Fast Whitespace Correction with Encoder-Only Transformers

Hannah Bast[1] and **Matthias Hertel**[1,2] and **Sebastian Walter**[1] *

[1]University of Freiburg
Department of Computer Science
Freiburg, Germany
{bast,swalter}@cs.uni-freiburg.de

[2]Karlsruhe Institute of Technology
Institute for Automation and Applied Informatics
Karlsruhe, Germany
matthias.hertel@kit.edu

Abstract

The goal of whitespace correction is to fix space errors in arbitrary given text. For example, given the text *whi te space correctio nwithTransf or mers*, produce *whitespace correction with Transformers*. We compare two Transformer-based models, a character-level encoder-decoder model and a byte-level encoder-only model. We find that the encoder-only model is both faster and achieves higher quality. We provide an easy-to-use tool that is over 900 times faster than the previous best tool, with the same high quality. Our tool repairs text at a rate of over 200 kB/s on GPU, with a sequence-averaged F_1-score ranging from 87.5% for hard-to-correct text up to 99% for text without any spaces.

1 Introduction

Most natural language processing applications assume a segmentation of the text into words. In English (and many other languages), this segmentation is typically achieved by splitting the text at space characters (and a few additional rules). However, many texts contain a significant amount of space errors, that is, spurious spaces or missing spaces. These can be due to OCR errors, imperfect extraction from PDF files, or typing errors. See Bast et al. (2021) for a more thorough discussion of the sources of such errors.

We consider the following *whitespace correction* problem: given a text in natural language, with an arbitrary amount of missing or spurious spaces, compute a variant of the text with correct spacing. The text may contain spelling errors, which make the task more difficult, but it's not part of the problem to correct them. However, as shown in Bast et al. (2021), with spaces repaired, spelling-correction algorithms do a much better job.

The best previous methods for whitespace correction achieve high F_1-scores, however, at the price of very slow processing speeds; see Section 2. This is a major obstacle for the practical use of such systems for large amounts of text. Our goal in this work is to provide a practical tool with a much higher speed, without sacrificing the high quality.

1.1 Contributions

We consider these as our main contributions:

• We provide a practical method for whitespace correction that is over 900 times faster than the best previous method, with the same high quality across a wide selection of benchmarks. On an A100 GPU, our tool repairs text at over 200 kB/s with a sequence-averaged F_1-score ranging from 87.5% (for hard-to-correct text) to 99% (for text without any spaces).

• We compare two Transformer-based models: a character-level encoder-decoder model (that outputs the repaired sequence), and a byte-level encoder-only model (that predicts for each gap between two characters whether there should be a space or not). Both models take existing spaces in the input into account. The encoder-only model is both faster and better.

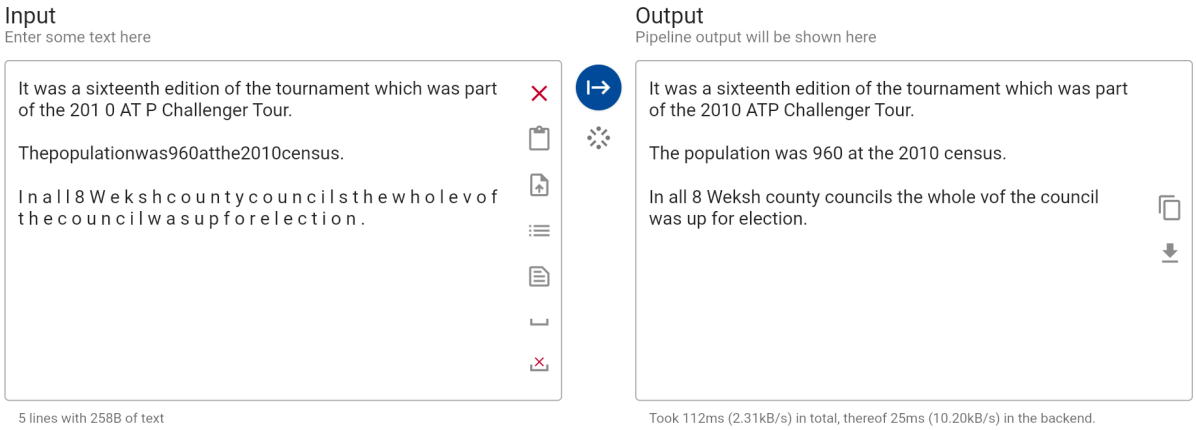
• We provide our whitespace correction models as a Python package with an easy-to-use command-line tool and as a web application; see Figure 1 and https://whitespace-correction.cs.uni-freiburg.de. The website provides links to public GitHub repositories with all our code, data, benchmarks, trained models, and a Docker setup. It also includes features to easily replicate our benchmark results and visualize model predictions on benchmarks.

2 Related Work

Recent work on spelling correction (Sakaguchi et al., 2017; Li et al., 2018; Pruthi et al., 2019; Jayanthi et al., 2020) and OCR postcorrection (Hämäläinen and Hengchen, 2019) predicts one

* Author contributions are stated in the end.

