Copy That! Editing Sequences by Copying Spans

Sheena Panthaplackel,[1]* Miltiadis Allamanis,[2] Marc Brockschmidt[2]
[1] The University of Texas at Austin, Texas, USA
[2] Microsoft Research, Cambridge, UK
spantha@cs.utexas.edu, {miallama, mabrocks}@microsoft.com

Abstract

Neural sequence-to-sequence models are finding increasing use in editing of documents, for example in correcting a text document or repairing source code. In this paper, we argue that common seq2seq models (with a facility to copy single tokens) are *not* a natural fit for such tasks, as they have to explicitly copy each unchanged token. We present an extension of seq2seq models capable of copying entire spans of the input to the output in one step, greatly reducing the number of decisions required during inference. This extension means that there are now many ways of generating the same output, which we handle by deriving a new objective for training and a variation of beam search for inference that explicitly handles this problem. In our experiments on a range of editing tasks of natural language and source code, we show that our new model consistently outperforms simpler baselines.

Introduction

Intelligent systems that *assist* users in achieving their goals have become a focus of recent research. One class of such systems are intelligent editors that identify and correct errors in documents while they are written. Such systems are usually built on the seq2seq (Sutskever, Vinyals, and Le 2014) framework, in which an input sequence (the current state of the document) is first encoded into a vector representation and a decoder then constructs a new sequence from this information. Many applications of the seq2seq framework require the decoder to copy some words in the input. An example is machine translation, in which most words are generated, but rare elements such as names are copied from the input. This can be implemented in an elegant manner by equipping the decoder with a facility that can "point" to words from the input, which are then copied into the output (Vinyals, Fortunato, and Jaitly 2015; Grave, Joulin, and Usunier 2017; Gulcehre et al. 2016; Merity et al. 2017).

Editing sequences poses a different problem from other seq2seq tasks, as in many cases, *most* of the input remains unchanged and needs to be reproduced. When using existing decoders, this requires painstaking word-by-word copying of the input. In this paper, we propose to extend a decoder with a facility to copy entire spans of the input to the output in a single step, greatly reducing the number of decoder steps required to generate an output. This is illustrated in Fig. 1, where our model inserts two new words into a sentence by copying two spans of (more than) twenty tokens each.

However, this decoder extension exacerbates a well-known problem in training decoders with a copying facility: a target sequence can be generated in many different ways when an output token can be generated by different means. In our setting, a sequence of tokens can be copied token-by-token, in pairs of tokens, in triplets, *etc.* or in just a single step. In practice, we are interested in copying the longest spans possible, as copying longer spans both speeds up decoding at inference time and reduces the potential for making mistakes. To this end, we derive a training objective that marginalises over all different generation sequences yielding the correct output, which implicitly encourages copying longer spans. At inference time, we solve this problem by a variation of beam search that "merges" rays in the beam that generate the same output by different means.

In summary, this paper (i) introduces a sequence decoder able to copy entire spans (Sect.); (ii) derives a training objective that encourages our decoder to copy *long* spans when possible; (iii) discusses a variation of beam search which matches our new training objective; and (iv) presents extensive experiments showing that the span-copying decoder improves on editing tasks on natural language and program source code (Sect.).

Model

The core of our new decoder is a span-copying mechanism that can be viewed as a generalisation of pointer networks used for copying single tokens (Vinyals, Fortunato, and Jaitly 2015; Grave, Joulin, and Usunier 2017; Gulcehre et al. 2016; Merity et al. 2017). Concretely, modern sequence decoders treat copying from the input sequence as an alternative to generating a token from the decoder vocabulary, *i.e.* at each step, the decoder can either generate a token t from its vocabulary or it can copy the i-th token of the input. We view these as potential *actions* the decoder can perform and denote them by $\mathsf{Gen}(t)$ and $\mathsf{Copy}(i)$.

Formally, given an input sequence $\boldsymbol{in} = in_0 \ldots in_{n-1}$, the probability of a target sequence $\boldsymbol{o} = o_0 \ldots o_{m-1}$ is commonly factorised into sequentially generating all tokens of

*Work done during an internship in Microsoft Research, Cambridge, UK.

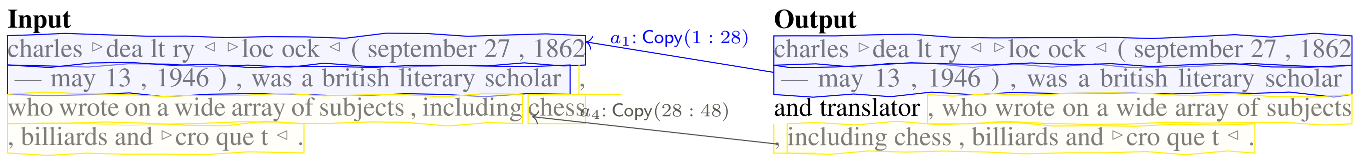

Input
charles ▷dea lt ry ◁ ▷loc ock ◁ (september 27 , 1862
— may 13 , 1946) , was a british literary scholar ,
who wrote on a wide array of subjects , including chess
, billiards and ▷cro que t ◁ .
a_1: Copy(1 : 28)
a_4: Copy(28 : 48)
Output
charles ▷dea lt ry ◁ ▷loc ock ◁ (september 27 , 1862
— may 13 , 1946) , was a british literary scholar
and translator , who wrote on a wide array of subjects
, including chess , billiards and ▷cro que t ◁ .

