INM706 Deep learning for sequence analysis coursework report

Kasper Groes Ludvigsen

City, University of London

GitHub: https://github.com/KasperGroesLudvigsen/deep_learning_for_sequence_analysis
Download data from (and see instructions in README):
https://1drv.ms/u/s!ApqC2EDsdE7tjL80dm7PHZx8w8Vtxw?e=apejim

Introduction

A drawback of many automatic speech recognition systems is that they typically generate text without punctuation. In addition to making the output difficult to read, the lack of punctuation complicates downstream tasks such as machine translation (Xu, Xie and Yao, 2016). It is therefore valuable to be able to automate the insertion of correct punctuation. Systems capable of inserting correct punctuation into text can also be a valuable tool for users of text editing software and other types of software in which text writing is essential.

Punctuation prediction can be regarded as a sequential labeling task where each word is labeled with a punctuation or not. As such, research has found recurrent neural networks (RNNs), particularly Long Short-Term Memory networks, to be well suited for punctuation prediction. It has been found that using a conditional random field (CRF) layer on top of the LSTM can improve performance in similar prediction tasks (e.g. Liu, Stolcke, Shriberg & Harper, 2005), although it was later found that a 2 layer bi-directional LSTM performed just as well on Chinese without the CRF on top (Xu, Xie and Yao, 2016). Convolutional neural networks have also been shown to be effective in punctuation prediction, with better precision but worse recall than bi-directional LSTMs (Zelasko, Szymanski, Mizgajski, Szymczak, Carmiel, Dehak, 2018). Researchers have also applied attention-based architectures to the punctuation prediction problem in Chinese (Chen et al, 2020; Yi & Tao, 2019), and although the authors claim that their transformer architectures produce state of the art results, they perform worse than the bidirectional LSTMs reported by Xu, Xie and Yao (2016). Finally, it has been suggested that integrating symbolic lexical knowledge into neural networks aids learning in natural language processing tasks (Plank & Klerke, 2019).

The objective of this coursework is two-fold: The primary objective is investigate the extent to which machine learning can accurately predict where to place commas in Danish text. To the best of my knowledge, such an endeavour has not previously been undertaken. Secondly, the objective is to compare the performance of models trained on sequences of word class tokens vs. sequences of word lemmas. More information about this is presented in the methodology section. Contrary to the research mentioned above, I am only interested in predicting the position of commas, not full-stops or other punctuation. This is because I will make the model freely available as a tool to help humans with correct punctuation, and placing commas correctly is arguably harder than placing full-stops correctly.

Methodology

The dataset

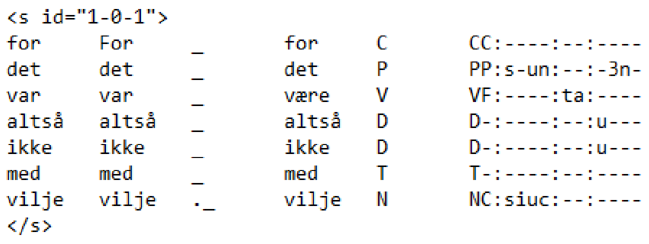
Many previous attempts at punctuation prediction have utilized both lexical and prosodic information, where lexical information relates to the written language and prosodic information relates to the spoken language (Xu, Xie and Yao, 2016). I only used lexical information as opposed to using both lexical and prosodic information. The reason for this is mainly that prosodic data was not available with the dataset, and it would not be available either in the intended use case for the model developed in this report. Furthermore, I would not expect prosodic information to be predictive of the position of commas, because intuitively it appears that cues (e.g. vowel length, pauses and loudness) in spoken language occur irrespective of the location of commas.

I used the Korpus 1990 dataset (Asmussen, n.d.), compiled by corpus linguist & language technology developer Jørg Asmussen, which was kindly made available to me by The Society for Danish Language and Literature upon request. The dataset contains 40 million Danish words in 2,130,531 sentences. Korpus 1990 data set consists of text that originates from the following primary types of sources and were collected between 1983 and 1992 (Norling-Christensen & Asmussen, 1998):

1. Books, magazines and newspapers (28 million words)
2. Manually transcribed radio and television broadcasts (3,8 million words)
3. Leaflets, booklets and pamphlets etc (2 million words)
4. Previous Danish corpora (4 million words)

The dataset does not contain information about which type of source individual sentences stem from. All words in the sentences have been e-POS (part of speech) tagged (Asmussen, 2015) and lemmatized. The e-POS tags include information about the word class of each word, e.g. “noun”, “verb” etc. The strength of this particular dataset is that its sentences originate from highly edited texts such as books and newspaper articles and it can therefore reasonably be assumed that commas are positioned correctly in the texts. Other more recent corpora such as Korpus 2010 contains texts from social media and online blogs which have not been edited to the same extent and therefore presumably are of lower grammatical quality. As per the way Korpus 90 is made, sentences do not contain full stops, because each sentence is extracted from between two full stops in the text from which it originates.

