Abstract

We present EdiT5\textsuperscript{1} – a novel semi-autoregressive text-editing model designed to combine the strengths of non-autoregressive text-editing and autoregressive decoding. EdiT5 is faster during inference than conventional sequence-to-sequence (seq2seq) models, while being capable of modeling flexible input-output transformations.

This is achieved by decomposing the generation process into three sub-tasks: (1) tagging to decide on the subset of input tokens to be preserved in the output, (2) re-ordering to define their order in the output text, and (3) insertion to infill the missing tokens that are not present in the input. The tagging and re-ordering steps, which are responsible for generating the largest portion of the output, are non-autoregressive, while the insertion step uses an autoregressive decoder.

Depending on the task, EdiT5 on average requires significantly fewer autoregressive steps, demonstrating speedups of up to 25x when compared to seq2seq models. Quality-wise, EdiT5 is initialized with a pre-trained T5 checkpoint yielding comparable performance to T5 in high-resource settings when evaluated on three NLG tasks: Sentence Fusion, Grammatical Error Correction, and Decontextualization while clearly outperforming T5 in low-resource settings.

1 Introduction

Pre-trained seq2seq models such as T5 (Raffel et al., 2020), BART (Lewis et al., 2020a), and MASS (Song et al., 2019) have established strong baselines for the majority of text-to-text translation tasks. A recent trend to massively scale up model sizes, e.g., all the way up to 540B params (Chowdhery et al., 2022), as well as the sizes of pretraining corpora, has further pushed the state-of-the-art without signs of reaching a plateau. From a practical point of view, running inference with such models is prohibitively expensive for most applications, which motivates the work on finding efficient recipes for model distillation, e.g., (Kim and Rush, 2016) and choosing a model architecture that can provide a better trade-off between performance on a given task and inference speed. A typical choice is to distill a large language model into a smaller seq2seq model, e.g., Transformer (Vaswani et al., 2017). In this paper we propose a novel model architecture EdiT5 which blends ideas from a seq2seq T5 (Raffel et al., 2020) and text-editing to provide faster inference without sacrificing on task performance.

Seq2seq-based models output text token-by-

\textsuperscript{1}Code and pre-trained models https://edit5.page.link/code
token from scratch, allowing them to model any kind of input-output relationship. However, for many real-world tasks this degree of generality is unnecessary, especially for monolingual tasks where the input and output texts have relatively high degrees of overlap. In such cases a natural approach is to cast conditional text generation as a text-editing task, where the model learns to construct target texts by applying a set of edit operations to the inputs (Malmi et al., 2022). Typically the set of edit operations is defined ahead of time (Omelianchuk et al., 2020; Malmi et al., 2019; Awasthi et al., 2019), which on the one hand limits the flexibility of the model to reconstruct arbitrary output texts from the inputs, but on the other, leads to latency improvements as the limited set of allowed operations significantly reduces the output vocabulary of the decoder. In this paper, we propose an approach which is both fast at inference time and flexible, able to model arbitrary rewrites.

**Faster inference.** A common method for achieving low latency in serving models is to reduce their size, thus reducing their computational cost. Doing so naively, however, often leads to inferior model quality, and much work has gone into finding better methods for model size reduction, such as distillation (Kim and Rush, 2016).

Regardless of model size, one of the major contributors to the total inference time for seq2seq models is the decoder, which generates the output sequence step-by-step. EDIT5 also relies on an autoregressive decoder, but generates the majority of the output sequence with its tagging and pointing networks, and as such the decoder makes far fewer steps.

**Flexible text-editing.** Recent text-editing approaches, e.g., (Awasthi et al., 2019; Malmi et al., 2019), are not as powerful as general purpose seq2seq approaches when it comes to modeling arbitrary input-output text transductions. EDIT5 supports open-vocabulary generation by relying on an autoregressive decoder. In the extreme case, where there is no overlap between the source and the target texts, it reduces to a vanilla seq2seq model generating the entire output from scratch. However, when the input and output overlap, it can benefit from the tagging and pointer networks to reconstruct the bulk of the output text that is further infilled (refined) by the autoregressive decoder.

**Warm start.** Training a high-precision text generation model typically requires large amounts of high-quality supervised data. Self-supervised techniques based on text in-filling (Rothe et al., 2020a; Lewis et al., 2020b; Raffel et al., 2020) have been shown to provide a crucial advantage over non-pre-trained models especially in low-resource settings. Hence, we design EDIT5 to be able to benefit from already existing pre-trained language models (specifically T5), where the final model is directly fine-tuned on the downstream task.