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 3: System Demonstrations, pages 389–399



Legend: ✕ Clear input, Paste clipboard into input, Upload a text file, Choose an example input, Load a benchmark (see Section 4.1), Insert whitespaces between all characters, Delete whitespaces between all characters, Copy output to clipboard, Download output as text file, Run model on input, Live as-you-type whitespace correction[1]

Figure 1: Our web interface for whitespace correction. The user can run any of our EO and ED models (panel for model selection not shown) on arbitrary text input. In the web app, there are tooltips instead of a legend.

word for every input token, without addressing space errors. To use these systems for text with a combination of spelling or OCR errors and whitespace errors, it is necessary to correct the spaces first. Other OCR postprocessing systems also try to correct space errors, but with limited success (Kissos and Dershowitz, 2016; D'hondt et al., 2017; Schulz and Kuhn, 2017; Nguyen et al., 2020). Bast et al. (2021) showed that space errors can be corrected separately with models that are robust against OCR and spelling errors.

Previous work on whitespace correction uses autoregressive models to determine the most likely segmentation of a given text into words. The used models are n-gram models, character-level recurrent neural network language models or character-level neural machine translation (NMT) models. Mikša et al. (2010) use a beam search with an n-gram language model to split unknown tokens in OCR'ed Croatian text into multiple words. Nastase and Hitschler (2018) use a character-level GRU NMT model to segment texts from the ACL anthology corpus. Soni et al. (2019) use n-gram statistics to split tokens in digitized historic English newspapers. Doval and Gómez-Rodríguez (2019) use a beam search with a character LSTM language model or n-gram language model for English word segmentation. Bast et al. (2021) use a beam search with a combination of a unidirectional character LSTM language model and a bidirectional LSTM whitespace classification model and introduce penalty terms to make use of existing spaces in the input text.

In contrast to previous work, our best approach does not use an autoregressive model, but instead addresses the task with a sequence-labeling classification model. We make use of the Transformer architecture, which improved the state of the art in many NLP tasks, including machine translation (Vaswani et al., 2017), language modeling (Radford et al., 2019), language understanding (Devlin et al., 2019), and Chinese word segmentation (Huang et al., 2020). We compare our approach with the previous best results from Bast et al. (2021) and other baselines.

3 Approach

We compare two approaches for the whitespace correction problem: A character-level encoder-decoder (ED) model and a byte-level encoder-only model (EO). Both models respect existing whitespace information in the input text. We pre-process the input text by removing duplicate, leading, and trailing whitespaces, and applying NFKC normalization[2].

3.1 Encoder-only (EO)

The EO approach treats the whitespace correction problem as a sequence-labeling task where we predict one of three repair tokens $\mathcal{R} = \{K, I, D\}$ for each character x_i in the input sequence

[1]For live as-you-type whitespace correction we limit the input size to 512 characters, because we simply correct the full input after every keystroke. Non-live correction supports larger inputs such as text files in the order of several MB.

[2]This is a form of Unicode normalization. It is only relevant for our EO models with byte input, see Section 3.1

$K \rightarrow$ Keep the character

$I \rightarrow$ Insert a space before the character

$D \rightarrow$ Delete the character

and use them afterwards to change the whitespacing in the input sequence accordingly.

For the EO approach, we directly input and embed UTF-8 encoded bytes instead of characters. We found the performance of using bytes directly to be on par with using characters, while introducing only a negligible runtime overhead. It also enables us to process any text without a character vocabulary or a special token for unknown characters. We also add sinusoidal positional encodings as defined by Vaswani et al. (2017) to the byte embeddings. To keep inference speeds high we aggregate all byte embeddings belonging to a character[3] into a single embedding **before** processing them further[4]: Within a grapheme cluster we first average the byte embeddings belonging to individual code points, then average the code point embeddings to get the final character embedding.

To process the character embeddings, we employ a standard Transformer encoder (Vaswani et al., 2017) with a linear output layer on top, which allows us to predict a probability distribution over the repair tokens $\mathcal{R}$ for each character in parallel. During inference we simply take the repair token with the highest probability for each character as output. However, we only allow the model to predict I between two non-space characters or D for space characters, otherwise we ignore the prediction and fall back to K:

$$y_i = \text{argmax}_{r \in \mathcal{R}}\, p(r \mid x, i) \text{ with } 1 \leq i \leq n,$$

$$y_i \leftarrow \begin{cases} K & \text{if } y_i = D \text{ and } x_i \neq \text{' '} \\ K & \text{if } y_i = I \text{ and } (x_i = \text{' '} \text{ or } x_{i-1} = \text{' '}) \\ y_i & \text{else.} \end{cases}$$

3.2 Encoder-decoder (ED)

For the ED approach, we tokenize the input text into a sequence of characters $x = (x_1, \ldots, x_n)$ with $x_i \in \mathcal{C}$. $\mathcal{C}$ is a character vocabulary containing all uppercase and lowercase letters of the English alphabet, common punctuation marks, and special tokens, e.g., for unknown characters.

[3]In accordance with the Unicode standard, we determine all characters in a UTF-8 encoded byte sequence using grapheme cluster boundaries.

[4]For languages using non-Latin alphabets, e.g. Russian, where characters are usually encoded into multiple bytes in UTF-8 this can make a large difference.