Figure 1: Sample edit generated by our span-copying model on the WikiAtomicEdits dataset on the edit representation task of Yin et al. (2019). ▷ and ◁ represent the BPE start/end tokens. The model first copies a long initial span of text $\mathsf{Copy}(1:28)$. The next two actions generate the tokens "and" and "translator", as in a standard sequence generation models. Then, the model again copies a long span of text and finally generates the end-of-sentence token (not shown).

the output.

$$p(\boldsymbol{o}_{[:m]} \mid \boldsymbol{in}) = \prod_{0 \le j < m} p(o_j \mid \boldsymbol{in}, \boldsymbol{o}_{[:j]}) \qquad (1)$$

Here, $\boldsymbol{o}_{[:j]}$ denotes the output tokens $o_0 \ldots o_{j-1}$, following Python's slicing notation. $p(o_j \mid \boldsymbol{in}, \boldsymbol{o}_{[:j]})$ is the probability of generating the token o_j, which is simply the probability of the $\mathsf{Gen}(t)$ action in the absence of a copying mechanism.[1] When we can additionally copy tokens from the input, this probability is the sum of probabilities of all correct actions. To formalise this, we denote evaluation of an action a into a concrete token as $[\![a]\!]$, where $[\![\mathsf{Gen}(t)]\!] = t$ and $[\![\mathsf{Copy}(i)]\!] = in_i$. Using $q(a \mid \boldsymbol{o})$ to denote the probability of emitting an action a after generating the partial output $\boldsymbol{o}$, we define

$$p(o_j \mid \boldsymbol{o}_{[:j]}) = \sum_{a, [\![a]\!] = o_j} q(a \mid \boldsymbol{o}_{[:j]}),$$

i.e. the sum of the probabilities of all correct actions.

Modelling Span Copying In this work, we are interested in copying whole subsequences of the input, introducing a sequence copying action $\mathsf{Copy}(i:j)$ with $[\![\mathsf{Copy}(i:j)]\!] = in_i \ldots in_{j-1}$. This creates a problem because the number of actions required to generate an output token sequence is not equal to the length of the output sequence anymore; indeed, there may be many action sequences of different length that can produce the correct output.

For example, Fig. 2 illustrates all action sequences generating the output $a\,b\,f\,d\,e$ given the input $a\,b\,c\,d\,e$. For example, we can initially generate the token a, copy it from the input, or copy the first two tokens. If we choose one of the first two actions, we then have the choice of either generating the token b or copying it from the input and then have to generate the token f. Alternatively, if we initially choose to copy the first two tokens, we have to generate the token f next. We can compute the probability of generating the target sequence by traversing the diagram from the right to the left. $p(\epsilon \mid a\,b\,f\,d\,e)$ is simply the probability of emitting a stop token, where ϵ denotes the empty sequence. $p(e \mid a\,b\,f\,d)$ is the sum of the probabilities $q(\mathsf{Gen}(e) \mid a\,b\,f\,d) \cdot p(\epsilon \mid a\,b\,f\,d\,e)$ and $q(\mathsf{Copy}(4:5) \mid a\,b\,f\,d) \cdot p(\epsilon \mid a\,b\,f\,d\,e)$, which re-use the term we already computed. Following this strategy, we can recursively compute the probability of generating the output token sequence by computing probabilities of increasing longer suffixes of it (essentially traversing the diagram in Fig. 2 from right to left).

[1]Note that all occurrences of p (and q below) are implicitly (also) conditioned on the input sequence $\boldsymbol{in}$, and so we drop this in the following to improve readability.

Formally, we reformulate Eq. (1) into a recursive definition that marginalises over all different sequences of actions generating the correct output sequence, following the strategy illustrated in Fig. 2. For this we define $|a|$, the length of the output of an action, *i.e.*, $|\mathsf{Gen}(t)| = 1$ and $|\mathsf{Copy}(i:j)| = j - i$. Note that w.l.o.g., we assume that actions $\mathsf{Copy}(i:j)$ with $j \le i$ do *not* exist, *i.e.* copies of zero-length spans are explicitly ignored.

$$p(\boldsymbol{o}_{[k:]} \mid \boldsymbol{o}_{[:k]}) = \sum_{\substack{a, \exists \ell. |a| = \ell \\ [\![a]\!] = \boldsymbol{o}_{[k:k+\ell]}}} q(a \mid \boldsymbol{o}_{[:k]}) \cdot p(\boldsymbol{o}_{[k+\ell:]} \mid \boldsymbol{o}_{[:k+\ell]}) \qquad (2)$$

Here, the probability of generating the correct suffix is only conditioned on the sequence generated so far and *not* on the concrete actions that yielded it. In practice, we implement this by conditioning our model of q at timestep k on a representation $\boldsymbol{h}_k$ computed from the partial output sequence $\boldsymbol{o}[:k]$. In RNNs, this is modelled by feeding the sequence of emitted tokens into the decoder, no matter how the decoder determined to emit these, and thus, one $\mathsf{Copy}(i:j)$ action may cause the decoder RNN to take multiple timesteps to process the copied token sequence. In causal self-attentional settings, this is simply the default behaviour. Finally, note that for numerical stability our implementation uses log-probabilities, implementing the summation of probabilities with the standard log-sum-exp trick.