EDIT5 decomposes the generation task into three steps: *tagging, pointing and insertion* (see Fig. 1). The tagger and pointer networks decide which source tokens to preserve and in which order they should appear in the output, thus allowing for arbitrary word dropping and reordering. The tagger is implemented using a non-autoregressive feedforward network, and pointing is implemented using a novel non-autoregressive pointing mechanism (Vinyals et al., 2015) combined with sinkhorn layers (Mena et al., 2018). The insertion network inserts/infills words which are present in the target sequence but do not appear in the source sequence. The network is implemented using an autoregressive transformer decoder, which attends to the tagged, reordered source sequence. The decoder predicts both the locations of where the token spans should be infilled, as well as the spans themselves.

We evaluate EDIT5 on three distinct text generation tasks: Sentence Fusion, Grammatical Error Correction (GEC), and Decontextualization, comparing to recent text-editing approaches and T5. Each task is unique in the editing operations required and the amount of training data available, which helps to better quantify the value of modeling decisions we have integrated into EDIT5.

Additionally, we explore the impact of training data size and model size on EDIT5. Finally we quantify the latency of EDIT5, providing a detailed analysis and comparison to T5.

2 **Model description**

The model architecture of EDIT5 resembles a vanilla Transformer (Vaswani et al., 2017) composed of an encoder and a decoder. EDIT5 decomposes the generation of a text $y$ from an input $x$ into three parts: predicting a sequence of edit tags $y^t$ (indicating whether a token from the input
should be copied to the output), a permutation of the input tokens $\pi$ (indicating the order that copied tokens should appear in in the output), and a sequence of tokens $y^d$ (indicating additional tokens that should be in the output, and where in the permuted input they should be inserted). $y^t$ and $\pi$ are modeled by the encoder, and $y^d$ by the decoder.

There are multiple ways to choose the triple $(y^t, \pi, y^d)$ for a given $(x, y)$ pair. During dataset creation we choose a single such triple for each training pair (see section 2.1 for details), in which case the probability of $y$ can be expressed as:

$$P(y|x) := \prod_{t=1}^{y^d} P(y^d_t|y^d_{<t}, y^t, \pi, x) \cdot P(\pi|x, y) \cdot P(y^t|x)$$

During inference, we first greedily set $y^t$ to maximize the third term, then $\pi$ to maximize the second term, and finally $y^d$ to maximize the first term. The output text $y$ is realized by applying the tags $y^t$ and permutation $\pi$ to the input sequence $x$ and then inserting the tokens $y^d$.

### 2.1 Text-editing encoder

The EdiT5 encoder consists of three steps: encoding, tagging, and pointing.

**Encoder.** The source sentence $x$ is first encoded using $N$ transformer layers into the hidden representations $h$.

**Tagging.** The tag sequence $y^t$ is constructed as follows: source tokens that must be copied are assigned the KEEP tag, tokens not present in the output are marked by the DELETE tag. Tags are predicted by applying a single transformer layer followed by a classification layer to the output of the encoder $h$, which is trained using cross-entropy:

$$L_{tagging} = -\sum_j^{x} \log P(y^t_j|f_t(h)_j)$$

where $y^t_j$ are the gold tags, $j$ is the index of the source token, and $f_t$ is a transformer layer followed by a classification layer. During inference we use $\text{argmax}$ to determine the tags, whereas during training we use the gold tags. The encoder hidden state is then updated to take these tags into account:

$$h^t_j = f_{te}([h_j; TE(y^t_j)])$$

Where $TE$ is a tag embedding layer, whose output is concatenated to the original hidden representation of the source sequence, before a feed-forward layer $f_{te}$ is applied.

**Pointing.** In many tasks it is helpful for the model to be able to rearrange the kept input tokens. For example, we can grammatically correct the sentence *Who are you?* to *Who are you* purely by reordering tokens from the input. In EdiT5 this is made possible thanks to its pointing mechanism. In contrast, in text editing approaches such as Malmi et al. (2019); Dong et al. (2019), correcting this sentence involves first deleting the words *you are* and then recreating them in the right order.

