_1 score of 94.9 percent. The specified `[CLS]` token, originally intended for classification, will not be used, as the aim is not to achieve a high accuracy, but rather to compare non-contextualised and contextualised word 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

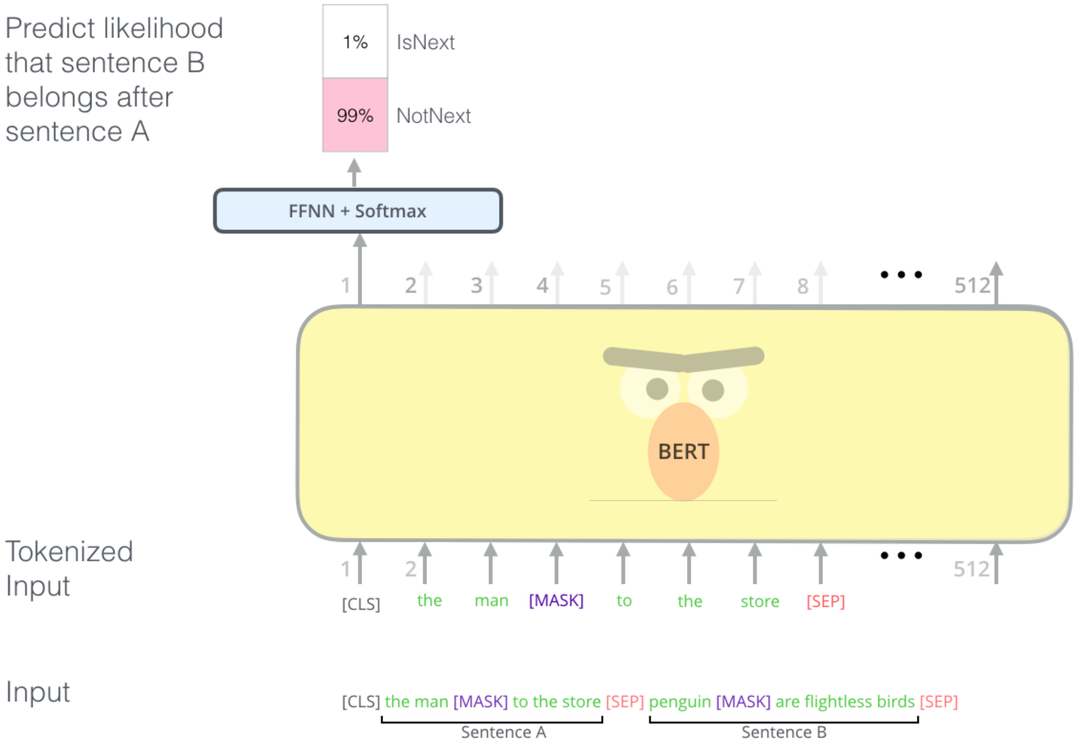

Predict likelihood that sentence B belongs after sentence A
1%
IsNext
99%
NotNext
FFNN + Softmax
1
2
3
4
5
6
7
8
...
512
BERT
Tokenized Input
1
2
...
512
[CLS]
the
man
[MASK]
to
the
store
[SEP]
Input
[CLS] the man [MASK] to the store [SEP] penguin [MASK] are flightless birds [SEP]
Sentence A
Sentence B

Figure 6: Bert for next sentence prediction.
Source: https://jalammar.github.io/illustrated-bert/

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 (similar to the concepts of AdaBoost). The result is a model that is 40 percent smaller and 60 percent faster while keeping 97 percent of BERTs understanding capabilities (Sanh et al. 2019, p. 5).

3 Data

3.1 Presentation

The data set used in this project consists of Amazon reviews categorised by one to five stars and is provided by Xiang Zhang[6]. The data is publicly available and can be downloaded on Google Drive: `amazon_review_full_csv.tar.gz`.