My models were either trained on a data set consisting of sequences of word classes (e.g. “noun”, “verb” etc.) or a data set consisting of sequences of lemmatized words. A later section described the reasoning behind this decision. The sequences were extracted from the raw data in Korpus 1990. The sentences of the Korpus 1990 data set are stored in text files and each sentence is enclosed by <s> and </s> tags. This is what a sentence from a text file looks like:



```
<s id="1-0-1">
for     For     _       for     C       CC:----:--:----
det     det     _       det     P       PP:s-un:--:-3n-
var     var     _       være    V       VF:----:ta:----
altså   altså   _       altså   D       D-:----:--:u---
ikke    ikke    _       ikke    D       D-:----:--:u---
med     med     _       med     T       T-:----:--:----
vilje   vilje   ._      vilje   N       NC:siuc:--:----
</s>
```

Each row represents a word in the sentence. The columns of interest are these: The third column, which specifies what character follows a token. Underscore indicates space. The fourth column contains the lemmatized version of the word and the sixth column contains the e-POS tag. From the e-POS tags, I only used the slice before the first colon, which specifies the word class. Across the entire data set, 6.5 % of words are succeeded by a comma. When only looking at sentences that do contain commas, 10 % of the words are succeeded by a comma. I decided to not discard sentences without commas because the model should learn that some sentences do not have commas. I did train a few models on a version of the data where sentences without commas were excluded, and the performance of the model was roughly the same as with commas. As it is unfeasible to ascertain the punctuation quality of all samples in the data set, I manually inspected a small subset of 100 sentences and was able to ascertain that the commas in these sentences were all placed correctly.

Table 1 - Class distribution

Percentage of words belonging to class 0 (word not succeeded by comma)	Percentage of words belonging to class 1 (word succeeded by comma)
93.5 %	6.5 %

On the use of word classes vs lemmatized words

I will now explain why I chose to use sequences of word classes in addition to sequences of lemmatized words. In Danish, it is the presence of words of different classes that determine where to place commas. For instance, in the sentence "She is an engineer and he is a kindergarten teacher", a comma must be inserted between "engineer" and "and", because that will result in two sub-sentences that both contain a subject (respectively "she" and "he") and a predicator ("is" in both sentences). I decided to make a dataset of sentences where each word is represented by its word class in order to reduce the number of unique tokens such that the model would have to learn associations between a smaller amount of tokens and the position of commas. The number of unique tokens in the word class sentence dataset is 41. The number of unique tokens in the lemma sentence dataset is 493,668. Fewer unique tokens means that the network "sees" much more often a token together with a sequence of tokens and a comma than it would with a larger number of unique tokens. To illustrate the idea, during training, the model will see the token "personal_pronoun" much more often than it will see "she", and it will not make a difference to the position of the comma what specific word (e.g. "he", "she", "they") the personal pronoun is. The use of word classes instead of lemmas is also motivated by the suggestion mentioned previously that using data containing lexical information aids learning in NLP tasks.

Input and target sequences

I split the data set into training, validation and testing data with a 80/20 split between training and testing data and a 80/20 split between training and validation data resulting in 426,106 sentences in test data, 1,363,540 sentences in training data, and 340,885 in validation data. All sentences are tokenized, so each input to the model is a sequence of tokens. The sequences vary in length from 1-45. Sequences longer than 45 tokens (roughly 2 % of the data set) were discarded. Sentences that are shorter than 45 are padded.

When using sequences of word classes, a sentence like "Han løb hurtigt, men ikke hurtigt nok" (which translates to "He ran fast, but not fast enough") would be `["personal_pronoun", "verb", "adverb", "adverb", "adverbial", "adjective", "adverb"]` and the target sequence would be `[0, 0, 1, 0, 0, 0, 0]` because the third word is followed by a comma.

In practice, the model outputs a tensor of size (*b*, *l*, *c*) where *b* is batch size, *l* is the length of the longest sequence in the batch and *c* is the number of classes. *c* is always two (comma/no comma) and the label for each observation in the sequence is decided by the largest value of the two. If the first value (index position 0) is largest, the observation in the sequence is labeled as 0 (no comma) and it is labeled as 1 if the second value (index position 1) is larger. Before feeding the model output to the loss function, PyTorch's `view()` function is applied to the output to make its size (*b*l, c*).

The data is input to the network in batches that are produced by a custom dataset class. The dataset class is adapted from Xu, Xie and Yao (2016) and the associated GitHub repository (Xu, n.d.).

Vocabulary sizes and batch sizes

I used different vocabulary sizes for the models that trained on sequences consisting of lemmatized words. By vocabulary I mean a collection of unique words. The vocabulary sizes I used were: 8,000, 50,000, 100,000, 200,000 and 493,668 (all unique lemmatized words) where the vocabulary consists of the *n* most frequent words, e.g. 8,000 most frequent words. Recall that the total dataset consists of 33,983,300 words. The 8,000 most frequent unique words account for 90.24 % of all words in the data set. 50,000 unique words account for 96.93 % of all words in the data set and 100,000 and 200,000 unique tokens account for 97.82 % and 98.28 % respectively. If a lemmatized word in a sentence is not in the vocabulary, it is replaced by the token "UNK" (short for "unknown"). See Table 2 below for an overview of vocabulary sizes used. I chose 8,000 as the smallest vocabulary size so as to not have too many tokens be assigned to the unknowns. I also experimented with different batch sizes, but 128 produced the best results. Co-incidentally, this batch size is also the one used in the best model by Xu et al (2016).

Table 2 - vocabulary sizes

Token type	**Vocabulary size(s)**
Word class	41
Word lemma	8,000, 50,000, 100,000, 200,000 and 493,668

Architectures

Although self-attention-based architectures (Vaswani et al, 2017) seem to be state-of-the-art for many NLP problems, my literature review shows that multilayer bidirectional LSTMs perform better than self-attention models in punctuation prediction tasks. I therefore opted for this type of architecture. In this section, I will further motivate the use of recurrent neural networks, and LSTMs in particular, for the prediction task as well as how they work, and describe the model architecture used in this coursework.