Given a sequence $x$ and the predicted tags $y^t$, the re-ordering model generates a permutation $\pi$. Our implementation is based on a pointer network (Vinyals et al., 2015), where an attention mechanism points to the next token. We follow Mallinson et al. (2020) which, unlike previous approaches where a decoder state attends over an encoder sequence, applies intra-attention, where source tokens attend to all other source tokens. As such the output of this model is a series of predicted pointings, where each source token predicts the token that comes after it. $\pi$ can easily be constructed by daisy-chaining these predicted pointers together, as seen in Fig. 2. We calculate attention using key-query attention, where we include an additional transformer layer prior to the key network:

$$\alpha_{m,j} = f^q(h^t_m, f^k(h^t)_j)$$

Where $\alpha_{m,j}$ is the unnormalized attention, $f^q$ is the query network, a single feed-forward layer, and $f^k$ is the key network, a transformer layer followed by a single feedforward layer.

Unlike Mallinson et al. (2020), we ensure a valid permutation is formed, i.e. no token is pointed to twice, by using sinkhorn layers (Mena et al., 2018), which normalizes over both the rows and the columns of the intra-pointer attention $\alpha$. Sinkhorn layers are defined as:

$$S^0 = \exp(\alpha)$$
$$S^t = T_c(T_r(S^{t-1}(\alpha)))$$

where $T_{r,m}^j(X) = \frac{X_{j,m}}{\sum_i X_{i,m}}$ is the column normalization operator and $T_{r,m}^j(X) = \frac{X_{j,m}}{\sum_t X_{j,t}}$ is the row normalization operator.
The loss for the pointing network is defined as:

\[
\mathcal{L}_{\text{pointing}} = CE(\pi | S(\alpha))
\]  

(7)

Where CE is the cross-entropy loss. During inference we use argmax to determine \( \pi \).

We use additional positional embeddings to update the hidden states with their new position (offset from 0). For example if Who you are? was reordered into Who are you?, the position information would be updated as \( _0 \text{Who} _2 \text{you} _3 \text{are} \).

\[
h^p_j = (h^f_j + PE(\pi_j))
\]  

(8)

where \( PE \) are learnt absolute positional embeddings (Devlin et al., 2019). These additional positional embeddings are masked out for those source words which do not appear in the target sequence. Finally we apply a transformer encoder layer to \( h^p \) forming the final encoded representation of the sequence \( h^f \). \( h^f \) captures the edits as well as the original sequence \( x \), and the decoder attends to this representation.

**Decoder.** We use a standard transformer decoder, which is tasked with inserting tokens which are in the output sequence but don’t appear within the input sequence. EdiT5 takes advantage of the pre-training of a T5 model, where T5 was pre-trained to infill missing spans. When pre-training T5 uses special tokens \( \langle \text{pos} \_i \rangle \) to indicate where missing spans should be inserted, as demonstrated in Figure 3. EdiT5 re-purposes these special tokens, using them to indicate at which position new tokens should be infilled. I.e. \( \langle \text{pos} \_1 \rangle \), indicates that the tokens should be inserted after the first token. As such the decoder first decodes a special position token and then decodes the inserted tokens which should appear after this token. For example to insert the cat after the first token, the decoder generates: \( \langle \text{pos} \_1 \rangle \) the cat. The decoder is trained with a standard cross-entropy loss:

\[
\mathcal{L}_{\text{insertion}} = -\sum_i |y_i^d| \log P(y_i^d | y_{<i}^d, h^f) 
\]  

(9)

Where \( i \) is the decoder index, and \( h^f \) is the encoder output. The loss for the entire model is defined as the sum of the three individual losses:

\[
\mathcal{L} = \lambda_1 \mathcal{L}_{\text{tagging}} + \lambda_2 \mathcal{L}_{\text{pointing}} + \lambda_3 \mathcal{L}_{\text{insertion}}
\]  

(10)

where \( \lambda_1, \lambda_2 \) and \( \lambda_3 \) are hyper-parameters determining the relative importance of tagging, pointing and insertion losses in the final loss.

**Pre-training.** While we initialize EdiT5 from T5 base, T5 was pre-trained with 12 decoder layers, and for EdiT5 we use a single decoder layer. To account for this change in the decoder layers, we perform additional pre-training. We use a pre-training objective which combines a T5 style span insertion task, with a generic text-editing denoising task, as used in BART (Lewis et al., 2020b). A source sentence is corrupted by dropping, swapping and adding spans (an example can be seen in Figure 3), and we task our model to reconstruct the original sentence. By introducing noise we are able to train the tagger to detect incorrect spans, and the pointer to reorder the sentence. The decoder then behaves like the T5 pre-training objective inserting the content of missing spans. Unlike BART’s pre-training, our approach is computationally cheap, as we do not decode the entire sequence when training, instead just decoding the missing spans.

