Neural Language Models for Spelling Correction

Master's thesis

Matthias Hertel

December 6, 2019

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik
Lehrstuhl für Algorithmen und Datenstrukturen

Working period

07. 06. 2019 – 06. 12. 2019

Supervisor

Prof. Dr. Hannah Bast

Examiners

Prof. Dr. Hannah Bast
Prof. Dr. Frank Hutter

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Abstract

Spelling correction is the task of automatically recovering the intended text from a misspelled text. This enhances digital communication and enables Natural Language Processing and Information Retrieval systems to work with misspelled text. We propose NLMspell, a spelling corrector based on a neural language model. NLMspell achieves correction F-scores of 91.5 % on artificial misspellings and 88.4 % on real misspellings. It outperforms our baselines methods based on traditional language models, as well as our neural machine translation approach to spelling correction, and the commercial product of a large Internet company.

Zusammenfassung

Die Rechtschreibkorrektur ist die Aufgabe, den beabsichtigten Text aus einem falsch geschriebenen Text automatisiert wiederherzustellen. Dies verbessert die digitale Kommunikation und ermöglicht es Systemen zur Verarbeitung natürlicher Sprache und zur Informationsbeschaffung, mit falsch geschriebenem Text zu funktionieren. Wir schlagen NLMspell vor, einen Rechtschreibkorrektor, der auf einem neuronalen Sprachmodell basiert. NLMspell erreicht Korrektur-F-Scores von 91,5 % bei künstlichen Schreibfehlern und 88,4 % bei echten Schreibfehlern. Es übertrifft unsere Baseline-Methoden, die auf traditionellen Sprachmodellen basieren, sowie unseren neuronalen maschinellen Übersetzungsansatz zur Korrektur der Rechtschreibung und das kommerzielle Produkt eines großen Internetunternehmens.

Acknowledgement

The author would like to thank

- Prof. Dr. Hannah Bast for support during all phases of this thesis.
- Markus Näther for support with the GPU cluster and hints on spelling correction literature and benchmarking.
- Frank Dal-Ri for technical support.
- Heike Hägle for administrative support.
- My family, girlfriend and friends for emotional support.
- You, the reader, for your interest in this thesis.

Contents

1. Introduction

1.1. Motivation

Misspelled text can be difficult to read for humans and is problematic for Natural Language Processing and Information Retrieval systems.

A lot of text is written with human-computer interfaces such as physical and digital keyboards. When writing, humans introduce noise into text due to fast typing or because they lack the knowledge about the correct orthography. With the upcoming of small smartphone and wearable devices, these interfaces become more vulnerable to noise. Although human reading is rather robust against misspellings, we need more time to read a misspelled text than to read a correctly spelled text [Rayner et al., 2006]. Misspellings can make a text incomprehensible or even negate the meaning of a text.

Almost all smartphone operating systems address this problem with autocorrection methods, which predict the intended text from the noisy input. When those systems fail in predicting the correct text, parsing the meaning of a sentence can become even harder, since the wrong predictions are more difficult to detect than the misspellings, and the predicted text can be orthographically less similar to the intended text than the misspelled input. Recently, the wrong-going of online conversations due to false autocorrections even became a common internet meme[1].

Misspelled text is not only problematic for human readers, but also for Natural Language Processing systems. Noisy input can break the performance of such systems, which usually assume correctly spelled input. This can lead to an Information Retrieval system not returning the desired results for a noisy query [Manning et al., 2010], or a mistranslation of an input sequence by Neural Machine Translation systems [Belinkov and Bisk, 2018]. Many Natural Language Processing systems represent words as word embedding vectors [Mikolov et al., 2013, Devlin et al., 2019]. When a word is misspelled, no pre-trained word embedding can be retrieved. Recent work has shown that text classifiers are prone

[1]See, for example, `http://www.autocorrectfail.org/`, *"Damn You, Autocorrect!"* by Jillian Madison, and *"Damn You, Autocorrect! 2"* by Lyndsey Saul.

to adversarial misspellings, meaning that a small change in the input text can flip the classification result, which can result in wrongly classified sentiment [Pruthi et al., 2019], spam filters not recognizing spam e-mails, toxicity detectors like Google's *Perspective* not recognizing toxic content, and hate speech detectors not detecting hate speech [Gong et al., 2019]. However, correcting the input text before handing it to the Natural Language Processing system can recover the system's performance.

1.2. Task definition

Spelling correction is the task of predicting the intended text $\bar{S}$ from a misspelled text S.

1.3. Outline

This thesis is divided as follows:

- Section 2 gives an introduction to recent developments in language modeling, neural machine translation and spelling correction.
- Section 3 explains background on the methods used in this work.
- Section 4 describes the datasets and benchmarks used to train and evaluate the different methods.
- Section 5 describes deep learning approaches to spelling correction.
- Section 6 describes the baseline methods.
- Section 7 describes the experiments to evaluate and compare the methods.
- Section 8 discusses limitations of this thesis.
- Section 9 gives a conclusion.
- Future work on the topic is proposed in section 10.
- Appendix A contains results of the hyperparameter optimization experiments.
- Appendix B describes a method to extract typos from the World Wide Web.

2. Related work and contribution

This section outlines recent progress in neural machine translation in 2.1, describes recent developments in neural language modeling in 2.2, presents related work on spelling correction in 2.3, and states our contribution to the spelling correction research in 2.4.

2.1. Neural machine translation

Recent progress in machine translation was achieved with sequence-to-sequence neural networks [Sutskever et al., 2014], that consist of a recurrent encoder network which transforms an input sequence into a fixed-size vector representation, and a recurrent decoder network which generates an output sequence from that representation.

This approach was improved with the introduction of the attention mechanism [Bahdanau et al., 2015], allowing the network to look back to specific parts of the input sequence that are relevant for the prediction of the next word in the output sequence, instead of having to encode the whole input sequence into a single vector. Using the attention mechanism, a new state of the art on English-to-French translation was set [Bahdanau et al., 2015]. A few months later, Google adopted the encoder-decoder neural machine translation approach with attention for their machine translation system [Wu et al., 2016].

Not much later, a new single-model state of the art on English-to-French translation was achieved with the Transformer architecture [Vaswani et al., 2017], that relies solely on the attention mechanism. Getting rid of the recurrence scheme of the encoder and decoder networks allowed for better parallelization and reduced the training cost.

2.2. Language modeling

A key challenge in language modeling is to capture long-term dependencies of words, which is difficult for neural networks and impossible for n-grams. Recently, two neural network architectures developed for Machine Translation allowed to build models that cover some long-term dependencies: those are recurrent neural networks with attention and Transformer networks.

In [Radford et al., 2018] a high-capacity language model based on the Transformer neural network architecture is used as an unsupervised initial point for supervised training for natural language inference, question answering, semantic similarity prediction and text classification, improving the state of the art on 9 out of 12 studied datasets.

A similar, but deeper model is trained on a large, heterogeneous corpus scraped from the World Wide Web in [Radford et al., 2019]. It is able to generate consistent texts that are hard to distinguish from human-written texts [Solaiman et al., 2019]. Without further supervised training the model achieves state-of-the-art results on common noun prediction in the Children's book test (selecting one out of 10 choices for an omitted word), on the LAMBADA dataset (predicting the last word of a sentence) and in the Winograd SCHEMA challenge (resolving ambiguities in text). The claim that the model is a zero-shot multi-task learner is supported by the fact that it can be used for reading comprehension, summarization, translation and question answering, although its performance on these tasks is poor compared to the state of the art.

2.3. Spelling Correction

2.3.1. Overview

A literature review on spelling correction is given in [Kukich, 1992], which is summarized in the following.

Subtasks Traditional spell checking algorithms divide the task into three subtasks:

1. Error detection: distinguishing whether or not a word is misspelled.
2. Candidate generation: generating a set of candidate words that are similar to the misspelled word.
3. Candidate ranking: ranking of the candidate words according to the probability that they are the intended word for the misspelled word, or determin-

ing only the most likely candidate word.

Error types Various sources of noise in human-written text exist:

1. Typographic errors, also known as typos, which are one of the following operations:
   a) Insertion of a character.
   b) Deletion of a character.
   c) Replacement of a character.
   d) Transposition: swapping two neighboring characters.
2. Cognitive errors
   a) Phonetic errors: wrong word choice due to similar pronunciation of words (e.g. confusing *there* with *their*).
   b) Grammatical errors: wrong word choice or wrong sentence structure due to the lack of knowledge of grammatical rules (e.g. confusing *his* with *her*).

Error classes Errors can be divided into two classes: *nonword errors* where the resulting token does not exist in the target language, and *real-word errors* where the resulting token is a valid word of the target language, but different from the intended word. Real-word errors are usually more difficult to detect and correct than nonword errors.

Context-free and context-dependent error correction Approaches to spelling correction can be divided into *context-free* error correction algorithms (sometimes called *isolated-word* error correction algorithms) and *context-dependent* error correction algorithms.

Context-free error correction algorithms address the problem of detecting and correcting misspelled words without considering the context a word appears in. Error detection is done by a lookup in a dictionary of correctly spelled words, or by analyzing whether a word's character n-grams are all legal n-grams of the language at hand. Candidate generation and ranking is done based on the minimum edit distance to the misspelled word, similarity keys, character n-grams, confusion probabilities, word frequencies or rule-based.

Context-free error correction algorithms can only detect and correct a fraction of all spelling errors. Namely, they are unable to detect real-word errors. Context is also needed to dissolve ambiguities. For example, consider the misspelled word *yello*. Is the intended word *hello*, *yellow*, or *yell*? The answer depends on the context: different corrections are plausible for the sequences *'Bananas are yello.'*, *'yello world'* and *'Don't yello at me.'*.

2.3.2. Noisy Channel Model

Early context-dependent spelling correctors rely on the Noisy Channel Model [Shannon, 1948]. Given an observed word t, the model finds the candidate word c that maximizes the conditional probability $p(c|t) = p(t|c) \cdot p(c)$, where $p(c)$ is the prior probability given by a language model and $p(t|c)$ is the channel model.

Candidate ranking In [Church and Gale, 1991], the Noisy Channel Model is used to rank candidate corrections for words that the Unix *spell* program flags as misspellings. The prior model is a unigram or bigram language model, and the channel model is based on character confusion frequencies. Candidate words are those words from a word list that can be transformed into the observed word with a single character insertion, deletion, replacement or transposition operation. The test set contains only cases with exactly two candidates (in addition to the observed word), where the gold-standard intended word is one of them. Using the context-free unigram language model, the model prefers the correct candidate in 87 % of the test cases. This improves to 90 % with the bigram model.

Real-word error correction In [Mays et al., 1991] the Noisy Channel model is used to correct real-word errors. Candidate words come from a vocabulary with 20,000 words. The prior model is a trigram language model and the channel model assigns a constant probability α to the observed word and divides the remaining $1 - \alpha$ uniformly to all candidate words which are one edit operation away from the observed word. The same edit operations as in [Church and Gale, 1991] are considered. The test set consists of 100 correctly spelled sentences and all sentences one can generate by replacing a single word in one of the correctly spelled sentences by one of the candidate words. For good choices of α, between

73.2 % and 79.0 % of the misspelled sentences are corrected, while between 1 % and 3 % of the correctly spelled sentences get changed.

2.3.3. Neural network approaches

Semi-character RNN [Sakaguchi et al., 2017] develop a recurrent neural network that corrects words with permuted internal letters. The approach is motivated by the *Cambridge effect*. The *Cambridge effect* is the finding that humans can easily read words with permuted letters, as long as the first and last letter of a word remain intact (*For epmxlae, try to raed tihs suceneqe.*). Words are represented as the concatenation of three vectors: the one-hot-encoded first letter, the internal letters as a bag of characters, and the one-hot-encoded last letter. A unidirectional LSTM gets the vector representations of the input words with permuted letters as input, and predicts a correctly spelled word from a vocabulary with 10,000 words for each input word. The model is tested on three noise types, all of which keep the first and last letter of a word intact: permutation of the internal characters (which does not affect the vector representation), deletion of a single internal character, and insertion of a single internal character. The model achieves 98.96 % word accuracy on words with permuted internal letters and 96.70 % accuracy on words with a single inserted letter. The word accuracy drops to 85.74 % on input words with deleted letters.

Nested RNN [Li et al., 2018] use a nested recurrent neural network model to correct typos. On the lowest layer, a unidirectional Gated Recurrent Unit (GRU) working on characters encodes each word into a vector. Those word vectors are fed into the next layer, which is a bidirectional GRU. The hidden states from the bidirectional GRU are used to predict one intended word for each input word. The model is trained on text with artificial noise based on phonetic similarity of words. For the evaluation, an annotator corrected the misspellings in the JFLEG test set [Tetreault et al., 2017] to create ground truth labels. The nested RNN model achieves an $F_{0.5}$-score of 69.39 %, which outperforms a character convolutional neural network (64 %) and the PyEnchant spell checker (54 %).

CCEAD [Ghosh and Kristensson, 2017] develop a method they call Correction and Completion Encoder Decoder Attention Network (CCEAD). The encoder is a combination of a Gated Recurrent Unit (GRU) and a convolutional Neural Network that both work on character-level. The decoder is a word-level GRU with attention to the encoder hidden states. Training and evaluation is done on sequences from OpenSubtitles, where typos extracted from Twitter were induced. They get 98.1 % word accuracy measured on all words including the ones that were not affected by noise, and 68.9 % sequence accuracy. However, our analysis of the Twitter typos revealed that only 2,466 unique words can be affected by noise, and 9,294 unique misspellings exist. Only 939 of the misspellings are ambiguous, while the other 8,355 translate to a single intended word. Since the model is trained and evaluated on the same typos (with different context), we can not tell whether it generalizes to new typos.

Spelling as a Foreign Language In [Zhou et al., 2017], the spelling correction task is modeled as a translation task. The task is to translate from English with misspellings to correct English. An encoder-decoder neural network with attention achieves 62.5 % sequence accuracy on misspelled e-commerce queries. The approach is compared to a statistical machine translation model, which achieves a similar result.

2.3.4. Limitations and the lack of a common benchmark

Most of the presented papers limit their problem to a fixed set of possible typos or target words. [Church and Gale, 1991] and [Mays et al., 1991] explicitly limit their algorithms to misspellings comprising a single character edit operation. [Sakaguchi et al., 2017] consider words with jumbled internal characters, but assume that the first and last character of a word are always correct. The neural network approaches restrict their output to a pre-defined set of target words, which contains between 3,000 and 20,000 words [Sakaguchi et al., 2017, Li et al., 2018, Ghosh and Kristensson, 2017]. Almost all approaches [Church and Gale, 1991, Mays et al., 1991, Sakaguchi et al., 2017, Li et al., 2018, Ghosh and Kristensson, 2017] predict one target word for each input word and are thereby not able to correct tokenization errors, where words can be split into multiple parts or multiple words can be merged. The least restricted approach is the one by [Zhou et al., 2017], but it is only evaluated in the e-commerce domain, which usually has a limited vocabulary and short queries.

There exists no common test benchmark, which makes comparisons between publications difficult. Some evaluate on unpublished datasets [Church and Gale, 1991], while others create synthetic test sets [Mays et al., 1991, Sakaguchi et al., 2017, Ghosh and Kristensson, 2017]. [Li et al., 2018] evaluate on typos from the public JFLEG grammatical error correction dataset, but annotated the gold standard manually and did not publish their ground truth.

Not only the benchmark datasets, but also the evaluation metrics differ between publications. [Church and Gale, 1991] and [Sakaguchi et al., 2017] report word accuracy measured on misspelled words. [Mays et al., 1991] and [Zhou et al., 2017] report sequence accuracy. [Li et al., 2018] report $F_{0.5}$-score. [Ghosh and Kristensson, 2017] report word accuracy on all words and sequence accuracy.

2.4. Contribution

This thesis contributes the following:

- We study the usage of a sophisticated neural language model as our main approach for the spelling correction task.
- We compare our main approach to a neural machine translation approach, a simple baseline spelling corrector based on word frequencies, a more sophisticated baseline corrector using a traditional n-gram language model, and the commercial product of a large internet company.
- All approaches except for the translation-based approach predict words from a vocabulary with 100,000 words, compared to 3,000 to 20,000 words in the previous neural-network-based approaches from the literature. The translation-based approach is not limited to words from a pre-defined vocabulary.
- Our methods can correct misspellings comprising up to two character edits. They are not restricted to character edits, but can also correct split words, merged words and mixtures of error types.
- We prepare two benchmarks, that contain artificial and real misspellings

with context and ground truth corrections.

- We propose an evaluation metric, which assesses precision and recall rates.
- The performance of our main approach improves upon the baselines and the commercial product on both benchmarks.

3. Background

This section gives background about text representations, neural networks, language models, sequence sampling and a distance measure for text. These concepts will be used in the following sections, so this section is a recommended read for readers who are not familiar with the mentioned topics.

3.1. Tokenization

Tokenization is the task of dividing text into smaller units called tokens. Depending on the application, we use two different tokenization procedures: one is space tokenization and the other is tokenization with a regular expression.

3.1.1. Space tokenization

In the evaluation part of this work, tokens are defined as the substrings that are separated by spaces. Space tokenization simply splits a text at all space positions.

For example, the space-split tokens of the sequence *“10$ is too much.”* are *“10$”*, *“is”*, *“too”* and *“much.”*.

3.1.2. Tokenization with a regular expression

Sometimes it is useful to separate words and punctuation marks. In a large text corpus, many words will appear with different punctuation marks attached to it. For example the space-split tokens *“world.”*, *“world?”* and *“world!”* could all appear in a text. Yet, we do not want them to be treated as three completely different tokens. Instead, we want the tokenizer to recognize that in all three cases the token *“word”* appears and is followed by different punctuation marks.

For that purpose, we tokenize text using a regular expression and the python *re* module.

The regular expression used is the following:

```
\d+[.,]\d.,]*\d|\w[\w’-]+\w|\w+|\S
```

It comprises four patterns divided by the “or” symbol |. The patterns are

1. \d+[.,]\d.,]*\d: Matches numbers separated by points or commas. For example *123* and *1,000,000.01*.
2. \w[\w'-]+\w: Matches hyphenated words and words with internal apostrophes. For example *word-level*, *non-hyphenated* and *O'Connor's*.
3. \w+: Matches non-hyphenated words. For example *word* and *München*.
4. \S: Matches all other symbols except spaces. For example punctuation marks and *$*.

For the example sequence *"10$ is too much."* the resulting tokens are *"10"*, *"$"* *"is"*, *"too"*, *"much"* and *"."*.

3.2. Byte pair encoding

Traditionally, Natural Language Processing systems process text either on the byte level, character level or word level. Using the byte or character representation, single elements contain little semantic information and sequence representations can get very long. Using the word representation, sequence representations are shorter and each element contains a lot of semantic information. However, the word representation has the disadvantage that one has to predefine a fixed vocabulary and all words that are not in the vocabulary are represented by the same *out-of-vocabulary (OOV)* symbol.

For the usage in neural networks, the input elements have to be encoded as vectors. Common encodings are the one-hot encoding or trained embeddings. For bigger vocabularies the one-hot-encoded word representations are bigger vectors, too. This leads to more parameters in the input and output layers of a neural network. The embedding size, on the other hand, can be fixed, but an embedding has to be trained for each word in the vocabulary, so that more words also lead to more parameters.

The byte pair encoding (BPE, [Sennrich et al., 2016]) interpolates between byte level and word level. BPE splits text into subwords, which are variable-length byte sequences. It is always able to represent all sequences entirely without the necessity of an *OOV* symbol.

