

MASTER'S THESIS 2021

Neural Sentence Filter to Improve Context in Cloze Tests

Ludvig Rasmus

Elektroteknik
Datateknik

ISSN 1650-2884

LU-CS-EX: 2021-44

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-44

**Neural Sentence Filter to Improve Context
in Cloze Tests**

Neuralt textfilter för förbättrad kontext i
lucktexter

Ludvig Rasmus

Neural Sentence Filter to Improve Context in Cloze Tests

Ludvig Rasmus
tfy14lra@student.lu.se

September 24, 2021

Master's thesis work carried out at EDIA B.V.

Supervisors: Mark Breuker, mark@edia.nl
Pierre Nugues, pierre.nugues@cs.lth.se
Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

Fill-in-the-blank tests, also known as *cloze tests*, are used in several educational contexts, of which a major one is language learning. Creating these tests for a learning platform is a process that is desirable to automate, but the generated tests are sometimes created from sentences with lacking context, which can make them impossible to solve for a language learner.

At the same time, in the last years, there have been several leaps forward in the field of language modeling, and with new approaches such as the *transformer*, word prediction and similar tasks are easier than ever before. In this thesis, I show that by using a sentence filter based on masked word prediction with a neural language model, it is possible to significantly improve the quality of provided context in automatically generated cloze tests.

Keywords: MSc, neural networks, transformers, BERT, cloze tests

Acknowledgements

I want to express huge thanks to my supervisor Pierre Nugues, for his continuous support throughout the entire project.

I am also deeply grateful to Henrik Gyllstad from the Centre for Languages and Literature, who performed the evaluation of the cloze tests. Without his help, there would be no results to present.

Lastly, I want to thank Mark Breuker and all of EDIA for hosting this project and letting me be part of such a fantastic company.

Contents

1	Introduction	7
1.1	Background	7
1.2	The Knowble Learning Platform	8
1.3	Project goal	8
1.4	Previous work	9
1.4.1	Language models	9
1.4.2	Cloze tests	10
2	Datasets	13
2.1	Data preprocessing	13
2.1.1	Text cleaning	13
2.1.2	Language skill level	15
3	Theoretical background	19
3.1	Statistical language modeling	19
3.2	Recurrent neural networks	19
3.3	Transformers	22
4	Model testing	25
4.1	Test cases	25
4.2	Model types	25
4.3	Testing methodology	26
4.4	Measurements	27
5	Cloze test pipeline	29
5.1	Pipeline description	29
5.2	Pipeline implementation	30
6	Evaluation	31
6.1	Evaluation of cloze tests	31
6.2	Data gathering for evaluation	32

6.3	Results	32
6.4	Discussion	32
6.4.1	Cloze tests	32
6.4.2	Future work	34
7	Conclusion	35
	References	37

Chapter 1

Introduction

1.1 Background

In the last few years, language modeling is an area that has seen several leaps forward with new approaches being utilized, from LSTM (Sundermeyer et al., 2012) to GRU (Cho et al., 2014) to Transformers (Vaswani et al., 2017). This thesis project does not aim to find new model types or to improve on these models, but rather aims to apply them in a new context.

Anyone who has ever attempted to learn a language through a course or language learning tool, has probably encountered the 'fill in the blank' task, which is also known as a *cloze task* or *cloze test*. While these tests have traditionally been created by humans, this method is subject to human error and bias in predictability, as well as clearly inefficient. Automating this process would be beneficial to any language learning service provider that works with cloze tests.

EDIA B.V., who hosted this project, has previously developed a method of generating cloze tests for a given target word from a large corpus of texts, based on picking the sentences that contain the easiest (i.e. most common) words. This is to hopefully choose sentences that aren't too advanced or somehow unnatural, which might cause predicting the word to be difficult, or even impossible. Unfortunately, this approach sometimes results in cloze tests that are difficult to solve, not because of long and complicated sentences, but because of lacking context.

The cloze test "We have new ___" is an example of it. In this test, there is no way for a test taker to predict the masked word in a good way, even if they are given options, since practically any plural or uncountable noun can fit in the gap. Furthermore, the test taker will not really have gained any new information about in which contexts and how the word can be used, which gives this cloze test a low quality from a didactic standpoint.

For both a human and a language model, a certain amount of context is required to be able to predict a masked token. Based on this simple fact, to solve the issue of unpredictable and non-didactic sentences, I propose a data pipeline based around letting a neural language

model predict the masked word. This is meant to result in more suitable sentences to create cloze tests from, for any given target word. I would argue that this solution is very intuitive, and what has to be tested is whether the prediction ability of the language model and a human are similar enough.

Testing different language models showed that the transformer-based BERT was the best candidate to use in the implementation. Letting an English professor then compare tests generated with the old method and the new one showed a significant increase in the solvable quality of the tests. While there are of course other factors to consider while making these tests, this does provide a very intuitive solution for a very crucial problem.

1.2 The Knowble Learning Platform

The learning platform this thesis relates to is called Knowble, and it lets aspiring English learners read news articles from a selection of different categories to choose from. When a user reads an article, words that they have either not seen before or only seen a few times are highlighted. Then, three of these new words are chosen, and a cloze test is generated for each of these three words. These sets of cloze tests can take one of three different forms:

1. *Dictation*, which gives the user voice samples of the word that should go in the gap for each cloze test.
2. *Drag words*, where the user is given five words to choose from, and is asked to drag the relevant ones to their respective correct cloze test gap.
3. *Finish words*, where the initial letters of the masked words are given, and the user is asked to complete them.

Sentences are chosen in the same way for each of these three types, so any sentence can appear in any type of cloze test. Figures 1.1, 1.2 and 1.3 show respective examples for each test type, taken from the Knowble platform. In the drag words test, a user has a small number of words to choose from, but also no visual or audio clues. Correspondingly in the dictation and finish words tests, the user gets clues, but also has a larger set of words to choose from, in the form of the words that get highlighted in the article text, as seen in Figure 1.4.

As mentioned briefly in the background, the sentences for the cloze tests are currently chosen based on probability, by picking the sentences with the lowest total vocabulary scores, where low scores are assigned to words that are more commonly used and high scores are assigned to more rarely used ones. This sentence choosing component is what we will attempt to improve in this thesis project.

1.3 Project goal

To summarize, the main goal of this thesis project is to devise a method to improve the quality of automatically generated cloze tests for the Knowble Platform. More specifically, making sure that the cloze tests contain enough context so that a learner has a fair chance to figure out the masked word. The intended method is to use a trained language model to filter out sentences that are unsuitable to generate tests from.

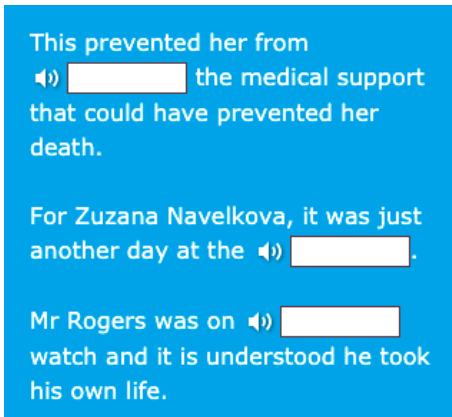


Figure 1.1: Example of a dictation cloze test.

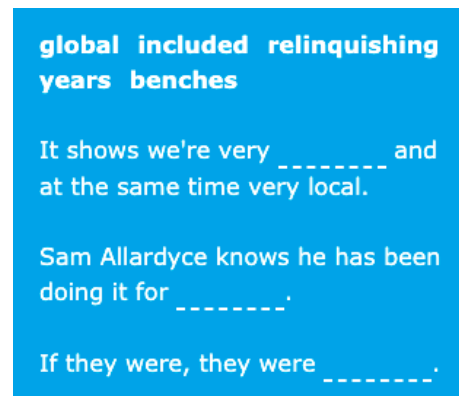


Figure 1.2: Example of a drag words cloze test.