**Dataset construction.** When constructing the training dataset, there are many possible combinations of \( y^1, \pi \) and \( y^y \) which could produce \( y \). For instance, all source tokens could be deleted and the decoder could then produce all the target tokens.
However to minimize latency, we wish to make the number of inserted tokens (i.e. the number of decoder steps) as small as possible, and maximize the number of kept tokens.

To produce alignments from a target sequence to a source sequence, we iterate left-to-right through characters in the target sequence, trying to find spans of target characters which appear in the sequence of source tokens, as described in Algorithm 1 (see Appendix A). Each source token can only be aligned to a single target span. Those target spans that can’t be aligned are instead inserted after the closest previous aligned source token. In cases where there are multiple possible alignments, e.g. the same token appears multiple times in the source, we align the target character span to produce the longest contiguous span of source tokens aligned with the target, i.e. where source tokens appear one-after-another in the target sequence. To find the longest contiguous span we compare the contiguous overlap between source and target for each possible alignment.

3 Experiments

We evaluate EDIT5 on three distinct text-editing tasks: Sentence Fusion, Grammatical Error Correction, and Decontextualization. In addition to reporting previously published results for each task, we also compare to FELIX (Mallinson et al., 2020), a recent non-autoregressive text-editing model, and a strong pre-trained T5 baseline implemented in the T5X framework (Roberts et al., 2022).

Modeling. For EDIT5 we initialize with a T5 base model with a 12-layer Transformer encoder, and single-layer Transformer decoder. Our code is based on the Tensorflow Model Garden’s (Hongkun Yu and Li, 2020) TF2 version of T5. After initializing with the T5 checkpoint, we further pre-train on the denoising objective (see Section 2.1) using the C4 corpus (Raffel et al., 2020), training for 100k steps.

For all experiments EDIT5 is trained using AdamW (Loshchilov and Hutter, 2019), additionally the learning rate was decayed using the validation set, and exact match is used for checkpoint selection. Tokenization is based on T5’s SentencePiece vocabulary (Kudo and Richardson, 2018), with a vocabulary size of 32k. We, however, modify the vocabulary, removing tokens which have punctuation as a suffix, and replacing them with additional span insertion special token, giving EDIT5 512 span insertion special token. Unless otherwise stated, we use an input sequence length of 128. We performed minimal hyper-parameter selection, which is discussed in the Appendix.

Task Analysis. The chosen tasks cover a diverse set of edit operations and a wide range of dataset sizes, varying from under 11 thousand data points to over 4.5 million. Table 1 provides dataset statistics including: the size, input sequence length, output sequence length for seq2seq models, the output sequence length for EDIT5, and the translation error rate (TER) (Snover et al., 2006) between the source and target sentences. We use TER to highlight unique properties of each task.

From Table 1 we see that for all tasks EDIT5 requires significantly fewer decoder steps than a seq2seq model, which results in significant latency savings. We also see that decontextualization has the longest input and output sequences, where the maximum input length of decontextualization is 512 tokens. Decontextualization has the highest TER, with the major contribution being deletion, which is due to the input sequence consisting of a paragraph, whereas the output is a single sentence. In contrast GEC, has the shortest input and output sequence, with the majority of the dataset consisting of a single input and a single output sentence. GEC has the lowest TER, however it has the highest insertion TER. Sentence fusion consists of two sentences being rewritten into a single sentence, and has a middling TER and sequence lengths. It also has the fewest substitutions.

3.1 Sentence Fusion

Sentence Fusion is the task of fusing independent sentences into a coherent output sentence(s) (Geva et al., 2019). It requires operations such as inferring the appropriate discourse connective, pronominalization, reordering the text to introduce relative clauses, and changing the order of the input sentences.

Data. We use the “balanced Wikipedia” portion of the DiscoFuse dataset (Geva et al., 2019) and also study the impact of training data size by creating four additional smaller subsets of DiscoFuse consisting of: 450,000 (10%), 45,000 (1%), 4,500 (0.1%) and 450 (0.01%) data points.