Modelling Action Choices It remains to explain how we model the per-step action distribution $q(a \mid \boldsymbol{o})$. We assume that we have per-token encoder representations $\boldsymbol{r}_0 \ldots \boldsymbol{r}_{n-1}$ of all input tokens and a decoder state $\boldsymbol{h}_k$ obtained after emitting the prefix $\boldsymbol{o}_{[:k]}$. This can be the state of an RNN cell after processing the sequence $\boldsymbol{o}_{[:k]}$ (potentially with attention over the input) or the representation of a self-attentional model processing that sequence.

As in standard sequence decoders, we use an output embedding projection applied to $\boldsymbol{h}_k$ to obtain scores $s_{k,v}$ for all tokens in the decoder vocabulary. To compute a score for a $\mathsf{Copy}(i:j)$ action, we use a linear layer $\boldsymbol{W}$ to project the concatenation $\boldsymbol{r}_i \| \boldsymbol{r}_{j-1}$ of the (contextualised) embeddings

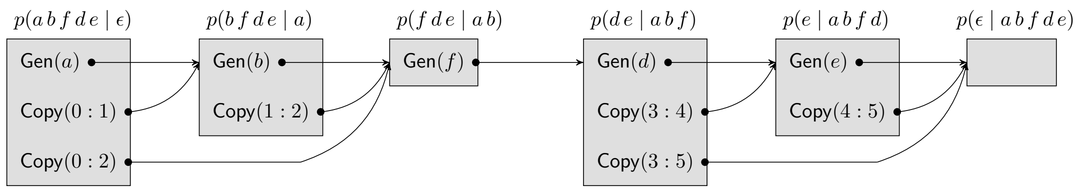


Figure 2: Illustration of different ways of generating the sequence $a\,b\,f\,d\,e$ given an input of $a\,b\,c\,d\,e$. Each box lists all correct actions at a given point in the generation process, and the edges after an action indicate which suffix token sequence still needs to be generated after it. We use ϵ to denote the empty sequence, either as prefix or suffix.

of the respective input tokens to the same dimension as $\boldsymbol{h}_k$ and then compute their inner product:

$$s_{k,[i:j]} = (\boldsymbol{W} \cdot (\boldsymbol{r}_i \| \boldsymbol{r}_{j-1})) \cdot \boldsymbol{h}_k^\top$$

We then concatenate all $s_{k,v}$ and $s_{k,[i:j]}$ and apply a softmax to obtain our action distribution $q(a \mid \boldsymbol{o})$. Note that for efficient computation in GPUs, we compute the $s_{k,[i:j]}$ for all i and j and mask all entries where $j \leq i$.

Training Objective We train in the standard teacher-forcing supervised sequence decoding setting, feeding to the decoder the correct output sequence independent of its decisions. We train by maximising $p(\boldsymbol{o} \mid \epsilon)$ unrolled according to Eq. (2). One special case to note is that we make a minor but important modification to handle generation of out-of-vocabulary words: *iff* the correct token can be copied from the input, Gen(UNK) is considered to be an incorrect action; otherwise only Gen(UNK) is correct. This is necessary to avoid pathological cases in which there is no action sequence to generate the target sequence correctly.

We found that using the marginalisation in Eq. (2) during training is crucial for good results. Our experiments (cf. Sect.) include an ablation in which we generate a per-token loss based only on the correct actions at each output token, without taking the remainder of the sequence into account (*i.e.*, at each point in time, we used a "multi-hot" objective in which the loss encourages picking any one of the correct actions). Training using this objective yielded a decoder which would most often only copy sequences of length one. In contrast, our marginalised objective penalises long sequences of actions and hence pushes the model towards copying longer spans when possible. The reason for this is that constructing an output sequence from longer spans implies that the required action sequence is shorter. As each action decision (q in Eq. (2)) "costs" some probability mass, as in practice q will assign some probability to incorrect choices. The marginalization in our objective ensures that the model is rewarded for preferring to copy longer (correct) spans, *i.e.* shorter action sequences, and hence fewer places at which probability mass is "spent". Note that we do not force the model to copy the longest possible sequence but instead allow the optimization process to find the best trade-off.

Beam Decoding Our approach to efficiently evaluate Eq. (2) at training time relies on knowledge of the ground truth sequence and so we need to employ another approach at inference time. We use a variation of standard beam search which handles the fact that action sequences of the same length can lead to sequences of different lengths. For this, we consider a forward version of Eq. (2) in which we assume to have a set of action sequences $\mathcal{A}$ and compute a lower bound on the true probability of a sequence $\boldsymbol{o}$ by considering all action sequences in $\mathcal{A}$ that evaluate to $o_0 \dots o_{k-1}$:

$$p(\boldsymbol{o}_{[:k]}) \geq \sum_{\substack{[a_1 \dots a_n] \in \mathcal{A} \\ [\![a_1]\!] \| \dots \| [\![a_n]\!] = \boldsymbol{o}_{[:k]}}} \prod_{1 \leq i \leq n} q(a_i \mid [\![a_1]\!] \| \dots \| [\![a_{i-1}]\!]) \tag{3}$$

If $\mathcal{A}$ contains the set of all action sequences generating the output sequence $o_0 \dots o_{k-1}$, Eq. (3) is an equality. At inference time, we under-approximate $\mathcal{A}$ by generating likely action sequences using beam search. However, we have to explicitly implement the summation of the probabilities of action sequences yielding the same output sequence. This could be achieved by a final post-processing step (as in Eq. (3)), but we found that it is more effective to "merge" rays generating the same sequence during the search. In the example shown in Fig. 2, this means to sum up the probabilities of the action sequences Gen(a)Gen(b) and Copy($0:2$), as they both generate the same output. To group action sequences of different lengths, our search procedure is explicitly considering the length of the generated token sequence and "pauses" the expansion of action sequences that have generated longer outputs. The pseudocode for this procedure is shown in Alg. 3, where `group_by_toks` merges different rays generating the same output.