The byte pair encoding is created in an iterative procedure using a training text corpus. Initially, the text in the training corpus is treated as a byte sequence. The initial subwords are the 256 bytes. The frequency of all consecutive byte pairs is estimated on the training corpus. The most frequent byte pair gets merged into a new subword with index 257 and all occurrences of the byte pair in the training corpus replaced by the new subword. Merges across token boundaries are prohibited, so that no subword can be longer than a token. This procedure is iterated for K merging steps. The resulting subword set has $K + 256$ elements. Short, frequent tokens get merged into a single subword quickly, while longer, infrequent tokens are represented by a sequence of multiple subwords.

We use the code and byte pair merges published along with [Radford et al., 2019], but limit the encoding to the first K merges, where K is a hyperparameter.

Example encodings Table 1 shows example subwords for three different byte pair encodings. With zero merges, the subwords equal the bytes of the sequence, here represented as characters. Short, frequent words like *"It"*, *"was"* and *"day"* are merged into a single subword already after 2,000 merges. After 10,000 merges, many frequent words are represented as single subwords.

merges	subwords
0	"I", "t", " ", "w", "a", "s", " ", "a", " ", "b", "r", "i", "g", "h", "t", " ", "c", "o", "l", "d", " ", "d", "a", "y", " ", "i", "n", " ", "A", "p", "r", "i", "l", ",", " ", "a", "n", "d", " ", "t", "h", "e", " ", "c", "l", "o", "c", "k", "s", " ", "w", "e", "r", "e", " ", "s", "t", "r", "i", "k", "i", "n", "g", " ", "t", "h", "i", "r", "t", "e", "e", "n", "."
2,000	"It", " was", " a", " br", "ight", " c", "old", " day", " in", " A", "pr", "il", ",", " and", " the", " cl", "oc", "ks", " were", " st", "ri", "k", "ing", " th", "ir", "te", "en", "."
10,000	"It", " was", " a", " bright", " cold", " day", " in", " April", ",", " and", " the", " cl", "ocks", " were", " striking", " th", "ir", "teen", "."

Table 1: Subwords of the sentence *"It was a bright cold day in April, and the clocks were striking thirteen."* for byte pair encodings with 0, 2,000 and 10,000 merge steps.

3.3. Neural networks

A neural network (see [Goodfellow et al., 2016] for an overview) models a function $\hat{y} = f(x)$ for inputs x. The function depends on parameters θ and can be written $\hat{y} = f(x, \theta)$. The parameters θ are fit with supervised learning.

3.3.1. Fully-connected neural networks

A fully-connected neural network consists of layers f_1 to f_n which are stacked on top of each other. Layers f_1 to f_{n-1} are called hidden layers, whereas the last layer f_n is called output layer.

The input x to a fully-connected neural network is a vector of size d, containing the values of d features. d is the input dimension of the fully-connected neural network. The first layer f_1 computes an output $a_1 = f_1(x)$ from the input x. Each following layer f_i computes an output $a_i = f_i(a_{i-1})$ from its preceding layer's output a_{i-1}.

Each layer f_i contains u_i neurons, also called units. Its output a_i is a vector of size u_i, where the ith entry is the activation value of the ith unit. The output a_n of the output layer f_n is the output of the fully-connected neural network. a_n has size u_n, which is the output dimension of the network.

$$\hat{y} = a_n$$

The function f_i is defined as follows, with W_i being a weight matrix of shape (u_i, u_{i-1}), where $u_0 = d$, b_i being a bias vector of size u_i, and ϕ a nonlinear function:

$$f_i(a_{i-1}) = \phi(W_i a_{i-1} + b_i)$$

The weight matrices W_i and biases b_i form the set of trainable parameters θ of the fully-connected neural network:

$$\theta = \{W_i, b_i \forall 1 \leq i \leq n\}$$

3.3.2. Activation function

Each layer of a neural network comprises a nonlinear function ϕ, which is called the activation function. Among others, a common choice for the activation function is the rectified linear unit (ReLU):

$$\mathrm{ReLU}(a) = \max(0, a)$$

3.3.3. Output function

For classification tasks, the output layer f_n of a neural network must return a probability distribution over all classes. For k classes, the desired output is a probability vector of size k, where the i^{th} entry is the estimated probability of class i given the input x. To be a valid probability distribution, the entries of the vector must sum up to one. The softmax output function transforms the internal activation $a = W_n x_{n-1} + b_n$ of the output layer into a valid probability distribution:

$$\mathrm{softmax}(a)_i = \frac{e^{a_i}}{\sum_{j=1}^{k} e^{a_j}}$$

3.3.4. Loss function

The loss function measures how close the predictions of a model f are to the true values. Let $X = \{x_1, ..., x_N\}$ be a set of inputs, $Y = \{y_1, ..., y_N\}$ the corresponding one-hot-encoded targets, $\hat{Y} = \{\hat{y}_1, ..., \hat{y}_N\}$ the predicted outputs, where $\hat{y}_i = f(x_i)$, and k classes. The crossentroy loss is defined as:

$$\mathrm{crossentropy}(Y, \hat{Y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} y_{i,j} \mathrm{log}(\hat{y}_{i,j})$$

3.3.5. Supervised training

The parameters ϕ of a neural network are trained with stochastic optimization techniques. In the beginning, all parameters are initialized randomly. Then, in each training step i a subset (X_i, Y_i) of the training examples (X, Y) is selected, such that $X_i \subset X$ and $Y_i \subset Y$. This subset is called training batch. In a forward pass of the neural network, the predicted outputs $\hat{Y}_i$ are computed. Then, the loss is computed as $\text{loss} = \text{crossentropy}(Y_i, \hat{Y}_i)$. Beginning with the parameters of the output layer, the gradients of the loss with respect to each layer's parameters is computed using the chain rule in a backward pass through the network. This procedure is called backpropagation. An optimizer uses the gradients to update the model's parameters such that the loss on the training batch reduces.

3.3.6. Optimizer

The Adam optimizer ([Kingma and Ba, 2015]) maintains an estimate of the first and second moment of each parameter's gradient, and updates the parameters with the following rule:

$$\phi_i \leftarrow \phi_i - \frac{\alpha \cdot m}{\sqrt{v} + \epsilon}$$

where ϕ_i is the i^{th} parameter, α the learning rate, m the first moment, v the second moment and $\epsilon = 10^{-8}$.

3.3.7. Learning rate decay

Learning rate decay is reducing the learning rate over time, so that in early training steps the learning rate is big, allowing the model to change rapidly, and in later steps, when the model is fine-tuned, the learning rate becomes smaller.

In addition to the initial learning rate α, exponential decay has two parameters: the decay rate γ and the number of decay steps k. At training step t, the learning rate is:

$$\text{learning\_rate}(t) = \alpha \cdot \gamma^{t/k}$$

3.3.8. Recurrent neural networks

Recurrent neural networks get sequences $x = (x^1, ..., x^T)$ as input. A recurrent neural network layer generates a hidden state vector h^t for each time step $1 \leq t \leq T$ with the following recursive procedure:

$$a^t = \sigma(Wx^t + Uh^{t-1} + b)$$

where h^t is the hidden state vector of size d_h, with h_0 the zero vector, W is a weight matrix of shape (d_h, d), U is a weight matrix of shape (d_h, d_h) b is the bias vector of size d_h, and σ is a nonlinear function. The hidden states can be fed into a fully-connected network to estimate probability distributions for classification tasks.

Figure 1 demonstrates the computational graph of a recurrent cell.

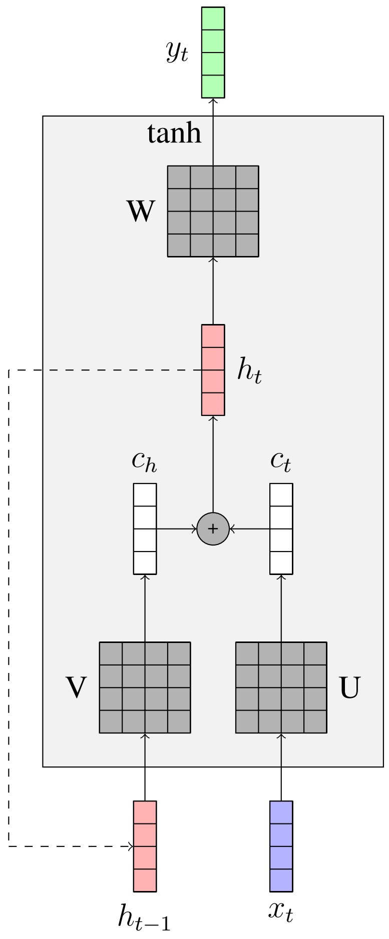


Figure 1: Computational graph of a recurrent cell. h_{t-1} is the previous hidden state, x_t the input vector, h_t the new hidden state and y_t the output vector. U, V and W the weight matrices. The $+$ denotes the element-wise sum of two vectors. Bias vectors are omitted in the figure for simplicity.

3.3.9. Long short-term memory

Recurrent neural networks as described above have difficulties capturing long-term relationships [Hochreiter et al., 2001] and suffer from the vanishing gradient problem [Hochreiter, 1991].

This motivates the long short-term memory (LSTM) cell introduced in [Hochreiter and Schmidhuber, 1997].

Here, the recursive procedure is as follows:

$$f^t = \sigma_g(W_f x^t + U_f h^{t-1} + b_f)$$

$$i^t = \sigma_g(W_i x^t + U_i h^{t-1} + b_i)$$

$$o^t = \sigma_g(W_o x^t + U_o h^{t-1} + b_o)$$

$$c^t = f^t \odot c^{t-1} + i^t \odot \sigma_h(W^c x^t + U^c h^{t-1} + b^c)$$

$$h^t = o^t \odot \sigma_h(c^t)$$

where f^t, i^t and o^t are the forget gate, input gate and output gate vectors of size d_h, W_f, W_i, W_o and W_c are weight matrices of shape (d_h, d), U_f, U_i, U_o and U_c are weight matrices of shape (d_h, d_h), b_f, b_i, b_o and b_c are bias vectors of size d_h, c^t is the internal state vector of size d_h, with c^0 the zero vector, h^t is the hidden state vector of size d_h, with h^0 the zero vector, σ_g is the sigmoid function, σ_h is the hyperbolic tangent function, and $\odot$ denotes the element-wise product.

The set of trainable parameters of a LSTM cell consists of the weight matrices and bias vectors:

$$\theta = \{W_f, W_i, W_o, W_c, U_f, U_i, U_o, U_c, b_f, b_i, b_o, b_c\}$$

Figure 2 demonstrates the computational graph of a LSTM cell.

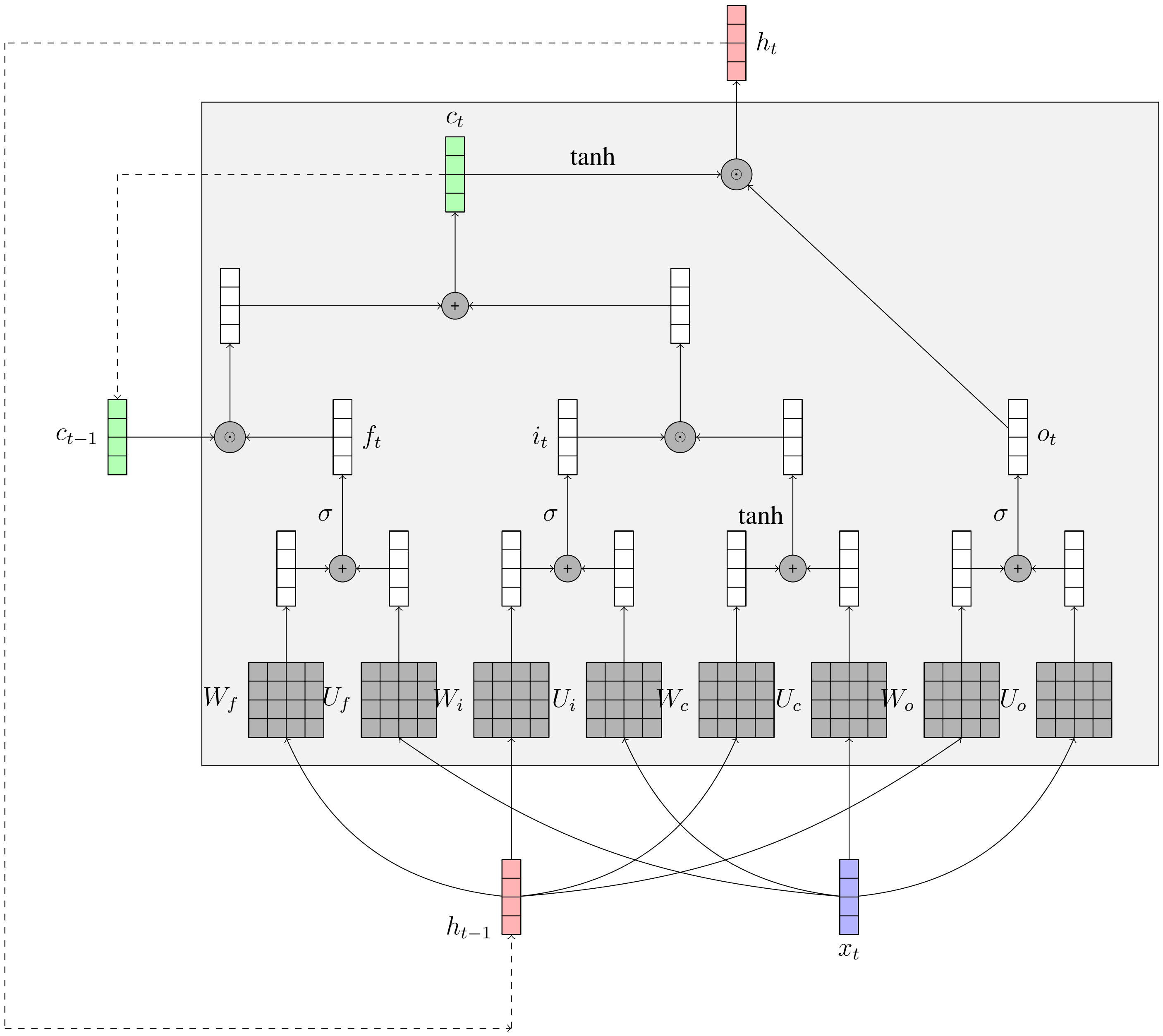


Figure 2: Computational graph of a LSTM cell. c_{t-i} and h_{t-1} are the previous hidden states, x_t is the input vector, c_t and h_t the new hidden state. h_t is passed to the next layer. All Ws and Us are weight matrices. f_t is the update vector, i_t the input gate and o_t the output gate. A $+$ denotes the sum of two vectors, and $\odot$ the element-wise product. Bias vectors are omitted in the picture for simplicity.

3.3.10. Backpropagation through time

The gradient computation for the stochastic optimization updates in recurrent neural networks or LSTMs works similar to the backpropagation for fully-connected networks. For T time steps, the network gets enrolled in a forward pass to generate the T hidden states, predicted outputs, and compute the loss. Then, the gradient of the loss with respect to each parameter is computed in a backward pass for all time steps. For a parameter θ_i, T gradients are computed, one for each time step, which get combined to the estimated gradient. Then, the optimizer uses the estimated gradient to update the parameters as in the fully-connected network.

3.3.11. Encoder-decoder networks

Sequence-to-sequence neural networks consist of two parts, the encoder and the decoder.

The encoder is a recurrent neural network that gets a sequence of vectors $\mathbf{x} = (x_1, ..., x_T)$ as input and generates hidden states h_1 to h_T with a recursive function:

$$h_t = f(x_t, h_{t-1})$$

In the vanilla encoder-decoder approach, the last hidden state of the encoder is passed to the decoder as context vector c:

$$c = h_T$$

The decoder is another recurrent neural network that estimates the probability distribution of the next word in the output sequence, given the context vector and the previous words:

$$p(y_t) = f(y_{t-1}, s_t, c)$$

where s_{t-1} is the hidden state of the recurrent neural network and the context vector is used as the initial hidden state:

$$s_0 = c$$

$$s_t = f(y_{t-1}, s_{t-1})$$

The encoder and decoder are trained jointly on bilingual pairs of sequences $x = (x_1, ..., x_T)$ and $y = (y_1, ..., y_N)$.

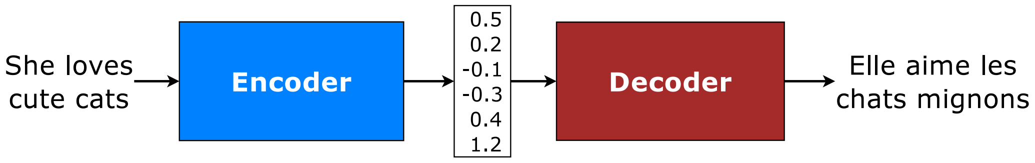


Figure 3: Encoder-decoder scheme for machine translation. The encoder transforms the input sequence into a fixed-size context vector, which is used by the decoder to generate the output sequence. Image from [Luong, 2016].

3.3.12. Attention

In the encoder-decoder approach described above, the neural network encodes the entire input sequence into a single vector h_T, which is then passed to the decoder. The attention mechanism introduced in [Bahdanau et al., 2015] is a way to relax this constraint by allowing the decoder to look at different parts of the input sequence at each step in the decoding process.

With attention, the conditional probability for the next word is defined as:

$$p(y_t) = f(y_{t-1}, s_t, c_t)$$

where s_t is the hidden state of the decoder network. Other than in the encoder-decoder approach without attention, the context vector c_t now differs for each word in the output sequence. It is a function of the decoder's current hidden state and all hidden states generated by the encoder:

$$c_t = f(s_t, (h_1, ..., h_T))$$

The context vector c_t is computed as a weighted average of the hidden states:

$$c_t = \sum_{i=1}^{T} \alpha_{it} h_i$$

where the weights α_{it} are given by an alignment model a:

$$\alpha_{it} = a(h_i, s_t)$$

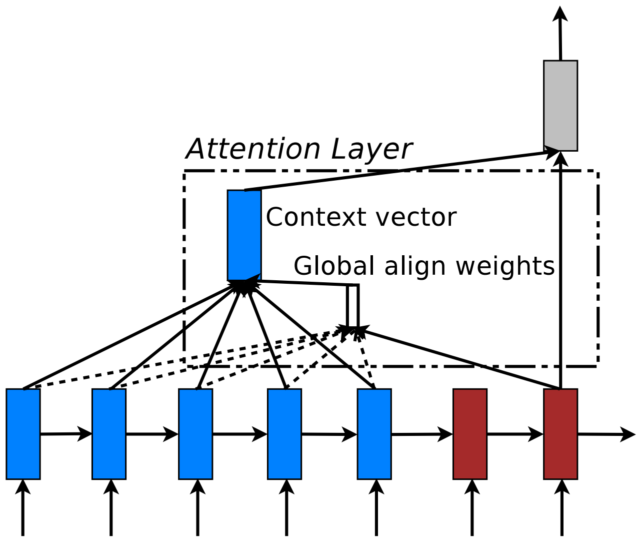


Figure 4: Attention layer. The encoder hidden states are blue, the decoder hidden states red. A context vector is computed as a weighted average of the encoder's hidden states. The gray vector is passed to the next layer of the decoder. Image from [Luong, 2016].

3.3.13. Self-attention

Unlike neural machine translation models that translate from an input sequence $X = (x_1, ..., x_T)$ to an output sequence $Y = (y_1, ..., y_{T'})$, neural language models predict the next element of a single sequence $S = (t_1, ...t_N)$.