Setup. Following Geva et al. (2019), we report Exact match, which is the percentage of exactly correctly predicted fusions. In addition to the T5
Table 1: Statistics across tasks: size of the dataset (Size), source length in tokens (Lsrc), target length in tokens (Ltgt), Edit5 insertion tokens (E5-Ins), and TER scores, including number of insertions (Ins), deletions (Del), substitutions (Sub), and shifts (Shift). Token counts are measured using a sentencepiece tokenizer and averaged over the development set.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Lsrc</th>
<th>Ltgt</th>
<th>E5-Ins</th>
<th>TER</th>
<th>Ins</th>
<th>Del</th>
<th>Sub</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence fusion</td>
<td>4.5M</td>
<td>42.5</td>
<td>41.1</td>
<td>5.8</td>
<td>10.92</td>
<td>2.49</td>
<td>0.41</td>
<td>3.75</td>
<td>0.62</td>
</tr>
<tr>
<td>GEC</td>
<td>2.3M</td>
<td>24.3</td>
<td>24.7</td>
<td>4.6</td>
<td>09.72</td>
<td>2.99</td>
<td>0.19</td>
<td>5.05</td>
<td>0.49</td>
</tr>
<tr>
<td>Decontextualization</td>
<td>11K</td>
<td>193.9</td>
<td>49.1</td>
<td>7.2</td>
<td>84.80</td>
<td>0.28</td>
<td>90.64</td>
<td>6.43</td>
<td>2.65</td>
</tr>
</tbody>
</table>

Table 2: Sentence fusion results (Exact Match, lower-cased) under various data conditions, latency (ms), and number of parameters.

<table>
<thead>
<tr>
<th>#Params</th>
<th>100%</th>
<th>10%</th>
<th>1%</th>
<th>0.1%</th>
<th>0.01%</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASERTAGGER</td>
<td>110M</td>
<td>53.80</td>
<td>47.31</td>
<td>38.46</td>
<td>25.74</td>
<td>12.32</td>
</tr>
<tr>
<td>FELIX</td>
<td>220M</td>
<td>61.31</td>
<td>52.85</td>
<td>45.45</td>
<td>36.87</td>
<td>16.96</td>
</tr>
<tr>
<td>Seq2Edits</td>
<td>279M</td>
<td>61.71</td>
<td></td>
<td></td>
<td></td>
<td>1.8</td>
</tr>
<tr>
<td>Edit5</td>
<td>141M</td>
<td>64.95</td>
<td>59.26</td>
<td>52.09</td>
<td>43.83</td>
<td>28.64</td>
</tr>
<tr>
<td>+ pre-training</td>
<td>141M</td>
<td>65.16</td>
<td>59.27</td>
<td>50.39</td>
<td>34.18</td>
<td>1.90</td>
</tr>
<tr>
<td>T5 base</td>
<td>220M</td>
<td>65.52</td>
<td>59.75</td>
<td>50.75</td>
<td>33.84</td>
<td>10.75</td>
</tr>
<tr>
<td>ROBERTA</td>
<td>380M</td>
<td>66.6</td>
<td></td>
<td></td>
<td></td>
<td>0.1%</td>
</tr>
<tr>
<td>AugBERT</td>
<td>157M</td>
<td>65.0</td>
<td></td>
<td></td>
<td></td>
<td>0.01%</td>
</tr>
<tr>
<td>T5 base</td>
<td>220M</td>
<td>65.52</td>
<td>59.75</td>
<td>50.75</td>
<td>33.84</td>
<td>10.75</td>
</tr>
<tr>
<td>ROBERTA</td>
<td>380M</td>
<td>66.6</td>
<td></td>
<td></td>
<td></td>
<td>0.1%</td>
</tr>
<tr>
<td>AugBERT</td>
<td>157M</td>
<td>65.0</td>
<td></td>
<td></td>
<td></td>
<td>0.01%</td>
</tr>
</tbody>
</table>

3.2 Decontextualization

Sentence decontextualization task was introduced by Choi et al. (2021). The goal is to rewrite an input sentence to make it stand-alone without the original context.

Data. We use the train, dev and test data from Choi et al. (2021), where sentences were selected from Wikipedia passages. Human annotators were asked to rewrite them, if possible, to be interpretable and grammatical without the context. We compare against T5 base, T5 xxl, FELIX, and a copy baseline. All models use a sequence length of 512.

Metrics. Following Choi et al. (2021), we report exact match, exact match when a sentence needs to be rewritten and SARI F1 (deletion and addition) on unigrams (Xu et al., 2016).