Complexity Our objective in Eq. (2) can be computed using the described dynamic program with complexity in $O(N^2)$, where N is the sequence length. On a GPU, this can be efficiently parallelised, such that for all reasonable sequence lengths, only a linear number of (highly parallel) operations is required. In practice, a slowdown is only observed during training, but not during beam decoding. For example, during training for the task BFP$_{\text{small}}$ (see Sect.), computing the per-step log-probabilities takes about 80ms per minibatch, whereas marginalisation takes about 52ms per minibatch. This constitutes 65% extra time required for the marginalisation.

```
def beam_search(beam_size)
  beam = [{toks: [START_OF_SEQ], prob: 1}]
  out_length = 1
  while unfinished_rays(beam):
    new_rays = []
    for ray in beam:
      if len(ray.toks) > out_length
        or ray.toks[-1] == END_OF_SEQ:
        new_rays.append(ray)
      else:
        for (act, act_prob) in $q(\cdot \mid$ ray.toks):
          new_rays.append(
            {toks: ray.toks $\|$ $[\![$act$]\!]$,
             prob: ray.prob*act_prob})
    beam = top_k(group_by_toks(new_rays),
                 k=beam_size)
    out_length += 1
  return beam
```

Figure 3: Python-like pseudocode of beam search for span-copying decoders.

Related Work

Copying mechanisms are common in neural NLP. Starting from pointer networks (Vinyals, Fortunato, and Jaitly 2015), such mechanisms have been used across a variety of domains (Allamanis, Peng, and Sutton 2016; Gu et al. 2016; See, Liu, and Manning 2017) as a way to copy elements from the input to the output, usually as a way to alleviate issues around rare, out-of-vocabulary tokens such as names. Marginalising over a single token-copying action and a generation action has been previously considered (Allamanis, Peng, and Sutton 2016; Ling et al. 2016) but these works do not consider spans longer than one "unit".

Most similar to our work, Zhou et al. (2018) propose a method to copy spans (for text summarization tasks) by predicting the start and end of a span to copy. However, they do not handle the issue of different generation strategies for the same output sequence explicitly and do not present an equivalent to our training objective and modified beam search. Dependent on the choice of "copied spans" used to train the model, it either corresponds to the case of training our method without any marginalisation, or one in which only one choice (such as copying the longest matching span) is considered. In our experiments in Sect. , we show that both variants perform substantially less well than our marginalised objective.

Our method is somewhat related to the work of van Merriënboer et al. (2017); Grave et al. (2019), who consider "multiscale" generation of sequences using a vocabulary of potentially overlapping word fragments. Doing this also requires to marginalise out different decoder actions that yield the same output: in their case, generating a sequence from different combinations of word fragments, in contrast to our problem of generating a sequence token-by-token or copying a span. Hence, their training objective is similar to our objective in Eq. (2). A more important difference is that they use a standard autoregressive decoder in which the emitted word fragments are fed back as inputs. This creates the problem of having different decoder states for the same output sequence (dependent on its decomposition into word fragments), which van Merriënboer et al. (2017) resolve by averaging the states of the decoder (an RNN using LSTM cells). Instead, we are following the idea of the graph generation strategy of Liu et al. (2018), where a graph decoder is only conditioned on the partial graph that is being extended, and not the sequence of actions that generated the graph.

Recently, a number of approaches to sequence generation avoiding the left-to-right paradigm have been proposed (Welleck et al. 2019; Stern et al. 2019; Gu, Wang, and Zhao 2019; Lee, Mansimov, and Cho 2018), usually by considering the sequence generation problem as an iterative refinement procedure that changes or extends a full sequence in each iteration step. Editing tasks could be handled by such models by learning to refine the input sequence with the goal of generating the output sequence. However, besides early experiments by Gu, Wang, and Zhao (2019), we are not aware of any work that is trying to do this. Note however that our proposed span-copying mechanism is also naturally applicable in settings that require duplication of parts of the input, *e.g.* when phrases or subexpressions need to appear several times in the output (*cf.* `obj` in Fig. 4 for a simple example). Finally, sequence-refinement models could also be extended to take advantage of our technique without large modifications, though we believe the marginalisation over all possible insertion actions (as in Eq. (2)) to be intractable in this setting. Similarly, Grangier and Auli (2017) present QuickEdit, a machine translation method that accepts a source sentence (*e.g.* in German) a guess sentence (*e.g.* in English) that is annotated (by humans) with change markers. It then aims to improve upon the guess by generating a better target sentence avoiding the marked tokens. This is markedly different as the model accepts as input the spans that need to be removed or retained in the guess sentence. In contrast, our model needs to automatically infer this information. Concurrently with this work, Stahlberg and Kumar (2020) proposed a model predicting a sequence of span-level edits on an input sequence towards producing the output, designed specifically for text-editing. In contrast our method is also applicable to non-editing problems (*e.g.*, summarization) which may require copying long spans from an input.