The ED approach uses an encoder-decoder Transformer model (Vaswani et al., 2017) with a linear output layer trained to translate sequences of characters with space errors into sequences without space errors by outputting characters one by one. At each output step t, we use the ED decoder to predict a probability distribution over $\mathcal{C}$ given the input sequence and the previous outputs $y_{<t} = (y_1, \ldots, y_{t-1})$. To ensure that the ED model only changes the whitespaces of a sequence during inference, we limit the set of possible outputs at each step to the space character or the next character to copy from the input sequence x_j:

$$y_t = \text{argmax}_{c \in \{\text{' '}, x_j\}}\, p(c \mid x, y_{<t}).$$

Sliding window Both the EO and ED approaches are trained with and limited to input sequences containing up to 512 tokens. Since real-world paragraphs often exceed this length bound, we use a sliding window approach during inference: We split input sequences into windows of 384 tokens and add the 64 tokens to the left and right of the window as additional context.[5] For a given sequence we run our model on each individual window separately and recombine the whitespace correction results of all windows afterwards. For example, a 950 byte long sequence would be split into three consecutive windows $w_1 = (0, 448, 64), w_2 = (64, 384, 64)$, and $w_3 = (64, 118, 0)$, specifying the sizes of the left context, the window itself, and the right context respectively. Note that at the beginning and end of the sequence, the left and right context sizes are 0, each window contains a maximum of 512 bytes, and all non-context sizes add up to 950.

We train a medium-sized and large model for each of the two approaches. All models use a hidden dimensionality of 512 and only differ in the number of encoder or decoder layers. See Table 1 for an overview over all models.

3.3 Training

We train our models on the publicly available training dataset from Bast et al. (2021).[6] This dataset consists of 108,068,848 paragraphs extracted from

[5]We experimented with 16, 32, 64, and 128 as context size, but found it to have very little effect on correction quality. To be on the safe side we finally chose 64 as context size leaving us with 384 tokens for the actual window. Also, because of missing left and right context, the windows can contain more than 384 tokens at the beginning and end of sequences.

[6]At https://whitespace-correction.cs.uni-freiburg.de

Model	#Parameters	#Layers
ED_{medium}	22.2 M	3 encoder, 3 decoder
ED_{large}	44.2 M	6 encoder, 6 decoder
EO_{medium}	19.0 M	6 encoder
EO_{large}	38.0 M	12 encoder

Table 1: We choose the number of layers such that the corresponding ED and EO models have comparable numbers of parameters.

arXiv and Wikipedia articles. To generate pairs of correctly and misspelled sequences, the authors inject OCR and spelling errors artificially into the paragraphs using error models derived from text corpora, typo collections, and random character transformations. Additionally, we inject space errors into the paragraphs:

- In 10% of the paragraphs we remove all spaces.
- In 10% of the paragraphs we have a space between each pair of adjacent characters.
- In the remaining 80% of the paragraphs we insert a space between two adjacent non-space characters with probability 10% and delete an existing space with probability 20%.

The EO approach naturally suffers from an unbalanced class distribution, causing our models to reach a plateau during early stages of training where they predict *K* for all characters. To counteract that, we use a focal loss (Lin et al., 2017) with $\gamma = 2$. This causes our models to overcome the plateau by decreasing the influence of the dominant and (mostly) easy-to-predict *K* class. We also tried using a regular cross-entropy loss and weighing classes *I* and *D* higher, but found it to perform worse while also having one more hyperparameter. For the full training details see Appendix A.

4 Experiments

4.1 Benchmarks

We evaluate on a total of eight benchmarks (see Table 2 for an overview), with the first six coming from Bast et al. (2021):

- Three benchmarks with text from Wikipedia, one with whitespace errors only, one with whitespace and spelling errors, and one without spaces but with spelling errors. These benchmarks are called **Wiki**, **Wiki+**, and **Wiki+ no␣** respectively.
- Two benchmarks based on text from arXiv with OCR errors and errors from PDF extraction, called **arXiv OCR** and **arXiv pdftotext**.

Benchmark	#Sequences	File size
Wiki	10,000	916 kB
Wiki+	10,000	916 kB
Wiki+no␣	10,000	778 kB
arXiv OCR	10,000	1.4 MB
arXiv pdftotext	10,000	1.4 MB
ACL	500	83 kB
Doval	1000	82 kB
Runtime	3,500	396 kB

Table 2: A benchmark simply consists of two text files. The first file contains the corrupted input text where each line corresponds to a sequence with whitespace and spelling errors. Its size is shown in the file size column. The second text file then specifies the groundtruth sequences without whitespace errors accordingly.

- One benchmark based on sequences from the ACL anthology dataset containing OCR errors. This benchmark is called **ACL**.
- The word segmentation benchmark **Doval** from Doval and Gómez-Rodríguez (2019), with 1,000 sequences without any whitespace.
- A **Runtime** benchmark for measuring the inference runtimes of our models and baselines, built by randomly sampling 500 sequences from each of the seven benchmarks above.