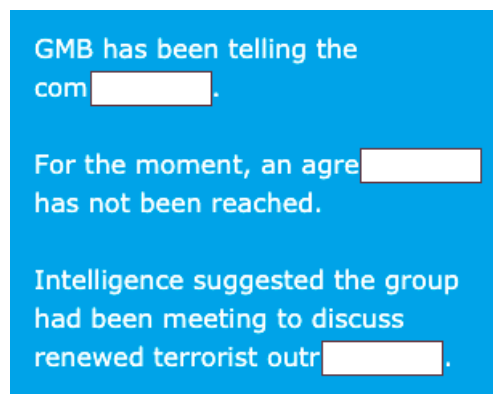


Figure 1.3: Example of a finish words cloze test.

1.4 Previous work

1.4.1 Language models

Language modeling is a way to create models that can predict probabilities of words in sequences, and this can easily be done with statistical methods by simply utilizing a large amount of text data and counting how often certain words occur and which other words they occur together with. This approach is often implemented in the form of *n-grams*, where sequences of *n* words are collected from a text, and this can be a surprisingly effective method, despite its conceptual simplicity (Mikolov et al., 2010).

For a long time, this was the preferred method of language modeling, but Mikolov et al. (2010) proposed a language model, based on a recurrent neural network (RNN) for sequential data prediction, that was superior to the statistical approach. Not long after, Sundermeyer et al. (2012) outperformed a standard RNN by incorporating so called LSTM (Long Short-Term Memory) architecture, a design first suggested by Hochreiter and Schmidhuber (1997), in the neural network. The main problem with regular RNNs that LSTM overcomes is the so-called vanishing gradient problem (Sundermeyer et al., 2012). An alternative to LSTM

The Duchess said that "growing up, I remember so much how it felt to not see yourself represented," in an interview with National Public Radio's (NPR) Samantha Balaban.

"So any child or any family hopefully can open this book and see themselves in it, whether that means glasses or freckled or a different body shape or a different ethnicity or religion," Meghan said.

While the book opens and ends with depictions of Meghan's family, there is a diverse range of fathers and sons depicted throughout the story.

Figure 1.4: Example of words being highlighted in an article.

in the form of gated recurrent units (GRU) was introduced by Cho et al. (2014) as a more easily implementable option, which Chung et al. (2014) found to be comparable to LSTM, and could not conclude if one type of unit performed better than the other.

In the past few years, RNNs have started to be replaced by a new design, called the transformer, in language modeling tasks. The first transformer was proposed by Vaswani et al. (2017), and its main advantage to RNNs is that it has more room for parallelization since it does not rely on sequential data processing in the same way that RNNs do. A transformer model that has garnered much attention recently is Google's model known as BERT, which reached state of the art results on as many as eleven natural language processing tasks (Devlin et al., 2019). Especially relevant to this project, Zhou et al. (2019) at Stanford University concluded that BERT is the state of the art for solving cloze tests, i.e. masked word prediction tasks.

1.4.2 Cloze tests

The cloze test as a pedagogical tool is based on a method for measuring readability proposed by Taylor (1953), that he referred to as a *cloze procedure*. He defined this as:

A method of intercepting a message from a "transmitter" (writer or speaker), mutilating its language patterns by deleting parts, and so administering it to "receivers" (readers or listeners) that their attempts to make the patterns whole again potentially yield a considerable number of cloze units.

Here, a *cloze unit* is defined as a successful attempt to fill in a missing part from a text. The pedagogical justification for using cloze tests lies in that, given a well formulated test, a test subject needs to have, and will therefore develop, an understanding not only of the words occurring, but also the meaning of the combination between them, as well as different types of patterns one might expect to occur.

This underlying purpose is what makes the definition of a cloze test narrower than simply a sentence completion test. Taylor himself formulated that it is not a method that is supposed

to test knowledge of specific bits of information (Taylor, 1953). As an example of a sentence completion test that does not serve the purpose of a cloze test, presenting an English learner with the test “Addis Ababa is the capital city of ___” does not provide didactic value towards English proficiency, but rather only tests the learner’s knowledge of geography.

Chapter 2

Datasets

2.1 Data preprocessing

Beyond the task of choosing sentences that make a good cloze test for a specific word, there is a crucial data preprocessing phase involved as well. This phase mainly involves generating the sentence corpus, which we will search through when looking for cloze test candidates, from a raw dataset.

2.1.1 Text cleaning

The base dataset consisted of 120,000 news articles scraped from the web, and will from here on referred to as the *Knowble dataset*. The news articles come from English language news sources based in different parts of the world, including the U.S., the U.K., India and South Africa to name a few. The types of news sources also vary, from more traditional news sources like the BBC, to more specialized ones like *Science Daily* and *Yahoo Finance*.

The desired corpus structure to search for cloze test candidates was a CSV file of sentences extracted from these articles. In this initial data structuring phase, sentences were then extracted from these articles, using the `sent_tokenize` function from the `nltk` package for Python, while filtering out sentences that should never be considered for cloze tests.

There were two main categories of sentences that were desirable to filter out at this data cleaning stage.

1. The first is sentences that would never be chosen for cloze tests, and would thus take up unnecessary space in the sentence corpus. This mostly includes sentences that are too short to provide any proper context, as well as sequences of characters that do not form any comprehensible words.
2. The other, more critical category, is sentences that could technically be chosen for cloze tests, since they might have a sequence of words that makes up a real sentence, but

could also contain clutter included from the dataset by the `nltk` sentence tokenizer.

Both of these categories can be included in what we can call *dirty data*, which is a common issue within large, unlabeled datasets. For example, Raffel et al. (2020) introduced a dataset known as the *Colossal Clean Crawled Corpus* (C4), which is a cleaned up version of a snapshot of Common Crawl. This is a huge digital archive of text scraped from the web, through reading HTML files and removing any non-text elements. This, however, results in a collection of data where a lot of it is not natural language, but rather source code and menu texts, among other things (Raffel et al., 2020).

One difference in this thesis is that the dataset is not only going to serve as training data for the self-trained model, but also as the source of the data that will be presented to the user. This is where the main problem lies in this approach, because we don't want to present users with cloze tests based on unnatural sentences since this might cause a user to learn wrongful information. The Knowble dataset, while scraped from the web, is also organized into complete news articles, separated in a CSV file. It is not nearly as cluttered as something like Common Crawl, but it does share some of the same issues. In some cases, text like image source references or website text unrelated to the article appeared in the data, which could result in sentences like the two following examples:

(ALEXEY DRUZHININ/ AFP/Getty Images) In November 2016, the Kremlin announced that Seagal had received Russian citizenship.

Role of Hezbollah sanctions in government formation TRENDING YOU MIGHT ALSO LIKE

In addition, the Knowble dataset also contained a surprising amount of strings composed of only question marks, as well as lengthy sequences of underscores. How these came to be included in the dataset is not entirely clear, it could be because of misinterpreted characters during data collection or some similar issue. Another category we want to avoid is sentences that are excessively long, since it can be distracting for the user, as well as not fitting properly into the interface where the cloze tests are presented. These factors combined resulted in the following list of conditions in which sentences have to fulfil each one to not be filtered out:

- Contains between 5 and 15 tokens.
- Does not contain sensitive or offensive terms.
- Ends in a punctuation character.
- Starts with an uppercase character.
- Does not contain uppercase character sequences longer than five characters.
- Does not contain the strings '??' or '___'.

The list of sensitive and offensive terms to avoid is the same as the one used by Raffel et al. (2020). The conditions about uppercase character sequences is to avoid sentences like the two presented above. This will most likely exclude some legitimate sentences, but in this case, it is more important not to include any faulty sentences in the corpus than not to miss

out on any legitimate ones. The same goes for the condition for starting with an uppercase character, which was added to cover cases, where the `sent_tokenize` function didn't work as intended. Cleaning this type of data often requires these types of approaches, where one has to find a common factor among undesirable data. Another example is from Raffel et al. (2020), where they filter out text that contains code, by filtering out any page that contains a curly bracket “{”, since it appears in many programming languages, but not in natural text.

What this all goes to show is that in whenever big, unlabeled datasets are involved, it is crucial to take samples from the data to see what types of patterns might occur in it, since each dataset is likely to have its own unique flaws.