Originally, the data is divided into training data with 3 000 000 data points and test data with 650 000 data samples. Though in this project the data will be merged and split differently. Each data point consists of a label, indicating how many stars the review

[6] http://xzh.me/

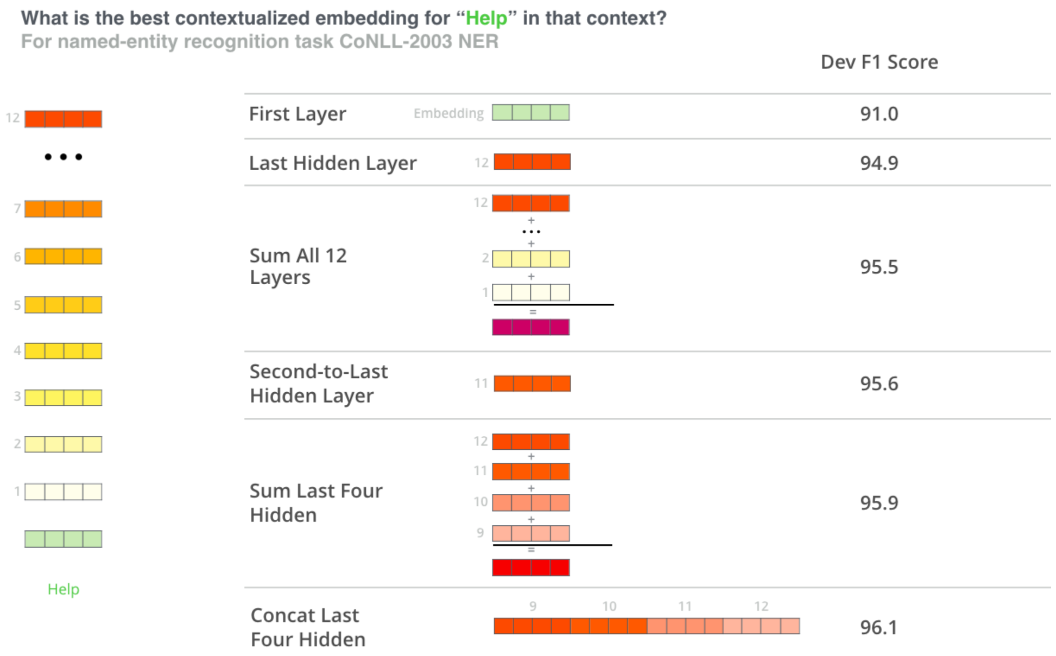


Figure 7: Using BERT for contextualised word embeddings.
Source: https://jalammar.github.io/illustrated-bert/

has, a title and the review text. One sample looks like this:

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

It can be seen in figure 8 that the class distribution is almost equal. The distribution of review lengths (words) can be seen in figure 9 and it shows an interesting pattern, having some. The percentiles can be found in table 1 and it can be sees that with 177 words 99 percent of the reviews do not need to be truncated.

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 start at zero: $0, 1, 2, 3$ and 4. Next, the data is split into 85 percent

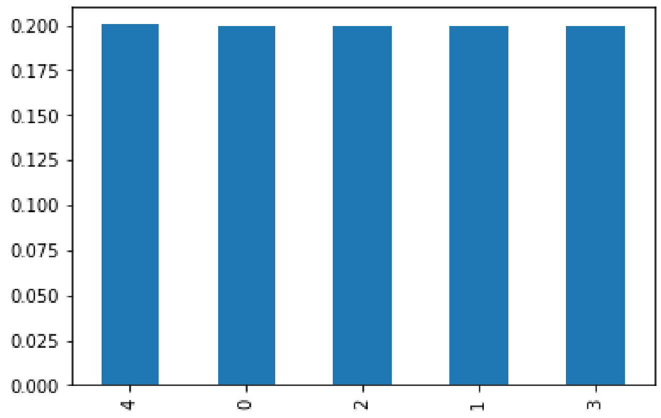


Figure 8: Class distribution.

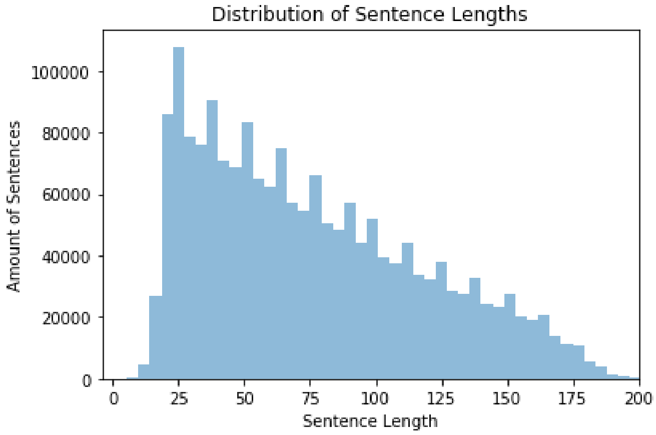


Figure 9: Distribution of review lengths.

training, 10 percent validation and 5 percent test data. In the following, for all sentences a padding of 200 has been used[7]. This lies way above the 99th percentile of 177 words as it accounts for two facts. Firstly, the sentences become longer due to punctuation, as they're tokenised as well, and secondly, some words are being further split during tokenisation.