4.2 Baselines

We reuse the following four baselines and their predictions from Bast et al. (2021):

- **Do nothing** This baseline keeps the input sequence unchanged. It is an interesting reference point for benchmarks with very few errors, like arXiv pdftotext.
- **Google** The authors copied erroneous sequences into a Google document[7] and applied all suggested space edits.
- **Wordsegment** Wordsegment is a Python package for word segmentation.[8] Before applying it to the text, all whitespaces are removed.
- **BID+** The best whitespace correction procedure from Bast et al. (2021), with the overall best hyperparameters (called *The One* in that paper). They perform a beam search with a combination of a unidirectional character-level LSTM language model and a bidirectional LSTM classification model to score whitespace insertions or deletions.

[7] At https://docs.google.com
[8] At https://github.com/grantjenks/python-wordsegment

In addition, we introduce an EO-like baseline called **ByT5**. Xue et al. (2022) released ByT5, a family of encoder-decoder Transformer models that input and output sequences of bytes and were pretrained on a masked token prediction task. We take the encoder of their smallest model (~217M parameters), add our byte-to-character aggregation scheme (see Section 3.1) and a linear output layer, and finetune it on 50M sequences from our training data for one epoch.

To our knowledge, the ByT5 models by Xue et al. (2022) are currently the only publicly available general purpose language models that work on character or byte level. Other openly accessible pretrained Transformer language models like the BERT (Devlin et al., 2019), T5 (Raffel et al., 2020), or OPT (Zhang et al., 2022) families are unsuitable for whitespace correction, because they all work with subword tokenization. Also, some of these are simply too large to permit a reasonable runtime and memory consumption.

4.3 Metric

Given two strings a and b that only differ in whitespaces, we define a function $\text{correction-ops}(a, b)$ that gives us the set of correction operations that we need to apply to turn a into b. A correction operation is a tuple $\langle r, i \rangle$ consisting of an insert or delete operation $r \in \{I, D\}$ and the character position $i, 1 \leq i \leq |a|$ at which the operation has to be applied. Given a benchmark sample $\langle s, g, p \rangle$ as a tuple of an input sequence s, ground truth sequence g, and predicted sequence p, we define a F_1-score as follows:

$$
\begin{aligned}
\mathcal{G} &= \text{correction-ops}(s, g) \\
\mathcal{P} &= \text{correction-ops}(s, p) \\
\text{TP} &= |\mathcal{G} \cap \mathcal{P}| \\
\text{FP} &= |\mathcal{P} \setminus \mathcal{G}| \\
\text{FN} &= |\mathcal{G} \setminus \text{P}| \\
\text{F}_1 &= \begin{cases} 1 & \text{if } |\mathcal{G}| = |\mathcal{P}| = 0 \\ \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}} & \text{else} \end{cases}
\end{aligned}
$$

For our evaluation metric we calculate the average F_1-score over all benchmark samples. With this metric the Do nothing baseline gives the percentage of benchmark samples without space errors.

4.4 Results

Quality Table 3 shows the sequence-averaged F_1-scores achieved by the baselines and our models on the 7 whitespace correction benchmarks.[9] Compared to the neural models, Google and Wordsegment perform poorly, sometimes even worse than the Do nothing baseline.[10] Our best models, ED_{large} and EO_{large}, perform on par with the best-so-far model BID+. In general, all four of our models achieve a high quality both in absolute terms as well as compared to the other baselines. We note, that EO_{large} performs slightly better than ED_{large} on 6 out of 7 benchmarks. A similar picture holds for the medium-sized variants. Comparing BID+ and EO_{large}, we see that the differences in F_1-score between them are rather small ($\leq 0.6\%$) across all benchmarks. ByT5 achieves good quality overall, but falls behind both of our large and on some benchmarks even medium-sized models. We hypothesize that this is mainly due to it being pretrained on text without spelling and whitespace errors, which makes it hard to transfer its language modeling capabilities to this vastly different input distribution. Increasing the model size consistently leads to F_1-score improvements across all benchmarks, both for the EO and the ED approach. See Appendix C for a showcase of some of the predictions and failure cases of our models on the benchmarks, and Appendix D for a visualization of attention maps.

Runtimes Table 4 shows runtime and GPU memory consumption on our Runtime benchmark (see Section 4.1). Wordsegment and BID+ use a batch size of 1 and cannot be easily modified to support larger batch sizes[11]. We also show the performance of all other models for batch size 1. That way, we can see which improvements come from the approach, and which from an increased batch size. Appendix B provides results for more batch sizes for ByT5, ED, and EO.

As expected, the encoder-only models EO and ByT5 outperform BID+ and ED by a large margin,

[9] We additionally evaluated ChatGPT (gpt-3.5-turbo, available at https://chat.openai.com) on 100 random sequences from our benchmarks via the OpenAI API. ChatGPT usually changed more about the sequence than just the whitespace errors. Apart from that, we found the corrections to be of high quality, but the runtime to be very slow. ChatGPT took 668 seconds to correct the 100 sequences, which equals a throughput of 0.015 kB/s and is over 1,000 times slower than EO_{large} with batch size 1.

[10] For example, arXiv pdftotext contains only few errors, and it is hard to make few fixes at the right places and not introduce new errors.

[11] In particular, BID+'s complicated decoding scheme including beam search caused its authors to implement inference only in an unbatched fashion.