The recurrent neural network predicts the next element given all previous elements. The version without attention mechanism is:

$$p(x_t|x_1, ...x_{t-1}) = f(x_{t-1}, h_{t-1})$$

and hidden states are generated recursively:

$$h_t = f(x_t, h_{t-1})$$

With attention, the model can look back at previously generated hidden states, and combine them to a context vector c_t:

$$p(x_t|x_1, ..., x_{t-1}) = f(x_{t-1}, h_{t-1}, c_t)$$

with

$$\begin{aligned} c_t &= f(h_t, (h_1, ..., h_{t-1})) \\ &= \sum_{i=1}^{T} \alpha_{it} h_i \end{aligned}$$

Given a sequence $X = (x_1, ..., x_T)$, the recurrent network generates hidden states h_1 to h_T. The hidden states are put into a matrix $H \in \mathbb{R}^{T \times d}$ an attention score matrix $S \in \mathbb{R}^{T \times T}$ is computed, where S_{ij} is the score for h_j when predicting x_i, and W a weight matrix:

$$S = HWH^\top$$

To prevent the network from attending to the current or future hidden states during training, the diagonal and all entries above the diagonal are masked with $-\infty$ before computing the softmax for each row, so that the masked scores become zero after softmaxing.

3.3.14. Transformer

[Vaswani et al., 2017] introduces the Transformer model, an encoder-decoder model that is solely based on the attention mechanism and does not include any recurrent operations. The dispensation from recurrent operations makes the Transformer faster than recurrent networks.

Figure 5 shows the Transformer's components. The left part of the figure is the encoder network, the right part is the decoder. N of the encoder and decoder cells, shown in the figure as gray boxes, are stacked on top of each other.

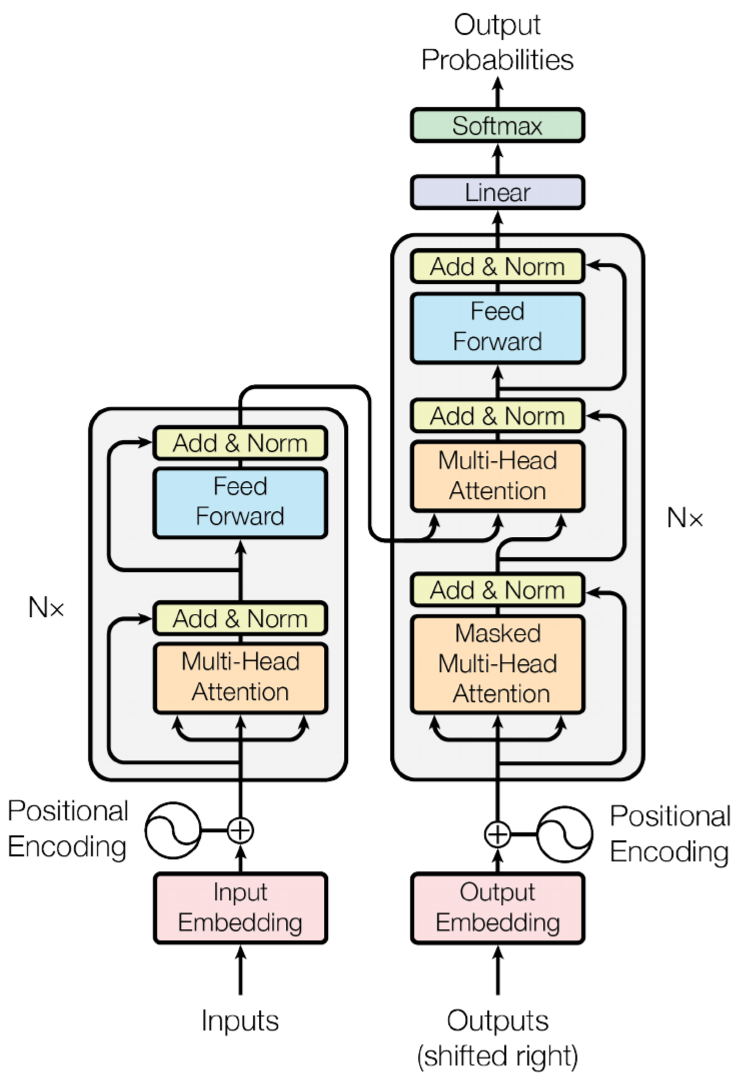


Figure 5: The Transformer architecture's components. N of the encoder and decoder layers, presented as gray boxes, are stacked on top of each other. Image from [Vaswani et al., 2017].

Inputs The input is a sequence $X = (x_1, ..., x_n)$. For language modeling and translation tasks, each value x_i is the index of a subword in a dictionary. The input sequence is split into tokens, which can be bytes, characters, subwords or words for language modeling and translation tasks. An array containing the tokens' labels in their sequential order is fed into the network.

Input embedding The embedding layer transforms the n input labels from x into a matrix $X \in \mathbb{R}^{n \times d_{\text{model}}}$. An embedding vector $e_i \in \mathbb{R}^{d_{\text{model}}}$ is learned for each input label i during training.

Positional encoding A positional signal is added to each of the rows in X to allow the Transformer to infer from the sequential order of the inputs.

The positional encoding is a mixture of sine and cosine functions with different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position in the sequence and i is the dimension in the encoding vector.

The positional encoding is a *relative encoding* since it holds that for any fixed offset k, BP_{pos+k} can be represented as a linear function of BP_{pos}. An advantage over learned positional encodings is that the sinusoidal encoding can be extrapolated to sequences of arbitrary length, which allows sequences during inference to be longer than the training sequences.

The positional encoding matrix for an input sequence has the same shape as the embedding matrix X. The two are added element-wise and the resulting matrix passed to the encoder.

Multi-Head Attention For queries Q, keys K and values V, scaled dot-product attention is:

$$\text{attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}}V)$$

Multiple heads allow the model to attend to multiple inputs at the same time step:

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

where

$$\text{head}_\text{i} = \text{attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Residual connections A residual connection is employed around each sublayer of the encoder and the decoder: $\mathrm{Residual}(\mathrm{Sublayer}, x) = x + \mathrm{Sublayer}(x)$

Layer Normalization Layer normalization [Ba et al., 2016] is applied to the output of each sublayer.

Position-wise feed-forward networks The feed-forward networks consist of two dense layers, of which the first has a ReLU activation function and the second a linear activation function. The first layer has d_{ff} units and the second layer has d_{model} units (usually $d_{ff} > d_{\mathrm{model}}$).

Outputs (shifted right) + embedding + positional encoding The output sequence $Y = (y_1, ..., y_{n'})$ is right-shifted, so that the first element in the sequence is a *<START>* symbol. The output elements are encoded and embedded in the same way as the inputs. A positional encoding is added to the embedding to encode the sequential order. The resulting matrix is given to the decoder, which predicts the next output at each position.

Decoder layers The decoder layers are similar to the encoder layers, but with two major differences. First, the self-attention scores get masked such that the decoder can not attend to future positions in the sequence. The corresponding attention scores are set to $-\infty$, so that after softmaxing they equal zero. Second, the decoder layers contain an additional multi-head attention layer that attends to the hidden states of the encoder layer. This cross-attention sublayer is located between the self-attention sublayer and the feed-forward sublayer of each decoder layer.

Output layer At each time step, the output layer of the decoder predicts a probability distribution for the next element in the sequence from the output of the last decoder layer. The output layer is a fully-connected layer with $|V|$ units using the softmax output function.

Transformer as a language model The Transformer decoder as a language model is introduced in [Liu et al., 2018]. It is constructed like the decoder of the encoder-decoder Transformer, but without the multi-head attention sublayer attending to the hidden states of an encoder.

3.4. Classification evaluation metrics

Accuracy is a metric for classification tasks with multiple classes. For binary classification tasks, however, the F-score based on precision and recall rates is often better suited.

3.4.1. Accuracy

Accuracy is defined as the number of cases where the model predicted the correct class, divided by the total number of cases.

$$\text{accuracy} = \frac{N_{\text{correct}}}{N}$$

3.4.2. Precision, recall, F-score

For binary classification tasks, it is often more interesting to look at precision and recall rates instead of accuracy. When the class labels are imbalanced, accuracy is a bad metric because a model that always predicts the more frequent class gets a high accuracy, although it is not a good model. Precision and recall rates better reflect a model's capability of detecting cases of the less frequent class.

The more frequent class is defined to be class zero and the less frequent class is class one. Let T be the set of cases where class one is the correct class, and P the set where the model predicts class one. The true positives TP are the cases where the model correctly predicts class one.

$$\text{TP} = T \cap P$$

The false positives FP are the cases where the model incorrectly predicts class one.

$$\text{TP} = P \setminus T$$

The false negatives FN are the cases where the model incorrectly predicts class zero.

$$\text{FN} = T \setminus P$$

Precision is the fraction of cases where the model is correct when it predicts class one.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall is the fraction of class one cases that were correctly predicted by the model.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The F_1-score combines precision and recall into a single metric. It is defined as the harmonic mean of precision and recall.

$$\text{F}_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

3.5. Language models

A language model estimates the likelihood of texts belonging to a language (see [Martin and Jurafsky, 2019] for an overview).

A sequence S is divided into N elements $S = (x_1, x_2, ..., x_N)$, and the language model models the probability of an element given the previous elements, $p(x_i|x_1, ..., x_{i-1})$. The elements can be bytes, characters, subwords or tokens.

Then, the sequence likelihood is the product of the elements' probabilities:

$$p(S) = \prod_{i=1}^{N} p(x_i|x_1, ..., x_{i-1})$$

3.5.1. N-gram models

An n-gram language model is based on the statistical frequency of groups of tokens. An n-gram is an ordered group of n tokens. The bigrams of the sentence

The cat eats fish. are *(The, cat)*, *(cat, eats)*, *(eats, fish)* and *(fish, .)*. Its trigrams are *(The, cat, eats)*, *(cat, eats, fish)* and *(eats, fish, .)*. The smallest n-grams with $n = 1$ are called unigrams. They are simply the tokens appearing in the sentence.

The conditional probability $p(w_i|w_1..w_{i-1})$ that a token w_i appears after tokens w_1 to w_{n-1} is estimated by Maximum Likelihood Estimation on a set of training sequences. Let $C(w_{i-n+1}..w_n)$ be the frequency of the n-gram ranging from w_{i-n+1} to w_n, and $C(w_{i-n+1}..w_{n-1})$ the frequency of the (n-1)-gram ranging from w_{i-n+1} to w_{n-1}. Then $p(w_i|w_1..w_{i-1})$ is estimated as:

$$p(w_i|w_1..w_{i-1}) = \frac{C(w_{i-n+1}..w_n)}{C(w_{i-n+1}..w_{n-1})}$$

The n-gram model can be combined with the models for the shorter unigrams to (n-1)-grams as given by the following recursive definition, where $0 \le \alpha \le 1$ is an interpolation factor:

$$p(w_i|w_{i-n+1}..w_{i-1}) =$$
$$\begin{cases} (1-\alpha)\cdot\dfrac{C(w_{i-n+1}..w_n)}{C(w_{i-n+1}..w_{n-1})} + \alpha\cdot p(w_i|w_{i-n+2}..w_{i-1}) & \text{if } C(w_{i-n+1}..w_{n-1}) > 0 \\ p(w_i|w_{i-n+2}..w_{i-1}) & \text{otherwise} \end{cases}$$

3.5.2. Neural language models

A neural language model uses a recurrent neural network or Transformer decoder to estimate the probability $p(x_i|x_1, ..., x_{i-1})$. The parameters of the model are trained on a text corpus.

Sequences are padded with a start-of-sequence symbol <*SOS*> and an end-of-sequence symbol <*EOS*>:

$$S = (\text{<}SOS\text{>}, x_1, ..., x_N, \text{<}EOS\text{>})$$

To be able to train on sequences with different lengths in the same batch, shorter sequences are padded to the right, or longer sequences cut after a certain length. The loss on the padding symbols is set to zero, so that it does not affect the training process.

When the neural language model works on the subword level, the probability $p(w|S)$ that the word w follows the sequence S is

$$p(w|S) = p(W|U) = \prod_{i=1}^{n} p(w_i|u_1, ..., u_N, w_1, ..., w_{i-1})$$

where $W = (w_1, ...w_n)$ are the subwords of w and $U = (u_1, ..., u_N)$ the subwords of S.

3.5.3. Perplexity

Perplexity is a measure of how well a language model predicts a test corpus W. It is the inverse of the probability that the model assigns to the corpus, normalized by the number of words in the corpus:

$$\begin{aligned} PP(W) &= p_{\text{LM}}(w_1, ..., w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{p_{\text{LM}}(w_1, ..., w_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p_{\text{LM}}(w_i|w_1, ..., w_{i-1})}} \end{aligned}$$

The lower the perplexity, the better does the model predict the training corpus. In order not to compute the product of many small probabilities, which will be rounded to zero in many programming languages, one can rewrite the formula as:

$$PP(W) = \exp(-\frac{1}{N}\sum_{i=1}^{N} log(p_{\text{LM}}(w_i|w_1, ..., w_{i-1})))$$

3.6. Generative sampling of sequences

A language model gives a probability distribution over the characters, subwords or words following a prefix. Three algorithms are used to generate sequences from those distributions.

3.6.1. Greedy decoding

Greedy decoding starts with an empty sequence and iteratively appends the most likely element to it, until it reaches the <*EOS*> symbol. The procedure is given in Algorithm 1.

Algorithm 1 Greedy Decoding

```
procedure GREEDYDECODING(model)        ▷ model is a language model.
    sequence := ""
    while ¬ sequence.ends_with(<EOS>) do
        sequence.append(model.get_most_likely_appendix(sequence))
    return sequence
```

3.6.2. Beam search

Beam search maintains a set of partial sequences. Those sequences are called beams. The size k of the beam set is a hyperparameter. In each iteration, the k most likely extensions to each beam are determined by the language model, and new candidate beams created by appending them to the beam. The k most likely beams are kept for the next iteration. Finally, the most likely beam ending with a <*EOS*> symbol is returned. Algorithm 2 gives the procedure in pseudocode.

Algorithm 2 Beam Search

```
procedure BEAMSEARCH(model, k)    ▷ model is a Language Model, $k$ is
                                    the number of beams.
    beams := {""}                             ▷ A single empty beam.
    terminated := {}             ▷ Terminated beams will be stored here.
    while |terminated| < k do
        new_beams := {}
        for beam ∈ beams do
            if beam.last_element() = <EOS> then
                terminated.add(beam)
            else
                for extension ∈ model.most_likely_extensions(beam, k) do
                    new_beams.add(beam + extension)
        beams := model.select_most_likely(new_beams, k)
    best_beam := model.select_most_likely(beams, 1)
    return best_beam
```

3.6.3. Best-first search

Greedy search and beam search might not find the most likely sequence due to the greedy decisions made in every iteration. Best-first search is a method to compute the most likely sequence, or the k most likely sequences. It is a breadth-first search guided by the probabilities of the partial sequences generated. The procedure is given in Algorithm 3.

Algorithm 3 Best First Search

```
procedure BESTFIRSTSEARCH(model, k)    ▷ model is a Language Model, k is the number of sequences to generate.
    V = model.vocabulary()             ▷ Characters, subwords or words.
    q := Queue()                       ▷ A minimum heap.
    q.insert(0, "")                    ▷ Empty sequence with score 0.
    terminated := {}                   ▷ Teminated sequences will be stored here.
    while $|$terminated$| < k$ do
        score, sequence := q.pop()
        if sequence.ends_with(<EOS>) then
            terminated.add(sequence)
        else
            for w $\in$ V do
                new_sequence = sequence.append(w)
                p = model.estimate(w|sequence)
                new_score = score - log(p)
                q.insert(new_score, new_sequence)
    return terminated
```

3.7. Edit distance

The edit distance between two strings a and b is the minimum number of basic edit operations needed to transform one into the other. The Damerau-Levenshtein edit distance [Boytsov, 2011] uses the following basic edit operations:

1. Insertion of a character.
2. Deletion of a character.
3. Replacement of a character by another character.
4. Transposition of two neighboring characters.

The restricted Damerau-Levenshtein edit distance is the Damerau-Levenshtein edit distance under the assumption that no substring is edited more than once. It is a lower bound of the unrestricted Damerau-Levenshtein edit distance. The restricted and unrestricted Damerau-Levenshtein edit distance differ in cases where the optimal edit operation sequence contains a replacement and insertion at the same position. Consider as an example the problem of transforming string *ab* into *bca*. In this case, the unrestricted Damerau-Levenshtein edit distance equals two, because a transposition of the two characters *a* and *b* followed by an insertion of a *c* between them transforms *ab* into *abc* with two basic edit operations. However, the restricted Damerau-Levenshtein edit distance equals three, because the two edit operations would edit the same substring *ab* and are therefore not allowed to be applied together.

Algorithm 4 computes the restricted Damerau-Levenshtein edit distance in time and space complexity in the order of the product of the lengths of the two given strings. It is a dynamic programming algorithm that maintains a matrix d, where entry $d_{i,j}$ is the restricted Damerau-Levenshtein edit distance between the prefix of a with length i and the prefix of b with length j.

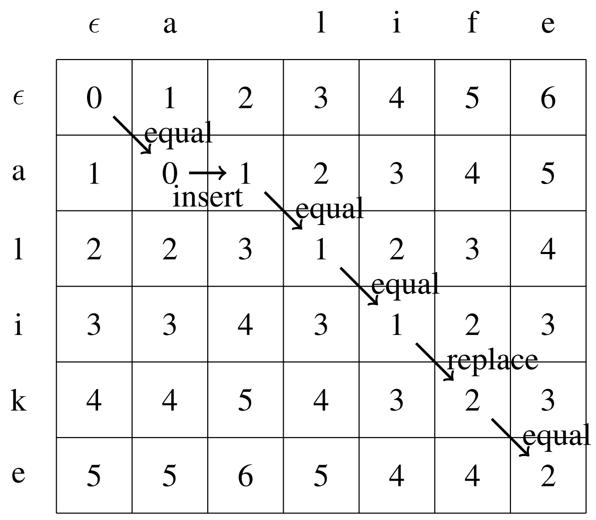
The character edit operations that transform a into b can be retrieved from the backtrace of matrix d. With each entry $d_{i,j}$ we store its predecessor and operation that led to the entry. Starting with the lower right corner of d, all predecessors and operations are retrieved, until the entry $d_{0,0}$ is reached.

To bias the algorithm towards leaving tokens intact, we prohibit replacements and transpositions that affect spaces. As a result, spaces can only be inserted or deleted.

Figure 6 shows an example edit distance matrix and edit operations retrieved from the backtrace of the matrix.

Algorithm 4 Restricted Damerau-Levenshtein Edit Distance

```
procedure EDITDISTANCE(a, b)                         ▷ a and b are one-indexed strings
    d := matrix[0 .. length(a), 0 .. length(b)]      ▷ $d \in \mathbb{Z}^{\text{length(a)}\times\text{length(b)}}$,
                                                       entry $d[i, j]$ is the edit distance between the prefix
                                                       of $a$ with length $i$ and the prefix of $b$ with length $j$
    for i := 0 .. length(a) do
        d[i, 0] := i
    for j := 1 .. length(b) do
        d[0, j] = j
    for i := 1 .. length(a) do
        for j := 1 .. length(b) do
            if a[i] = b[j] then
                cost := 0
            else
                cost := 1
            d[i, j] := d[i-1, j] + 1                              ▷ deletion
            d[i, j] := minimum(d[i, j], d[i, j-1] + 1)            ▷ insertion
            d[i, j] := minimum(d[i, j], d[i-1, j-1] + cost)       ▷ replacement
            if i $>$ 1 $\wedge$ j $>$ 1 $\wedge$ a[i] = b[j-1] $\wedge$ a[i-1] = b[j] then
                d[i, j] := minimum(d[i, j], d[i-2, j-2] + cost)   ▷ transposition
    return d[length(a), length(b)]
```



	ϵ	a		l	i	f	e
ϵ	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
l	2	2	3	1	2	3	4
i	3	3	4	3	1	2	3
k	4	4	5	4	3	2	3
e	5	5	6	5	4	4	2

Figure 6: Edit distance matrix and operations that transform the string *"alike"* into *"a life"*. Spaces can only be inserted or deleted.