Text can be treated as sequential data because the “value” (meaning) of a word is dependent on the “value” of prior words, much like the value of a stock is dependent on the value of the stock at the previous time step in traditional time series data. A central problem in sequential data processing tasks is learning long-term dependencies; the basic problem being that gradients propagated over many stages tend to vanish most of the time, or sometimes explode (Goodfellow, Bengio & Courville, 2016). Recurrent neural networks (RNNs) are a type of neural network specifically designed to process sequential data, and they can scale to longer sequences than would be practical for conventional feed forward networks. They can be seen as operating on a sequence that contains vectors $x^{(t)}$ where t is a time step[1] index that ranges from 1 to τ, and they make use of parameter sharing which makes them able to generalize across samples of different lengths . This characteristic is particularly important when a specific piece of information, such as a specific word or word class, can occur at different t's (Goodfellow, Bengio & Courville, 2016).

Although vanilla RNNs are better suited for sequential data than conventional feed forward networks, RNNs suffer from short term memory. Long Short-Term Memory networks, which are a type of RNN, is a type of network that addresses the short term memory that characterises vanilla RNNs by being able to learn longer term dependencies (Goodfellow, Bengio & Courville, 2016). LSTMs can therefore be said to “remember” past information for a larger number of timesteps than conventional RNNs. LSTMs are therefore suited for natural language processing tasks such as the one at hand where the position of a comma can be dependent on distant tokens.

An LSTM network consists of one or more LSTM cells (a cell is depicted in Figure 1). The central element of the LSTM cell is the cell state which is depicted as the horizontal line in the top of the cell in Figure 1. The cell state is sometimes referred to figuratively as the information highway of an LSTM network. The information that is transported via this highway is regulated by a number of internal mechanisms called gates:

1. The forget gate is the first gate through which input to the network is passed. The forget gate applies a sigmoid function to the input vector thereby reducing the values to numbers between 0 and 1 (and summing to 1). The output of the forget gate is multiplied by the cell state values, thereby removing the impact of the values in the cell state that are multiplied by 0, or “forgetting” these values.
2. The input gate which regulates what is going to be stored in the cell state. The input gate consists of two layers through which the input vector is passed. The output of the two layers are multiplied. These two layers are:
    a. A sigmoid layer which can be said to decide which values to update.
    b. A tanh layer which creates new candidate values that could be added to the state.
3. The output gate which regulates what information is recursively passed on to the “next” LSTM cell. The output gate has a sigmoid layer through which the input vector is passed

[1] Time steps does not need to be literal points in time - it sometimes just refers to the position in a sequence

and a tanh layer through which the cell state is passed. The outputs of these two layers are multiplied.

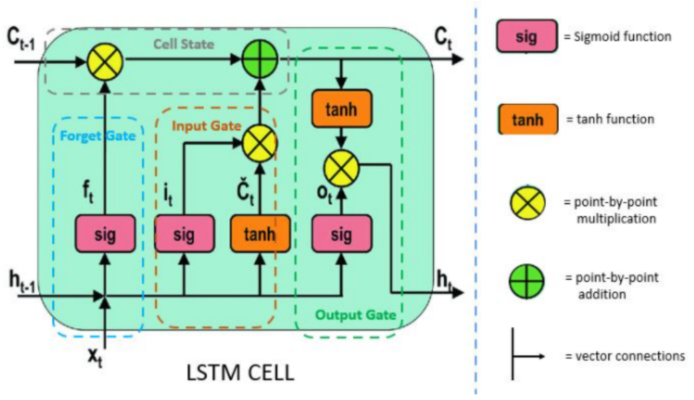


Figure 1: The LSTM cell (Singhal, 2020)

LSTM networks can be either unidirectional, i.e. using only information from previous *t-x* time steps, or bi-directional, i.e. using information from both *t-x* and *t+x* timesteps (Schuster & Paliwal, 1997). This is achieved by “duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as-is as input to the first layer and providing a reversed copy of the input sequence to the second” (Brownlee, 2017, “Bidirectional LSTMs”). I implement a bi-directional LSTM because whether a word is followed by a comma is highly influenced by the subsequent words. For instance, a comma placing rule of thumb in Danish is that a sequence of words between two commas (or between a period and a comma) must contain at least one predicator, but it may contain two in cases where the predicator relates to the same subject, as in “She danced and sang”. Thus, any predictive model should benefit greatly from being able to look ahead and see that in the example sentence above, a comma should not be placed after “danced” because there is another predicator ahead that relates to the same subject. In some situations, the use case of the final model will prevent the use of bi-directional LSTMs, for instance in the prediction of future values of a stock price, where values at t+1, t+2, …, t+n cannot be used to make predictions about the value of t. This, however, is not the case for punctuation prediction.

The architecture I used in this coursework has two bidirectional LSTM cells and one fully connected layer at the end. I made the number of LSTM layers parameterisable and experimented with 2, 3, 4, and 5 layers. The size of the hidden layers are controlled via a parameterised constructor argument in the model class, and I experimented with both 512 and 1024. During training, I performed in-place gradient clipping on the model parameters to prevent gradients from exploding. The gradient norm is parameterizable, but I used a norm of 5 for all models.

Loss function

I used the cross entropy loss function which is a loss function that can measure the performance of a classification model that outputs a probability between 0 and 1 (Loss functions

- ML Glossary documentation, n.d.). In a binary classification setting, log loss and cross entropy loss calculate the same quantity and can be used interchangeably (Brownlee, 2019). As seen in Figure 2, the larger the divergence between the predicted probability and the target label, the larger the cross entropy loss. Cross entropy loss for binary classification is given by

$$\frac{1}{N}\sum_{i=1}^{N} -\left(y_i \times log(p_i)\right) + (1 - y_i) \times log(1 - p_i))$$

where log is the natural log, *y* is the class label (0 or 1) for the ith observation and *p* is the predicted probability of class 1.