Analysis. Results in Table 3 show that Edit5 achieves a higher exact match scores, and SARI delete score when compared to T5 base, with a significant drop in latency and using fewer parameters. T5 base achieves significantly higher SARI add, suggesting its better at inserting new tokens, which is unsurprising as Edit5 is primarily focused on copying the source sequence. Both T5 and Edit5 achieve significantly higher numbers than FELIX. Edit5 and T5 base, however, still achieve a significantly lower score than the T5 xxl, which can be explained by the difference in model size.

3.3 Grammatical Error Correction

GEC requires systems to identify and fix grammatical errors in a given input text.
We evaluate on the standard GEC test set BEA (Bryant et al., 2019), and use BEA-DEV for checkpoint selection. For pre-training we use an artificial GEC dataset C4_200M of 200M sentences (Stahlberg and Kumar, 2021). We then fine-tune on cLang-8 (Rothe et al., 2021), a distilled version of the Lang-8 learners corpus (Mizumoto et al., 2011).

**Setup.** We report ERRANT F0.5 scores for BEA. We report additional gT5/gFelix baseline numbers from Rothe et al. (2020b), where T5/Felix models were trained only on cLang-8. For pre-training we sampled 0.2% examples from the training set to use as a development set, and train till convergence as measured on this development set.

We additionally measure the impact that model size has on quality and latency, training T5 and EdiT5 small, base, and large models. To make the latency comparison fairer, we also train single-decoder-layer variants of the T5 models we call T5 Slim. To further ensure a fair latency comparison between EdiT5 and T5 we use the same framework for both models. Additionally, we do not perform EdiT5 specific pre-training.

**Results.** From Table 4, we see that all models outperform their equivalent gT5/gFelix models, which is not surprising as the latter models were trained on less data. A surprising result is that the T5 slim variants achieve comparable scores to the full T5 models while having significantly lower latency. Comparing EdiT5 against T5 models, we see up to ∼1 point differences in F0.5 scores between models of the same size (small/base/large), however EdiT5 produces speed ups between 10x and 25x.

In Figure 4, we study the latency–quality trade-offs of T5, T5 slim, and EdiT5 models. We omit Felix from this analysis, because Felix achieves a significantly lower score. We focus on the 95 percentile latency, as it is often the case that users require that a model returns a result within a fixed latency budget. We see that EdiT5 drops less than 0.25 F0.5 points comparing across model sizes, whilst being significantly faster. Additionally for a given latency budget of 5ms, no full T5 model would fit, and only the T5 slim small would fit, whereas both EdiT5 small and base fit. Comparing EdiT5 base against T5 slim small, we see that EdiT5 scores 3 F0.5 points higher, whilst being faster. For any latency budget under 20ms, EdiT5 is quicker and offer better results than T5 and T5 slim. For latency budgets above 20ms, T5 slim large scores slightly (<0.25 F0.5) higher than EdiT5, and if latency is not a factor then gT5 xxl should be used.

## 4 Latency analysis

The tasks on which EdiT5 outperforms seq2seq models in latency are those that have overlap be-
between sources and targets, but it’s unclear how much overlap is required for EDiT5 to produce latency savings. To answer this question, we split EDiT5 base, T5 base and T5 slim base into components whose latencies we measure separately and compare. Details on how latencies are measured can be found in the Appendix C.

A seq2seq model decomposes into two parts: the encoder (we include the input embedding here, so we refer to this as encoder* below), and the decoder. EDiT5 has both of these parts, but also includes a third part (which we call its overhead), comprising of pointer realization and additional transformer layers. To make our analysis simpler and more task-agnostic, we make two simplifying assumptions. First, we assume the worst-case that no tokens are deleted by EDiT5 and there are no padding tokens in the input², in practice this is not the case, and provides significant latency savings for EDiT5. Second, we assume that decoder latency is linear in the number of decoder steps³. Both of these assumptions benefit the latency of seq2seq models more than EDiT5.

Results. In Table 5 we present latencies of encoder*, worst-case EDiT5 overhead and the per-step latency of a decoder under various input-length conditions. We see the overhead added by EDiT5 even in the worst-case is small.

From these results we can derive a simple rule for when EDiT5 will provide a net latency benefit. Compared to T5 slim base⁴, EDiT5 base must save on average 4 decoder steps with an input length of 128, and 7 steps with an input length of 512.