2.1.2 Language skill level

One key thing to consider when picking sentences for a test, is the skill level of the user. If a sentence contains words that the user has not yet learned, the test might be unsolvable. There are different ways to approach this, of which two are:

- Store a list of words the user has learned, and check if sentences contain words that do not appear in this list.
- Classify sentences to required language level and only choose those within the user's current level.

If using the first method, one has to consider granting for example proper nouns some sort of exception from the rule, since they are likely to not be in the user's list of learned words, but this fact does not necessarily impact how comprehensible a sentence is. As an example, take the sentence:

Shahkar said one military personnel and a civilian were wounded in the blast on the outskirts of Charikar, the provincial capital.

From the words in this sentence, a learner will likely not have *Shahkar* and *Charikar* in their list of learned words, but may nonetheless conclude that *Shahkar* could denote a person, and *Charikar* a city. The same applies for different punctuation symbols, like commas and question marks, since these also don't appear in the lists.

One issue with the aforementioned method is that it requires consideration of learned words for each unique user. What would also follow from this, is that unless there was a set order that every user would encounter each new word in, that is if every unique user's word list was expanded in the same order, cloze tests would have to be generated for each unique user at runtime. This would include checking every sentence in the corpus to not include any words that haven't been learned, which would require tokenization of these sentences, and possibly lemmatization of the tokens, to mention just two potential operations. In summation, this would be far too inefficient and time consuming to be a feasible option.

Instead, the second approach of using a language level classification was deemed more suitable for use in this project, by creating a collection of sentence corpora, each based on a specific language level. One commonly used way of classifying language proficiency levels is the Common European Framework of Reference for Languages (CEFR) level system, which ranks proficiency into six different levels, from A1 to C2.

EDIA has previously also developed a model to classify texts to the required CEFR level to be able to read them. This model however, is based on a 9-level scale as seen in Figure 2.1, with three extra levels in the form of A2+, B1+ and B2+ being added for more specificity. Using this, sentences were extracted from the article corpora, filtered on the conditions from the previous section, into CEFR level-based sentence corpora.

Because the model is meant to classify texts rather than individual sentences, the classification was done on article level, which meant that all sentences from one article were assigned the same CEFR level. This might seem like an issue, but texts in general tend to keep a similar language level throughout, providing that it is the product of a single author, and if the model was used on separate sentences the classification would be skewed, since this is not what the model was intended for. Furthermore, splitting texts into sentences and sending each separately to the classifier would be computationally heavy.

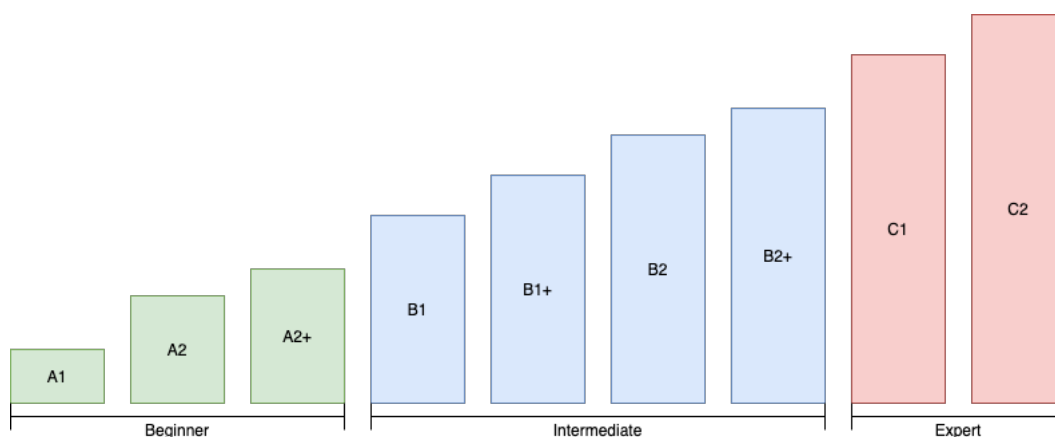


Figure 2.1: The 9-level CEFR scale.

After classifying the Knowble dataset into sentence corpora, it was clear that the dataset favored texts belonging to the higher end of the CEFR scale. To gain sentences for the lower levels, like A1 and A2, other sources had to be used. For this, a dataset with articles from Simple English Wikipedia was utilized, from which 25,000 Wikipedia articles were used. Simple English is a separate language option on Wikipedia, and includes a simpler language and generally shorter articles than the regular English option, intended for individuals with more limited English skills. Table 2.1 shows the resulting number of sentences in each CEFR-based sentence corpus.

As the numbers show, the lower CEFR levels have a significantly lower number of sentences at their disposal. At first glance, this might seem like a big problem, but if a user is going to use Knowble to read news articles, they are likely already past the lowest levels of the CEFR scale.

Classifying the target word was done in a different way, and instead utilized a Global Scale of English (GSE), score mapping on word level from the Pearson website (SOURCE), which is a numeric scale from 10 to 90. This was then converted to its closest CEFR counterpart according to Figure 2.2, where the GSE score is the upper bound for the corresponding CEFR level, to be consistent with the sentence classification. What this means is, for example, that any score greater than 36 and smaller than or equal to 43 will be converted to CEFR level A2+.

To increase the chances of finding a sentence fit for a cloze test, especially for words on the

CEFR level	#sentences
A1	583
A2	1 666
A2+	7 186
B1	34 872
B1+	71 512
B2	99 655
B2+	147 904
C1	136 139
C2	3 381

Table 2.1: Number of sentences in each corpus

GSE	CEFR
30	A1
36	A2
43	A2+
51	B1
59	B1+
67	B2
76	B2+
85	C1
90	C2

Table 2.2: GSE score to CEFR level conversion

lower end of the CEFR scale, a mapping was created describing which sentence corpora we are allowed to search through for each given word CEFR level. In general, after checking the word's corresponding sentence corpus, we allow checking two levels below and then one level above, so if the word has CEFR level B2, we first check sentence corpus B2, then B1+, then B1 and finally B2+. For the lower levels A1 and A2, some exceptions had to be made. Firstly, as seen earlier, both of their corresponding sentence corpora ended up being quite small, despite the additional simpler datasets, and secondly, there is none at all or very limited material of lower CEFR level. Because of this, for these levels, we allow checking through higher level sentence corpora to a certain extent.

The whole initial filtering process, which involves filtering out dirty data as mentioned in the previous section, as well as dividing the approved sentences into CEFR-based corpora, is visualized in Figure 2.2, where the end result is a collection of sentence corpora based on CEFR level.

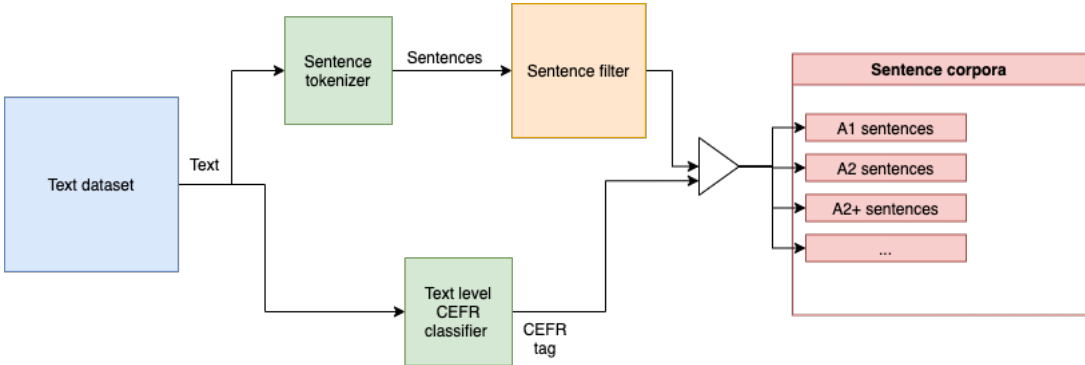


Figure 2.2: Chart describing the initial filter process.

Chapter 3

Theoretical background

This chapter will discuss some relevant theoretical background for the project.