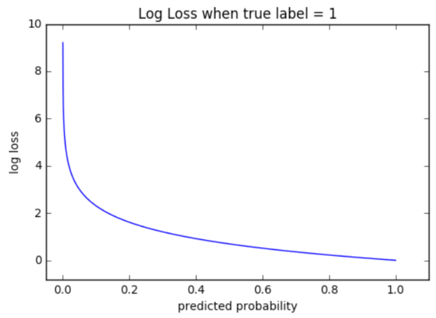


Figure 2: Cross entropy loss (Loss functions - ML Glossary documentation, n.d.)

Learning rates

Learning rate is the most important hyperparameter to tune (Goodfellow, Bengio & Courville, 2016). I therefore experimented with several learning rates and also experimented with adaptive learning rates and schedules. I used PyTorch's `ReduceLROnPlateau` to adjust the learning rate every time loss had plateaued. I implemented the scheduler with a patience of 10 epochs, a factor of 0.1, which means the learning rate was multiplied by 0.1 each time loss plateaued, an improvement threshold of 0.001 and relative threshold mode such that a new loss was only considered an improvement if it was lower than:

best loss * (1 - 0.001)

I also experimented with the triangular policy cyclic learning rate (Smith, 2017). This adaptive learning rate method eliminates the need to experimentally find the best values and schedule for the global learning rates. The method achieves this by letting the learning rate vary cyclically between boundary levels during training. Even though the method eliminates the need for manually experimenting with learning rates, it does require that reasonable learning rate boundaries are set. I identified which boundary learning rates to use by training for 1 epoch and recording the losses and learning rates for each batch. I then visualized the learning rates and losses in Figure 3 below. According to Smith (2017) the boundary values to use are those that lie between the point where accuracy starts to increase and the point where accuracy decreases or becomes ragged. This can be determined visually. Instead of using accuracy to determine the boundary levels, I used loss as the loss was more readily available via the loss function and saved compute time. I decided to use 1e-7 as the lowest learning rate and 0.002 as the largest learning rate based on the plot.

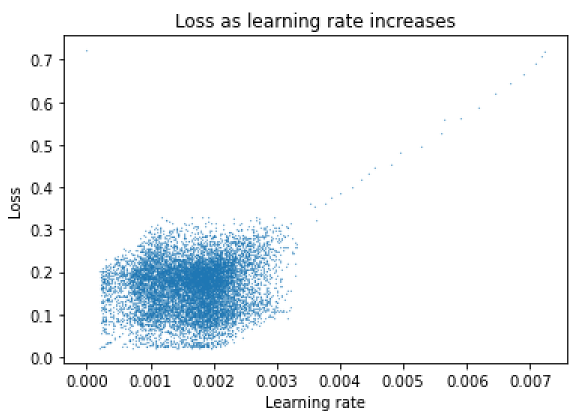


Figure 3: Loss as learning rate increases

Optimizers

For models trained with constant learning rate, I used the Adam optimizer. For models trained with cyclic learning rate, I had to use the standard Stochastic Gradient Descent optimizer because Adam was not compatible with the cyclic learning rate package from Pytorch. Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions. It is computationally efficient, requires little memory, is invariant to diagonal rescaling of the gradients and is well suited for large datasets and models with many parameters (Kingma & Ba, 2014)

Evaluation metrics

I evaluate model performance by *precision*, *recall* and their harmonic mean - the *F1* score. I use recall and precision because these metrics are essential to evaluating the performance of a classification model in the face of class imbalance (Google, n.d.). I use the F1 score because it conveys the balance between precision and recall (Brownlee, 2014).

Precision attempts to answer the question: “What proportion of positive identifications was actually correct?” (Google, n.d.) and is given by:

$$\text{Precision} = \frac{True\ positives}{True\ positives + False\ positives}$$

Recall attempts to answer the question “What proportion of actual positives was identified correctly?” (Google, n.d.) where positives are class label 1 (comma), i.e. false negatives penalize recall. Recall is given by:

$$\text{Recall} = \frac{True\ positives}{True\ positives + False\ negatives}$$

F1 is the harmonic mean of precision and recall and is given by:

$$\text{F1} = \frac{True\ positives}{True\ positives + \frac{1}{2}(False\ positives + False\ negatives)}$$

I do not use the evaluation metric *accuracy* because it is not a very informative evaluation metric in the face of class imbalance. Only 6.5 % of the words in the data set are followed by a comma, so the model could achieve an accuracy of 93.5 % without any ability to predict where to place commas by simply predicting the class label 0 (no comma) for all words.

Early Stopping

I used the early stopping functionality developed by Sunde (2020)[2] to stop model training early if the loss stops decreasing. The EarlyStopping object records model loss every epoch and stops model training before the number of epochs have been used up if the model loss does not improve for `patience` epochs in a row. It is also possible to specify `delta` which defines what counts as a loss improvement. I used the default value which is 0, meaning that any loss decrease is considered an improvement. The `patience` parameter in combination with `delta` thus specifies *convergence*, which is a stop condition for an optimization algorithm where a stable point is located and further iterations are not expected to yield further performance improvement (Kochenderfer & Wheeler, 2019). Let’s consider an example:
Suppose the model’s loss decreases every epoch for the first three epochs. After the fourth epoch, the loss is not better than it was after epoch three. If `patience=5`, the model will continue to train for another five epochs after which it will stop training if the loss has still not improved, i.e. the model will train for eight epochs in total. If the loss improves, the early stopping counter is set to 0 again.