The padding is applied after tokenisation, so that all sentences have the same length. The tokeniser used for both, Word2Vec and BERT, is the DistilBERT tokeniser provided by the HuggingFace library[8]. For Word2Vec, only the 1 000 000 most common embeddings are taken due to computational constraints. Inferentially, all other words are

[7] 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.

[8] https://huggingface.co/transformers/model_doc/distilbert.html#distilberttokenizer

considered unknown.

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

4 Method

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[9].

The first step is to create the non-contextualised or contextualised embeddings for the reviews. In the case of Word2Vec, an embedding layer[10] is used, mapping the token Ids to non-contextualised word-embeddings of shape (200×300) 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 Word2Vec embeddings. The embeddings are loaded into the embeddings layer using the Gensim library (Řehůřek and Sojka 2010). For the contextualised word embeddings, the tokenised sentences are fed to DistilBERT[11] and the output of the last encoder is then taken as the embedding. The pretrained model *distilbert-base-uncased* is being used and all weights are frozen during training.

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

FFN (Word2Vec) $X \rightarrow$ Embedding → Flattening → Linear → Sigmoid → Dropout → Linear → LogSoftMax

LSTM (Word2Vec) $X \rightarrow$ Embedding → LSTM → Dropout → Linear → Sigmoid → LogSoftMax

FFN (DistilBERT) $X \rightarrow$ DistilBERT → Flattening → Linear → Sigmoid → Dropout → Linear → LogSoftMax

LSTM (DistilBERT) $X \rightarrow$ DistilBERT → LSTM → Dropout → Linear → Sigmoid → LogSoftMax

The last linear layer is always used to map into one of the five classes of Amazon reviews, thus the labels. Furthermore, dropout (Srivastava et al. 2014) is being applied

[9] https://github.com/flennic/text-mining-project

[10] https://pytorch.org/docs/stable/nn.html?highlight=embedding#torch.nn.Embedding

[11] https://huggingface.co/transformers/model_doc/distilbert.html

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.

Since the data set is balanced, accuracy can be used as the evaluation measurement. Additionally, it provides a single value to adequately estimate the performance of the models.

5 Results

All four model combinations and their achieved accuracies can be seen in table 2. The accuracies shown for training have dropout enabled[12] 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 up to the current batch. Figure 11 shows the accuracies for the FFN and figure 12 shows the accuracies for the LSTM, both using Word2Vec and DistilBERT embeddings respectively. The lowest baseline accuracy is around 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 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 embeddings and peaks at 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 their validation accuracies, which seems a bit odd. This pattern can be seen for for the different model combinations. For the FFN, using Word2Vec, the test accuracy is way lower than the training or validation accuracy[13].

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.

[12] Enabling another prediction without dropout would have been too time consuming for this project.

[13] Due to these results the source code has been investigated but no issues have been found. Therefore these results are treated as valid.

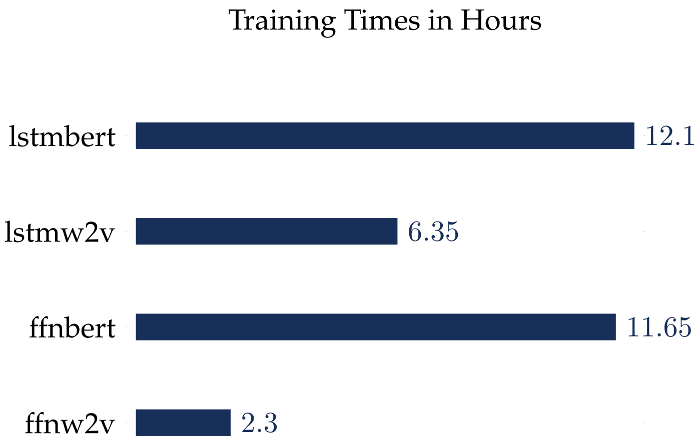


Figure 10: Training times for the different model combinations.