3.1 Statistical language modeling

The most basic method of language model is the purely statistical one. Typically, calculating the probability of a word sequence is simply done through conditional probabilities, such as:

$$P(w_1 \dots w_i) = P(w_1) \times P(w_2|w_1) \times \dots \times P(w_i|w_1 \dots w_{i-1}) \quad (3.1)$$

For large values of i , this can be difficult to calculate, so in many applied contexts, it is common to assume that the probability of a word mainly depends on the two preceding ones. This is known as the *trigram assumption*, and thus only requires collecting n-grams of sizes up to three from a training corpus. A key issue with this approach, is that any n-gram that is not seen in the training corpus will automatically have a probability of 0. To solve this issue, *smoothing* techniques can be utilized to redistribute the probabilities (Goodman, 2001).

3.2 Recurrent neural networks

The simplest implementation of a neural network is the Feed-forward Neural Network (FNN), where the data, as the name suggests, moves forward in one direction from beginning to end. However, they have never been heavily utilized for language modeling, since n-grams are conceptually simpler and yet capture much of the same data as an FNN might do (Sundermeyer et al., 2013).

It wasn't until the Recurrent Neural Network (RNN) that neural networks became a good option for language modeling. The main advantages of RNNs as compared to traditional FNNs are first of all the ability to cyclically store data in the network, and thus produce an output that not only depend on the last input, but also previously seen data, and secondly

that RNNs allow for variable length input, whereas FNNs always have a fixed input size (Mikolov et al., 2010). Starting with a simple three-layered RNN, with one input layer, one hidden (state) layer and one output layer, the input to the hidden layer will be a concatenation of the current word representation and the previous hidden layer output (of the previous time step). This method of using a previous state simply as an additional input is known as *truncated backpropagation*, and the model is usually called a *simple recurrent network* or SRN (Bodén, 2002), and was first suggested by Elman (1990). Assuming an extended, multi-layer SRN, the computation of each layer can be described with the following equation, taken from `pytorch`:

$$h(t) = \tanh(W_{ih}x(t) + b_{ih} + W_{hh}h(t-1) + b_{hh}), \quad (3.2)$$

where $h(t)$ is the hidden state of the current layer, $h(t-1)$ is the hidden state of the layer at the previous time step, $x(t)$ is the input, W is the weight matrix of the current layer, and b_{ih} and b_{hh} are possible biases. In this design, for each computation, only one earlier state is used, and the error of this state is used to modify the weights in the hidden layers. However, rather than just inputting the state of the previous time step, it can be extended to include an arbitrarily large amount of previous states, which is known as *backpropagation through time*, or BTT (Bodén, 2002).

BTT does not come without its flaws, though. As it happens, RNNs are difficult to train using BTT, because of a phenomenon known as the vanishing gradient problem, which means that any error that is backpropagated will eventually either decay and vanish, or grow at an exponential rate (Hochreiter and Schmidhuber, 1997). What causes this is that the error is scaled by a factor at every time step, and this factor is in practice almost always greater than or less than one.

To solve this, LSTM (Long Short-Term Memory) was introduced by Hochreiter and Schmidhuber (1997). Here the corresponding factor is fixed to one, but this alone results in a unit with limited learning potential. This caused additional components, called *gating units* to be added to the design. In the original report by Hochreiter and Schmidhuber (1997), the LSTM unit only contained an *input gate* and an *output gate*, whose main role is regulating read and write access to the unit. However, Gers et al. (1999) showed that this design was flawed, in that it had issues with processing continual or exceptionally long time series, since this could cause unbounded growth of some of the unit's internal values. Their solution to this problem was adding a *forget gate* to the LSTM unit, whose role is to reset memory blocks where the contents are no longer used. With this addition, the LSTM as we know it today was created. For all inputs $x(t)$, an LSTM layer computes the following:

$$h(t) = o(t) * \tanh(c(t)) \quad (3.3)$$

where $*$ is the element-wise (Hadamard) product, and $c(t)$ and $o(t)$, as well as their dependencies, are given by:

$$\begin{aligned} c(t) &= f(t) * c(t-1) + i(t) * g(t) \\ o(t) &= \sigma(W_{io}x(t) + b_{io} + W_{ho}h(t-1) + b_{ho}) \\ g(t) &= \tanh(W_{ig}x(t) + b_{ig} + W_{hg}h(t-1) + b_{hg}) \\ f(t) &= \sigma(W_{if}x(t) + b_{if} + W_{hf}h(t-1) + b_{hf}) \\ i(t) &= \sigma(W_{ii}x(t) + b_{ii} + W_{hi}h(t-1) + b_{hi}) \end{aligned}$$

Here, $i(t)$, $o(t)$ and $f(t)$ are the input, output, and forget gates, respectively, and $c(t)$ is the so called *cell state*, which handles the memory of the LSTM unit. In Figure 3.1, the three gating units are clearly visualized, and \tilde{c} represents the new cell state.

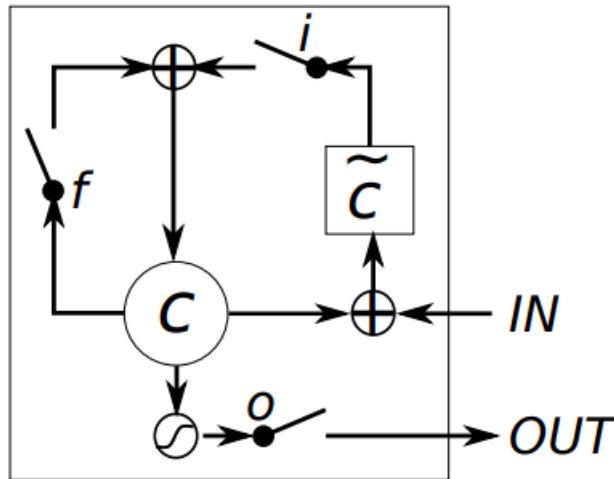


Figure 3.1: LSTM unit as presented by Chung et al. (2014).

This all makes the LSTM unit somewhat complicated to implement, which called for a simpler alternative. This caused the GRU, or Gated Recurrent Unit, to be introduced by Cho et al. (2014). A GRU has two types of gating units, compared to the three different ones in an LSTM unit, and the computation for GRU layers is represented by the function, again taken from `pytorch`:

$$h(t) = (1 - z(t)) * n(t) + z(t) * h(t - 1) \quad (3.4)$$

where $n(t)$ and $z(t)$ are given by:

$$\begin{aligned} n(t) &= \tanh(W_{in}x(t) + b_{ir} + r_t * (W_{hn}h(t - 1) + b_{hn})) \\ r(t) &= \sigma(W_{ir}x(t) + b_{ir} + W_{hr}h(t - 1) + b_{hr}) \\ z(t) &= \sigma(W_{iz}x(t) + b_{iz} + W_{hz}h(t - 1) + b_{hz}) \end{aligned}$$

In these equations, σ is a logistic sigmoid function, $r(t)$ computes the *reset gates* and $z(t)$ the *update gates*. The reset gates control whether the GRU should use or ignore previous hidden states, while the update gates control how much information is actually carried over from these states. What this means in practice is that units that have active reset gates will be capturing more short-term dependencies, while units with more active update gates will capture more of the long-term dependencies (Cho et al., 2014). Figure 3.2 shows a GRU unit, and comparing it with Figure 3.1 it is easy to see that the design of the GRU is comparably simpler.

In an evaluation of the three mentioned approaches, Chung et al. (2014) deemed LSTM and GRU superior to the more traditional RNN with a \tanh unit, but could not conclude whether one of the gated units was superior to the other and simply deemed them to be comparable, and that the choice between the two might depend on the task at hand.