Results

In this section I present and analyse my results. First, I will compare the performance of models trained on sequences of word classes against that of models trains on sequences of word lemmas. Then, I will compare my results to those reported in academic papers claiming state-of-the-art performance on similar punctuation prediction tasks, showing that my results are superior.

Training on sequences of word lemmas vs word classes

Contrary to my expectations, the best performance was achieved with a model trained on sequences of word lemmas and not on sequences of word classes. The learning rate was 1e-3, and the LSTM had five hidden layers with a hidden size of 1024. The size of the vocabulary was 100,000. Figure 4 plots the loss of the model over epochs. With respect to precision, the difference is negligible, but with respect to recall, training on sequences of word lemmas produces a recall that is 8.5 percentage points, or 12 %, above that of the best performance

[2] The code for the EarlyStopping class is in the module pytorchtools.py

achieved with word class tokens. See Table 3 for a comparison. The results can be reproduced by running the files `reproduce_lemma.ipynb` and `reproduce_class.ipynb` respectively.

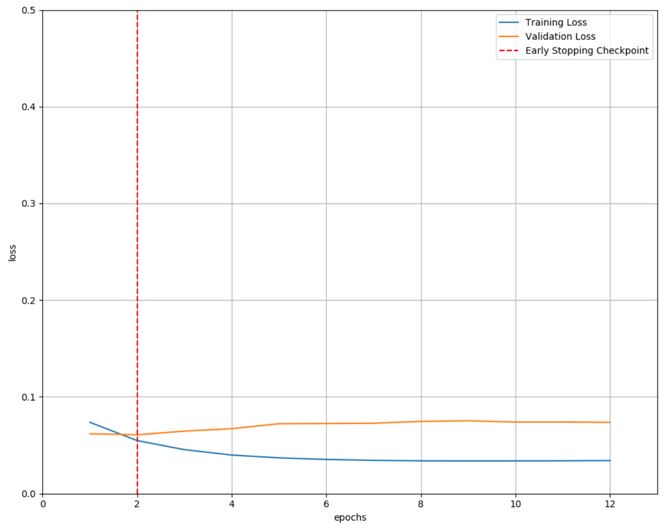


Figure 4: Loss over epochs of best model

Table 3: Model performance: Sequences of word class tokens vs word lemma tokens					
Precision		**Recall**		**F1**	
Word class	*Word lemma*	*Word class*	*Word lemma*	*Word class*	*Word lemma*
81.9 %	82 %	70 %	78.5 %	74 %	80 %

Below, I will present a few other notable observations from my results.

Effect of vocabulary size on performance when using word lemmas
My results show that vocabulary size is an important hyperparameter to experiment with. Recall that for models trained on sequences of lemmatized words, I used 5 different vocabulary sizes: 8,000, 50,000, 100,000, 200,000 and 493,668. The latter constitutes all unique lemmatized words in the dataset. The results clearly show that a vocabulary of size 493,668 is too large. This is made evident by the poor performance yielded by models trained with a vocabulary of this size. The performance of the best model trained with the vocabulary of size 493,668 produced a recall of 8.21 % and precision of 7.57 %, i.e. it hardly learned anything. The performance difference between a vocabulary size of 8,000 and 50,000 is 2 in terms of F1, and the difference between 50,000 and 100,000 is negligible. At 200,000, performance is the same as with 8,000. See Figure 5 for an overview of F1 scores with different vocabulary sizes.

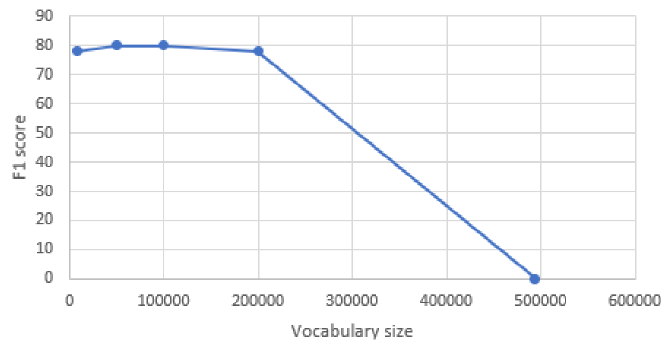


Figure 5: F1 score as vocabulary size increases

Effect of learning rate on performance
A constant learning rate performed better than adaptive learning rates. A constant learning rate of 1e-3 turned out to produce the best results with both word lemmas and word classes. The performance of the learning rate impact is bigger on recall than on precision. 1e-5 and 1e-7 gave slightly worse performance, while learning rates larger than 1e-3 gave significantly worse performance. A learning rate of 1e-3 not only clearly gave the best performance, but also in much faster training because fewer epochs were needed to obtain the best model.

The effect of number of LSTM layers
For a vocabulary size of 50,000, recall increases with the number of LSTM layers at the expense of a slightly worse precision, which shows that with more layers, the model becomes better able at correctly identifying where to place commas thereby increasing recall, but it comes at the expense of sometimes misclassifying tokens as class 1 (comma). See Table 4 below for an overview.