Model	ACL	Wiki+ no␣	Wiki+	Wiki	arXiv OCR	arXiv pdftotext	Doval
Do nothing	62.0	4.1	86.9	35.0	62.0	64.3	0.8
Google	75.6	16.7	91.9	76.0	83.9	86.1	-
Wordsegment	47.4	85.9	35.9	63.8	66.8	62.5	-
ByT5	87.1	98.7	95.7	98.2	96.5	94.8	99.4
BID+	87.8	99.0	**98.0**	98.8	**97.6**	95.5	99.9
ED_{medium}	86.0	98.9	96.5	98.5	96.7	95.4	99.5
ED_{large}	**87.8**	99.1	97.2	98.9	97.1	95.9	99.8
EO_{medium}	86.6	99.2	96.9	98.8	97.2	95.8	99.7
EO_{large}	87.5	**99.3**	97.6	**99.0**	97.3	**96.1**	**99.9**

Table 3: Sequence-averaged F_1-scores for our models and baselines on the whitespace correction benchmarks from Section 4.1. We show the best result for each benchmark in bold.

due to being able to correct each character in the input text simultaneously instead of one after another. They are even significantly faster than the (non-neural) Wordsegment baseline. Even with a batch size of 1, EO_{large} is over 75 times faster than the previous best model BID+. With a batch size of 128, EO_{large} achieves 213 kB/s, which is over 900 times faster than BID+.

Memory All of our models require relatively little memory and can easily be run on a standard GPU with reasonable batch sizes. We carried out additional tests with our EO_{large} model on a NVIDIA GeForce GTX 1080 Ti GPU, where we reached batch sizes of 305 and 399 for full precision and mixed precision inference, respectively, before getting OOM errors. We also determine a rough estimate of the amount of GPU memory required per sequence/batch element in Table 4. One can use these values to approximate the amount of GPU memory required for running a model with a certain batch size.

To investigate how performance scales with model size, we additionally trained another EO model called EO_{larger}, which has 18 layers and ~3.3 times more parameters than EO_{large}. EO_{larger} improves upon EO_{large} on every benchmark, but only by small margins with an average gain of 0.15 percentage points in sequence-averaged F_1-score. Compared with EO_{large}, it also requires about three times more memory and runs only at 60% of its speed. Here, the tradeoff between quality improvement, speed loss, and increase in memory consumption seems less favorable than the one between EO_{large} and EO_{medium}. We leave it for future work to investigate how much performance can improve further by using even larger models.

5 Demo

We provide open access to all of the EO and ED models in form of a Python package which can be installed via pip[12] or from source. The package comes with a command line tool and a Python API both of which enable the user to correct whitespace errors in arbitrary text. Additionally, the command line tool provides an option for running a whitespace correction JSON API and thereby enables access to our models not only from Python code but also from other programming and development environments.

In addition to the Python package, we also provide a web interface at https://whitespace-correction.cs.uni-freiburg.de that allows users to correct whitespaces in arbitrary text; see Figure 1. In particular, the web app allows to evaluate the output against an error-free ground truth and provides quick access to all of the used benchmarks. Therefore, the easiest way to reproduce the results of our models on all benchmarks as presented in this paper is through the web interface.

For more information visit our GitHub repositories at https://github.com/ad-freiburg/whitespace-correction and https://github.com/ad-freiburg/text-correction-benchmarks.

6 Conclusion

We have shown that carefully trained encoder-only Transformers perform whitespace correction with

[12] At https://pypi.org/project/whitespace-correction

Model	Batch size	Runtime seconds	Throughput sequences / sec	Throughput kB / sec	GPU memory MiB	GPU memory MiB / sequence
Wordsegment	1	246.1	14.22	1.6	-	-
ByT5	1	31.9	109.6	12.7	974	~51
BID+	1	1,725	2.0	0.2	-	-
ED_{medium}	1	1,402	2.5	0.3	144	~36
ED_{large}	1	2,432	1.4	0.2	234	~36
EO_{medium}	1	13.8	253.9	29.4	118	~28
EO_{large}	1	22.5	155.4	18.0	184	~28
ByT5	128	4.2	833.6	96.4	7,402	~51
ED_{large}	128	151.8	23.1	2.7	4,832	~36
EO_{large}	128	1.9	1,842	213.0	3,704	~28

Table 4: Inference runtimes on our Runtime benchmark (see Section 4.1) on a NVIDIA A100 GPU and an Intel Xeon Platinum 8358 CPU with mixed precision enabled. Wordsegment and BID+ only support a batch size of 1. Results for more batch sizes for ByT5, ED, and EO can be found in Appendix B.

the same high quality as the best previous work, but over 900 times faster. A classical encoder-decoder Transformer can achieve the same quality, but with little to no gains in runtime speed. Our software is open source and accessible as a Python package as well as via a dedicated website.

A logical next step in this line of work is to combine whitespace correction with spelling correction. Recent large language models like GPT-3 (Brown et al., 2020) can solve both tasks at once with high quality, but they are slow and expensive to run due to their size and autoregressive decoder architecture. By separating the tasks, as advocated by Bast et al. (2021), each can be solved efficiently with specialized models, like our EO models for whitespace correction. It remains an interesting open question whether a single model can achieve both high quality and reasonably cheap inference.