4. Datasets

4.1. Wikipedia

We use Wikipedia as a corpus to train and evaluate the methods of this work.

Wikipedia has advantages over other text corpora that are frequently used in language modeling. First, it is bigger than the Reuters and Europarl corpora, so that models are unlikely to overfit to text seen in the training data. Second, the texts come from a variety of topics, so that the resulting models will work on multiple domains. Third, one can assume that most of the text is spelled correctly, due to the active community continuously correcting errors. This is advantageous over web-scraped corpora like ClueWeb or WebText (the latter not being publicly available).

4.1.1. Text extraction

We downloaded the English Wikipedia dump from June 20, 2019.[2] It contains all articles with their metadata in XML format. The raw text of the articles was extracted using the WikiExtractor script by Giuseppe Attardi.[3]

4.1.2. Preprocessing

The articles were split into a development and a test set, containing 10,000 articles each, and the remaining training set. Articles were further split into paragraphs by splitting at line breaks. Leading and trailing whitespaces were deleted from the paragraphs, and empty paragraphs removed. Special whitespace characters like the non-breaking space where replaced with the normal space.

4.1.3. Statistics

Table 2 gives the number of articles and paragraphs in the Wikipedia training, development and test sets.

[2]File enwiki-20190620-pages-articles-multistream.xml.bz2 from `https://dumps.wikimedia.org/enwiki/20190620/`.

[3]Downloadable at `https://github.com/attardi/wikiextractor/wiki`.

	training set	development set	test set
articles	5,860,206	10,000	10,000
paragraphs	43,103,197	74,112	73,600

Table 2: Statistics of the Wikipedia training, development and test sets.

4.2. Typo collection

As a source of typos we use a collection by Peter Norvig.[4] It contains lists of misspellings for 7,841 correctly spelled words. The typos are collected from Wikipedia[5] and by Roger Mitton[6] We removed 19 pairs from the collection, where the misspelling and the intended text were equal.

[4]`http://norvig.com/ngrams/spell-errors.txt`, accessed July 23, 2019
[5]`http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines`
[6]`https://www.dcs.bbk.ac.uk/~ROGER/corpora.html`

5. Approaches

5.1. NLMspell: spelling correction with a neural language model

5.1.1. Procedure

Overview The Neural Language Model Spelling Corrector (NLMspell) generates candidate corrections for each token in the input sequence, estimates the likelihood of the candidate corrections using a neural language model, and finds the approximately most likely intended sequence with beam search.

NLMspell can correct misspellings that comprise up to two of the following edit operations, in any combination:

- Character edit: the insertion or deletion of a character, replacement of a character by another character or transposition of two neighboring characters.
- Token merge: the merge of two tokens by removing the space between them.
- Token split: the split of a token into two tokens by inserting a space.

Candidate corrections First, the input sequence S is split into tokens $(t_1, ..., t_n)$. Here, the tokens are simply the parts of S that are separated by spaces.

NLMspell processes the tokens from left to right. At position i, candidate corrections are generated based on tokens t_i and t_{i+1}. The 100,000 most frequent words from the training data form the vocabulary V that is used to generate candidate corrections.

Before generating candidate corrections for a token, punctuation marks are split from the token using the RegexTokenizer. After generating candidate corrections for the punctuation-free token, the punctuation marks are re-attached to the candidate corrections. For example, for the input sequence *"Hello wurld!"*, the tokens are *"Hello"* and *"wurld!"*. For the second token, candidates are generated for the punctuation-free token *"wurld"*. Assume *"world"* is one of the generated candidates. The exclamation mark is re-attached to the candidate to form the correction candidate *"world!"*.

The candidate set C_i for token t_i contains the following candidates:

- **Observed candidate:** The observed token t_i.
- **Edit candidates:** All words from V which have an edit distance to t_i smaller or equal to two.
- **Split candidates:** All splits of t_i into two tokens t_{left} and t_{right}, such that $t_{\text{left}} \in V$ and $t_{\text{right}} \in V$.
- **Merge candidate:** The token t_i merged with the next token t_{i+1}, if the resulting token t' is in V.
- **Split-edit candidates:** For all splits of t_i into two tokens t_{left} and t_{right}, such that $t_{\text{left}} \in V$, candidates $(t_{\text{left}}, c_{\text{right}})$ are generated for words $c_{\text{right}} \in V$ such that the edit distance between t_{right} and c_{right} is one. Analog, candidates $(c_{\text{left}}, t_{\text{right}})$ are generated if $t_{\text{right}} \in V$ such that the edit distance between c_{left} and t_{left} is one.
- **Merge-edit candidates:** All words from V that have an edit distance of one to the merge of t_i with t_{i+1}.
- **Double-split candidates:** All splits of t_i into three tokens such that all three tokens are in V.
- **Double-merge candidates:** t_i merged with the next two tokens, if the merged token is in V.
- **Merge-split candidates:** All splits of the merge of t_i with t_{i+1} into two tokens, such that the two tokens are in V.

Probability estimation Given a prefix $\hat{S}_i$, for each candidate correction c_j from C_i the probability $p(c_j|\hat{S}_i)$ that c_j follows $\hat{S}_i$ is estimated with the neural language model.

To do so, the prefix $\hat{S}_i$ is encoded into a sequence of subwords $U = (u_1, ..., u_{|U|})$ and the candidate correction c_j is encoded into a sequence of subwords $W = (w_1, ..., w_{|W|})$. The neural language model is used to estimate the probabilities for the subwords in W following the subword sequence U. The probability of candidate c_j is the product of the probabilities of its subwords W:

$$p(c_j|\hat{S}_i) = p(W|U) = \prod_{i=1}^{|W|} p(w_i|u_1, ..., u_{|U|}, w_1, ..., w_{i-1})$$

The subword encodings of some words are prefixes of subword encodings of other words. The probability of a prefix will always be greater than the probability of the longer word, because the product of the subword probabilities can only decrease with more subwords. To get a better estimate of the probability $p(c_j|\hat{S}_i)$ we multiply the subword probability with the probability that c_j is a complete word. Complete words are followed by a space. We define the probability $p_{\text{complete}}(c_j, \hat{S}_i)$ that c_j is a complete word as the sum of the probabilities of all subwords that begin with a space:

$$p_{\text{complete}}(c_j, \hat{S}_i) = \sum_{s \in \text{SPACE}} p(s|\hat{S}_i, c_j)$$

where SPACE is the set of subwords beginning with a space. We then redefine the candidate probability:

$$p(c_j|\hat{S}_i) = p(W|U) \cdot p_{\text{complete}}(c_j, \hat{S}_i)$$

Beam search The neural language model uses only the left context to estimate the probability of a candidate correction. However, the right context gives additional information that can help to determine the most likely correction. To make use of the right context, NLMspell performs a beam search. Thereby, instead of choosing the most likely correction greedily based only on the left context, it maintains a set of candidate sequences and postpones the decision about the most likely correction to a later step, where it will have seen some of the right context.

For example, given the input sequence *"The ct eats fish."*, at the second step (processing *"ct"*), the language model gives the candidate sequence *"The act"* a higher probability than *"The cat"*. However, both sequences are kept as candidate sequences, and at the third step (now processing *"eats"*) the probability of *"The cat eats"* becomes greater than the probability of *"The act eats"*. In this example, the right context word *"eats"* helped to find the correct intended word for *"ct"*.

The beam search proceeds as follows: Let B_i be the set of candidate sequences maintained by NLMspell in iteration i. Each candidate sequence $b_l \in B_i$ is assigned a score, that reflects how likely the candidate sequence is, given the observed tokens t_1 to t_{i-1}. Initially, B_1 contains just the empty sequence.

The tokens are processed from t_1 to t_n iteratively. In iteration i, candidate correction set C_i for token t_i is generated. Each candidate correction $c_j \in C_i$ is appended to each candidate sequence $b_l \in B_i$. That gives a candidate sequence set B_i^{new} that contains $|B_i| \cdot |C_i|$ new sequences. For each new candidate sequence $b_{l,j}^{\text{new}}$, that was created by appending candidate correction c_j to candidate sequence b_l, a score is computed based on the score of b_l, the probability $p(c_j|b_l)$ estimated by the neural language model, and the similarity between c_j and observed token t_i. The k new candidate sequences from B_i^{new} with the best scores are kept as the set B_{i+1} for the next iteration. The size k of the candidate sequence set is a hyperparameter of NLMspell.

The merge, merge-split, merge-edit and double-merge candidates are candidates not only for token t_i, but for tokens t_i to t_{i+1} or t_{i+2} combined. The candidate sequences that are created with such a candidate are not added to B_{i+1}, but to B_{i+2} or (if it is a double-merge candidate) to B_{i+3}.

After n iterations, NLMspell terminates with a set of k candidate sequences B_{n+1}. The candidate sequence with the best score gets returned as the predicted sequence $\hat{S}$ for input sequence S.

Candidate scoring We score the candidates based on two assumptions:

1. Candidates that get a high probability by the language model are more likely to be correct than candidates with a low probability.
2. Intended sequences are orthographically similar to the observed sequences.

As the score for a candidate, we use its negative log likelihood estimated by a language model and punish it depending on the edit distance to the observed token. The smaller the score, the better.

The punishment is necessary to bias the beam search towards candidate sequences that are similar to the input sequence. If we were to score the candidates without any punishment, the beam search would result in a sequence that is likely spelled correctly, but not necessarily the intended sequence for the input sequence. For example, the candidate *"The"* would always get the same score, whether the observed sequence starts with *"The"*, *"Te"*, *"Then"*, *"This"*, *"That"*, *"Thus"*, *"He"* and many other tokens for which *"The"* is contained in the candidate set. We want the candidate *"The"* to get a high score when the observed token is *"The"*, a medium score when the observed token is *"Te"*, and a low score when the observed token is *"He"*. This is achieved by punishing the score of a candidate depending on the candidate's edit distance to the observed token.

As the score of a candidate sequence $b_l \in B_{i+1}$ we use the negative log likelihood of the sequence plus a punishment for the number of edit operations needed to transform the observed tokens t_1 to t_i into b_l. The smaller the score, the better.

A candidate sequence b is a sequence of candidate corrections $(c^1, ..., c^i)$. Each candidate correction c^j has a probability $p(c^j|c^1, ..., c^{j-1})$ estimated by the neural language model. The likelihood of the sequence b is the product of its candidate corrections' probabilities. It follows that the log likelihood is the sum of the logarithms of those probabilities. To compute the log likelihood of a new candidate sequence $b_{l,j}^{\text{new}}$, which is created by appending c_j to candidate sequence b_l, we add the logarithm of $p(c_j|b_l)$ to the log likelihood of b_l.

$$\log(p(b_{l,j}^{\text{new}})) = \log(p(b_l)) + \log(p(c_j|b_l))$$

The punishment for a candidate correction c_j is zero when c_j is equal to the observed token t_i. Otherwise we add a punishment $\lambda \cdot \text{edit\_distance}(c_j, t_i)$. As with the log likelihood, we sum up the punishments for the individual candidate corrections that form a candidate sequence to get the punishment for the entire sequence.

Putting the log likelihood and the punishments together, the score of a new candidate sequence $b_{l,j}^{\text{new}}$, that is created by appending candidate correction c_j to candidate sequence b_l, is computed from the score of b_l, the probability $p(c_j|b_l)$

and the punishment value:

$$\text{score}(p(b_{l,j}^{\text{new}})) = \text{score}(b_l) - \log(p(c_j|b_l)) + \lambda \cdot \text{edit\_distance}(c_j, t_i)$$

5.1.2. Runtime improvements

NLMspell has two potential runtime bottlenecks: one is the candidate generation, and the other is the estimation of the probability for many candidate corrections.

For the generation of the candidate corrections C_i given a token t_i, all words from V with an edit distance to t_i smaller or equal to two are to be determined. The naïve way to find these words is to compute the edit distance between t_i and every word in V, and keep the words with edit distance smaller or equal to two. However, with $|V| = 100,000$, this results in 100,000 edit distance computations per input token, which takes multiple seconds runtime.

After the candidate correction set C_i is computed, the probability for each candidate correction $c_j \in C_i$ as an appendix to every candidate sequence $b_l \in B_i$ has to be evaluated with the neural language model. For some input tokens, C_i contains more than 2,000 candidate corrections. With $k = 10$ candidate sequences in Beam Search, that results in more than 20,000 candidate evaluations in one iteration of the Beam Search algorithm. The naïve solution to this problem would be to generate all 20,000 candidate sequences and let the neural language model estimate their probabilities one by one. This can be done in finite time, but would take minutes to hours to evaluate.

Fast candidate generation An observation allows us to define a lower bound on the edit distance between t_i and the words in V, which we then use to pre-filter V rapidly. We then only compute the edit distance between t_i and a much smaller pre-filtered set $C_i' \subset V$ to determine C_i.

We observe that, when we define sets W_i for every word w_i in V, such that W_i contains all strings that can be generated from w_i by deleting up to two characters, words w_i and w_j within an edit distance smaller or equal to two have at least one common element in $W_i \cap W_j$. We call W_i the stump set of word w_i.

The intuition is as follows: a character insertion, deletion or replacement affects up to one character in a word. If w_i can be transformed into w_j with two or less edit operations, at maximum two characters in w_i can be not present in w_j, and the other way around. This holds also for the transposition of two characters: regard the words *"abcd"* and *"badc"* which have edit distance two. Removing the *a* and the *c* in both words results in the common stump *"bd"*.

We now know that the edit distance between token t_i and word $w_j \in V$ is greater than two if the stump set T_i of token t_i and the stump set W_j of w_j do not have any element in common. In a preprocessing phase, we compute the stump sets W_j for all words $w_j \in V$. For every stump, we compute the set of words in V that have this stump in their stump set. We call this the stump-word index. Now, given an observed token t_i, we compute the stump set T_i of t_i, and retrieve the words for every stump in T_i from the stump-word index. This set C'_i is much smaller than V, which saves us a lot of edit distance computations.

An example shows that the stump method gives only a lower bound on the edit distance, and that we therefore still have to compute the edit distance between t_i and every element in C'_i to determine the candidate correction set C_i. Consider the words *"aabb"* and *"bbcc"*. Both have the stump *"bb"* in common. However, the edit distance between the two words is 4 and therefore greater than 2.

Semi-fast probability estimation The estimation of all candidates corrections as appendices to all candidate sequences is sped up by re-using hidden states and batching candidates. However, a lot of evaluations remain, which is why we call the method the semi-fast probability estimation.

With every candidate sequence b_i we store the neural language model's hidden state for every subword. When appending candidate c_j to b_i, we start from the last hidden state and process the subwords from c_j iteratively. This saves us re-computing the hidden states for the subwords in b_i for every iteration and every candidate.

Additionally, we group multiple candidate corrections into batches and process them altogether. When candidate corrections do not have the same number of subwords, shorter candidate corrections are padded. Only the subword probabilities without padding are used to compute a candidate's probability, and the last hidden state not corresponding to a padding symbol is stored with the candidate.

Candidate pre-filtering To reduce the number of word probability estimations, we use the candidate corrections' frequency in the training data as a heuristic to pre-select a subset of promising candidates, and only evaluate their probability.

The edit candidates and the split-edit candidates are pre-filtered such that both sets contain only the K most frequent candidates. For the edit candidates, simply the word frequency in the training data is used to determine the most frequent candidates. For the split-edit candidates, which consist of two words, frequencies of the 10,000,000 most frequent bigrams are used to determine the most frequent candidates. If a split-edit candidate is not among the most frequent 10,000,000 bigrams, its frequency is set to 0. For those candidate corrections the frequency of its less frequent word is used as a tie-breaker.

5.1.3. Neural language model architecture and training hyperparameters

We use a recurrent neural network as the language model in NLMspell. We expect that a good language model is crucial for a good performance of NLMspell. In the design of the neural language model, many decisions have to be made, which affect the performance of the neural language model.

Hyperparameter search space When designing the neural language model, we have to make the following decisions:

- How many **byte pair merges** to use for the encoding?
- How many **LSTM layers** does the model incorporate?
- How many **hidden units** per LSTM layer?
- How many **dense layers** to stack on top of the LSTM layers?
- How many **hidden units** per dense layer?
- Does the model incorporate **self-attention**?
- What is the learning rate or **learning rate schedule**?
- What **batch size** to use during training?
- **How long** to train the model?

With our approach we have already fixed some design decisions:

- **Unidirectional model.** A bidirectional model would need to deal with noisy input, since misspellings can appear everywhere in the sequence. We found it convenient to use a unidirectional model and process the input sequence from left to right, such that the left context is always corrected by the model before predicting the next correction.
- **Recurrent model.** Recently, transformer models have shown great success in language modeling and translation tasks. However, they are parameter-heavy and training seems sensitive to hyperparameters. In our experiments, we found recurrent models to train better with the resources available. More details can be found in appendix A.1.
- **ReLU.** We always use ReLU as the activation function for dense layers because we believe the effect of changing the activation function to be small, and we do not want to increase our hyperparameter search space unnecessarily.
- **Adam optimizer**. We use the Adam optimizer because we hope for the per-parameter adaptive learning rates to stabilize training and found the training loss of our models to decrease rapidly, even when just using the default hyperparameters.

Figure 7 depicts the computational graph of a neural language model with attention.

Hyperparameter optimization issues We are to test 20 different network architectures with 6 different learning rates and 4 different decay rates A grid search on that search space would require to train and evaluate $20 \cdot 6 \cdot 4 = 480$ models. Since training a model can take more than a day, we are in the range of multiple years runtime required for grid search.