An alternative to sequence generation models for edits is the work of Gupta et al. (2017), who propose to repair source code by first pointing to a single line in the output and then only generate a new version of that line. However, this requires a domain-specific segmentation of the input – lines are often a good choice for programs, but (multi-line) statements or expressions are valid choices as well. Furthermore, the approach still requires to generate a sequence that is similar to the input line and thus could profit from our span-copying approach.

Experimental Evaluation

We evaluate our RNN-based implementation on two types of tasks. First, we evaluate the performance of our models in the setting of learning edit representations (Yin et al. 2019) for natural language and code changes. Second, we

	WikiAtEds	GitHubEdits	C# Fixers
Yin et al. (2019)	72.9%	59.6%	n/a
S2S+COPYTOK	67.8%	64.4%	18.8%
S2S+COPYSPAN	**78.1%**	**67.4%**	**24.2%**

Table 1: Accuracy for edit representation task.

consider correction-style tasks in which a model has to identify an error in an input sequence and then generate an output sequence that is a corrected version of the input. In the evaluation below, S2S+COPYTOK denotes a variant of our S2S+COPYSPAN model in which the decoder can only copy single tokens. For all experiments, we use a single NVidia K80 GPU.

Edit Representations

We first consider the task of learning edit representations (Yin et al. 2019). The goal is to use an autoencoder-like model structure to learn useful representations of edits of natural language and source code. The model consists of an edit encoder $f_{\triangle}(x_-, x_+)$ to transform the edit between x_- and x_+ into an edit representation. Then, a neural editor $\alpha(x_-, f_{\triangle}(x_-, x_+))$ uses x_- and the edit representation to reconstruct x_+ as accurately as possible. We follow the same structure and employ our S2S+COPYSPAN decoder to model the neural editor α. We perform our experiments on the datasets used by Yin et al. (2019).

Our editor models have a 2-layer biGRU encoder with hidden size of 64, a single layer GRU decoder with hidden size of 64, tied embedding layers with a hidden size of 64 and use a dropout rate of 0.2. In all cases the edit encoder $f_{\triangle}$ is a 2-layer biGRU with a hidden size of 64. The GRU decoders of both models use a Luong-style attention mechanism (Luong, Pham, and Manning 2015).

Editing Wikipedia First, we consider the task of learning edit representations of small edits to Wikipedia articles (Faruqui et al. 2018).[2] The dataset consists of "atomic" edits on Wikipedia articles without any special filters. Tab. 1 suggests that the span-copying model achieves a significantly better performance in predicting the exact edit, even though our (nominally comparable) S2S+COPYTOK model performs worse than the model used by Yin et al. (2019). Our initial example in Fig. 1 shows one edit example, where the model, given the input text and the edit representation vector, is able to generate the output by copying two long spans and generating only the inserted tokens. Note that the WikiAtomicEdits dataset is made up of only insertions and deletions. The edit shown in Fig. 1 is generally representative of the other edits in the test set.

Editing Code We now focus on the code editing task of Yin et al. (2019) on the GitHubEdits dataset, constructed from small (less than 3 lines) code edits scraped from C# GitHub repositories. These include bug fixes, refactorings and other code changes. Again, the results in Tab. 1 suggest that our span-based models outperform the baseline by predicting the edited code more accurately.

Yin et al. (2019) also use the edit representations for a one-shot learning-style task on a "C# Fixers" dataset, which are small changes constructed using automatic source code rewrites. Each edit is annotated with the used rewrite rule so that the dataset can be used to study how well an edit representation generalises from one sample to another.

As in Yin et al. (2019), we train the models on the larger and more general GitHubEdits dataset. To evaluate, we compute the edit representation $f_{\triangle}(x_-, x_+)$ of one sample of a group of semantically similar edits in C# Fixers and feed it to the neural editor with the source code of another sample, *i.e.*, compute $\alpha(x'_-, f_{\triangle}(x_-, x_+))$. We repeat this experiment by picking the first 100 samples per fixer, computing the edit representation of each one and applying the edit to the other 99. The results of this process are shown in the last column of Tab. 1, suggesting that our span-copying models are able to improve on the one-shot transfer task as well.

Note that these results are not exactly comparable with those presented in Yin et al. (2019), as they randomly select 10 pairs (x_-, x_+), compute their edit representation and then for a given x'_- compute $\alpha(x'_-, f_{\triangle}(x_-, x_+))$ for each of the 10 edit representations, finally reporting the best accuracy score among the 10 candidates. Since this process cannot be replicated due to the random selection of samples, we instead opted for the reproducible process described above.