Due to working directly with bytes, our EO models could be trained and applied across multiple languages. However, because datasets and benchmarks for whitespace correction in non-English languages are yet to be created, we leave the development of multilingual models for future work.

Author contributions

Most of the work behind this paper was done by S.W., with M.H. and H.B. acting as advisers. In particular, S.W. did all of the implementation work and the evaluation. All authors wrote the paper, with S.W. doing the largest part.

Acknowledgement

The authors acknowledge support by the state of Baden-Württemberg through bwHPC. M.H. is funded by the Helmholtz Association's Initiative and Networking Fund through Helmholtz AI.

Ethics Statement

Our models are trained on a mix of Wikipedia and arXiv articles, so they are naturally exposed to the biases represented in that data. However, since the models are restricted to only changing the whitespacing in text either by design (for the EO models) or by the decoding procedure (for the ED models), we consider the ethical concerns of using our models to be small.

References

Hannah Bast, Matthias Hertel, and Mostafa M. Mohamed. 2021. Tokenization repair in the presence of spelling errors. In *Conference on Computational Natural Language Learning (CoNLL '21)*, pages 279–289.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies (NAACL-HLT '19), pages 4171–4186.

Eva D'hondt, Cyril Grouin, and Brigitte Grau. 2017. Generating a training corpus for OCR post-correction using encoder-decoder model. In *International Joint Conference on Natural Language Processing (IJCNLP '17)*, pages 1006–1014.

Yerai Doval and Carlos Gómez-Rodríguez. 2019. Comparing neural- and N-gram-based language models for word segmentation. *Journal of the Association for Information Science and Technology*, 70(2):187–197.

Mika Hämäläinen and Simon Hengchen. 2019. From the paft to the fiiture: A fully automatic NMT and word embeddings method for OCR post-correction. In *International Conference on Recent Advances in Natural Language Processing (RANLP '19)*, pages 431–436.

Weipeng Huang, Xingyi Cheng, Kunlong Chen, Taifeng Wang, and Wei Chu. 2020. Towards fast and accurate neural chinese word segmentation with multi-criteria learning. In *International Conference on Computational Linguistics (COLING '20)*, pages 2062–2072.

Sai Muralidhar Jayanthi, Danish Pruthi, and Graham Neubig. 2020. NeuSpell: A neural spelling correction toolkit. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP Demos '20)*, pages 158–164.

Ido Kissos and Nachum Dershowitz. 2016. OCR error correction using character correction and feature-based word classification. In *International Workshop on Document Analysis Systems (DAS '16)*, pages 198–203.

Hao Li, Yang Wang, Xinyu Liu, Zhichao Sheng, and Si Wei. 2018. Spelling error correction using a nested RNN model and pseudo training data. *CoRR*, abs/1811.00238.

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR '19)*.

Mladen Mikša, Jan Šnajder, and Bojana Dalbelo Bašic. 2010. Correcting word merge errors in croatian texts. *International Conference on Formal Approaches to South Slavic and Balkan Languages (FASSBL '10)*.

Vivi Nastase and Julian Hitschler. 2018. Correction of OCR word segmentation errors in articles from the ACL collection through neural machine translation methods. In *International Conference on Language Resources and Evaluation (LREC '18)*.

Thi-Tuyet-Hai Nguyen, Adam Jatowt, Nhu-Van Nguyen, Mickaël Coustaty, and Antoine Doucet. 2020. Neural machine translation with BERT for post-OCR error detection and correction. In *Joint Conference on Digital Libraries (JCDL '20)*, pages 333–336.

Danish Pruthi, Bhuwan Dhingra, and Zachary C. Lipton. 2019. Combating adversarial misspellings with robust word recognition. In *Conference of the Association for Computational Linguistics (ACL '19)*, pages 5582–5591.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Keisuke Sakaguchi, Kevin Duh, Matt Post, and Benjamin Van Durme. 2017. Robsut wrod reocginiton via semi-character recurrent neural network. In *Conference on Artificial Intelligence (AAAI '17)*, pages 3281–3287.

Sarah Schulz and Jonas Kuhn. 2017. Multi-modular domain-tailored OCR post-correction. In *Conference on Empirical Methods in Natural Language Processing (EMNLP '17)*, pages 2716–2726.

Sandeep Soni, Lauren F. Klein, and Jacob Eisenstein. 2019. Correcting whitespace errors in digitized historical texts. In *Workshop on Computational Linguistics for Cultural Heritage, Social Sciences, Humanities and Literature (LaTeCH '19)*, pages 98–103.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS '17)*, pages 5998–6008.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. 2022. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. Opt: Open pre-trained transformer language models.

A Training setup

We train all our medium and large sized models for three epochs over the full training data with a maximum sequence length of 512 and up to 65,536 tokens per batch. This amounts to about 1.2M training steps. We warmup the learning rate linearly to 10^{-4} over the first percent of training steps and decay it afterwards towards zero using a cosine schedule. We use AdamW (Loshchilov and Hutter, 2019) with $\beta_1 = 0.9$, $\beta_2 = 0.999$, a weight decay of 0.01, and clip gradients to a norm of 1. We also use a dropout rate of 0.1 throughout our models. Finally, we keep the model checkpoint corresponding to the lowest validation loss as our final model. Training takes about 4-5 days for all of our models on a single NVIDIA V100/A100 GPU.