Finally, collating the results in Table 5 with the number of decoder steps performed by EDiT5 and T5 in Table 1, we see that whereas in T5 the decoder latency dominates the latency of encoder*, in EDiT5 this is no longer the case. For instance for GEC, at 24.7 decoder steps on average required to construct the output, T5 slim spends 3.7x more time in its decoder than in encoder*. EDiT5 however spends less time in its decoder than in encoder*, as such the encoder* is now the latency bottleneck.

<table>
<thead>
<tr>
<th>Component</th>
<th>Len. 128</th>
<th>Len. 512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoder*</td>
<td>0.98</td>
<td>2.65</td>
</tr>
<tr>
<td>Worst-case EDiT5 overhead</td>
<td>0.49</td>
<td>1.16</td>
</tr>
<tr>
<td>1 layer decoder per-step</td>
<td>0.15</td>
<td>0.17</td>
</tr>
<tr>
<td>12 layer decoder per-step</td>
<td>1.26</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Table 5: Mean latencies (in milliseconds, ± 0.01ms) measured for the components of EDiT5 and T5 models for various input lengths. EDiT5 overhead is normally input dependent, but we estimate worst-case latency.

decoder model which has become a general-purpose tool for a variety of sequence-transduction tasks, establishing many new state-of-the-art results (Raffel et al., 2020; Rothe et al., 2021). However, two considerable challenges hindering the productionizing of T5-based models are the high latency caused by autoregressive decoding and the need for having a relatively large number of training examples despite the fact that pre-training makes T5 more sample efficient. Recently, it has been found that the sample efficiency problem can be mitigated by performing in-context few-shot learning, but this typically requires scaling up the model size even further (Brown et al., 2020; Chowdhery et al., 2022), increasing the latency.

To reduce latency, a number of non-autoregressive (NAT) seq2seq methods have been proposed for neural machine translation (Gu et al., 2018, 2019; Du et al., 2021) but a quality gap compared to autoregressive methods still exists. To decrease the gap, it is common to run the NAT methods iteratively, which, however, limits the inference speed advantage over autoregressive methods (Lee et al., 2018). In contrast, we show that for tasks where inputs and outputs overlap, we can maintain an order-of-magnitude speed-up without compromising on the model quality by treating the problem as a text-editing task and producing the output in a single pass.

A number of text-editing models have been proposed as a faster and more sample efficient alternative to seq2seq models like T5 (Awasthi et al., 2019; Malmi et al., 2019; Omelianchuk et al., 2020; Mallinson et al., 2020). Another recently proposed approach to speed up the inference time of Transformer models is called aggressive decoding (Sun et al., 2021; Ge et al., 2022).

Closest to our work, Mallinson et al. (2020) show that adding pointing mechanism for reordering and a separate insertion model allow their text-editing model, FELIX, to produce an arbitrary output in
a flexible manner. 

FELIX is a non-autoregressive model which first predicts the tokens to keep, their order, and the locations at which to insert new tokens. Then it runs a separate model based on a BERT masked language model for inserting new tokens. In contrast, EDT5 employs a single, end-to-end model which has an autoregressive insertion component. This enables more accurate insertions, while keeping the latency low, given that most of the tokens can be copied from the source non-autoregressively. Other text-editing models that employ autoregressive insertion include EditNTS (Dong et al., 2019), the text-normalization model by Zhang et al. (2019), Seq2Edits (Stahlberg and Kumar, 2020), ESC (Chen et al., 2020) and LEWIS (Reid and Zhong, 2021). However, unlike EDT5, these models perform also the edit operation prediction autoregressively, making them potentially slower at inference time.

6 Conclusions

In this paper we have proposed EDT5 a low latency solution to text generation, that achieves comparable or better results, across three distinct tasks, to a strong T5 baseline whilst achieving inference latencies that are up to 25x quicker than the baseline model.

In the future we wish to explore the following ideas: 1) The impact of distillation for EDT5. Distillation has previously been shown to be particularly advantageous to non-autoregressive models. 2) Exploring the impact that quantization has on both latency and quality. 3) Applying EDT5 to additional languages. EDT5 makes no language specific assumptions and we plan to apply it to languages other than English.

Limitations

A limitation of EDT5, and text-editing models in general, is the assumption of overlapping text between the input and output sequences. For instance, in machine translation the overlap between source and target is minimal to none. As such EDT5 would decode the entire target sequence, thus offering no latency saving.