Alternatives to grid search are Hyperband [Jamieson and Talwalkar, 2016] and BOHB [Falkner et al., 2018], which both build on the Successive Halving algorithm [Karnin et al., 2013]. Successive Halving trains models for a short training time budget, evaluates them, and only trains the models further which perform well after the short training time. However, we find that our models' performance correlates negatively over training time budgets, which is why we do not use Hyperband and the approaches built upon it. In initial experiments we see

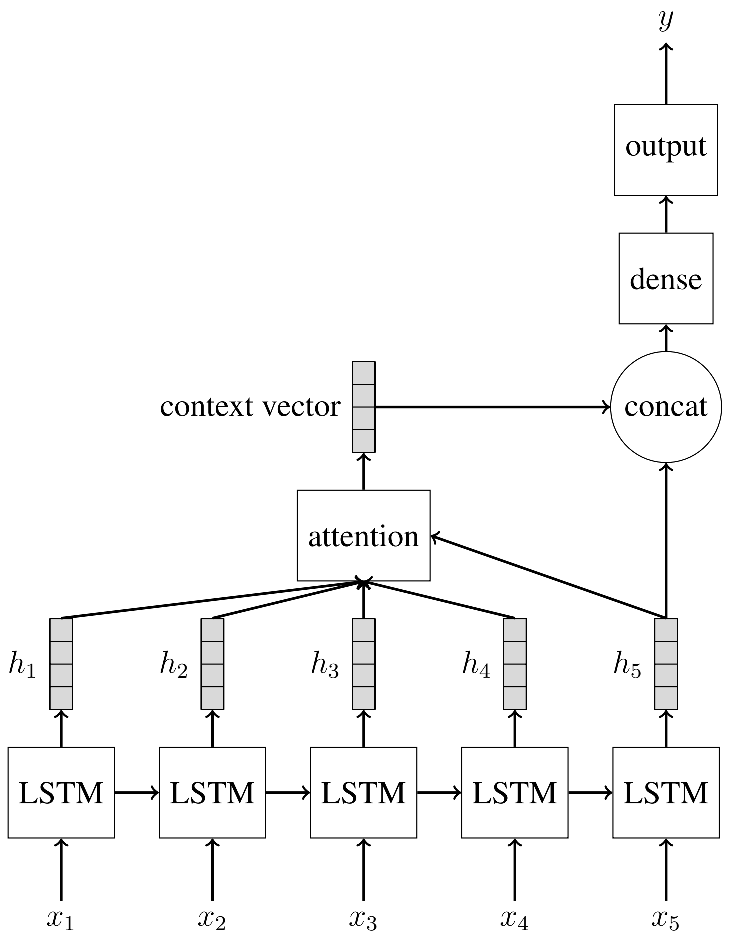


Figure 7: A recurrent neural language model with attention.

that small models train faster than big models, but big models have better performance when trained for a long time. We see training loss curves of different models cross after 18 hours training time and later. If we were to eliminate models after a short training time budget, we would only keep the models that train fast in the beginning, and eliminate the models that train more slowly but have better end performance.

Phased hyperparameter grid search Instead of using grid search or Successive Halving to optimize the hyperparameters of our neural language model, we use a procedure we call phased grid search. We assume that the choice of the network architecture affects the model's performance more than the learning rate used to train it, as long as the learning rate is in a reasonable range. In our experiments we see that with the default learning rate 0.001 the training loss decreases rapidly in early training steps. Therefore we decide to first train models with 20 different architectures and the default learning rate for one day each, in order to find the best architecture. Training the 20 models is distributed to 20 machines to save wall-clock time. The results of this experiment can be found in appendix A.2.

In the second phase, we test different initial learning rates for the best architecture. Models are trained for 3,000 steps. The results of this experiment can be found in appendix A.2.

In the third phase, we test different decay rates. Models are trained for 20,000 steps. The results of this experiment can be found in appendix A.2.

Overall, the entire hyperparameter search took less than two days wall-clock time.

We finally choose a model with 10,000 byte pair merge steps, a single LSTM layer with 1024 units, self-attention and a single dense layer with 1024 units, and train it for 24 hours on a Nvidia Titan X GPU with a batch size of 64, and a learning rate of 0.003 decaying every 1,000 training steps with a decay rate of 0.95.

5.1.4. Punishment value

We use the development benchmarks to find the optimal punishment value λ.

The development benchmarks contain pairs of noisy and intended sequences. First we compute the ground truth corrections for all misspellings in an intended sequence. To do so, we compute the character edit operations that transform the noisy into the intended sequence, and apply these operations token-wise. If a token from the noisy sequence does not get changed while transforming the noisy into the intended sequence, it is correctly spelled in the noisy sequence. Otherwise, it is a misspelling.

For every token in the noisy sequence there are two possibilities:

1. The token is spelled correctly. Usually there is a candidate correction that gets a higher probability from the language model. However, the score of

the candidate correction will be punished depending on λ. We compute the minimum λ needed to preserve the correctly spelled token.

2. The token is misspelled. Usually the ground truth correction gets a higher probability from the language model. However, the score of the ground truth correction will be punished depending on λ. We compute the maximum λ such that the token gets corrected.

For case 1, consider a misspelled token t_i and its ground truth correction c_i. Usually the probability of c_i is greater than that of t_i - otherwise NLMspell has no chance of correcting t_i. However, NLMspell will punish the score of c_i by $\text{edit\_distance}(t_i, c_i) \cdot \lambda$. That means, if λ is too high, NLMspell will not correct t_i. We get the probabilities $p(t_i|S)$ and $p(c_i|S)$ from the neural language model, where S is the intended sequence until c_i (c_i not included). The scores of t_i and c_i are the following:

$$\text{score}(t_i) = -\text{log}(p(t_i|S))$$

$$\text{score}(c_i) = -\text{log}(p(c_i|S)) + \text{edit\_distance}(t_i, c_i) \cdot \lambda$$

It follows that for

$$\lambda < \frac{\text{log}(p(c_i|S)) - \text{log}(p(t_i|S))}{\text{edit\_distance}(t_i, c_i)}$$

NLMspell will correct t_i and thereby generate a true positive.

For case 2, consider a correctly spelled token t_i. Usually there are candidate corrections that have a higher probability than t_i. However, NLMspell will punish their score by $\text{edit\_distance}(t_i, c_i) \cdot \lambda$. That means, if λ is high enough, NLMspell will preserve t_i. We compute the candidate set C_i for t_i and estimate the probability $p(c_j|S)$ for each candidate correction c_j, where S is the intended sequence until t_i. The scores for t_i and all c_j are the same as above. It follows that for

$$\lambda < \max_{c_j \in C_i} \frac{\text{log}(p(c_j|S)) - \text{log}(p(t_i|S))}{\text{edit\_distance}(t_i, c_j)}$$

NLMspell will replace t_i and thereby generate a false positive.

This procedure is done for every pair of sequences in a development benchmark. The resulting threshold λ-values for which NLMspell will generate a true positive or false positive are stored in a list and sorted. Then, we move the punishment value λ from infinity to the lowest threshold value. For each threshold value we count the number of true positives TP and false positives FP that get generated if λ was set to this value. To compute an F-score we also need the number of false negatives FN. Since we know the total number of misspellings M in the development benchmark, we can compute the number of false negatives as:

$$\mathrm{FN} = \mathrm{M} - \mathrm{TP}$$

Finally, we set λ to the threshold value that gives the highest F-score on the development benchmark.

The threshold values are computed under the assumption that the model is able to correct the left context perfectly. In practice this assumption will often not hold, and many corrections will depend on previous corrections. However, making this assumption allows us to compute the thresholds in a single run over the development data, instead of having to simulate multiple beam searches with different thresholds.

We slightly modify the approach described above to mimic the beam search performed by NLMspell and make use of the advantage that NLMspell can postpone decisions to the next beam search iteration. When computing the scores for an observed token and the candidate corrections, we do not only look at the probability of a token or correction given the left context, but also the probability that the next token of the intended sequence follows the observed token or candidate.

$$\mathrm{score}(t_i) = -\log(p(t_i|S)) - \log(p(g_{i+1}|S, t_i)$$

$$\mathrm{score}(c_i) = -\log(p(c_i|S)) - \log(p(g_{i+1}|S, c_i) + \mathrm{edit\_distance}(t_i, c_i) \cdot \lambda$$

where g_{i+1} is the next token from the intended sequence.

5.2. TranslationSpell: spelling correction with a neural machine translation model

The Neural Machine Translation Spelling Corrector (TranslationSpell) models the spelling correction task as a machine translation problem. The task is to translate from English with spelling errors to correct English.

5.2.1. Artificial training data

TranslationSpell incorporates a neural machine translation model that is trained on sequence pairs $(S, \bar{S})$, where S is a misspelled sequence and $\bar{S}$ the intended correctly spelled sequence.

The Wikipedia training sequences form the set of correctly spelled sequences. To generate misspelled sequences, for each character in the sequence a random error is introduced with probability $p = 0.05$. Errors are sampled from the following set:

1. Insertion of a random letter from a-z or A-Z.
2. Deletion of a character.
3. Replacement of a character by a random letter from a-z or A-Z.
4. Transposition of two neighboring characters.

The four error types are sampled with uniform probability.

5.2.2. Model architecture

The neural machine translation model is an encoder-decoder neural network with attention. Figure 8 depicts the computational graph of the model.

Encoder The encoder is a bidirectional LSTM cell. That is a combination of two LSTM cells: one generates hidden states processing the input sequence in forward direction, and the other in backward direction. Each cell has 1024 units. We choose to represent the input sequence as bytes, since a byte representation is more robust to noise than a subword representation. The subwords of a token can change entirely if a single character of the token is changed, whereas most of its bytes will remain intact.

For a byte sequence $S = (x_1, ..., x_n)$, the forward LSTM cell generates hidden states f_1 to f_n processing S from x_1 to x_n, and the backward LSTM cell generates hidden states b_n to b_1 processing S from x_n to x_1. For each x_i, the forward and backward hidden states f_i and b_i are concatenated to a hidden state vector $h_i = \mathrm{concat}(f_i, b_i)$ that encodes information about x_i and the left and right context.

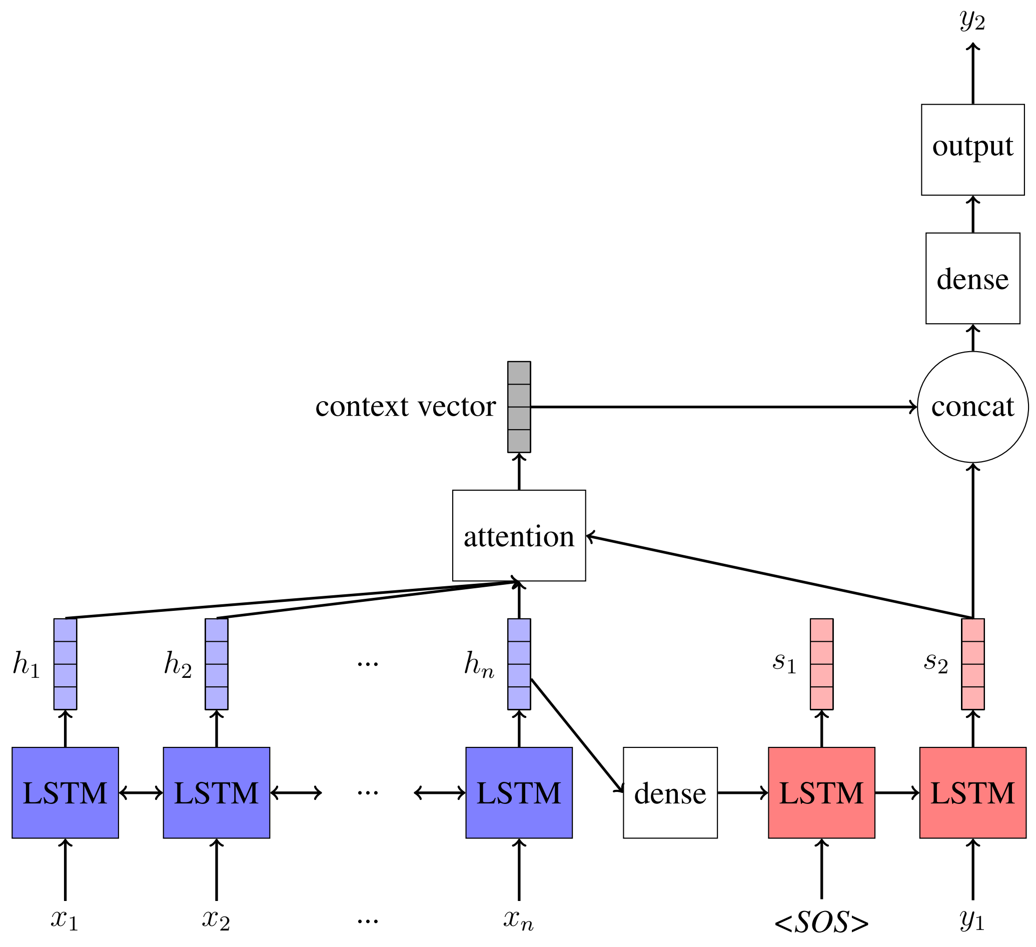


Figure 8: A neural translation model with attention. The bidirectional LSTM of the encoder is blue, the LSTM of the decoder red. The dense layer between the two LSTMs is the gate. <*SOS*> is a start symbol.

Decoder The decoder is a unidirectional LSTM cell with attention to the encoder. A dense layer and an output layer are stacked on top to predict the next subword.

The last hidden state h_n generated by the encoder is used as the initial hidden state of the decoder. For h_n to be usable in the decoder, we must pass it through a gate which transforms it to a vector of the same size as the decoder LSTM cell's hidden state. The gate is a dense layer that has as many units as the decoder LSTM cell.

The decoder LSTM cell has 1024 units. Given the previous hidden state and a subword, the it produces a hidden state $\bar{h_t}$. A context vector c_t is computed with the attention mechanism, where $\bar{h_t}$ is the query and the encoder hidden states h_1 to h_n are the keys and values.

The context vector c_t and the decoder hidden state $\bar{h_t}$ are concatenated and fed into a dense layer with 1024 units and ReLU activation, followed by a softmax output layer that predicts the next subword. Subwords come from a byte pair encoding with 2,000 merge steps.

5.2.3. Training

The training loss is the categorical crossentropy of the target sequence. The model is trained with the Adam optimizer [Kingma and Ba, 2015], with the default learning rate 0.001, no decay, and a batch size of 64. We let the model train for two days on a Nvidia GTX 1060 GPU.

5.2.4. Prediction

Before generating a prediction for an input sequence, the input sequence is split into sentences with the NLTK tokenizer [Bird et al., 2009]. This is done because translation models work better on shorter sequences, and sentence splitting is often robust to misspellings. Then, the sentences are translated independently.

First, the encoder is used to encode the input sentence. Then, the most likely target sentence according to the decoder is found with a best-first search.

We find that in some cases TranslationSpell predicts sentences that are too short, ignoring parts of the input sentence. To circumvent this problem, we punish the log likelihood of a predicted sequence by adding $-0.99 \cdot \mathrm{ed}$, where ed is the edit distance between the input sequence and the predicted sequence. The short sequences have a higher edit distance which results in a stronger punishment. We then continue the best-first search until all sequences in the search queue are less likely than the most likely punished sequence. The most likely punished sequence is returned as predicted sequence $\hat{S}$.

6. Baselines

6.1. UnigramSpell: a context-free baseline spelling corrector

As a baseline, we develop UnigramSpell, a context-free spelling corrector based on a vocabulary of correct words and their frequencies in the training data. The 100,000 most frequent words form the vocabulary V.

Error detection UnigramSpell processes the tokens of an input sequence $S = (t_1, ...t_n)$ from left to right. Tokenization is done with the regular expression method. All tokens that are numbers, punctuation marks or words contained in V remain unchanged. All other tokens are considered to be misspellings and replaced by a word from V.

Word replacement For a token t_i that is not in V, and therefore considered a misspelling, a candidate set C_i is generated the same way as in NLMspell. Then, t_i gets replaced by the single-edit candidate $c \in C_i$ that is most frequent in the training data. For split candidates, which consist of multiple words, the frequency of the least frequent word is considered. If no single-edit candidate exists in C_i, t_i gets replaced by the most frequent two-edit candidate. If C_i is empty, t_i does not get replaced.

6.2. NgramSpell: a context-dependent baseline spelling corrector

As a context-dependent baseline, we develop NgramSpell, a spelling corrector using an n-gram language model. NgramSpell is the same as NLMspell, but uses an n-gram language model instead of the neural language model to score the candidate corrections. All other parts - candidate generation, beam search, punishments, fitting of punishment values - are the same as in NLMspell.

N-gram model An n-gram language model is used for the candidate selection in NgramSpell. The model is an interpolation of a unigram model, a bigram model and a trigram model. The unigram model uses the frequencies of all words in the complete training data to estimate unigram probabilities.

The bigrams and trigrams, however, do not all fit into memory. Therefore a stochastic estimation of the 30,000,000 most frequent bigrams and trigrams is used. We train the bigram and trigram models on 10,000,000 training paragraphs. After each 1,000,000 paragraphs, we remove all bigrams and trigrams from memory, which are not among the 30,000,000 most frequent.

The interpolation factor α is optimized on 1,000 development paragraphs by testing 101 values ranging from 0 to 1 with uniform step size, and choosing the value for α that minimizes perplexity. The best interpolation factor found was $\alpha = 0.45$.

6.3. Commercial baseline

The Google docs spelling corrector is used as a commercial baseline. The misspelled benchmark sequences are copied into a Google docs document, and all suggestions by the spelling corrector are applied, until no more edits are suggested.

7. Experiments

This chapter explains the different experiments we did to analyse the language models and spelling correctors under study. A comparison of different language models is given in 7.1. The spelling correction benchmarks and evaluation metric are given in 7.2 and 7.3, followed by a comparison of the different spelling correctors in 7.4. In 7.5 we test different variants of NLMspell, and in 7.6 we test the robustness of the spelling correctors to varying error rates.

All models are implemented in TensorFlow [Abadi et al., 2015] and, if not stated differently, trained on two Nvidia Titan X GPUs with data parallelism.

7.1. Language models

For a comparison of different kinds of language models, we train a recurrent neural language model without attention, a recurrent neural language model with attention and a Transformer language model on two Nvidia Titan X GPUs with data parallelism for 24 hours each. All models use a byte pair encoding with 10,000 merge steps, which was found to be better than a smaller encoding during the hyperparameter optimization.

For the recurrent models with and without attention we choose the best architecture found during the hyperparameter optimization. That is a single-layer LSTM with 1024 units, followed by a dense layer with 1024 units. The learning rate is set to 0.003 and decayed with a factor of 0.95 every 1,000 training steps.

To find a good Transformer architecture, we test models with one to six layers. All other hyperparameters are equal to the ones stated in [Vaswani et al., 2017]. We test different learning rate schedules for each architecture and train models for 5,000 steps on a Nvidia GTX 1060 GPU. The learning rate schedules equal the schedules in [Vaswani et al., 2017], but with the learning rate multiplied by 2, 1 or 0.5. The learning rate is increased linearly during the first 4,000 steps and then decayed exponentially. We evaluate the models' perplexity on 1,000 development paragraphs, and then train models with one to six layers using the learning rate that gave the lowest perplexity for each number of layers. The results for these models can be found in appendix A.1. The 3-layer model gives the lowest perplexity. However, we suspect that on two Nvidia Titan X GPUs we can train a deeper model, because we can increase the batch size. We therefore decide to train a 4-layer Transformer model and set the batch size to 64, such that a training step takes less than one second.

Table 3 gives a comparison of the recurrent models and the Transformer model, as well as the small model from [Radford et al., 2019][7] and our trigram model, all evaluated on 10,000 development sequences. We observe that the top three models are attention models. Despite having more parameters than the recurrent models, the transformer model could be trained on more sequences than the recurrent models in 24 hours. However, our best model is the recurrent model with attention. Our Transformer model performs almost as well as the recurrent model with attention. Overall the model from [Radford et al., 2019] is the best model.

model	parameters	sequences	perplexity
n-gram	40M	10.0M	378.9
LSTM	29M	7.0M	157.0
LSTM+attention	32M	6.9M	*103.3*
Transformer	44M	7.3M	106.5
GPT small [Radford et al., 2019]	117M	-	**78.7**

Table 3: Per-token perplexity of our trigram model, recurrent model without attention, recurrent model with attention, 4-layer Transformer model, and the small model from [Radford et al., 2019] on 10,000 development sequences, together with the number of parameters of each model and the number of sequences it was trained on. The best result is marked bold. Our best result is marked italic.

7.2. Spelling correction benchmarks

To evaluate the quality of spelling correction methods, a benchmark of misspelled sequences with the corresponding correct sequences is needed. To the best of our knowledge no such dataset is publicly available. We therefore generate two

[7] Downloaded from `https://github.com/openai/gpt-2`

benchmarks with different types of misspellings: artificial, randomized typos and realistic typos from a collection of typos.