B Batched runtimes

Model	Batch size	Runtime seconds	Throughput sequences / sec	Throughput kB / sec	GPU memory MiB	GPU memory MiB / sequence
ByT5	16	5.0	696.2	80.5	1,732	~51
	128	4.2	833.6	96.4	7,402	
ED_{medium}	16	174.4	20.1	2.3	686	~36
	128	106.9	32.7	3.8	4,744	
ED_{large}	16	264.03	13.2	1.5	774	~36
	128	151.8	23.1	2.7	4,832	
EO_{medium}	16	2.2	1,596	184.5	550	~28
	128	1.6	2,143	247.7	3,618	
EO_{large}	16	2.7	1,277	147.7	620	~28
	128	1.9	1,842	213.0	3,704	

The table above shows runtimes using batched inference on our Runtime benchmark (see Section 4.1) on a NVIDIA A100 GPU and an Intel Xeon Platinum 8358 CPU with mixed precision enabled. For each model we report both batch size 16 and 128. We only show the models from Table 4 that support batched inference in their implementation. The EO models reach a throughput of well over 200 kB/s with a batch size of 128 while requiring less than 4GB of GPU memory. We consider them to be fast and memory efficient enough to be used in practical applications, even on less powerful end-user devices like laptops.

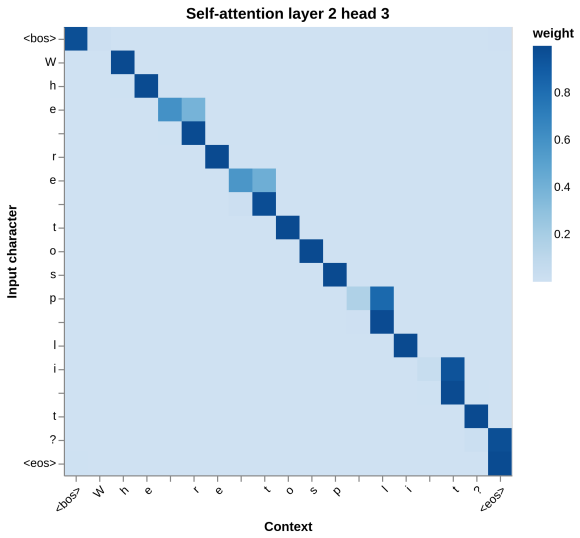
C Sample predictions and failure cases

We present some predictions and failure cases of our EO_{large} model on selected benchmark sequences in the following. In accordance with our website at https://whitespace-correction.cs.uni-freiburg.de we mark correct changes (true positives) in the input text with green, incorrect changes (false positives) with red, and missing changes (false negatives) orange. See our website if you want to evaluate and visualize our models' predictions on your own texts. Marked whitespaces are shown as # for visualization purposes.