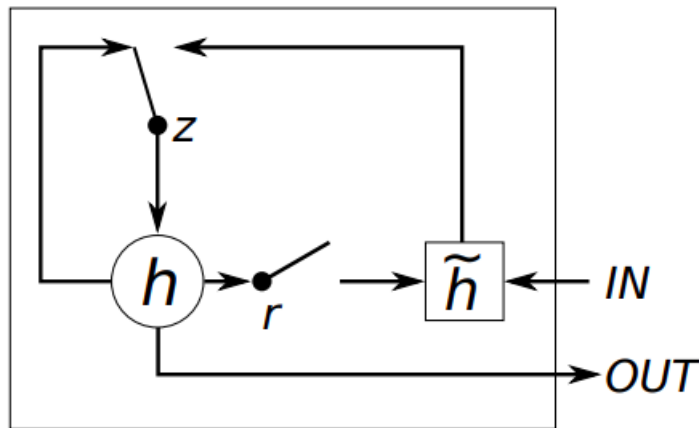


Figure 3.2: GRU unit as presented by Chung et al. (2014).

3.3 Transformers

The transformer was first introduced by Vaswani et al. (2017). The problem they address within recurrent models is that they rely on sequential processing of data while training, which makes them inefficient when the input or output sequences become long. The design that they suggest is not based on recurrence, but rather relies on an attention mechanism to find dependencies in the data. This enables more parallelization of the data, which greatly reduces training times. An attention function is described as projecting the input onto three matrices named Q , K and V for query, keys and values respectively.

The attention mechanism is based on “Scaled Dot-Product Attention”, which given the matrices Q , K and V , as seen in Figure 3.3, is calculated by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (3.5)$$

Rather of simply applying this function once to an input, the authors suggest applying this function to different projections of the data, to increase chances of learning different dependencies in the input. The result of each project is stored in a separate *attention head*, which are then concatenated into what is called “Multi-Head Attention”, seen in Figure 3.4, as formulated in Vaswani et al. (2017):

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

In these equations, h is the amount of parallel attention layers, and all the W matrices are model parameter matrices, whose dimensions are decided by d_{model} , which is the dimension of the model input, and their respective linear projection dimensions. These are d_k for the queries and keys, and d_v for the values.

Expanding on this, Devlin et al. (2019) added bidirectional attention to the transformer, and introduced BERT, or Bidirectional Encoder Representations from Transformers. While

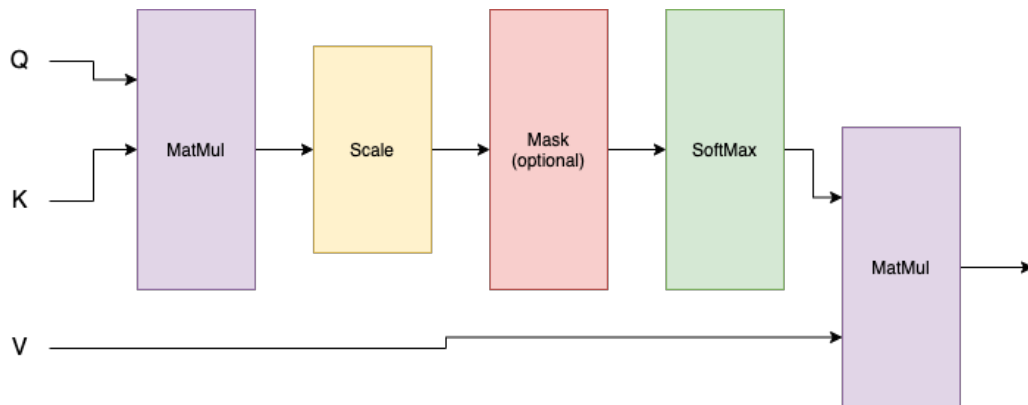


Figure 3.3: Scaled Dot-Product Attention as described by Vaswani et al. (2017).

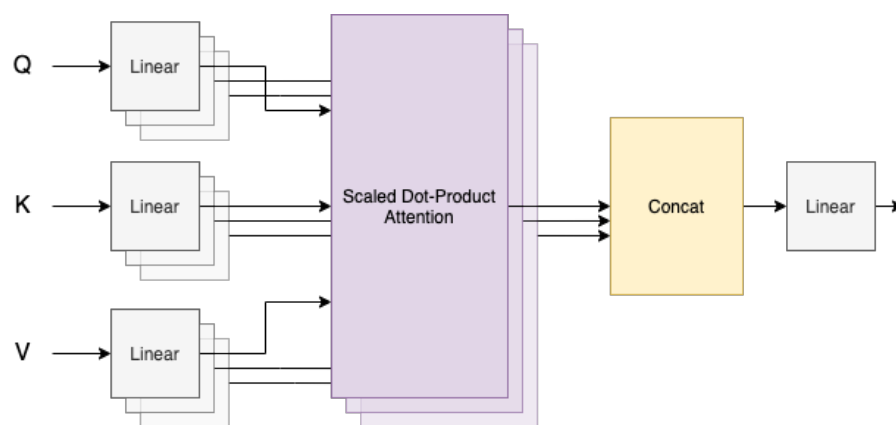


Figure 3.4: Multi-Head Attention as described by Vaswani et al. (2017).

most language models rely on left to right context to predict a word, bidirectional models also allow for right to left context, which means that they capture contexts on both sides of a word from both directions.

To allow easy usage in a range of different tasks, BERT is pre-trained on two tasks, and can then be fine-tuned to perform other, perhaps more specific, tasks. The first of these two training tasks is masked word prediction, which in practice is the same as solving cloze tests. This task is meant to give BERT a “deep bidirectional representation” of the data, since if the word was not masked, the bidirectional representation would cause the prediction of the word to be trivial. The training is done by randomly choosing 15% of the tokens in a sequence, and then, for each of the chosen tokens, replacing them with a mask token 80% of the time, a random token 10% of the time, and the token itself 10% of the time.

The second task that BERT is trained on is next sentence prediction, which is meant to capture dependencies between sentences, rather than just within them. This is done through a binary classification in which training examples 50% of the time are generated with two sentences that actually follow one another in the corpus, and 50% of the time with two sentences where a random sentence is following the first one, with these training samples being labeled as “IsNext” and “NotNext”, respectively (Devlin et al., 2019).

Another thing that is noteworthy in BERT is its approach to word embeddings. In the

input to the model, each token of the text is represented by an embedding that is a sum of three different embeddings. These are:

- Token embedding, which represents the string itself
- Segment embedding, which tells us which of the two sentences the token belongs to
- Position embedding, which represents at which position the token occurs

This allows the word embeddings in BERT to capture a lot of relevant context-specific data in an efficient way, and is shown in Figure 3.5. This combined with the aforementioned bidirectionality and pre-training is what allows BERT to perform so well on a big variety of tasks.

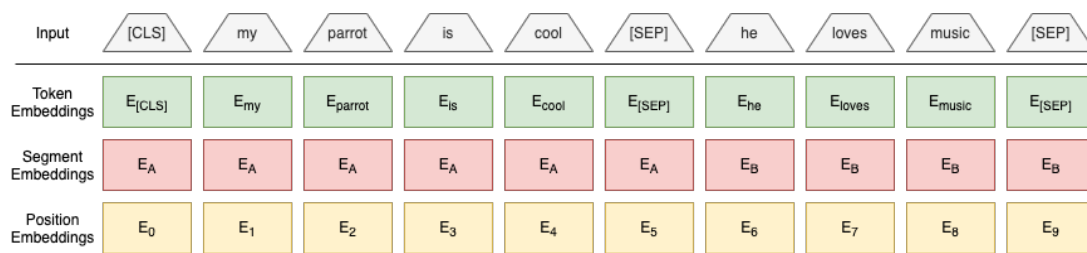


Figure 3.5: BERT word embeddings as described by Devlin et al. (2019).

Chapter 4

Model testing

This chapter will present the first main phase of the experiments, namely testing models to see which one is best suited for the later implementation phase.

4.1 Test cases

Five different test cases of masked word prediction were used, all based around a single word class, to limit the label space since training a GRU model to predict any type of token was deemed infeasible. The first task was predicting five of the most common prepositions in the English language (*at, in, of, on, to*). After this task, the number of possible prepositions was increased to 12, adding *as, for, from, off, out, up, and down*, and the final task for preposition prediction included all 54 common prepositions as specified by Cambridge Dictionary. These tasks will from now be referred to as *5P, 12P* and *54P*.