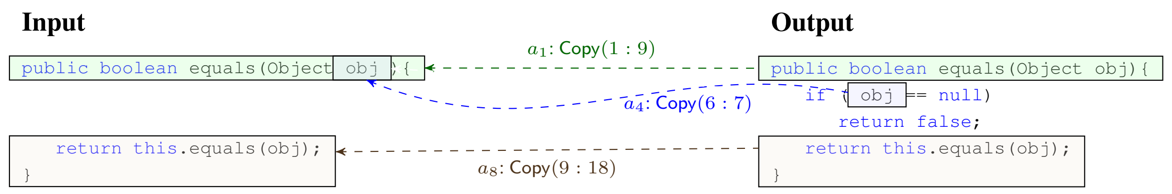
Overall, while our S2S+COPYTOK model is roughly on par with the numbers reported in Yin et al. (2019), our new model S2S+COPYSPAN clearly sets a new state of the art. This improvement can be attributed to the ability of the model to copy larger spans which allows it to better represent the relevant edits.

Correction Tasks

Correction tasks were one of the core motivations for our new decoding strategy, as they usually require to reproduce most of the input without changing it, whereas only few tokens are removed, updated or added. In our experiments, we focus on source code corrections, but provide an indication that S2S+COPYSPAN would work also for natural language.

Code Repair Automated code repair systems (Monperrus 2018) are commonly composed of two components, namely a (heuristic) component that suggests potentially fixed versions of the input, and an oracle (*e.g.*, a harness executing a test suite) that checks the candidates for correctness. Recent software engineering research has started to implement the heuristic component using seq2seq models (Chen et al. 2018; Tufano et al. 2019; Lutellier et al. 2019). The models are usually viewed as language models (conditioned on the faulty code) or employ standard neural machine translation pipelines mapping from "faulty" to "correct" code. The task usually only requires minor changes to the input code and consequently most of the input is copied into the output. We believe that our model is a natural match for this setting.

[2]According to Yin et al. (2019), a part of the data was corrupted and hence they used a smaller portion of the data.



```
public boolean equals(Object obj){
    return this.equals(obj);
}
```

```
public boolean equals(Object obj){
if (obj == null)
    return false;
    return this.equals(obj);
}
```

Figure 4: Generation example in BFP$_{\text{small}}$ (edited for clarity). S2S+COPYSPAN learns to copy long spans while generating the necessary edits. The non-highlighted output tokens are generated using Gen(t). All other tokens are copied from the input.

To test this hypothesis, we use the two bug-fix pair (BFP) datasets of Tufano et al. (2019). The BFP$_{\text{small}}$ dataset contains pairs where each snippet has at most 50 tokens and the BFP$_{\text{medium}}$ dataset has Java snippets containing from 50 up to 150 tokens. In these datasets, the input is code with some form of a bug, whereas the output is correct code. This corpus was constructed by scraping Git commits and filtering for those with commit messages suggesting that the edit fixes a bug. For both the S2S+COPYTOK and S2S+COPYSPAN models we employ a 2-layer biGRU as an encoder and a single layer GRU decoder. We use embeddings with 32 dimensions and GRUs with hidden units of size 128. Note that the vocabulary size for this task is just 400 by construction of the dataset. We employ a Luong-style (Luong, Pham, and Manning 2015) attention mechanism in the the decoders of both models.

Tab. 2 shows the results of our models, as well as the original results reported by Tufano et al. (2019). Overall, the S2S+COPYSPAN model performs better on both datasets, achieving a new state of the art. This suggests that the span-copying mechanism is indeed beneficial in this setting, as becomes clear in a qualitative analysis. Fig. 4 shows an example (slightly modified for readability) of a code repair prediction and the span-copying actions. In this case, the model has learned to copy all of the input code in chunks, extending it only by inserting some new tokens in the middle.

We use this task to consider four ablations of our model, clarifying the impact of each of the contributions of our paper. To study the effect of marginalising over all correct choices (Eq (2)), we compare with two alternative solutions. First, we train the model to always copy the longest possible span. The results shown in Tab. 2 indicate that this has a substantial impact on results, especially for results obtained by beam search. We believe that this is due to the fact that the model fails to capture the entire spectrum of correct actions, as the objective penalises correct-but-not-longest copying actions. This leads to a lack of informed diversity, reducing the benefits of beam search.

Second, we consider an objective in which we use no marginalisation, but instead train the model to predict any one of the correct actions at each step, without any preference for long or short copied spans – this corresponds to the approach of Zhou et al. (2018). Our results show that this is competitive on shorter output sequences, but quickly degrades for longer outputs. We believe that this is due to the fact that the model is not encouraged to use as few actions as possible, which consequently means that producing a correct output can require dozens or hundreds of prediction steps.

We also evaluated our modified beam search in Alg. 3 in comparison to standard beam search and greedy decoding on the code repair task. The results show (small) improvements when considering only the top result, but substantial gains when taking more decoder results into account.

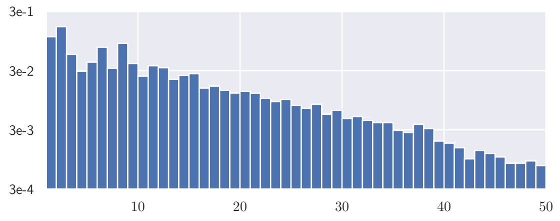
For a quantitative analysis, we additionally compute statistics for the greedy decoding strategy of S2S+COPYSPAN. In Fig. 5, we plot the frequency of the lengths of the copied spans for BFP$_{\text{small}}$ and BFP$_{\text{medium}}$. Given that the merging mechanism in beam decoding does *not* offer a unique way for measuring the length of the copied spans (actions of different lengths are often merged), we disable beam merging for these experiments and employ greedy decoding. Overall, the results suggest that the model learns to copy long sequences, although single-copy actions (*e.g.*, to re-use a variable name) are also common. In the BFP$_{\text{small}}$ dataset, S2S+COPYSPAN picks a Copy$(\cdot : \cdot)$ action with a span longer than one token about three times per example, copying spans 9.6 tokens long on average (median 7). Similarly in BFP$_{\text{medium}}$, S2S+COPYSPAN copies spans of 29.5 tokens long (median 19) This suggests that the model has learned to take advantage of the span-copying mechanism, substantially reducing the number of actions that the decoder needs to perform.