Table 4: Effect of number of LSTM layers on performance

LSTM layers	Vocabulary size	Precision	Recall	F1
2	50,000	82.14	75.16	0.78
3	50,000	81.7	76.43	0.79
4	50,000	81.73	77.73	0.8

Models converge early
Most models trained with word lemma vocabularies and learning rates of 1e-3 or larger converge very early - usually after 1-3 epochs, while models trained with learning rates below 1e-3 need significantly more epochs - between 20 and 24. There is a natural straight forward explanation for the observation that models take longer to converge with smaller learning rates, namely that the learning rate controls how much the weights of the network are updated after each epoch. The lower the learning rate, the smaller the weight updates. Figuratively put, the steps taken towards the minimum of the loss surface are smaller, and more steps (epochs) are therefore needed. However, explaining why the model converges after a few epochs when using a learning rate of 1e-3 is not so straight forward. A model that is not complex enough for the

problem can converge early, but even with more and larger LSTM layers, the model converges after 2 epochs. To increase my confidence that the early convergence is not the result of landing in a local minimum on the loss surface, I also trained a model with a patience of 250, and the results were the same: No performance improvement occurred after 2 epochs. This leads me to speculate that the models converge early because there is simply not more information that can be extracted from the data set.

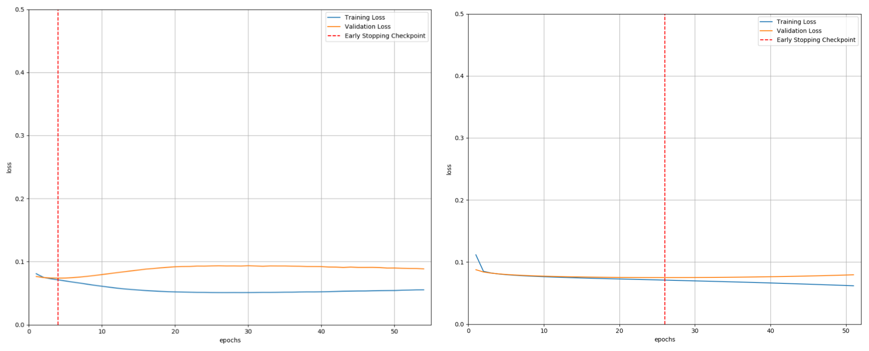


Figure 6: Loss over epochs, training with word class tokens. Left panel: learning rate = 1e-3. Right panel: Learning rate = 1e-5

Comparing my results to state-of-the-art results

In this section, I will compare my results to state-of-the-art results reported in scientific articles. The purpose of the comparison is to give an indication of the quality of my work, although the results are not directly comparable because the stem from models trained on different data sets in different languages.

The performance of my best model is superior to the results reported in Xu et al (2016), who used a two-layer bi-directional LSTM, and Chen et al (2020) who used an attention-based architecture. It is worth noting that the results reported by the other papers are obtained from different models, i.e. the model that produced the best recall is not the model that produced the best precision. This is not the case for my result, as my performance scores seen in Table 5 below all stem from the same model. It is also interesting that in two out of three cases, bidirectional LSTMs perform better than the attention-based architectures used by Chen et al (2020). The implications of these results are discussed in the *Discussion* section.

Table 5. Comparing comma prediction performance to state-of-the-art results

Precision			***Recall***			***F1***		
Xu et al (2016)	*Chen et al (2020)*	*My results*	*Xu et al (2016)*	*Chen et al (2020)*	*My results*	*Xu et al (2016)*	*Chen et al (2020)*	*My results*
77.99	68.8	**82**	72.30	69.8	**78.5**	75.04	69.3	**80**

Discussion

Addressing the performance difference between models trained on word classes vs. word lemmas

Contrary to my expectations, the best performance was achieved with a model that was trained on sequences of word lemmas rather than word classes. The justification for using sequences of word classes was that it would simplify the prediction problem by removing information from the data. Although I believe the justification for experimenting with word classes to be reasonable, I interpret the results to indicate that too much information is removed when using sequences of word classes. As an example, there is a rule in Danish comma setting that a comma must always be placed before the Danish equivalent of “but”, whose word class is “conjunction”, but such a rule does not exist for other conjunctions. When using word classes, the model cannot learn this rule because it treats all conjunctions the same.
On the other hand, too much information is also detrimental to model performance, at least for the architecture and model complexity I used. This was clearly showcased in the poor performance of the model trained on a vocabulary of ~500,000 unique tokens. So in sum, the results show that vocabulary size is an important hyperparameter to experiment with.

LSTMs vs self-attention models for punctuation prediction

When comparing my model performance and the results of Xi et al (2016) with that of Chen et al (2020) - both of which train their models on Chinese texts - it seems that LSTMs are better suited for this prediction task than self-attention-based architectures. The difference may, however, come down to the datasets used, the hyperparameters used etc., and I therefore cannot firmly conclude that LSTMs are better than transformer models in this prediction task. In order to draw such conclusions, a more thorough investigation is needed. Going forward, it would be interesting to train a self-attention based model on my dataset to see if it could outperform the LSTM models.

Potential improvements

Here, I will briefly discuss how to potentially improve my results.

Mitigating class imbalance
It might be that the results could be improved by using techniques that address the class imbalance that characterises the data set. Generally, three categories of techniques exist for addressing class imbalances: resampling techniques, cost sensitive learning and ensemble methods (Haixiang, Yijing, Shang, Mingyun, Yuanyue & Bing, 2017). Here, I briefly discuss the extent to which these techniques could be useful to my prediction problem.