The models were also tested on predicting the 25 most common verbs taken from EF, with all verb forms included which resulted in around 200 labels, and lastly on the 50 most common nouns taken from the Oxford Dictionary, with both singular and plural forms included. Similarly as above, these tasks will be referred to as *25V* and *50N*.

4.2 Model types

The different models that were considered were initially a self-created model based on a recurrent neural network, as well as a BERT model. The BERT model used was a version known as BertForMaskedLM, which has a masked language model head already ready to use, which meant that no fine-tuning was necessary.

The self-made model had a GRU-based architecture, which consisted of an embedding layer, a Bidirectional GRU layer, and a dense layer with softmax activation. Bidirectional GRU was used instead of a regular GRU to hopefully increase performance, and to gain a

better comparison between it and BERT. To avoid overfitting during training, the Bidirectional GRU layer was surrounded by two dropout layers with a rate of 0.5.

The GRU model was trained specifically for one task at a time, while with BERT, as mentioned before, there was no need for any training. What this meant was that the GRU models only considered the specified limited label space for the current task, whereas BERT considers the whole dictionary it was trained on when predicting a masked word, which is around 30,000 words. We wanted to see how BERT would perform given the same limited amount of options for a more meaningful comparison, which led to a third model variant, Multiple-choice BERT, or McBERT for short.

McBERT differs from standard BERT in that it, like the GRU model, is given the available labels for the specified task, and is only allowed to choose from these when making its prediction. The way this works is that McBERT first makes a regular BERT prediction over the whole available dictionary. Then the predictions are iterated over until one of the available labels is found. So for the task of predicting the most common five prepositions, the predictions are iterated through until one of these prepositions is found.

Another difference between the GRU model and the BERT models is that the latter take whole sentences as input when predicting, while the former takes a symmetrical limited context surrounding the masked word. In the experiments the context in both directions were four tokens, with padding tokens at the start and end of sentences.

4.3 Testing methodology

The first step of model testing involved creating the GRU model and training it on a sufficient amount of data for each task. Due to the heavy computations involved in training the model, Google Colab was used, which allows for code execution on a hosted GPU.

The GRU model was created with the Python package `keras` with layers as specified in the previous chapter.

Samples for training and testing were extracted from a collection of 10,000 articles from the Knowble dataset in the 5P, 12P and 54P tasks, whereas in 25V and 50N, 20,000 articles were used. This was simply an effect of verbs and nouns being more diverse than prepositions, and as such they yield fewer samples from the same collection of data. For comparison, data extraction from 10,000 articles for the 5P task yielded 362,744 samples, while 20,000 articles for the 50N task yielded 249,340 samples. Furthermore, since all forms of the verbs and nouns were used, the label space in the 25V and 50N tasks were larger than for the preposition prediction tasks. A lemmatizer in the form of `WordNetLemmatizer` from `nlTK` was used to find the base form of words, which for 25V resulted in around 200 unique labels of different verb forms, and for 50N in around 100 labels, roughly corresponding to singular and plural forms, with some exceptions.

60 percent of the total amount of samples were used for training the GRU model, and 20 percent each for validation and test data. GRU model training samples consisted of context from a window with a width of 4 tokens surrounding the target word as feature data, and the target word as the label. As an example, if we have the sentence *The bird is sitting in the tree.* and the target word is *in*, the feature vector and label of the extracted sample will be represented as ([the, bird, is, sitting, the, tree, ., <PAD>], in). The GRU model as it was implemented is visualized in Figure 4.1.

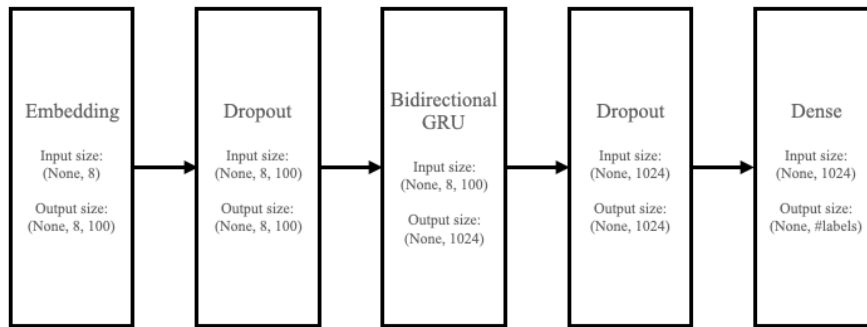


Figure 4.1: Rough visualization of the GRU network. The output size of the Dense layer depends on the size of the label space.

The GRU model was trained for between 5 and 10 epochs, depending on the task. For tasks with fewer labels, overfitting tended to occur at an earlier stage, so fewer epochs were used in training to hopefully reach an optimal result. For example, for the 5P task, training was done for 5 epochs, while for 25V and 50N it was done for 10 epochs. The loss function used was categorical cross-entropy, and the optimizer was RMSprop. Some initial experiments showed no significant difference in performance between RMSprop and Adam, another popular optimizer. Adam was however far slower to train with than RMSprop, which led to the choice of the latter.

While the GRU model was tested on the whole collection of test data, the BERT models were tested on 5000 test samples, since they happen to have a very slow prediction function compared to the GRU model.

4.4 Measurements

The results of the model testing phase can be found in Table 4.1 and 4.2, showing the test accuracy for each model in every task. The test accuracy was calculated with the `accuracy_score` function from the Python package `scikit-learn`, and simply calculates the fraction of samples for which we predict the correct label.

Task	GRU	McBERT
5P	0.825	0.942
12P	0.738	0.898
54P	0.647	0.830
25V	0.604	0.796
50N	0.539	0.757

Table 4.1: Results from testing the GRU and McBERT models.

As shown in Table 4.1, in every task, McBERT outperformed the GRU model. In fact, as seen in Table 4.2, even when BERT is not given the available labels, it performs better than the GRU model. Given that BERT has already been proven as the state of the art in so

Task	BERT
5P	0.898
12P	0.846
54P	0.795
25V	0.695
50N	0.559

Table 4.2: Results from testing the regular BERT model.

many tasks, this was not very surprising, but there are some interesting things to note if we compare the results between BERT and McBERT.

For the preposition tasks, the scores do not differ much, which suggests that prepositions are less ambiguous to predict. This should not be surprising, since they are often part of fixed grammatical constructions, such as “believe **in**”, “**in** spite **of**” and “moving **on**”. On the other hand, the scores for the noun prediction task differ a lot, likely since nouns are often more ambiguous, and can be more easily replaced by other nouns without violating grammatical rules.

The obvious choice for an implementation based on these results would seem to be McBERT. However, there were a few issues with using McBERT in the intended applied context. If it were feasible to generate tests from scratch at runtime for each article instance, McBERT would be a perfect fit. Imagine that we have a five words for which we want to generate cloze tests, and we input these to a McBERT-based test generator. Then we could make sure that each of these tests was optimally chosen given both their target word and the other words, since we can guarantee that none of the other four words fit better in the gap than the intended target word. Unfortunately, since there are computing capacity constraints involved, the best solution to an applied implementation would be to generate one or a few cloze tests for each word that we could want a user to learn, and store these in a database. Since we don’t know which words will occur together in practice, there is no good way to do this with McBERT. Rather than generating cloze tests that are perfectly suited to a specific scenario, the aim should be to generate cloze tests that are suitable in a wider context. Because of this, regular BERT was deemed to be a better choice for the applied implementation.

Chapter 5

Cloze test pipeline

After choosing which model to use in the final product, the second main phase of the project involved implementing the whole pipeline for generating cloze tests, including the neural language model filter for sentences.

5.1 Pipeline description

The implementation was done through a pipeline consisting of several stages, as visualized in Figure 5.1, where the input to the pipeline is a word, and the end output is a generated cloze test for that word.