We also find that the S2S+COPYTOK model tends to (mistakenly) assign higher scores to the input sequence, with the input sequence being predicted as an output more often compared to the span-copying model: the MRR of the input sentence is 0.74 for the baseline S2S+COPYTOK model compared to 0.28 for the S2S+COPYSPAN model in the BFP$_{\text{small}}$ dataset. This suggests that the strong bias towards copying required of the baseline model (as most of the decoding actions are single-token copies) negatively impacts the tendency to generate *any* change.

Grammar Error Correction A counterpart to code repair in natural language processing is grammar error correction (GEC). Again, our span-copying model is a natural fit for this task. However, this is a rich area of research with highly optimised systems, employing a series of pretraining techniques, corpus filtering, deterministic spell-checkers, *etc.* As we believe our contribution to be orthogonal to the addition of such systems, we evaluate it in a simplified setting. We only compare our S2S+COPYSPAN model to our baseline S2S+COPYTOK model, expecting results substantially below the state of the art and only highlighting the *relative*

	Accuracy	Accuracy@20	MRR	Structural Match
On BFP$_{\text{small}}$				
Tufano et al. (2019)	9.2%	43.5%	—	—
S2S+COPYTOK	14.8%	42.0%	0.177	18.2%
S2S+COPYSPAN	**17.7%**	**45.0%**	**0.247**	**21.2%**
S2S+COPYSPAN (always copy longest)	14.2%	33.7%	0.174	14.2%
S2S+COPYSPAN (no marginalization)	16.9%	43.4%	0.210	20.2%
On BFP$_{\text{medium}}$				
Tufano et al. (2019)	3.2%	22.2%	—	—
S2S+COPYTOK	7.0%	23.8%	0.073	9.4%
S2S+COPYSPAN	**8.0%**	**25.4%**	**0.105**	**13.7%**
S2S+COPYSPAN (always copy longest)	7.2%	20.0%	0.090	10.8%
S2S+COPYSPAN (no marginalization)	2.5%	11.1%	0.035	3.7%

Table 2: Evaluation of models on the code repair task. Given an input code snippet, each model needs to predict a corrected version of that code snippet. "Structural Match" indicates that the generated output is identical to the target output up to renaming the identifiers (*i.e.*, variables, functions).


(a) BFP$_{\text{small}}$ ($\mu = 9.6$, median: 7)

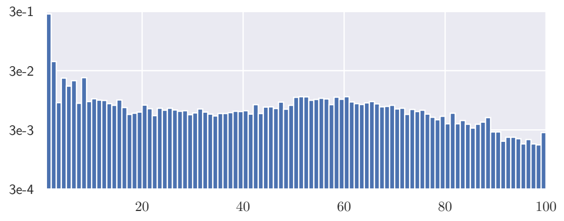

(b) BFP$_{\text{medium}}$ ($\mu = 29.5$, median: 19)

Figure 5: Length histograms of Copy($\cdot$: $\cdot$) actions during beam decoding in log-y scale. Beam merging is disabled for computing the statistics of this experiment. For BFP$_{\text{small}}$ 11.2% of the copy actions are single-copy actions, whereas for BFP$_{\text{medium}}$ 27.1% of the actions are single-copy actions. This suggests that S2S+COPYSPAN uses long span-copying actions in the majority of the cases where it decides to take a span-copying action.

	Prec.	Recall	$F_{0.5}$
S2S+COPYTOK	**34.9%**	6.4%	0.1853
S2S+COPYSPAN	28.9%	**10.4%**	**0.2134**

Table 3: Evaluation on Grammar Error Correction (GEC) (Bryant, Felice, and Briscoe 2017). Note: our models use no pretraining, spell checking or other external data, commonly used in GEC tasks.

improvement our contribution offers. Our models have a 2-layer bi-GRU encoder with a hidden size of 64, a single layer GRU decoder with hidden size of 64, tied embedding layer of size 64 and use a dropout rate of 0.2.

We use training/validation folds of the FCE (Yannakoudakis, Briscoe, and Medlock 2011) and W&I+LOCNESS (Granger 1998; Bryant et al. 2019) datasets for training and test on the test fold of the FCE dataset. These datasets contain sentences of non-native English students along with ground-truth grammar error corrections from native speakers. Tab. 3 shows the results computed with the ERRANT evaluation metric (Bryant, Felice, and Briscoe 2017), where we can observe that our span-copying decoder again outperforms the baseline decoder. Note that the results of both models are substantially below those of state of the art systems (*e.g.* Grundkiewicz, Junczys-Dowmunt, and Heafield (2019)), which employ (a) deterministic spell checkers (b) extensive monolingual corpora for pre-training and (c) ensembling.