Noise induction Two different benchmarks are created to evaluate the performance of the approaches on the correction of artificial and realistic misspellings. For both types of misspellings a development and a test benchmark is created. This is done by inducing two different noise types into 1,000 paragraphs from the Wikipedia development and test sets. Only paragraphs with a maximum length of 1024 characters are considered, which equals the maximum sequence length during training of the translation model used in TranslationSpell. The paragraphs without noise form the ground truth sequences, and the noisy paragraphs the input sequences.

Artificial misspellings The *artificial* benchmark contains sentences with artificial, random misspellings. Each token that is not a number, punctuation mark or symbol is affected by noise with a probability of 20 %. If a token is affected by noise, we introduce a single error with probability 80 %, and two errors with probability 20 %. This follows the finding in [Damerau, 1964], that misspellings comprising a single character edit operation account for about 80 % of all misspellings. Errors are drawn from the following distribution:

- With 80 % probability a character edit is chosen. This can be a character insertion, deletion, replacement or the transposition of two neighboring characters.
- With 10 % probability a split operation is chosen. A space is inserted at a randomly chosen position.
- With 10 % probability a merge operation is chosen. The space after the token is removed.

When introducing two errors, every combination of error types is possible. Errors are applied in the following order: merge, edit, split. When introducing two character edits, it is ensured that no character is affected twice. Insertable characters are all lowercase letters a-z and uppercase letters A-Z. Merges are only applied when the token to merge with is not a number, punctuation mark or symbol. Splits are only applied when the token has length two or greater. Table 4 gives examples for all combinations of error types.

correct token(s)	errors	error type(s)	misspelling
algorithm	1	edit	algoritXhm
algorithm	1	split	algo rithm
algorithm runs	1	merge	algorithmruns
algorithm	2	edit, edit	aglorihm
algorithm	2	edit, split	amlgori thm
algorithm	2	split, split	a lgor ithm
algorithm runs	2	merge, edit	algorithrmruns
algorithm runs	2	merge, split	algo rithmruns
algorithm runs in	2	merge, merge	algorithmrunsin

Table 4: Examples of artificial misspellings for all combinations of error types that can be generated in the *artificial* benchmark.

Real misspellings The other benchmark contains misspellings from the typo collection. In the following, the benchmark will be called the *realistic* benchmark. The collection contains isolated misspellings with the corresponding intended texts. The misspellings and intended texts are mostly single words, but can be up to three words.

Before inducing the misspellings into paragraphs from the Wikipedia development and test sets, we split the misspelling-intended pairs into a development and a test set, both containing half of the pairs. We use the same ground truth paragraphs as in the *artificial* benchmark, and induce misspellings from the development set into the development paragraphs, and misspellings from the test set into the test paragraphs.

We extend the 80-20 split between single-edit and multi-edit misspellings reported in [Damerau, 1964] to more than two edit operations. We assume that a misspelling with k or more character edits is four times less likely than a misspelling with $k-1$ character edits. That is, we give single-edit misspellings 80 $\%$ probability, two-edit misspellings 16 $\%$ probability, three-edit misspellings 3.2 $\%$ probability and so on. As a formula, the probability for a misspelling with k

benchmark	single-edit	multi-edit	split	merge	mixed	nonword	real-word	total
artificial	5348	1015	651	1266	493	7294	1479	8773
realistic	3520	564	7	4	7	2448	1654	4102

Table 5: Number of ground truth tokens affected by the different error types in the test benchmarks.

edits is:

$$p_{\text{edits}}(k) = 0.8 \cdot 0.2^{k-1}$$

When inducing misspellings into a ground truth sequence $\bar{S}$ to generate a noisy sequence S, we proceed as follows. First, $\bar{S}$ is tokenized into tokens $(\bar{t_1}, ..., \bar{t_n})$. The tokens are processed from left to right. Each token is affected by noise with a base probability $p = 0.2$. If a token $\bar{t_i}$ is affected by noise, the number of edit operations is sampled following p_{edits}. Then, all misspellings for $\bar{t_i}$, $(\bar{t_i}, \bar{t_{i+1}})$ and $(\bar{t_i}, \bar{t_{i+1}}, \bar{t_{i+2}})$ that comprise the sampled number of edit operations are retrieved from the misspelling collection. One of the retrieved misspellings is selected at random and appended to S. If the collection does not contain a misspelling with the sampled number of edit operations, t_i is not affected by noise. If t_i is not affected by noise, it is simply appended to S.

Table 5 gives statistics about the error types sampled for the two test benchmarks, and 6 shows example sequences.

7.3. Spelling correction evaluation metric

Goal An evaluation metric for the spelling correction task has to measure two things:

1. How many of the misspelled tokens does the spelling corrector correct?
2. How many wrong corrections does the spelling corrector predict?

We are going to answer these two questions with precision and recall rates and combine the two rates into a single F-score.

Definition In the evaluation, we are given triples of sequences $(S_{\text{in}}, S_{\text{true}}, S_{\text{pred}})$ where S_{in} is the noisy input sequence, S_{true} the ground truth sequence and S_{pred}

benchmark	text
ground truth	Tverdislav was subsequently accused by rival boyars of helping the ruling princes to stifle the republic in Novgorod. Mutual accusations between boyar factions broke into a civil war in 1218, when the prince declared Tverdislav's posadnikship over, but this act was deemed unlawful by all the parties and Tverdislav managed to retain his position.
artificial	Tverdislav aws subsequently accused by rivBal boyars of hleping the ruling princes to sile thveG republli c in Novgorod. Mutual accusations between boyar factions broke into C Fcivil war in 1218, when the prince declared Tverdislav's posadnikship over, butO this act was deemed unlawful by all the pnrties and Tverdislav managed to retain his position.
realistic	Tverdislav wase subsequently accused by rival boyars of helping the ruling princes to stifle the republic in Novgorod. Mutual accusations between boyar factions broke into I civil war in 1218, when the prince declared Tverdislav's posadnikship over, bat this act was deemed unlawful by all tie parties anf Tverdislav managed to retain hed position.

Table 6: An example ground truth sequence and corresponding noisy sequences from the two benchmarks.

the predicted sequence. Each sequence is tokenized by splitting at spaces.

$$S_{\text{in}} = (t_1^{\text{in}}, ..., t_{n_{\text{in}}}^{\text{in}})$$

$$S_{\text{true}} = (t_1^{\text{true}}, ..., t_{n_{\text{true}}}^{\text{true}})$$

$$S_{\text{pred}} = (t_1^{\text{pred}}, ..., t_{n_{\text{pred}}}^{\text{pred}})$$

We use the following definitions to compute precision, recall and F-score:

- True positive: a ground truth token $t_i^{\text{true}} \in S_{\text{true}}$ that is misspelled in the input sequence S_{in} is correctly predicted in S_{pred}.
- False negative: a ground truth token $t_i^{\text{true}} \in S_{\text{true}}$ that is misspelled in the input sequence S_{in} is not correctly predicted in S_{pred}.
- False positive: the spelling corrector predicts the change of an input token $t_i^{\text{in}} \in S_{\text{in}}$ that results in a token in S_{pred} that is not in S_{true}.

For the determination of true positives, false negatives and false positives, we need to compute the following:

- Which tokens of the ground truth sequence are misspelled in the input sequence? The indices of these tokens are the ground truth positive set misspelled.
- Which tokens of the ground truth sequence are present in the predicted sequence? The indices of these tokens are the correct prediction set restored.
- Which tokens of the input sequence are changed by the spelling corrector? The indices of these tokens are the changed set changed.
- Which of the predicted tokens are correct? The indices of these tokens are the set of correct predicted tokens correct.

Then, true positives TP, false negatives FN and false positives FP can be computed:

$$\text{TP} = \text{misspelled} \cap \text{restored}$$

$$\text{FN} = \text{misspelled} \setminus \text{restored}$$

$$\text{FP} = \text{changed} \setminus \text{correct}$$

Notice that the sets contain indices instead of words. By index we mean the position of the token in the sequence. This is necessary because the same word can occur multiple times in a sequence, and it can be correct in one case and wrong in the other.

Example Consider the following input sequence S_{in}, ground truth sequence S_{true} and predicted sequence S_{pred}:

$$\begin{aligned} S_{\text{in}} &= \text{“\textit{Te cute cteats delicious fi sh.}”} \\ S_{\text{true}} &= \text{“\textit{The cute cat eats delicious fish.}”} \\ S_{\text{pred}} &= \text{“\textit{The cute act eats delicate fi sh.}”} \end{aligned}$$

The misspelled set contains the indices of *“The”*, *“cat”*, *“eats”* and *“fish.”*. Those are the tokens from the ground truth sequence that are misspelled in the input sequence.

$$\text{misspelled} = \{1, 3, 4, 6\}$$

The restored set contains the indices of *“The”*, *“cute”* and *“eats”*. Those are the tokens from the ground truth sequence that are present in the predicted sequence.

$$\text{restored} = \{1, 2, 4\}$$

The changed set contains the indices of *“Te”*, *“cteats”* and *“delicious”*. Those are the tokens from the input sequence that are changed by the spelling corrector.

$$\text{changed} = \{1, 3, 4\}$$

The correct set contains the indices of *“Te”* and *“cute”*. Those are the tokens from the input sequence for which the spelling corrector made the correct prediction.

$$\text{correct} = \{1, 2\}$$

Note that the indices in misspelled and restored refer to the ground truth sequence, whereas the indices in changed and correct refer to the input sequence.

The true positives are *“The”* and *“eats”*. Those are the tokens from the ground truth sequence that are misspelled in the input sequence and restored in the predicted sequence.

$$\text{TP} = \text{misspelled} \cap \text{restored} = \{1, 4\}$$

The false negatives are *"cat"* and *"fish."*, Those are the tokens from the ground truth sequence that are misspelled in the input sequence and not restored in the predicted sequence.

$$\text{FN} = \text{misspelled} \setminus \text{restored} = \{3, 6\}$$

The false positives are *"cteats"* and *"delicious"*. They are tokens from the input sequence that are changed by the spelling corrector but did not result in the correct tokens from the ground truth sequence.

$$\text{FP} = \text{changed} \setminus \text{correct} = \{3, 4\}$$

The token *"cute"* is a true negative. It is present in all three sequences. True negatives will not be used any further in the evaluation.

Procedure We compute the sets misspelled, restored, changed and correct from the sequences S_{true}, S_{in} and S_{pred} under the assumption that the noise induction process and the spelling corrector both change as less characters as possible.

From the backtrace of the edit distance matrix between S_{in} and S_{true} we get the character edit operations that transform S_{in} into S_{true}. All tokens from S_{true} that are affected by an edit operation are in the set misspelled. The removal of a space between two tokens affects both tokens.

We do the same for the sequences S_{in} and S_{pred} to get the tokens from S_{in} that are changed by the spelling corrector. Those tokens are in the set changed.

Next, we compute the token matching between S_{true} and S_{pred}. This is done with a dynamic programming procedure computing a matrix $D \in \mathbb{Z}^{n_{\text{true}} \times n_{\text{pred}}}$, where entry $D_{i,j}$ equals the number of matching tokens between the first i tokens from S_{true} and the first j tokens from S_{pred}. The procedure uses the following recursive definition:

$$D_{i,j} = \max \begin{cases} D_{i-1,j}, & \\ D_{i,j-1}, & \\ D_{i-1,j-1} + 1 & \text{if } t_i^{\text{in}} = t_j^{\text{pred}} \\ D_{i-1,j-1} & \text{otherwise} \end{cases}$$

with the base cases $D_{0,j} = 0 \ \forall j$ and $D_{i,0} = 0 \ \forall i$. From the backtrace from $D_{n_{\text{in}},n_{\text{pred}}}$ to $D_{0,0}$ we get the pairs $(t_i^{\text{in}}, t_j^{\text{pred}})$ such that the third line of the recursive definition is used to compute $D_{i,j}$. These are the matching pairs. All t_i^{in} in the matching pairs are in the set restored and all t_j^{pred} are correctly predicted tokens.

Finally we need to compute the set correct. It contains all tokens from S_{in} for which the spelling corrector made the correct prediction. We first group tokens from S_{in} together that get merged by the spelling corrector. To do so we look at the edit operations that transform S_{in} into S_{pred} and group tokens together if the space between them gets removed. Then, for each merged group we want to know the corresponding predicted tokens from S_{pred}. For that, the number of spaces that the spelling corrector inserts into the merged group is counted. If k spaces are inserted into a merged group, i.e. the merged token is split k times, the group results in $k + 1$ tokens. A merged group, such that the previous merged groups result in i predicted tokens and the group is split k times, results in the $(i + 1)^{\text{th}}$ to $(i + 1 + k)^{\text{th}}$ tokens in S_{pred}. If all the resulting tokens in S_{pred} are correctly predicted tokens, the tokens of the merged group are in the set correct.

Example (continued) Consider again the sequences S_{in}, S_{true} and S_{pred} from our example:

$$S_{\text{in}} = \text{“\textit{Te cute cteats delicious fi sh.}”}$$
$$S_{\text{true}} = \text{“\textit{The cute cat eats delicious fish.}”}$$
$$S_{\text{pred}} = \text{“\textit{The cute act eats delicate fi sh.}”}$$

The edit operations transforming S_{in} into S_{true} are:

- Insertion of an *h* after position 1, transforming *“Te”* into *“The”*.
- Insertion of an *a* after position 9, transforming *“cteats”* into *“cateats”*.
- Insertion of a space after position 10, splitting *“cateats”* into *“cat”* and *“eats”*.

- Deletion of a space at position 28, merging *"fi"* and *"sh."* into *"fish"*.

The affected tokens from S_{true} are *"The"*, *"cat"*,*"eats"* and *"fish"*. Their indices are in misspelled.

The edit operations transforming S_{in} into S_{pred} are:

- Insertion of an *h* after position 1, transforming *"Te"* into *"The"*.
- Insertion of an *a* after position 8, transforming *"cteats"* into *"acteats"*.
- Insertion of a space after position 10, splitting *"cateats"* into *"cat"* and *"eats"*.
- Three replacements of *o*, *u* and *s* at positions 21 to 23 with *a*, *t* and *e*, transforming *"delicious"* into *"delicates"*.
- Deletion of an *s* at position 24, transforming *"delicates"* into *"delicate"*.

The affected tokens from S_{in} are *"Te"*, *"cteats"* and *"delicious"*. Their indices are in changed.

The token matching between S_{true} and S_{pred} matches *"The"*, *"cute"* and *"eats"*. These are the tokens that are present in both sequences. Their indices in S_{true} are in restored. Their indices in S_{pred} are the set of correctly predicted tokens.

The spelling corrector does not merge any tokens. The only token that gets split is *"cteats"*. We therefore get the predictions *"Te"* $\rightarrow$ *"The"*, *"cute"* $\rightarrow$ *"cute"*, *"cteats"* $\rightarrow$ *"act eats"*, *"delicious"* $\rightarrow$ *"delicate"*, *"fi"* $\rightarrow$ *"fi"* and *"sh."* $\rightarrow$ *"sh."*. The input tokens that result only in correctly predicted tokens are *"Te"* and *"cute"*. Their indices are in correct.

From the four sets misspelled, restored, changed and correct we get the true positives, false negatives and false positives as described above, which are then used to compute precision, recall and F-score for the entire benchmark.

Detailed evaluation: error types To analyze if the spelling correctors perform different on different types of errors we distinguish the following types:

- Single-edit: a token is replaced by another token and the two tokens have edit distance one.
- Multi-edit: a token is replaced by another token and the two tokens have edit distance greater one.
- Split: a token is split into two tokens by inserting a space.

- Merge: two tokens are merged by removing a space.
- Mixed: multiple splits, multiple merges or a mix of splits, merges and character edits.

The error types of the true positives and false negatives are determined by the ground truth character edit operations transforming S_{in} into S_{true}. The error types of the false positives are determined by the character edit operations executed by the spelling corrector, that is, the operations transforming S_{in} into S_{pred}.

Error types refer to the type of the ground truth misspelling or predicted misspelling, not to the edit operations executed by the spelling corrector. That is, for example, if the spelling corrector inserts a space into an input token, it predicts a merge error, since it predicts that the token is the merge of two tokens.

Detailed evaluation: error classes To analyze how well the spelling correctors work on different classes of errors, we distinguish between nonword and real-word errors. We define that the 100,000 most frequent words in the training data are words, and all other strings are nonwords.

Tokens from the intended sequence, which are misspelled in the input sequence, are real-word errors if the misspelling is a word. If the misspelling consists of multiple tokens due to a token split, we require all tokens to be words to make the misspelling a real-word error. Otherwise, the misspelling is a nonword error.

False positives are classified as real-word errors if they involve an input token that is a word. If the input token is a nonword, the false positive is a nonword error.

Spelling error detection metric We introduce another metric that evaluates how well a spelling corrector detects spelling errors. The detection metric does not require that the spelling corrector finds the correct intended word, but measures how many of the misspelled input tokens get changed versus how many of the correctly spelled input tokens get changed.

Let M be the misspelled input tokens and C the input tokens changed by the spelling corrector. Then, true positives TP, false positives FP and false negatives

FN are defined as follows:

$$\mathrm{TP} = M \cap C$$

$$\mathrm{FP} = C \setminus M$$

$$\mathrm{FN} = M \setminus C$$

Precision, recall and F-score are computed from the numbers of true positives, false positives and false negatives.

This is a useful metric for two reasons: First, there are scenarios where it is enough to flag misspellings and let a user decide for the correct spelling instead of automatically correcting it, e.g. as a writing aid software. Second, it gives insight into the spelling corrector's functionality: if a spelling corrector often detects misspellings but does not correct them, it needs a better language model to select the right candidate correction.

7.4. Spelling correction results

7.4.1. Artificial benchmark

Table 7 gives the results on the benchmark with artificial misspellings. With an F-score of 91.5 %, NLMspell outperforms the other spelling correctors. It has the highest precision and highest recall among all approaches.

It is noteworthy that the context-dependent baseline, NgramSpell, is the second-best approach, outperforming the context-free baseline and the commercial baseline as well as TranslationSpell. This suggests that little context is often enough to find the correct word. However, there are cases where a better language model helps: replacing the traditional language model in NgramSpell with the neural language model in NLMspell improves the F-score by 3.4 % absolute. Comparing the context-free UnigramSpell to the context-dependent NgramSpell, we find that using context improves the F-score by 24.2 % absolute.

All approaches except for TranslationSpell show higher precision than recall scores. The most conservative method is the commercial baseline, with a high precision but low recall. The fact that TranslationSpell has higher recall than precision suggests that it has too much freedom in generating sequences that differ from the input sequences, thereby introducing many false corrections.

Regarding the error detection metric, we see the same pattern as for the correction metric: NLMspell has the highest F-score, followed by NgramSpell, TranslationSpell, the commercial baseline and UnigramSpell. However, the detection F-scores of NLMspell and NgramSpell are closer than the correction F-scores. Both approaches are almost equally well on detecting misspellings, but the more sophisticated model in NLMspell helps to find the correct intended word. The gap between the detection F-scores and correction F-scores for TranslationSpell, the commercial baseline and UnigramSpell is big. The commercial baseline has difficulties with randomly inserted uppercase letters. It often predicts the correct word, but keeps the wrong capitalization. UnigramSpell often predicts the wrong word because it does not make use of the context. TranslationSpell is good in detection misspellings, but has not developed a good language model to predict the correct intended word.

Models that have a higher correction F-score also achieve a higher sequence accuracy. However, it is difficult to solve a paragraph in its entirety. The best sequence accuracy is below 50 %.