Resampling techniques revolve around reducing class imbalance either by selecting minority class examples more frequently (over-sampling the minority class) or selecting only a subset of the majority class (under-sampling). For large data sets, it is recommended to use a combination of two different over-sampling techniques (Haixiang, Yijing, Shang, Mingyun, Yuanyue & Bing, 2017): 1) Random oversampling, which randomly duplicates samples from the minority class, 2) and SMOTE (Chawler, Bowyer, Hall & Kegelmeyer, 2002). Although resampling techniques can sometimes improve model performance, it is not immediately clear how such techniques could be applied in a situation where each sample is a sequence of values and the objective is to predict a label for each observation in the sequence. In contrast to a situation where the objective is to predict one label for an entire sequence, such as classifying the sentiment of a sentence, the objective of the prediction task at hand in this report is to predict a label for each word in a sentence. In a situation where each sentence has one label, the class imbalance issue could be addressed by simply oversampling sentences of the minority class, but it seems the solution is not as simple when a value must be predicted for each word in the sentence. Going forward, it would be interesting to explore how to apply resampling techniques on a data set like mine.

Cost sensitive learning is about differentiating the cost imposed on the mis-classification of a class. PyTorch's implementation of the cross entropy loss function allows for the specification of a `weight` parameter which specifies a manual rescaling weight given to each class. The weights are applied when the loss is averaged across observations for a minibatch (CrossEntropyLoss, n.d.). With this parameter, I could reduce the weight of the most frequent class (class 0 - no comma) thereby reducing its influence on the loss.

Ensemble methods combine several models and have in some cases been shown to perform better than single models. Boosting is a general ensemble method that creates a strong classifier from an ensemble of weak classifiers, and AdaBoost is a specific instance of a boosting method that has enjoyed widespread success (Brownlee, 2016). If I were to take this project further, the AdaBoost algorithm would seem like a good starting point.

Other strategies for learning rate
A wide plethora of techniques exist for adapting the learning rate during model training. I only tried out two different techniques - reduce learning rate on plateau and cyclical learning rates. The results could potentially be improved with other learning rate adaptation techniques. One popular technique is the optimisation algorithm AdaGrad (Duchi, Hazan & Singer, 2011) which adapts the learning rate to the parameters. AdaGrap performs smaller updates to parameters associated with frequently occurring features, and larger updates to less frequently occurring features. It would be interesting to evaluate to effectiveness of this optimisation algorithm on a data set like the one used in this coursework which is characterised by class imbalance.

Investigate correctness of the commas in the data set
I have measured the model's ability to produce accurate comma predictions and found that it performs reasonably on the test data. The model's performance on the test data is, however, no guarantee that it will perform at a similar level on new data. Even if the model scored 1 on all evaluation metrics (recall, precision etc.), this in and of itself does not mean that the model has learned to set commas correctly - merely that it has learned a function that accurately maps

inputs to outputs in the test data. If a large portion of the commas in the training and test data were placed incorrectly (but consistently incorrect), the model would achieve good results on the data set, but not generalize well outside of the data set. The extent to which the model can be expected to generalize well to samples outside of the data set used in this report hinges on, of course, the quality of the punctuation in the training data, i.e. the degree to which commas are objectively placed correctly.

In hindsight, it might have been productive to verify the correctness of sentences that do not contain commas. One way to do so could be to compute the average number of words that appear before the first comma in a sentence, and then manually inspect a sample of sentences without commas whose length are longer than that number. Another way could be to sample sentences that contain more than some amount, say 3, nouns. The reasoning behind the latter suggestion is that in grammatical parlance, nouns are *subjects* and in written Danish, clauses must be separated by commas, and each subordinate clause must contain a subject. In the sentence “Ian ran far, but not as far as Sophie” there are two clauses, “Ian ran far” and “but not as far as Sophie”. Each contains a subject, “Ian” and “Sophie” respectively. The first clause in the sentence “Ian, Peter, Donald and Elizabeth ran far, but not as far as Sophie” contains four subjects, but such sentences appear to be rarer than sentences with fewer subjects. Thus, the more nouns are in a sentence, the rarer the sentence would be if it was correct to not have a comma in it. I did train a few models on a version of the data where sentences without commas were excluded, and the performance of the model was roughly the same as with commas.

Conclusion

The primary objective of this report was to investigate the extent to which machine learning can accurately predict where to place commas in Danish text. My best model - a five layer, bi-directional LSTM followed by a fully connected layer - shows promising performance with a precision of 82 %, recall of 78.5 % and F1 score of 80 %, which is noticeably better than what has been achieved on data sets in other languages. Secondly, the objective was to compare the performance of models trained on sequences of word class tokens vs. sequences of word lemma tokens. Contrary to my expectations, the best performance was achieved with a model trained on sequences of word lemmas and not on sequences of word classes. The word class vocabulary only contained 41 unique tokens as compared to upwards of 500,000 for the word lemma vocabulary if all unique tokens are included. The inferior performance of the model trained on sequences of word class tokens is likely due to the information that is lost when only 41 tokens are used. My results also show that the vocabulary size can affect model performance, and the best performance in my case was achieved with a vocabulary size of 100,000.

References

An overview of ensemble methods for binary classifiers in multi-class problems Experimental study on one-vs-one and one-vs-all schemes | *Elsevier Enhanced Reader* (no date). doi: 10.1016/j.patcog.2011.01.017.

Asmussen, J. (2015) 'Design of the ePOS tagger', p. 12.

Asmussen, J. (no date) *korpusdk*. Available at: https://korpus.dsl.dk/resources/details/korpusdk.html#english (Accessed: 26 May 2021).

Brownlee, J. (2014) 'Classification Accuracy is Not Enough: More Performance Measures You Can Use', *Machine Learning Mastery*, 20 March. Available at: https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/ (Accessed: 22 August 2021).