Both models using DistilBERT word embeddings have the highest training times with 12.1 hours for the LSTM and and 11.65 hours for the FFN. Around half of the time is needed for the LSTM with Word2Vec word embeddings with 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.

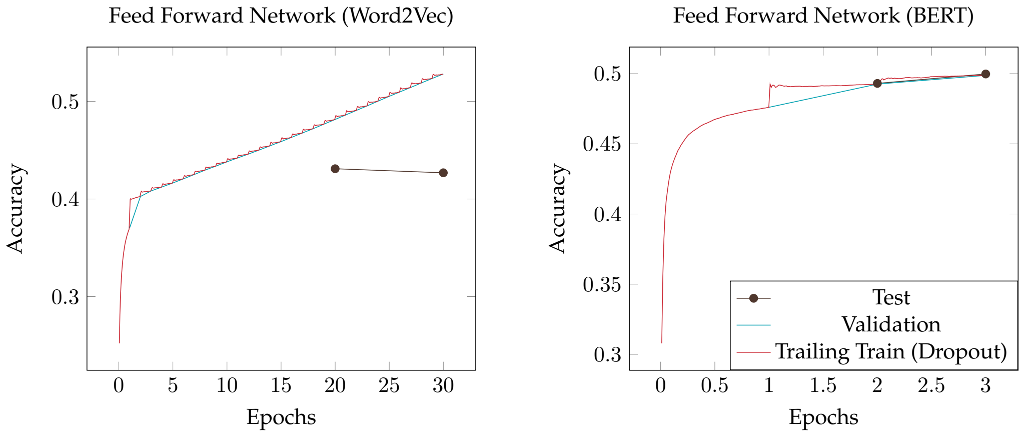


Figure 11: Accuracies for the Feed Forward Networks.

6 Discussion

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.

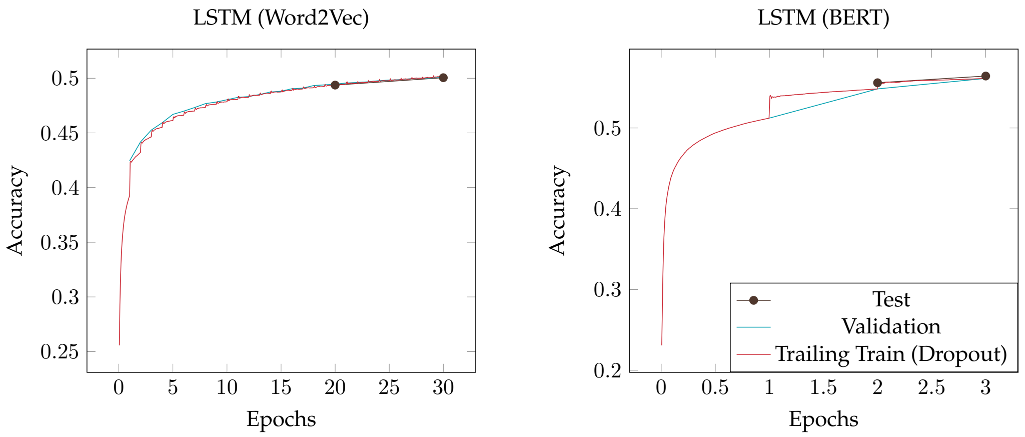


Figure 12: Accuracies for the LSTMs.

Apart from achieving a higher accuracy, also less iterations through 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. The FFN using Word2Vec has seen the data 40 times more often and the LSTM using Word2Vec has seen the data 20 times more often compared to their DistilBERT counterparts. Note that for calculating these factors, 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 each epoch[14]. The overfit can be especially seen for the FFN using Word2Vec embeddings. Looking at figure 11, it can be seen that the test accuracy for the FNN using Word2Vec becomes actually lower going from 20 to 30 training epochs.