Error classes Table 8 gives the analysis for real-word and nonword errors. All approaches perform better on nonword than on real-word errors. NLMspell has the smallest gap between the two error classes, 5.2 % absolute correction F-score. On both error classes, the top three approaches are the same as in the overall analysis, in the same order: NLMspell performs best, followed by NgramSpell and TranslationSpell. The only difference to the overall analysis is that UnigramSpell is slightly better on nonword errors than the commercial baseline, whereas the commercial baseline is better overall. UnigramSpell does not correct any real-word errors by design.

Error types Table 9 gives the results on the same benchmark, broken down to the different types of misspellings. We observe that NLMspell has better F-scores than the other approaches across all error types.

Multi-edit replacements seem more difficult to correct than misspellings of the other types. All approaches have much lower F-scores for multi-edit replacements than for single edit replacements. This includes our best approach, NLM-spell, where the F-score for multi-edit replacements is 11.1 % absolute lower than for single-edit replacements.

Random splits and merges on the other hand seem to be particularly easy to solve, with NLMspell getting correction F-scores above 95 % on these cases.

Artificial test set

model	detection			correction			sequence acc.
	precision	recall	F-score	precision	recall	F-score	
UnigramSpell	87.1 %	71.6 %	78.6 %	67.4 %	60.8 %	63.9 %	17.3 %
NgramSpell	97.4 %	95.0 %	96.2 %	89.3 %	87.0 %	88.1 %	43.1 %
commercial	96.7 %	73.2 %	83.3 %	75.3 %	58.6 %	65.9 %	22.8 %
NLMspell	97.5 %	95.7 %	**96.6 %**	92.5 %	90.6 %	**91.5** %	**49.5** %
TranslationSpell	88.8 %	93.7 %	91.2 %	75.1 %	77.0 %	76.0 %	28.2 %

Table 7: Results on the test set with artificial errors. The best results are marked bold.

7.4.2. Realistic benchmark

Table 10 gives the results on the benchmark with real misspellings. We observe that the realistic misspellings are more difficult to correct than the random noise. Almost all approaches have lower F-scores on the realistic benchmark than on the artificial benchmark. This is because the realistic benchmark contains a greater fraction of real-word errors than the artificial benchmark, which are more difficult to detect and correct. The best correction F-score on the realistic benchmark is 3.1 % absolute lower than the best result on the artificial benchmark. However, since there are less misspellings per sequence in the realistic benchmark than in the artificial benchmark, the best sequence accuracy is 7.9 % higher.

The commercial baseline is the only approach that has a higher correction F-score on the realistic than on the artificial benchmark, despite having a lower correction F-score. It seems to be designed to correct common misspellings instead of random noise.

Artificial test set

model	error class	detection			correction		
		precision	recall	F-score	precision	recall	F-score
UnigramSpell	nonword	87.1 %	96.5 %	91.6 %	67.4 %	73.1 %	70.1 %
	real-word	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
NgramSpell	nonword	98.0 %	96.7 %	**97.3 %**	90.7 %	88.8 %	89.7 %
	real-word	95.6 %	90.1 %	92.8 %	82.0 %	78.3 %	80.1 %
commercial	nonword	97.7 %	81.2 %	88.7 %	76.4 %	63.4 %	69.3 %
	real-word	92.2 %	50.3 %	65.1 %	66.6 %	34.7 %	45.6 %
NLMspell	nonword	98.0 %	96.0 %	97.0 %	93.7 %	91.2 %	**92.4 %**
	real-word	96.1 %	94.8 %	**95.5 %**	87.0 %	87.4 %	**87.2 %**
TranslationSpell	nonword	95.1 %	94.9 %	95.0 %	79.6 %	78.3 %	78.9 %
	real-word	74.1 %	90.0 %	81.3 %	57.2 %	70.5 %	63.2 %

Table 8: Performance of the spelling correctors evaluated for different error classes on the test set with artificial errors. The best results are marked bold.

Comparing the performance of the different spelling correctors on the realistic benchmark, we find a similar pattern like on the artificial benchmark. NLMspell has the best F-score, best precision and best recall. The difference to NgramSpell is greater than on the artificial benchmark, with a 7.2 % absolute difference in correction F-score. The commercial baseline and TranslationSpell have swapped their positions on the scoreboard, the commercial baseline now being better than TranslationSpell. The correction F-score of TranslationSpell drops by dramatic 16.0 % absolute. It appears that TranslationSpell can not generalize well from the randomized noise it was trained on to real misspellings.

Error classes Table 11 gives the analysis on the realistic benchmark broken down to nonword and real-word errors. NLMspell is best in the correction of both error classes.

The performance gap between the correction of real-word and nonword errors is greater than on the random benchmark for the two best approaches: 8.7 % and

Artificial test set

model	error type	detection			correction		
		precision	recall	F-score	precision	recall	F-score
UnigramSpell	single-edit	92.8 %	80.7 %	86.3 %	69.9 %	69.4 %	69.7 %
	multi-edit	76.6 %	88.4 %	82.1 %	51.9 %	41.4 %	46.0 %
	split	100.0 %	27.4 %	43.0 %	99.0 %	16.0 %	27.5 %
	merge	72.1 %	96.3 %	82.5 %	73.9 %	83.0 %	78.2 %
	mixed	42.9 %	54.3 %	47.9 %	16.7 %	8.7 %	11.5 %
NgramSpell	single-edit	96.9 %	95.2 %	96.0 %	89.1 %	88.6 %	88.9 %
	multi-edit	96.4 %	93.6 %	95.0 %	75.0 %	70.6 %	72.8 %
	split	99.3 %	96.3 %	97.8 %	93.9 %	95.2 %	94.6 %
	merge	99.8 %	95.5 %	**97.6 %**	99.7 %	91.4 %	95.3 %
	mixed	96.5 %	91.4 %	93.9 %	88.1 %	80.9 %	84.4 %
commercial	single-edit	97.4 %	75.5 %	85.0 %	78.1 %	58.8 %	67.1 %
	multi-edit	91.4 %	59.7 %	72.3 %	36.7 %	29.5 %	32.7 %
	split	97.9 %	70.2 %	81.8 %	85.5 %	72.7 %	78.6 %
	merge	97.5 %	92.8 %	95.1 %	96.8 %	80.6 %	88.0 %
	mixed	96.3 %	65.5 %	78.0 %	53.2 %	40.8 %	46.2 %
NLMspell	single-edit	97.1 %	95.6 %	**96.3 %**	93.0 %	91.6 %	**92.3 %**
	multi-edit	96.1 %	95.8 %	**95.9 %**	81.0 %	81.3 %	**81.2 %**
	split	99.1 %	96.7 %	**97.9 %**	95.4 %	95.2 %	**95.3 %**
	merge	99.4 %	94.7 %	97.0 %	99.7 %	92.3 %	**95.8 %**
	mixed	99.2 %	94.4 %	**96.7 %**	90.5 %	88.4 %	**89.4 %**
TranslationSpell	single-edit	93.7 %	92.4 %	93.1 %	80.3 %	78.1 %	79.2 %
	multi-edit	86.1 %	92.9 %	89.4 %	54.4 %	54.0 %	54.2 %
	split	94.1 %	97.0 %	95.5 %	78.1 %	90.8 %	83.9 %
	merge	96.0 %	95.1 %	95.5 %	97.4 %	86.6 %	91.7 %
	mixed	61.8 %	96.4 %	75.3 %	37.6 %	69.2 %	48.7 %

Table 9: Performance of the spelling correctors evaluated for different error types on the test set with artificial errors. The best results are marked bold.

Realistic test set

model	detection			correction			sequence acc.
	precision	recall	F-score	precision	recall	F-score	
UnigramSpell	67.3 %	59.5 %	63.1 %	50.5 %	44.7 %	47.4 %	22.0 %
NgramSpell	89.7 %	86.5 %	88.1 %	82.7 %	79.8 %	81.2 %	45.7 %
commercial	92.2 %	60.1 %	72.7 %	85.9 %	56.0 %	67.8 %	35.0 %
NLMspell	92.7 %	93.6 %	**93.1 %**	88.2 %	88.7 %	**88.4 %**	**57.4 %**
TranslationSpell	71.6 %	75.6 %	73.5 %	61.2 %	58.9 %	60.0 %	30.8 %

Table 10: Results on the test set with realistic errors. The best results are marked bold.

16.1 % absolute F-score difference for NLMspell and NgramSpell, compared to 5.2 % and 9.6 % on the artificial benchmark. TranslationSpell and the commercial baseline perform poorly on real-word errors.

The only performance increase in comparison to the artificial benchmark is seen for the commercial baseline on nonword errors. Apparently it is not designed to remove random noise, but to correct typical misspellings. However, it is the second-worst corrector on real-word errors, only beating UnigramSpell, which does not deal with real-word errors by design.

Error types Table 12 gives the analysis on the realistic benchmark, broken down to the different error types. NLMspell is the best approach across all error types except for merged words, where NgramSpell is better.

7.4.3. Error analysis

It follows a qualitative analysis of the wrong predictions made by our best approach, NLMspell, on both benchmarks.

A fraction of false positives is due to unusual capitalization in the ground truth data. For example, NLMspell corrects *"Knockout Tournament"* into *"knockout tournament"* and *"the eltway"* into *"the beltway"* instead of *"the Beltway"*, but the ground truth is upper case (apparently these are the names of a tournament

Realistic test set

model	error class	detection			correction		
		precision	recall	F-score	precision	recall	F-score
UnigramSpell	nonword	67.3 %	100.0 %	80.4 %	50.5 %	75.0 %	60.3 %
	real-word	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
NgramSpell	nonword	91.8 %	99.6 %	95.5 %	83.6 %	90.8 %	87.1 %
	real-word	85.6 %	67.3 %	75.4 %	80.8 %	63.4 %	71.0 %
commercial	nonword	93.8 %	83.0 %	88.0 %	87.9 %	77.7 %	82.5 %
	real-word	85.5 %	26.5 %	40.5 %	77.7 %	23.8 %	36.5 %
NLMspell	nonword	93.2 %	99.3 %	**96.2 %**	89.0 %	94.7 %	**91.8 %**
	real-word	91.8 %	85.1 %	**88.3 %**	86.8 %	79.8 %	**83.1 %**
TranslationSpell	nonword	85.6 %	93.3 %	89.3 %	68.9 %	74.1 %	71.4 %
	real-word	49.3 %	49.8 %	49.6 %	45.8 %	36.6 %	40.7 %

Table 11: Performance of the spelling correctors evaluated for different error classes on the test set with realistic errors. The best results are marked bold.

Realistic test set

model	error type	detection			correction		
		precision	recall	F-score	precision	recall	F-score
UnigramSpell	single-edit	82.6 %	60.3 %	69.7 %	62.8 %	50.9 %	56.3 %
	multi-edit	16.2 %	56.2 %	25.1 %	13.0 %	6.7 %	8.9 %
	split	0.0 %	5.0 %	0.0 %	0.0 %	0.0 %	0.0 %
	merge	11.2 %	100.0 %	20.2 %	1.2 %	100.0 %	2.4 %
	mixed	2.5 %	37.5 %	4.7 %	0.0 %	0.0 %	0.0 %
NgramSpell	single-edit	91.5 %	90.5 %	91.0 %	85.2 %	86.5 %	85.8 %
	multi-edit	77.9 %	62.4 %	69.3 %	64.2 %	37.6 %	47.4 %
	split	73.3 %	90.0 %	80.8 %	41.2 %	100.0 %	58.3 %
	merge	40.0 %	100.0 %	**57.1 %**	57.1 %	100.0 %	**72.7 %**
	mixed	26.7 %	50.0 %	34.8 %	14.3 %	57.1 %	22.9 %
commercial	single-edit	96.1 %	63.9 %	76.8 %	90.4 %	60.7 %	72.6 %
	multi-edit	72.8 %	35.3 %	47.5 %	65.8 %	26.6 %	37.9 %
	split	47.8 %	85.0 %	61.2 %	19.4 %	100.0 %	32.6 %
	merge	21.4 %	50.0 %	30.0 %	13.3 %	50.0 %	21.1 %
	mixed	38.7 %	37.5 %	38.1 %	9.7 %	42.9 %	15.8 %
NLMspell	single-edit	93.8 %	96.7 %	**95.2 %**	89.7 %	94.5 %	**92.1 %**
	multi-edit	86.6 %	74.1 %	**79.9 %**	76.5 %	52.7 %	**62.4 %**
	split	71.9 %	100.0 %	**83.6 %**	53.8 %	100.0 %	**70.0 %**
	merge	20.0 %	50.0 %	28.6 %	33.3 %	50.0 %	40.0 %
	mixed	60.0 %	87.5 %	**71.2 %**	42.9 %	85.7 %	**57.1 %**
TranslationSpell	single-edit	86.9 %	78.7 %	82.6 %	72.9 %	65.6 %	69.0 %
	multi-edit	52.5 %	57.6 %	55.0 %	28.6 %	17.9 %	22.0 %
	split	29.2 %	50.0 %	36.9 %	4.4 %	42.9 %	8.0 %
	merge	23.3 %	50.0 %	31.8 %	6.5 %	50.0 %	11.4 %
	mixed	22.8 %	62.5 %	33.4 %	0.9 %	42.9 %	1.8 %

Table 12: Performance of the spelling correctors evaluated for different error types on the test set with realistic errors. The best results are marked bold.

and a road).

Some errors are due to ambiguous cases that are hard to resolve. For example, for the input *“lacl tournament”*, NLMspell predicts *“last tournament”* instead of *“local tournament”*. Both corrections have edit distance two to the input. The fact that three characters of *“local”* are present in *“lacl”*, compared to two characters of *“last”* helps humans to resolve the case. However, NLMspell does not make use of this information.

Many false positives are introduced in names. Apparently the language model has a preference for some frequent names over other, less frequent names. For example, NLMspell corrects *“Michaeli”* into *“Michael”*.

Some wrong predictions result in grammatical errors, like predicting *“the strike protested”* for *“the striker protested”*, where it should be *“the strikers protested”*.

Misspelled abbreviations result in false negatives, because the abbreviations are either not in the vocabulary of NLMspell or very infrequent.

Most false negatives are because of infrequent words. For example, *“recategorizedbythe”* does not get split, *“Kölenr”* not corrected into *“Kölner”* and *“fadv-tracked”* not into *“fast-tracked”*.

Some false negatives result from the fact that NLMspell does not correct misspellings with more than two edits. For example, it does not correct *“take”* when it should be *“took”*, and *“a”* when it should be *“the”*. However, we doubt that the signal from the language model is strong enough that these cases could be corrected if the true word was a candidate correction.

All in all we see very few cases where we think a better language model could improve the corrections. Most errors by NLMspell can be explained with its candidate generation procedure, its preference for less edit operations, and infrequent, difficult cases. However, an exception for the correction of names and abbreviations could improve NLMspell.

7.4.4. Runtimes

Total runtimes of the approaches on the two test benchmarks are given in table 13.

UnigramSpell is by far the fastest algorithm. Most input tokens appear in

corrector	artificial	realistic
UnigramSpell	5.5	2.5
NgramSpell	4,790.0	4,967.2
NLMspell	17,150.5	18,458.7
TranslationSpell	3,134.1	2,308.8

Table 13: Total runtimes of the spelling correctors on the two benchmarks, measured in seconds. Each benchmark consists of 1,000 paragraphs from Wikipedia.

UnigramSpell's vocabulary and are not modified, therefore not requiring much runtime. In most of the remaining cases, UnigramSpell evaluates only the single-edit candidates, which are much less than the two-edit candidates. NgramSpell evaluates all candidates for all input tokens and is therefore much slower than UnigramSpell.

NLMspell evaluates only a subset of the candidates, but uses a neural language model with many parameters for the estimation, which makes it slower than NgramSpell. The probability estimation takes most of NLMspell's runtime, the candidate generation only a small fraction of it.

During the best-first-search, TranslationSpell appends a subword to the sequence with the highest likelihood in each iteration. Unlikely candidates are therefore never evaluated, which makes TranslationSpell faster than NgramSpell.

7.5. NLMspell variants

Next we test different variants of NLMspell. We are interested in whether NLMspell achieves better results when evaluating more candidate corrections, and whether the attention mechanism in the neural language model improves the performance.

We test a variant of NLMspell that evaluates 500 candidates of each type, instead of 100. However, since this slows down the algorithm, we reduce the number of beams to 3 instead of 10.

Another variant is NLMspell with the language model replaced by a recurrent

NLMspell variants

model	detection			correction			sequence acc.
	precision	recall	F-score	precision	recall	F-score	
original	97.5 %	95.7 %	**96.6 %**	92.5 %	90.6 %	**91.5 %**	**49.5 %**
more candidates	97.5 %	95.7 %	**96.6 %**	92.1 %	90.0 %	91.0 %	49.4 %
no attention	97.9 %	95.0 %	96.4 %	92.3 %	89.5 %	90.9 %	47.7 %

Table 14: Comparison of the three different variants of NLMspell on the artificial test benchmark. The best results are marked in bold.

neural network without attention. We take the model we trained for 24 hours on two Nvidia Titan X GPUs.

We evaluate the two variants and the original variant on the artificial test benchmark. The detection and correction results are given in table 14. We observe that the greater number of beams in the original variant of NLMspell is more important than the increased number of considered candidates. When increasing the number of candidates of each type to 500 while reducing the number of beams to 3, the correction F-score drops by 0.5 % absolute and the sequence accuracy by 0.1 % absolute. It appears that there are few cases where the attention mechanism helps to find the correct candidate. When using the model without attention, the correction F-score drops by 0.6 % absolute and the sequence accuracy by 1.8 % absolute.

When evaluating more candidates and using less beams, the average runtime per paragraph increases from 17.1 seconds to 21.0 seconds. The model without attention on the other hand is faster than the model with attention, with an average runtime of 13.7 seconds.

7.6. Robustness

Next we are interested in how the performance of the spelling correctors changes when the rate of misspellings in the input sequences varies. We generate two new artificial test benchmarks with the same ground truth sequences as in the previous experiment. In one benchmark we introduce an error into each token with probability $p = 0.1$, in the other benchmark with probability $p = 0.5$. 22.3 % of the test paragraphs contain no errors when $p = 0.1$, and 5.1 % when $p = 0.5$.

The correction evaluation of the spelling correctors on these two benchmarks is given in table 15.

We observe that the F-scores for $p = 0.1$ are lower than for $p = 0.2$ and $p = 0.5$ for all spelling correctors. For all spelling correctors the drop in precision is higher than the drop in recall. This is natural since there are less positive examples in the ground truth data, so all approaches will produce more false positives and less true positives. In addition, the punishment values for NgramSpell and NLMspell are fit on a development benchmark with an error rate of $p = 0.2$, that is assuming a higher base error rate than with $p = 0.1$. This leads to the spelling correctors making too many corrections. The same holds for TranslationSpell where every character in the input sequences has a probability of 5 % to be corrupted. The sequence accuracy, on the other hand, increases with less misspellings.

All approaches are robust to a higher error rate of $p = 0.5$. The F-score of NgramSpell drops only by 0.2 % absolute compared to the benchmark with $p = 0.2$, and increases by 0.1 % absolute for NLMspell, by 2.5 % absolute for UnigramSpell and by 0.5 % absolute for TranslationSpell. UnigramSpell does not use the context to correct misspellings, so misspellings do not interact with each other, which makes it robust to more misspellings. TranslationSpell was trained on sequences containing many misspellings, which seems to fit with the $p = 0.5$ benchmark. However, the sequence accuracies decrease with higher error rates for all approaches, since correcting an entire sequence becomes more difficult. The best approach gets half as many sequences right with $p = 0.5$ compared to $p = 0.1$.