Conclusion

We have presented a span-copying mechanism for commonly used encoder-decoder models. In many real-life tasks, machine learning models are asked to edit a pre-existing input. Such models can take advantage of our proposed model. By correctly and efficiently marginalising over all possible span-copying actions we can encourage the model to learn to take a single span-copying action rather than multiple smaller per-token actions.

Of course, in order for a model to copy spans, it needs to be able to represent all possible plans which is $O(n^2)$ to the input size. Although this is memory-intensive, $O(n^2)$ representations are common in sequence processing (*e.g.* in transformers). In the future, it would be interesting to investigate alternative span representations. Additionally, directly optimising for the target metrics of each task (rather than negative log-likelihood) can further improve results.

References

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, 2091–2100.

Bryant, C.; Felice, M.; Andersen, Ø. E.; and Briscoe, T. 2019. The BEA-2019 shared task on grammatical error correction. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*.

Bryant, C.; Felice, M.; and Briscoe, E. J. 2017. Automatic annotation and evaluation of error types for grammatical error correction. In *ACL*.

Chen, Z.; Kommrusch, S.; Tufano, M.; Pouchet, L.-N.; Poshyvanyk, D.; and Monperrus, M. 2018. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808* .

Faruqui, M.; Pavlick, E.; Tenney, I.; and Das, D. 2018. WikiAtomicEdits: A Multilingual Corpus of Wikipedia Edits for Modeling Language and Discourse. In *Proc. of EMNLP*.

Granger, S. 1998. The computer learner corpus: A versatile new source of data in SLA rsearch. *Learner English on Computer* .

Grangier, D.; and Auli, M. 2017. QuickEdit: Editing text & translations by crossing words out. *arXiv preprint arXiv:1711.04805* .

Grave, E.; Joulin, A.; and Usunier, N. 2017. Improving neural language models with a continuous cache. In *International Conference on Learning Representations (ICLR)*.

Grave, É.; Sukhbaatar, S.; Bojanowski, P.; and Joulin, A. 2019. Training Hybrid Language Models by Marginalizing over Segmentations. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Grundkiewicz, R.; Junczys-Dowmunt, M.; and Heafield, K. 2019. Neural Grammatical Error Correction Systems with Unsupervised Pre-training on Synthetic Data. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, 252–263.

Gu, J.; Lu, Z.; Li, H.; and Li, V. O. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393* .

Gu, J.; Wang, C.; and Zhao, J. 2019. Levenshtein Transformer. *arXiv preprint arXiv:1905.11006* .

Gulcehre, C.; Ahn, S.; Nallapati, R.; Zhou, B.; and Bengio, Y. 2016. Pointing the unknown words. In *Association for Computational Linguistics (ACL)*.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. 2017. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Lee, J.; Mansimov, E.; and Cho, K. 2018. Deterministic non-autoregressive neural sequence modeling by iterative refinement. *arXiv preprint arXiv:1802.06901* .

Ling, W.; Grefenstette, E.; Hermann, K. M.; Kočiský, T.; Senior, A.; Wang, F.; and Blunsom, P. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* .

Liu, Q.; Allamanis, M.; Brockschmidt, M.; and Gaunt, A. L. 2018. Constrained Graph Variational Autoencoders for Molecule Design. In *International Conference on Learning Representations (ICLR)*.

Luong, M.-T.; Pham, H.; and Manning, C. D. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* .

Lutellier, T.; Pang, L.; Pham, H. V.; Wei, M.; and Tan, L. 2019. ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair. *CoRR* abs/1906.08691.

Merity, S.; Xiong, C.; Bradbury, J.; and Socher, R. 2017. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*.

Monperrus, M. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51(1): 17.

See, A.; Liu, P. J.; and Manning, C. D. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*.

Stahlberg, F.; and Kumar, S. 2020. Seq2Edits: Sequence Transduction Using Span-level Edit Operations. *arXiv preprint arXiv:2009.11136* .

Stern, M.; Chan, W.; Kiros, J.; and Uszkoreit, J. 2019. Insertion transformer: Flexible sequence generation via insertion operations. *arXiv preprint arXiv:1902.03249* .

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Neural Information Processing Systems (NeurIPS)*.

Tufano, M.; Watson, C.; Bavota, G.; Penta, M. D.; White, M.; and Poshyvanyk, D. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28(4): 19.

van Merriënboer, B.; Sanyal, A.; Larochelle, H.; and Bengio, Y. 2017. Multiscale sequence modeling with a learned dictionary. *arXiv preprint arXiv:1707.00762* .

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Neural Information Processing Systems (NeurIPS)*, 2692–2700.

Welleck, S.; Brantley, K.; Daumé III, H.; and Cho, K. 2019. Non-monotonic sequential text generation. *arXiv preprint arXiv:1902.02192* .

Yannakoudakis, H.; Briscoe, T.; and Medlock, B. 2011. A new dataset and method for automatically grading ESOL texts. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, 180–189. Association for Computational Linguistics.

Yin, P.; Neubig, G.; Allamanis, M.; Brockschmidt, M.; and Gaunt, A. L. 2019. Learning to represent edits. In *International Conference on Learning Representations (ICLR)*.

Zhou, Q.; Yang, N.; Wei, F.; and Zhou, M. 2018. Sequential copying networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*.