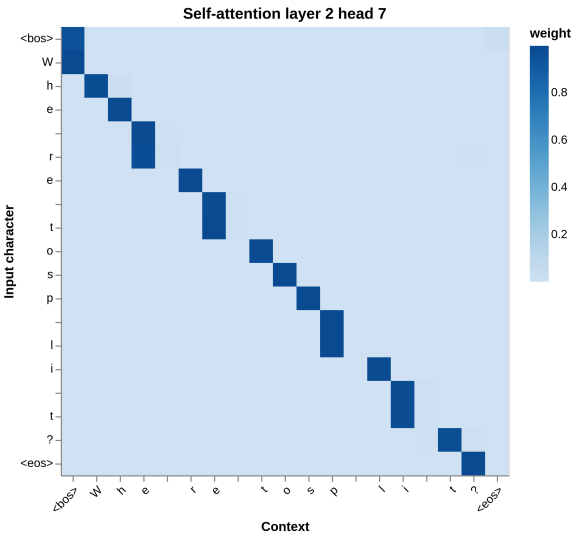
Input	Prediction	Comment
UnlikeinCanada,theAmericanStatesareresponsiblfortheorganisationoffederalelectionsintheUnitedStates.	Unlike#in#Canada,#the#American#States#are#responsible#for#the#organisation#of#federal#elections#in#the#United#States.	For sentences that contain no spelling errors and no exotic words, our model is almost always able to perfectly correct the sequence, even if it contains no whitespaces at all.
The Exit Players have trained with members of Improvised Shakespeare, Paralellogramophonograph, andi Baby Wants Candy, os well as instructors from the Groundlings, the PIT, iO, and Coldtowne Theater.	The Exit Players have trained with members of Improvised Shakespeare, Paralellogramo#phonograph, andi Baby Wants Candy, os well as instructors from the Groundlings, the PIT, iO, and Coldtowne Theater.	Complex composite words or proper names are sometimes split or merged by our model. To correctly identify them one requires either domain knowledge or very good language understanding.
He has also played league chess in the Chess Bundesliga, for Porz an Werder Bremen.	He has also played league chess in the Chess Bundesliga, for Porzan Werder Bremen.	Similar to the sample above. Our model predicts that *Porzan Werder Bremen* is what the chess club is called, but actually this should have been *Porz and Werder Bremen*, where Porz is a German city near Cologne.
Brian Catling (born 1948 in London) is an English sculptor, poet, novelist, film maker and perofmrance artist.	Brian Catling (born 1948 in London) is an English sculptor, poet, novelist, filmmaker and perofmrance artist.	Here our model merges the words *film* and *maker*, which according to the Cambridge dictionary is also a valid English word. A reminder that not all benchmark ground truths are unambiguous.
X stems from Y X eases Y *Y results in X Y is related to X *X is result of Y X is linked to Y	X stems from Y X eases Y *Y results in X Y is related to X *X is result of Y X is linked to Y	Here we would expect our model to either insert a whitespace before both Y and X, or remove the whitespace before * in both cases. Instead it keeps the sequence unchanged.
Ju ly 1 ducat ion Programs Beginning after J a n ~ a r y 1, 1976 Roger R o s e ~ b l a t t , Divi-sion Director -202-382-5891 Procjrhm grants for c r i t i c a l re-examination of t h e content, o r g a h i z a t i o n , and method of presenta t ion of a group of related courses or an ordered program of study in the humanities. The central topic can be a region, culture, era, etc.; o r a program can be defined by a cur r -i cu la r level. L i m i t , $ 1 8 0 , 0 0 0 i n three years.	July 1 ducation Programs Beginning after Ja#n#~#a#ry 1, 1976 Roger Rose#~#b#l#a#t#t, Divi-sion Director -202-382-5891 Procjrhm grants for critical re-examination of the content, orgahization, and method of presentation of a group of related courses or an ordered program of study in the humanities. The central topic can be a region, culture, era, etc.; or a program can be defined by a curr-icular level. Limit, $180,000 in three years.	Exotic characters within words which are not often seen during training can cause or model to not be able to correctly merge the full word. Here the model missed to correctly predict the words *Jan~ary* and *Rose~blatt*.

D Attention visualization

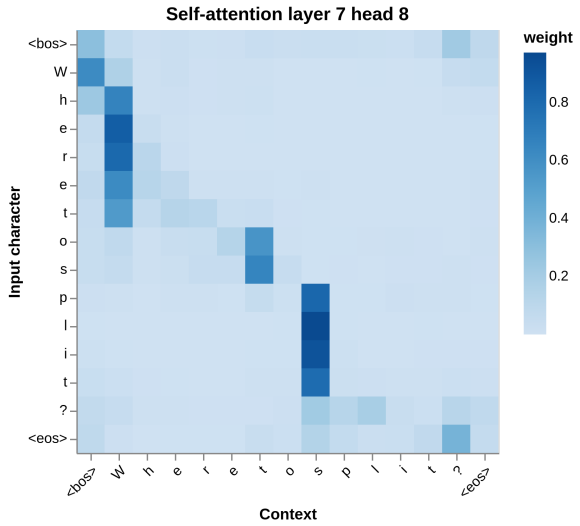
We look at the self-attention maps produced during the forward pass of our EO models. In earlier layers we mostly find local or character-specific attention patterns, while in the middle and later layers our models ultimately seem to learn to identify word boundaries.
All of the following self-attention maps are normalized row-wise, meaning each row displays the attention distribution for the input character on its left over all input characters, which we call context in the figures.



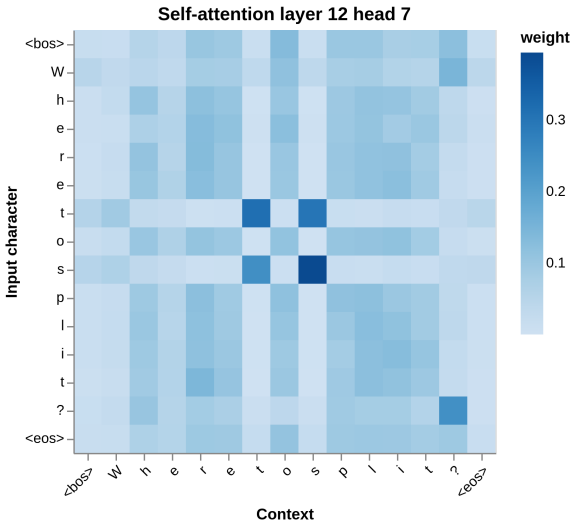
(a) Attention head in layer 2/12 looking at the following non-whitespace character.
Input text is *Whe re tosp li t?*.



(b) Attention head in layer 2/12 looking at the previous non-whitespace character.
Input text is *Whe re tosp li t?*.



(c) Attention head in layer 7/12 identifying the individual words in the input text (indicated by the vertical bars).
Input text is *Wheretosplit?*.



(d) Attention head in layer 12/12 marking where whitespaces should be inserted into the input text.
Input text is *Wheretosplit?*.

Exemplary attention maps of selected heads produced in early, middle, and late layers while running our EO_{large} model. Input texts are deliberately chosen make the attention patterns as clear as possible.