Robustness experiment

model	error probability $p = 0.1$				error probability $p = 0.5$			
	precision	recall	F-score	seq. acc.	precision	recall	F-score	seq. acc.
UnigramSpell	58.1 %	59.3 %	58.7 %	23.8 %	73.9 %	60.3 %	66.4 %	9.5 %
NgramSpell	85.5 %	85.2 %	85.4 %	55.1 %	90.2 %	85.5 %	87.8 %	25.5 %
NLMspell	89.7 %	89.7 %	**89.7 %**	**61.6 %**	93.5 %	89.7 %	**91.6 %**	**30.8** %
TranslationSpell	69.3 %	77.2 %	73.1 %	37.8 %	77.8 %	75.2 %	76.5 %	14.2 %

Table 15: Correction results on artificial benchmarks with varying error probabilities. The best result on each benchmark is marked in bold.

8. Limitations

Error types In this work we focus on the correction of typographic errors, excluding grammatical errors, punctuation errors and cognitive errors.

Our approaches to spelling correction are developed and evaluated under the assumption that misspellings are likely to be orthographically similar to the intended text. In cases where this assumption does not hold, our approaches will not perform well.

In particular NLMspell is limited to the correction of misspellings comprising up to two character edit operations, and fails for misspellings comprising three or more edit operations.

Cognitive errors like word choice errors and grammatical errors will follow particular patterns that could be exploited by statistical and machine learning methods. Since we do not have training data for such errors at hand, we limit our approach to the correction of typographic errors where the misspelled word is orthographically similar to the intended word.

We do not address punctuation errors, because we believe them to be cognitive errors rather than typographic errors. However, language models could be used to correct punctuation with similar approaches like in this work.

Closed vocabulary correction Our baselines and NLMspell predict words from a fixed vocabulary. In our experiments, we used the 100,000 most frequent words from the training data as possible target words. While that covers most intended words for the misspelled input sequences, there is a fraction of words that can not be corrected. Those will be rare words like names, hyphenated compound words and words from foreign languages.

TranslationSpell on the other hand is open-vocabulary. The translation model predicts a sequence of subwords which can be any output string, without vocabulary restrictions. However, TranslationSpell performed worse than the closed-vocabulary approaches in our experiments.

9. Conclusion

Attention models are better language models. The best three language models studied in our work incorporate attention mechanisms: the Transformer model from [Radford et al., 2019] performed best, followed by our recurrent neural network with attention and our Transformer model.

Neural language models improve spelling correction. We have shown that the recent progress in neural language modeling can raise the performance in the field of spelling correction over the performance of traditional approaches.

Our best spelling corrector, NLMspell, achieves correction F-scores of 91.5 % and 88.4 % on artificial and real spelling errors, being better than an alternative approach using a trigram language model, a neural machine translation approach and context-free and commercial baselines.

Real-word errors and multi-edit errors are difficult. The correction of real-word errors and multi-edit errors remain difficult problems. All methods under study perform worse on those error types than on nonword errors and errors comprising a single edit operation.

Artificial, randomly induced noise is easier to correct than real misspellings. Humans tend to make more real-word errors, which are difficult to solve.

Posing spelling correction as a machine translation task is challenging. When posing the spelling correction problem as a machine translation task, we face three major difficulties.

First, it is crucial that the model does not overfit the training data. We circumvented this problem by generating training data with a randomized procedure, so that the model sees a variety of misspellings during training.

Second, the training data has to reflect the patterns of realistic misspellings. We hoped that the model trained on artificial, randomized misspellings, would generalize to real misspellings, which are a subset of the randomized misspellings. However, our analysis showed that the translation model is much better in correcting artificial errors than in correcting real errors.

Third, the model has to deal with ambiguities. Before doing this work, we believed the spelling correction task to be easier than the machine translation task, since most of the input tokens are correctly spelled and can simply be copied to the output sequence. In addition, a translation system has to deal with word orders differing in the input and output language, as well as with cases where one word in the input sequence can translate to multiple words in the output sequence, or the other way around. A noisy sequence with misspellings, on the other hand, can usually just be corrected from left to right. However, the work on NLMspell showed that a lot of ambiguities lie in the spelling correction task. When considering all words within an edit distance of two to an observed input token, the number of candidates often exceeds 2,000. We are not aware of any pair of languages where one word in the one language can be translated to as many different words in the other.

10. Future work

The future work on neural language models for spelling correction will follow two directions: one is to extend NLMspell to more error types, and the other is to improve TranslationSpell.

Extension to more error types NLMspell generates candidate corrections for every input token following a fixed procedure. Currently this does not include candidates with an edit distance greater than two to the observed token. We do not think that all those candidates can be included, because that would be too many candidates that NLMspell can not score in reasonable amount of time. However, a fixed set of common error patterns, like typing a single character instead of two equal characters, could be used as additional edit operations to create candidates with more than two edit operations.

An error type that can be supported by NLMspell in the future is punctuation errors. For tokens ending with a punctuation mark, the token without punctuation mark could be added to the candidate set. For tokens not ending with a punctuation mark, one could generate candidates ending with the common punctuation marks.

Runtimes With average runtimes of more than 18 seconds per paragraph, NLMspell is more of theoretical interest than of practical use for most applications. However, it shows how far spelling correction can get when using neural language models. To make a practical spelling corrector out of NLMspell, the number of candidate evaluations must be reduced by introducing a better preselection strategy.

Language models trained on more diverse text Our models are trained and evaluated on Wikipedia. Text in Wikipedia usually follows certain patterns, including repeating topics and sentence structures. We doubt that the models generalize well to other domains, for example digital communication, where language patterns are different. One could train several models on different domains and plug them into different versions of NLMspell, which can then be used on different domains. An alternative is to train a single model on a diverse dataset containing texts from multiple sources, such that a single version of NLMspell will work on multiple domains.

Improved translation model The machine translation approach has the potential to make use of the patterns that are present in real misspellings.

Our experiments have shown that training on random noise is not enough to correct real misspellings. More realistic noise could be induced using a large set of misspellings extracted from a noisy corpus like ClueWeb or other World Wide Web corpora. Appendix B presents a typo extraction method. To prevent overfitting to the extracted misspellings, one could proceed similar to [Xie et al., 2018] and first train a noise model on the misspellings, which will then be used to sample misspellings for all words in the training corpus, including words that originally did not appear in the extracted misspellings.

In principle, the decoder of a translation model should be able to learn a language model as good as the neural language models from our work, and together with the encoder could be trained as an end-to-end spelling corrector, without the necessity of generating candidate corrections, computing edit distances and fitting punishment values as in NLMspell. Predicting subwords from a byte pair encoding, this would even be an open-vocabulary spelling corrector.

Transformer networks have recently become state of the art in neural machine translation, and are promising to work best for spelling correction as a machine translation task, too.

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

[Ba et al., 2016] Ba, L. J., Kiros, J. R., and Hinton, G. E. (2016). Layer Normalization. *CoRR*, abs/1607.06450.

[Bahdanau et al., 2015] Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Belinkov and Bisk, 2018] Belinkov, Y. and Bisk, Y. (2018). Synthetic and Natural Noise Both Break Neural Machine Translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[Bird et al., 2009] Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

[Boytsov, 2011] Boytsov, L. (2011). Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1).

[Church and Gale, 1991] Church, K. W. and Gale, W. A. (1991). Probability scoring for spelling correction. *Statistics and Computing*, 1(2):93–103.

[Damerau, 1964] Damerau, F. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176.

[Devlin et al., 2019] Devlin, J., Chang, M., 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186.

[Falkner et al., 2018] Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 1436–1445.

[Ghosh and Kristensson, 2017] Ghosh, S. and Kristensson, P. O. (2017). Neural Networks for Text Correction and Completion in Keyboard Decoding. *CoRR*, abs/1709.06429.

[Gong et al., 2019] Gong, H., Li, Y., Bhat, S., and Viswanath, P. (2019). Context-Sensitive Malicious Spelling Error Correction. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2771–2777.

[Goodfellow et al., 2016] Goodfellow, I. J., Bengio, Y., and Courville, A. C. (2016). *Deep Learning*. Adaptive computation and machine learning. MIT Press.

[Hochreiter, 1991] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*, 91(1).

[Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

[Jamieson and Talwalkar, 2016] Jamieson, K. G. and Talwalkar, A. (2016). Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, pages 240–248.

[Karnin et al., 2013] Karnin, Z., Koren, T., and Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. In *International Conference on Ma-*

chine Learning, pages 1238–1246.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Kukich, 1992] Kukich, K. (1992). Techniques for Automatically Correcting Words in Text. *ACM Comput. Surv.*, 24(4):377–439.

[Li et al., 2018] Li, H., Wang, Y., Liu, X., Sheng, Z., and Wei, S. (2018). Spelling Error Correction Using a Nested RNN Model and Pseudo Training Data. *CoRR*, abs/1811.00238.

[Liu et al., 2018] Liu, P. J., Saleh, M., Pot, E., Goodrich, B., Sepassi, R., Kaiser, L., and Shazeer, N. (2018). Generating Wikipedia by Summarizing Long Sequences. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[Luong, 2016] Luong, M.-T. (2016). *Neural machine translation*. PhD thesis, Stanford University.

[Manning et al., 2010] Manning, C., Raghavan, P., and Schütze, H. (2010). Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103.

[Martin and Jurafsky, 2019] Martin, J. H. and Jurafsky, D. (2019). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, 3rd edition*. Draft available at `https://web.stanford.edu/~jurafsky/slp3/`.

[Mays et al., 1991] Mays, E., Damerau, F. J., and Mercer, R. L. (1991). Context based spelling correction. *Inf. Process. Manage.*, 27(5):517–522.

[Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119.

[Pruthi et al., 2019] Pruthi, D., Dhingra, B., and Lipton, Z. C. (2019). Combating Adversarial Misspellings with Robust Word Recognition. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5582–5591.

[Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

[Rayner et al., 2006] Rayner, K., White, S., Johnson, R., and Liversedge, S. (2006). Raeding wrods with jubmled lettres: there is a cost. *Psychological science*, 17(3):192–193.

[Sakaguchi et al., 2017] Sakaguchi, K., Duh, K., Post, M., and Durme, B. V. (2017). Robsut Wrod Reocginiton via Semi-Character Recurrent Neural Network. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 3281–3287.

[Sennrich et al., 2016] Sennrich, R., Haddow, B., and Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423.

[Solaiman et al., 2019] Solaiman, I., Brundage, M., Clark, J., Askell, A., Herbert-Voss, A., Wu, J., Radford, A., and Wang, J. (2019). Release strategies and the social impacts of language models. *CoRR*, abs/1908.09203.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112.

[Tetreault et al., 2017] Tetreault, J. R., Sakaguchi, K., and Napoles, C. (2017).

JFLEG: A Fluency Corpus and Benchmark for Grammatical Error Correction. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers*, pages 229–234.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008.

[Whitelaw et al., 2009] Whitelaw, C., Hutchinson, B., Chung, G., and Ellis, G. (2009). Using the Web for Language Independent Spellchecking and Autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP 2009, 6-7 August 2009, Singapore, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 890–899.

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, abs/1609.08144.

[Xie et al., 2018] Xie, Z., Genthial, G., Xie, S., Ng, A. Y., and Jurafsky, D. (2018). Noising and Denoising Natural Language: Diverse Backtranslation for Grammar Correction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 619–628.

[Zhou et al., 2017] Zhou, Y., Porwal, U., and Konow, R. (2017). Spelling Correction as a Foreign Language. *CoRR*, abs/1705.07371.

The author does not take any responsibility for the content of the linked web pages.

A. Hyperparameter experiments

A.1. Transformer models

layers	parameters	batch size	sequences	perplexity
1	22.9M	32	5.8M	218
2	29.9M	16	4.5M	196
3	37.0M	16	3.7M	**183**
4	44.1M	8	3.0M	300
6	58.3M	8	2.6M	426

Table 16: Perplexity of different Transformer language models trained for 24 hours on a GTX 1060 GPU. The best result is marked in bold.

A.2. Recurrent language models

Neural network architectures In the first phase of the phased grid search, we search for the best architecture for the neural language model. Performing a grid search on the architecture search space, we train neural language models with 20 different architectures for one day each on a Nvidia GTX 1060 GPU. The architectures contain byte pair encodings with 2,000 or 10,000 merge steps, one or two LSTM layers with 512, 1024 or 2048 units, always a single dense layer on top with the same number of units as for the LSTM layers, and either with or without self-attention. The batch size is set to 128, or smaller if 128 leads to out-of-memory errors. Perplexity of a development set with 1,000 sequences is evaluated for each model and the architecture giving the lowest perplexity is chosen for the final model. The results of the first phase are found in table 17.

BPE merges	LSTM units	attention	parameters	batch size	sequences	perplexity
2000	512	no	4.4M	128	15.0M	330
		yes	*5.2M*	*128*	*15.7M*	*189*
	2 × 512	no	6.5M	128	12.1M	327
		yes	7.3M	128	12.0M	169
	1024	no	13.0M	128	9.0M	243
		yes	16.2M	64	5.4M	163
	2 × 1024	no	21.4M	64	4.2M	273
		yes	24.6M	64	3.8M	143
	2048	no	42.8M	64	3.2M	250
		yes	55.4M	32	1.5M	200
10000	512	no	12.6M	128	6.9M	233
		yes	13.4M	64	5.9M	155
	2 × 512	no	14.7M	64	5.1M	244
		yes	15.5M	64	5.0M	156
	1024	no	29.4M	64	3.9M	186
		yes	32.6M	64	3.5M	**136**
	2 × 1024	no	37.8M	32	1.9M	229
		yes	40.9M	32	1.9M	148
	2048	no	75.6M	32	1.7M	241
		yes	88.2M	16	0.8M	200

Table 17: Evaluation of different neural language model architectures. Values for empty table entries are equal to the value in the previous line. Perplexity is measured per token on a development set with 1,000 sequences. The best result is marked in bold. All models except for the model written in italic are trained for 24 hours on a GTX 1060 GPU. The *italic* model was trained for 30.5 hours accidentally.

Learning rates In the second phase, we test different learning rate schedules for the chosen architecture. To reduce the number of training runs, we optimize the learning rate schedule in two steps: first, we find a good initial learning rate, and second, we test different decay rates. To find a good learning rate, we train models for 3,000 training steps, varying the learning rate between 0.01, 0.005, 0.003, 0.002, 0.001 and 0.0005. Training a model takes approximately 75 minutes. The initial learning rate resulting in the model with the lowest perplexity on the development set is chosen as the initial learning rate for training the final model.

learning rate	perplexity
0.01	∞
0.005	551
0.003	**373**
0.002	433
0.001	466
0.0005	627

Table 18: Evaluation of different initial learning rates for the best architecture from A.2. Each model is trained for 3000 steps. The best result is marked in bold.

Decay rates Finally, we are interested in finding a good decay rate. Starting with the best initial learning rate found in the previous experiment, we train different models, decaying the learning rate every 1,000 training steps with decay rates 1 (no decay), 0.99, 0.95 or 0.5. We train each model for 20,000 steps. Training a model takes approximately 9 hours. Again the perplexity of each model on the development set is evaluated and the learning rate schedule that gave the lowest perplexity is chosen for training the final model.

decay rate	perplexity
1	212
0.99	196
0.95	**156**
0.5	257

Table 19: Evaluation of different decay rates for the best architecture from A.2 with the best initial learning rate from A.2. Each model is trained for 20,000 steps. The best result is marked in bold.

B. Typo extraction method

Similar to [Whitelaw et al., 2009], we use a World Wide Web corpus as a source of typos with corresponding intended words. Text from the World Wide Web is noisy, in the sense that it consists of correctly spelled and misspelled words. We make the following assumptions:

1. Correctly spelled words are more frequent than misspelled words.
2. Misspelled words are similar to the corresponding intended words.
3. Misspelled words and corresponding intended words appear in the same contexts.

We use ClueWeb as World Wide Web text corpus. Raw text was extracted from 2 out of 244 folders of ClueWeb by searching for lines that start with a <*p*> tag and contain no other HTML tag, except for an optional <\p> tag in the end. The resulting corpus contains 17,006,105 paragraphs.

Word frequency counts and trigram frequency counts were computed from the resulting sequences. We look at trigrams $t = (l, w, r)$ and $t' = (l', w', r')$ such that $l = l'$, $r = r'$, i.e. w and w' appear in the same context. We add w' to the misspellings of w, if one of the following cases is met:

1. The edit distance between is w and w' is one, w at least 100 times more frequent than w' and t at least 200 times more frequent than t'.
2. The edit distance between is w and w' is two, w at least 10,000 times more frequent than w' and t at least 1,000 times more frequent than t'.

The resulting typo collection contains 37,795 misspellings for 5,240 correctly spelled words. Table 20 shows examples of extracted typos.

intended word	misspellings
especially	Especialy, eYpecially, epecially, epsecially, escpecially, esecially, esepcially, esopecially, espacially, espacialy, espaecially, espcecially, espcially, especailly, especaily, especally, especcially, especcialy, especiall, especiallin, especiallly, especiallt, especiallyÂ, especialy, especialyl, especiaslly, especilally, especilly, especitally, espeically, espepcially, espescially, espesially, espoecially, esspeically, expecially, expecialy, specally, speciallt, specically
is	Bis, Bs, Cs, Dis, Ds, Eis, Fs, Gis, Hs, Js, Ls, Mis, Ms, Nis, Os, Ps, Ris, Rs, Sis, Ss, Tis, Ts, Us, Vis, Vs, Ws, Ys, _s, ais, bis, bs, cis, cs, dis, ds, eis, es, fis, fs, gis, gs, hs, i's, iA, iB, iD, iI, iOs, iQ, iX, ia, ias, ib, ic, ics, id, ids, ie, ies, ifs, ig, ih, ihs, ii, iis, ij, ik, il, im, ins, io, ios, ip, ips, iq, ir, isa, isc, ise, ish, isi, isl, ism, isn, iso, iss, ist, isÂ, ius, iv, ix, iz, js, ks, lis, ls, ms, ns, ois, os, ps, qs, ris, rs, si, sis, ss, tis, ts, vis, vs, wis, ws, yis, ys
world	orld, owrld, wiorld, wirld, woald, wodld, woeld, woerld, woirld, wold, wolrd, woprld, wordd, wordl, workd, worl, worldm, worldt, worldy, worlf, worlk, worls, wotld, wotrld, wourld, woyrld, wrld, wrold, wurld, zorld
University	Ubiversity, Uiversity, Uniersity, Unievrsity, Univerisity, Univeristy, Univerity, Universirty, Universiry, Universit, Universita, Universite, Universitys, Universiy, Universiyt, Universiyty, Universtity, Universtiy, Universty, Univesity, Univesrity, Univewrsity, Univsersity, Unniversity, Unversity, Unviersity, Unviversity
of	Ef, Of, af, bof, cf, dof, ef, eof, f, ff, fo, gf, iof, lf, nof, o, oF, oK, ob, oc, od, oe, ofa, ofd, ofg, ofm, ofr, oft, ofÂ, og, ogf, oh, oi, oif, ok, ol, om, oof, op, opf, orf, os, osf, ot, ou, ouf, ov, ow, oy, pf, rf, sof, tof, uf, yf
Stanford	Standford
Google	Goggle, Googe, Googgle, Googl, Googled, Googls, Goole, Goolge, Gooqle, iGoogle, oogle

Table 20: Examples of typos extracted from ClueWeb.