The first step is the CEFR classification of the target word, after which the available sentence corpora are identified. These are then searched through in the specified order, until a sentence containing the target word is found. Each sentence corpus is randomized before iterating, to increase the chances of generating a different cloze test for the same word between instances. This sentence is sent to the BERT language model filter, where it is inputted to the `approve` function. This function tries to predict the gap where the target word occurs, and if the target word is in the top three predictions and the confidence level is over a specified threshold value, the function returns true. This is different from the model testing phase, where we only considered BERT's top prediction. This might seem like it could give way to cloze tests of lesser quality but this is why the confidence measure is utilized, to ensure that the word still fits well in the cloze test gap.

If the sentence is approved, it gets added to a list of candidates. If the list gets to a length of ten candidates, the iteration is interrupted and the list is returned. If the list is empty after iterating through the first corpus, we move on to the next one that is allowed for the target word's CEFR level, and if no sentence is found after iterating through all, we simply return an error message.

After extracting the cloze test sentence candidates, we choose the one with the highest probability, through a function that calculates the joint probability distribution, the block

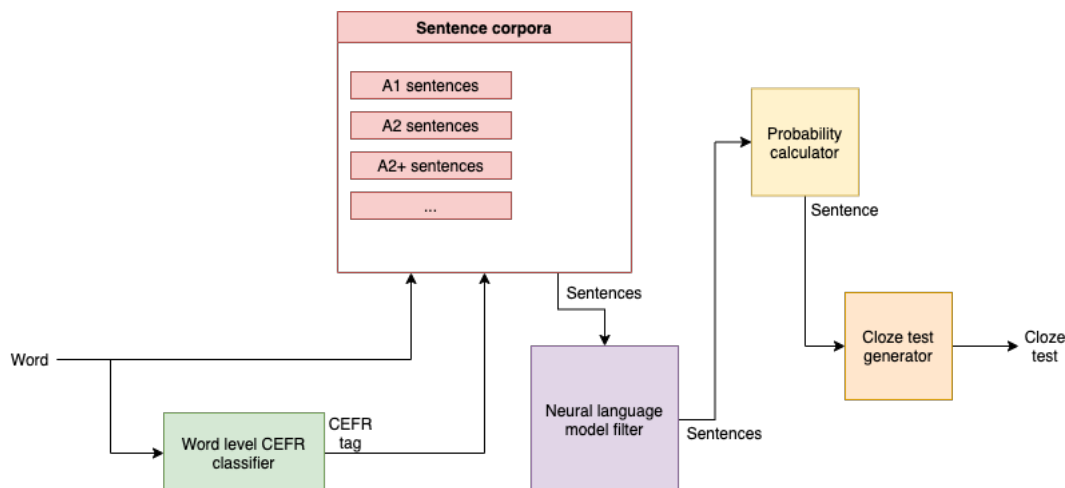


Figure 5.1: Chart describing the main pipeline from word to cloze test.

marked as “Probability calculator” in Figure 5.1. This functions works by tokenizing the sentence, and essentially trying to predict each token in it, and then calculating the cross-entropy loss between the predicted and the actual sentence. This is to increase chances of getting a sentence that isn’t too specific, since it can feel a bit awkward and too detached from the text if the sentence for example contains a bunch of names of people completely unrelated to the article. After the most probable sentence has been identified, a cloze test is generated by simply tokenizing this sentence, replacing the target word, wherever it occurs, with a blank, and then putting the sentence back together through detokenization.

5.2 Pipeline implementation

The implementation was completely written in Python using the Spyder Python IDE and was done with a client-server design through the gRPC Python package. In this design, the client registers the words that a user wants to generate cloze tests for, and sends them to the server. The client class only does one thing apart from this, which is parsing the input of the user into separate words. The server class on the other hand holds all the blocks specified in Figure 5.1. Some of these are objects with their own class, more specifically the word level CEFR classifier and the neural language model filter. These are created when the server starts up, and both act as attributes to the server class. The sentence corpus also serves as an attribute, in the form of a dictionary that is generated from the CEFR sentence files, also when starting up the server. Lastly, the probability calculator and cloze test generator blocks are not objects but are represented by class methods of the server.

Chapter 6

Evaluation

6.1 Evaluation of cloze tests

A question that was not immediately easy to answer was how the quality of the cloze tests should be evaluated. After some consideration, the only feasible option seemed to be asking an expert to evaluate the quality of the cloze tests generated with the new method as compared with the previous one.

The evaluation data was taken from the Knowble platform in the form of 50 news articles and associated cloze tests, for a total of 150 cloze tests. Then, half of these cloze tests were exchanged with ones generated with the new method, naturally with the same target words. For these samples, only drag words cloze tests were used. This was a result of several factors.

Dictation tests were excluded since integrating voice samples into the evaluation was not deemed feasible. It was also easier for the evaluation to only consider one type of cloze test, since it makes results easier to compare. And lastly, the drag words cloze tests are what one could call the “weakest link” of the three types, since it is the most reliant on the sentence having sufficient context. If context is lacking in dictation, the user can simply listen to the voice sample, and if it’s lacking in a finish words cloze test, having the first letters given is likely to heavily guide the user in the right direction.

Since all sentences can be used for all three types of tests, the assumption was made that if a sentence is fitting in a drag words test, it would also work well with the other types of tests.

The expert, a professor of English language, was then asked to evaluate the tests in terms of how easy they were to solve. We assume that if a professor of English has issues figuring out which word goes in which gap, the tests are probably not fit for the cause.

6.2 Data gathering for evaluation

Gathering the data for the evaluation step was done manually by fetching articles with associated cloze tests from the Knowble platform. This proved a rather tedious step, since Knowble often presents the same cloze test for several articles, and we wanted to avoid duplicates in the evaluation data.

For half of the articles, the three cloze tests were then exchanged for ones generated with the new method, for the same target words. Initially, a confidence threshold value of 15 was used, and if it failed to generate any cloze tests because the condition was too strict, the threshold was first lowered to 12. If still no sentence was found, it was lowered yet again to 10.

6.3 Results

Table 6.1 shows the results of the cloze test evaluation.

	Old method	New method
#Approved	34	67
#Not approved	41	8
Proportion approved	0.453	0.893

Table 6.1: Results from the evaluation of the cloze tests.

6.4 Discussion

6.4.1 Cloze tests

From the results seen in Table 6.1, it's safe to say that the statistics show a significant improvement from the old method. What these results suggest is that a language model can indeed be used to simulate the experience of an average human user to a high degree.

Figures 6.1 and 6.2 show two sets of cloze tests where in one, all three were approved and in one, none were approved. Additionally, in this case the approved ones were generated with the new method and the non-approved with the old one.

Starting with Figure 6.1, the correct words to fill in the blanks are 'profits', 'digital' and 'sales', in order. For each of these words, there are other words in the sentence providing important context. In the first test, 'given to charity' and 'from the concert' together imply something of value that can be given and received. 'Sales' could go with 'from the concert' but cannot be 'given to charity', and neither can 'companies'. In the third test, one could argue that both 'sales' and 'profits' qualify, but since one of them has already been used, the test is solvable through process of elimination.

As for Figure 6.2, the correct words in order are 'system', 'target' and 'included'. Looking at the first test here, any of the words except 'included' can fit in the gap, and the same goes for the second test. One could argue that the third test is solvable since no other word fits

grammatically, but the test does not provide a good example of how the word ‘included’ is commonly used. Also, if there was any other adjective or plural noun in the available words, the test would likely become much more difficult to solve. While the new method might not be 100% proof, these three examples still provide good examples of the types of sentences that will definitely be avoided.

Fill in the blanks!

sales, profits, digital, system, companies

- All the ____ from the concert were given to charity.
- Windows Media Player is a ____ media player made by Microsoft.
- The company has annual ____ of over \$5.84 billion.

Figure 6.1: Example of cloze test set where all of the tests were approved.

Fill in the blanks!

included, network, target, information, system

- That is the best ____ we have.
- That was the ____ from the beginning.
- If they were, they were ____.

Figure 6.2: Example of cloze test set where none of the tests were approved.

Which five words the user is allowed to choose from in a drag words test can affect how the test is perceived, and how these words are picked has not been altered in the new model. If an even smaller evaluation data set was used, this could be a randomness factor that might skew the results, but I would argue that the amount of samples combined with the big performance difference is enough to say that this cannot have affected the end results in any substantial way. For that same reason, I also argue that the method would be beneficial to use for the other two types of cloze tests that were not included in the evaluation data.