An additional limitation is that all of our experiments were done on English tasks. It is unclear how EDT5’s pointing mechanism would behave with languages which have a less strict word-order, such as Czech.

Finally, we have measured latency only on V4 TPUs, and thus it is unclear how the performance would behave on different graphics cards or on CPUs. As such to determine if EDT5 offers a good trade-off between quality and latency, one must measure latency on the target device.

Acknowledgement

We thank Sebastian Krause, Sascha Rothe, and Hongkun Yu for useful discussions, suggestions and feedback. We also thank Shankar Kumar and Felix Stahlberg for providing feedback on an earlier draft of the paper.

References


Mengyun Chen, Tao Ge, Xingxing Zhang, Furu Wei, and Ming Zhou. 2020. Improving the efficiency of grammatical error correction with erroneous span detection and correction. In Proceedings of the 2020


### A Alignment Algorithm

#### B Training Details

All models were trained on 4x4 or 8x8 TPUs, all Edit5 models completed training (including Edit5 pre-training) in under a day. T5 large pre-training large took 2 days to complete and was done using a 4x4 TPU.

#### B.1 Hyper-Parameters Selection

For T5 we compared the T5 1.0 and T5 1.1 version using the base model on the validation sets and found that T5 1.1 performed better, as such used T5 1.1. For Edit5 we used the BEA dev set, finding that T5 1.0 base performed better than T5 1.1 and selected 1.0 for all experiments.

For T5 we used the recommended fine-tuning settings, including using the adafactor optimizer

**Algorithm 1: Edit5 Alignment**

| Data: source // List of tokens                  |
| Data: target // List of characters             |
| Result: alignments                             |
| buffer ← ∅                                      |
| alignments ← {}                                 |
| i ← 0                                          |
| while i < len(target) do                        |
| max_length ← 0                                  |
| max_index ← 0                                   |
| j ← i + 1                                      |
| while j < len(target) do                        |
| source_index, overlap_length ← contiguous_length(target[i:j], source) |
| if overlap_length > max_length then             |
| max_length ← overlap_length                     |
| max_index ← source_index                        |
| j ← j + 1                                      |
| if max_overlap_length > 0 then                  |
| source[max_index] ← ∅                           |
| alignment ← (i,j,max_index,buffer)              |
| alignments.append(alignment)                   |
| buffer ← ∅                                      |
| i ← i + 1                                      |
| else                                           |
| buffer ← buffer + target[i]                    |
| i ← i + 1                                      |

(Shazeer and Stern, 2018), with a learning rate of 0.001. For Edit5 we used AdamW with default settings and the default learning rate of 3e-4.

**DiscoFuse.** For both Edit5 and T5 we experimented with 3 different batch sizes 128, 256, 1024. For 100% and 10%, there was not a noticeable difference in the DEV set exact match performance, so we chose 1024 as it converged the quickest. For 1% and lower, we found that a batch size of 128 performed the best on the dev set.

**Decontextualization.** For Edit5 we experimented with the batch size 128, 256, 1024 and found that 256 offered the best exact match and used this. We also slightly modified the pre-processing code, bracketing the target sequence with [CLS] and [SEP], which helped the alignment code.

**GEC.** For both Edit5 and T5 we used the T5 recommended number of tokens per batch of: batch size = 512, maximum sequence length = 128. We however note that T5 used the inverse: batch size = 128, maximum sequence length = 512. For T5 and EditT5 we disabled learning rate warmup when fine-tuning on cLang-8. Two additional hyperparameters were set for EditT5, during pre-training on C4_200M, we noted that EditT5 train set performance was lower than T5, as such we disabled
dropout on the additional EdiT5 specific transformer layers. We additionally used the dev set to set the values of lambda for equation 10. We experimented with tagging/pointing λ being 1, 2, 10, or equal to the number of tokens. Where λ equal to the number of tokens produced the best results.

C Latency measurement

To report latency for a model, we run inference on a Cloud TPU V4 chip with batch size 1 and report the time spent in computations on the device. This approach ignores some practical contributors to latency, such as memory transfers between the host and device, but we found it also reduced noise significantly, while focusing on the main performance differences between EdiT5, T5 and T5 slim (the amount of computation they each perform). To further minimize spurious latency differences, both EdiT5 and the baseline models are based on the same T5 implementation, found in TensorFlow Model Garden (Hongkun Yu and Li, 2020).