Using contextualised word embeddings, the training times for the models are a lot higher, the biggest difference is shown between the LSTM using DistilBERT embeddings and the FFN using Word2Vec embeddings with a factor of almost 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 negligible. This might be due to a parallelisation during the training process, as the implementation used here utilises multiple data loaders. This could be exploited in the future, as parallelised hardware easier available (see core counts for CPU and GPU which are basically made for parallel execution) than improved single thread performance. Another interesting finding is that, despite the fact that DistilBERT basically is a bidirectional encoder, an LSTM is still necessary to extract all the information from the contextualised embeddings. The FFN using word embeddings from DistilBERT is way less accurate than the equivalent LSTM. Also note, that due to computational limitations, a simple unidirectional LSTM with two layers and a small hidden state is being used. Therefore, one presumption is that a larger BiLSTM might achieve even higher accuracies.

[14] This is due to lazy loading the data form disk as the whole data does not fit into memory.

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

7 Conclusion

Contextualised word embeddings are indeed more powerful than their non-contextualised counterparts. The trade-off for this is a high computational demand. Some of the additionally demanded computational power is compensated by parallelisation, nevertheless it is still high, especially considering that DistilBERT and not BERT has been 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 actually more than 60 percent faster compared to BERT.

With deep learning already requiring a lot of computational power, these new models climb to the next level, which indeed is a problem. It is not only the price for the required hardware, but also environmental issues, which raise concern. Nonetheless, these new transformers capture more information and achieve better results. It can also be seen that, at least to some degree, the research society is aware of this problem and tries to reduce computation demands. Here DistilBERT is used, Google also released ALBERT (Lan et al. 2019) which is a lighter version of BERT.

Still the question remains, if throwing more hardware at our problems will be a long-term solution for tackling new and more challenging problems.

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:

Lan, Zhenzhong et al. (2019). *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. arXiv: 1909.11942 [cs.CL].

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

```
{
    "orig_train_path": "data/original/train.csv",
    "orig_test_path": "data/original/test.csv",
    "processed_data_folder": "data/processed/",
    "cached_model_path": "checkpoints/2019-12-17_15-40_LstmWord2VecModelInteractor.pth",
    "word2vec_path": "data/embeddings/GoogleNews-vectors-negative300.bin",
    "splits": [
        0.85,
        0.1,
        0.05
    ],
    "padding": 200,
    "embeddings": 1000000,
    "categories": 5,
    "run_model": "lstm_w2v",
    "load_cached_model": false,
    "models": {
        "ffn_w2v": {
            "data_loader_workers": 4,
            "batch_size": 8192,
            "learning_rate": 0.0001,
            "epochs": 30,
            "embedding_size": 300,
            "dropout": 0.25,
            "hidden": 256
        },
        "lstm_w2v": {
            "data_loader_workers": 2,
            "batch_size": 1024,
            "learning_rate": 5e-05,
            "epochs": 30,
            "embedding_size": 300,
            "dropout": 0.25,
            "lstm_layers": 2,
            "lstm_hidden": 128,
            "lstm_dropout": 0.25,
            "gradient_clip": 5
        },
        "ffn_bert": {
            "data_loader_workers": 1,
            "batch_size": 256,
            "learning_rate": 0.0001,
            "epochs": 3,
            "embedding_size": 768,
            "dropout": 0.25,
            "hidden": 256,
            "max_batches_per_epoch": 64
        },
        "lstm_bert": {
            "data_loader_workers": 1,
            "batch_size": 256,
            "learning_rate": 5e-05,
            "epochs": 3,
            "embedding_size": 768,
            "dropout": 0.25,
            "lstm_layers": 2,
            "lstm_hidden": 128,
            "lstm_dropout": 0.25,
            "gradient_clip": 5
        }
    },
    "device": null,
    "seed": 42,
    "log_level": 20,
    "cache": true
}
```

8.2 Model Settings DistilBERT

```
{
  "activation": "gelu",
  "attention_dropout": 0.1,
  "dim": 768,
  "dropout": 0.1,
  "finetuning_task": null,
  "hidden_dim": 3072,
  "initializer_range": 0.02,
  "is_decoder": false,
  "max_position_embeddings": 512,
  "n_heads": 12,
  "n_layers": 6,
  "num_labels": 2,
  "output_attentions": false,
  "output_hidden_states": false,
  "output_past": true,
  "pruned_heads": {},
  "qa_dropout": 0.1,
  "seq_classif_dropout": 0.2,
  "sinusoidal_pos_embds": false,
  "tie_weights_": true,
  "torchscript": false,
  "use_bfloat16": false,
  "vocab_size": 30522
}
```