In the data generation step, there were sometimes words for which no sentences were found. If lowering the confidence threshold value did not help, the issue was either that no sentence containing the word existed, or that BERT could not predict the given word. BERT being unable to predict a word was in all observed cases caused by the word not existing in BERT’s vocabulary list. Words not in this list are interpreted as several smaller segments by BERT’s tokenizer, and are thus not considered when predicting a gap of a single mask token. A solution to this would be to first check how the target word is tokenized by BERT, and then replace the word in the sentences with the appropriate amount of mask tokens.

6.4.2 Future work

There are a few aspects of cloze test generation that this project does not involve. One of these is ambiguity between homographs. As of now, we are simply given a word as a string identified from the article the user is reading, and we try to find a fitting cloze test given only the raw string. What is not taken into account is the actual meaning of this string, which can be an issue where a word can have several meanings. A desirable extension would be to extract the words from the article with contextual embeddings, from which we then could look for sentences, and generate cloze tests where the word is used in a similar fashion to how it's being used in the article.

The choice of which words to generate cloze tests for is also an aspect to work with, by choosing the words intentionally. Especially for the drag words test, which words the user is allowed to choose from will greatly affect how the test is perceived by a user. There are too many angles to consider in this aspect to generalize, but for example, if only one of the words to choose from is a noun, it's likely that fitting it into a gap will be trivial since all of the other words would make the completed sentence ungrammatical.

Using the method suggested in this thesis, it would be possible to later add more customization to the choice of target words, for example generating a set of cloze tests where all the words are of the same word class, and still make sure that the tests aren't too ambiguous. Using the theory applied in McBERT could be useful to make sure that each target word fits best only in its own cloze test, but the current method with the top three predictions combined with the threshold value makes it at least very unlikely that any other target word than the actual one fits best, given a cloze test.

For now, the cloze tests are only generated for single words, but something that could be useful for a learner is to also generate tests for specific grammatical constructions and phrases. As briefly touched upon in the Model testing chapter, prepositions are often part of constructions like verb phrases, and which prepositions are used for certain verbs often vary between languages, which can be challenging for learners. In English you 'believe in' something, whereas in Swedish you 'believe on/at', for example. Adding this type of functionality would make users not only increase vocabulary, but also learn grammatical rules in a more active way than is currently possible.

Another problem with the new method is that because of its slow speed, it is not feasible to run the code live in a service, but rather it should be used to generate cloze tests for expected words in advance. For a completely dynamic approach, a faster solution would have to be utilized. This is likely to become easier as new, faster and even more precise language modeling methods are discovered.

Chapter 7

Conclusion

The goal of this thesis project was to improve the quality of provided context in automatically generated cloze tests. Looking at the results, despite the rather small evaluation set, it's safe to say that goal was achieved. This is not particularly unexpected, since what was done is simply to ask a language model if it thinks there is enough context provided to predict a masked word. The solution is intuitive and conceptually simple, and solves one of the key issues with cloze test generation.

BERT proved to be a great model for the current approach, although its slow prediction did have some consequences. With newer and better approaches, this method will only get easier and more efficient to use.

There still remains several other aspects of cloze test generation to consider in future work. These include choosing actively which target words to group together and generate tests for, and in the case of the Knowble platform, making sure the cloze tests make sense in relation to the article, which in part depends on solving word disambiguation.

In short, the approach suggested in this project solves the aspect it intends to, but there still remains others to consider in order to further improve the quality of automatically generated cloze tests.

References

- Bodén, M. (2002). A guide to recurrent neural networks and backpropagation. In *THE DALLAS PROJECT, SICS TECHNICAL REPORT T2002:03, SICS*.
- Cho, K., van Merriënboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.
- Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.
- Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.
- Gers, F., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2.
- Goodman, J. (2001). A bit of progress in language modeling. *CoRR*, cs.CL/0108005.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.
- Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In Kobayashi, T., Hirose, K., and Nakamura, S., editors, *INTERSPEECH*, pages 1045–1048. ISCA.

- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1 – 67.
- Sundermeyer, M., Oparin, I., Gauvain, J.-L., Freiberger, B., Schlüter, R., and Ney, H. (2013). Comparison of feedforward and recurrent neural network language models. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8430–8434.
- Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *Proc. of Interspeech 2012*, pages 194–197.
- Taylor, W. L. (1953). “cloze procedure”: A new tool for measuring readability. *Journalism & Mass Communication Quarterly*, 30:415 – 433.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
- Zhou, J., Xu, J., and Yang, J. B. (2019). Cs 224n project: Cloze answer generator. Technical report, Department of Computer Science, Stanford University.

Appendices

EXAMENSARBETE Neural Sentence Filter to Improve Context in Cloze Tests**STUDENT** Ludvig Rassmus**HANDLEDARE** Pierre Nugues (LTH), Mark Breuker (EDIA)**EXAMINATOR** Jacek Malec (LTH)

Godkännande av meningar till lucktexter med neuralt nätverk

POPULÄRVETENSKAPLIG SAMMANFATTNING Ludvig Rassmus

Lucktexter är en typ av test som används inom ett antal olika områden, bland annat språkinläring. Detta projekt gick ut på att ta fram en metod för att garantera att testerna innehåller tillräckligt kontext för att en användare skall kunna lösa dem.

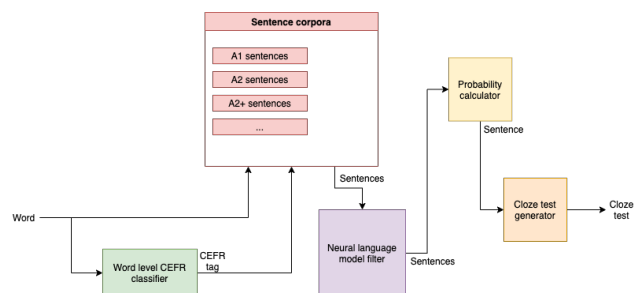
På senare år har det gjorts många framsteg inom området språkmodellering, som lätt förenklat går ut på att beräkna sannolikheter för sekvenser av ord. Med nya tekniker har det blivit lättare än någonsin att skapa och träna träffsäkra maskinlärningsmodeller som kan användas inom en mängd olika områden. Ett sådant användningsområde utforskade jag i detta projekt, i form av lucktexter (engelska Cloze tests).

Lucktexter är texter där ett eller flera ord tagits bort för att en elev skall fylla i dem. För att en lucktext ska kunna ha något pedagogiskt värde, så måste den kanske framförallt innehålla tillräckligt med kontext runt saknade ord att det går att gissa vad som tagits bort. Till exempel, om en elev får texten "Jag går till ___" och ombeds att fylla i det saknade ordet, finns det alldeles för lite kontext i meningen för att det skall vara rimligt att lista ut.

En tidigare metod för att ta fram lucktexter automatiskt använder sig av en stor samling text att hämta meningar ifrån. Givet ett ord som man vill generera ett test för, söker man efter meningar som innehåller det önskade ordet, och väljer sedan den mest sannolika meningen för att generera ett test från. Problemet är att denna metod ofta resulterar i meningar som den ovan, med för lite kontext. För att lösa detta problem använde jag

samma metod, men med ett extra steg för att välja en passande mening. Detta steg innebär att en språkmodell baserad på ett neuralt nätverk får testa att prediktera det saknade ordet, och om modellen lyckas hitta rätt ord med en tillräckligt hög säkerhet görs antagandet att en människa antagligen också kan det.

Figuren nedan beskriver processen där indatan är ett ord som vi vill generera en lucktext till, och resultatet är en lucktext genererat från en passande mening hämtad från vår textsamling.



Med hjälp av en docent i engelska utvärderades de automatiskt genererade testerna, och resultaten visar att med hjälp av en språkmodell kan vi garantera att nästan 90% av testerna håller god kvalitet ur kontextsynvinkel, till skillnad från ungefär 45% med den gamla metoden.