Brownlee, J. (2016) 'Boosting and AdaBoost for Machine Learning', *Machine Learning Mastery*, 24 April. Available at: https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/ (Accessed: 22 August 2021).

Brownlee, J. (2017) 'How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras', *Machine Learning Mastery*, 15 June. Available at: https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/ (Accessed: 31 May 2021).

Brownlee, J. (2019) 'A Gentle Introduction to Cross-Entropy for Machine Learning', *Machine Learning Mastery*, 20 October. Available at: https://machinelearningmastery.com/cross-entropy-for-machine-learning/ (Accessed: 21 August 2021).

Chawla, N. V. *et al.* (2002) 'SMOTE: Synthetic Minority Over-sampling Technique', *Journal of Artificial Intelligence Research*, 16, pp. 321–357. doi: 10.1613/jair.953.

Chen, Q. *et al.* (2020) 'Controllable Time-Delay Transformer for Real-Time Punctuation Prediction and Disfluency Detection', in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8069–8073. doi: 10.1109/ICASSP40776.2020.9053159.

Classification: Precision and Recall | Machine Learning Crash Course (no date). Available at: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall (Accessed: 22 August 2021).

CrossEntropyLoss — PyTorch 1.9.0 documentation (no date). Available at: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (Accessed: 22 August 2021).

Duchi, J., Hazan, E. and Singer, Y. (2011) 'Adaptive Subgradient Methods for Online Learning and Stochastic Optimization', p. 39.

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. Available at: https://www.deeplearningbook.org/ (Accessed: 21 April 2021).

Haixiang, G. *et al.* (2017) 'Learning from class-imbalanced data: Review of methods and applications', *Expert Systems with Applications*, 73, pp. 220–239. doi: 10.1016/j.eswa.2016.12.035.

Kingma, D. P. and Ba, J. (2014) 'Adam: A Method for Stochastic Optimization', *arXiv:1412.6980 [cs]*. Available at: http://arxiv.org/abs/1412.6980 (Accessed: 30 August 2021).

Kochenderfer, M. J. and Wheeler, T. A. (2019) *Algorithms for Optimization*. Cambridge, MA, USA: MIT Press.

korpusdk (no date). Available at: https://korpus.dsl.dk/resources/details/korpusdk.html (Accessed: 16 February 2021).

Liu, Y. *et al.* (2005) 'Using Conditional Random Fields for Sentence Boundary Detection in Speech', in *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. *ACL 2005*, Ann Arbor, Michigan: Association for Computational Linguistics, pp. 451–458. doi: 10.3115/1219840.1219896.

Loss Functions — ML Glossary documentation (no date). Available at: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html (Accessed: 21 August 2021).

Norling-Christensen, O. and Asmussen, J. (1998) 'The corpus of the Danish dictionary', *Lexikos*, 8, pp. 223–242. doi: 10.4314/lex.v8i0.

Olah, C. (2015) *Understanding LSTM Networks -- colah's blog*. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (Accessed: 30 May 2021).

Plank, B. and Klerke, S. (2019) ‘Lexical Resources for Low-Resource PoS Tagging in Neural Times’, in *Proceedings of the 22nd Nordic Conference on Computational Linguistics*. Turku, Finland: Linköping University Electronic Press, pp. 25–34. Available at: https://www.aclweb.org/anthology/W19-6103 (Accessed: 31 May 2021).

Schuster, M. and Paliwal, K. K. (1997) ‘Bidirectional recurrent neural networks’, *IEEE Transactions on Signal Processing*, 45(11), pp. 2673–2681. doi: 10.1109/78.650093.

Singhal, G. (2020) *Introduction to LSTM Units in RNN*. Available at: https://www.pluralsight.com/utilities/promo-only?noLaunch=true (Accessed: 30 May 2021).

Smith, L. N. (2017) ‘Cyclical Learning Rates for Training Neural Networks’, *arXiv:1506.01186 [cs]*. Available at: http://arxiv.org/abs/1506.01186 (Accessed: 26 May 2021).

Sunde, B. M. (2020) *Early Stopping for PyTorch*. Available at: https://github.com/Bjarten/early-stopping-pytorch/blob/f1a4cad7ebe762c1e3ca9e74c0845a555616952b/pytorchtools.py (Accessed: 22 August 2021).

Vaswani, A. *et al.* (2017) ‘Attention Is All You Need’, *arXiv:1706.03762 [cs]*. Available at: http://arxiv.org/abs/1706.03762 (Accessed: 1 June 2021).

What usually causes a neural network to stop improving early? - Quora (no date). Available at: https://www.quora.com/What-usually-causes-a-neural-network-to-stop-improving-early (Accessed: 23 August 2021).

Xu, K. (no date) *kaituoxu/X-Punctuator*, *GitHub*. Available at: https://github.com/kaituoxu/X-Punctuator (Accessed: 26 May 2021).

Xu, K., Xie, L. and Yao, K. (2016) ‘Investigating LSTM for punctuation prediction’, in *2016 10th International Symposium on Chinese Spoken Language Processing (ISCSLP)*. *2016 10th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, pp. 1–5. doi: 10.1109/ISCSLP.2016.7918492.

Yi, J. and Tao, J. (2019) ‘Self-attention Based Model for Punctuation Prediction Using Word and Speech Embeddings’, in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7270–7274. doi: 10.1109/ICASSP.2019.8682260.

Żelasko, P. *et al.* (2018) ‘Punctuation Prediction Model for Conversational Speech’, *arXiv:1807.00543 [cs]*. Available at: http://arxiv.org/abs/1807.00543 (Accessed: 30 May 2021).