Iterative Decoding for Compositional Generalization in Transformers

Anonymous ACL submission

Abstract

Deep learning models generalize well to in-distribution data but struggle to generalize compositionally, i.e., to combine a set of learned primitives to solve more complex tasks. In sequence-to-sequence (seq2seq) learning, transformers are often unable to predict correct outputs for longer examples than those seen at training. This paper introduces *iterative decoding*, an alternative to seq2seq that (i) improves transformer compositional generalization in the PCFG and Cartesian product datasets and (ii) evidences that, in these datasets, seq2seq transformers do not learn iterations that are not unrolled. In iterative decoding, training examples are broken down into a sequence of intermediate steps that the transformer learns iteratively. At inference time, the intermediate outputs are fed back to the transformer as intermediate inputs until an end-of-iteration token is predicted. We conclude by illustrating some limitations of iterative decoding in the CFQ dataset.

1 Introduction

Deep learning architectures achieve state-of-the-art (SOTA) results in a wide array of machine learning problems, where their impressive performance is attributed to their ability to generalize (Goodfellow et al., 2016; LeCun et al., 2015). However, this ability is typically limited to generalization under the statistical learning paradigm, i.e., in-distribution generalization, and does not encompass generalizing compositionally. Compositional generalization is the ability of a model to combine a set of learned primitives to execute more complex tasks. For instance, for a ground robot whose motion planner has learned to execute the instructions "walk", "jump", and "jump right", generalizing compositionally would be to be able to execute the instruction "walk right" (Lake and Baroni, 2018).

In machine learning, compositional generalization is desirable for two reasons. First, because it is a crucial aspect of intelligence observed in both humans and classical artificial intelligence. In humans, a prevailing example is the way children can solve complicated mathematical expressions after being taught basic arithmetics. Second, because it can increase a model's data efficiency. By endowing models with the ability to extrapolate to unseen examples involving compositions of primitives not seen during training, compositionality acts as a mechanism for data augmentation.

In this paper, our goal is to increase compositional generalization in transformers with particular focus on natural language-like tasks, where compositionality is key. Understanding that for models to execute composite tasks they need to be taught *how to compose*, we introduce *iterative decoding*, an alternative to sequence-to-sequence (seq2seq) learning that decomposes the process of mapping the inputs to the outputs of each example into a sequence of intermediate steps which the transformer learns to perform iteratively. During training, each input-output pair is converted into a sequence of "intermediate input-intermediate output" pairs. During prediction, the predicted intermediate outputs are adapted into the subsequent intermediate inputs, which are fed back to the transformer until an end-of-iteration (EOI) token is produced.

Our main contributions are (i) showing that iterative decoding, especially when combined with architectural modifications such as relative attention (Shaw et al., 2018) and copy decoding (Gu et al., 2016), improves compositional generalization in transformers trained on PCFG (top left of Fig. 1) (Hupkes et al., 2020) and Cartesian product (bottom left of Fig. 1), and (ii) evidencing that, in these datasets, seq2seq transformers cannot learn iterations unless they are unrolled. PCFG is a string editting dataset in which we consider two compositionally hard splits. In Cartesian product, the goal is to generalize to longer input vectors than those on which the model is trained. We also present

PCFG

```
IN: copy swap_first_last E18 B11 T13 W20

OUT: W20 B11 T13 E18
```

Cartesian product

```
IN: 1 4 3 [SEP] a d

OUT: 1 a 1 d 4 a 4 d 3 a 3 d
```

CFQ

```
IN: Did a person marry a cinematographer ,
influence M1 , and influence M2

OUT: SELECT count(*) WHERE {
 ?x0 a ns:people.person .
 ?x0 ns:influence.influence_node.influenced M1 .
 ?x0 ns:influence.influence_node.influenced M2 .
 ?x0 ns:people.person.spouse_s ?x1 .
 ?x1 a ns:film.cinematographer .
 FILTER ( ?x0 != ?x1 ) }
```

Figure 1: Examples of input-output pairs in the PCFG, Cartesian product, and CFQ datasets.

numerical results on CFQ (Keysers et al., 2019), a semantic parsing dataset consisting of natural language questions and SPARQL queries (right of Fig. 1). The results obtained on CFQ evidence a limitation of iterative decoding, which can be sensitive to the ordering of the intermediate steps.

We believe that iterative decoding can potentially improve transformer performance in other compositionally hard problems in NLP. Nevertheless, the definition of the intermediate steps for these tasks is dataset-specific and is out of the scope of this work. Our goal is not to propose iterative decoding as a "one-size-fits-all" solution, but rather to show that, when a heuristic is available (as is the case of the datasets we consider), iterative decoding can improve the ability of transformers to generalize compositionally [cf. Remark 1]. In the case of PCFG, Cartesian and CFQ, the specific construction of the intermediate steps is detailed in the experimental results section (Secs. 4.1 through 4.3).

2 Background

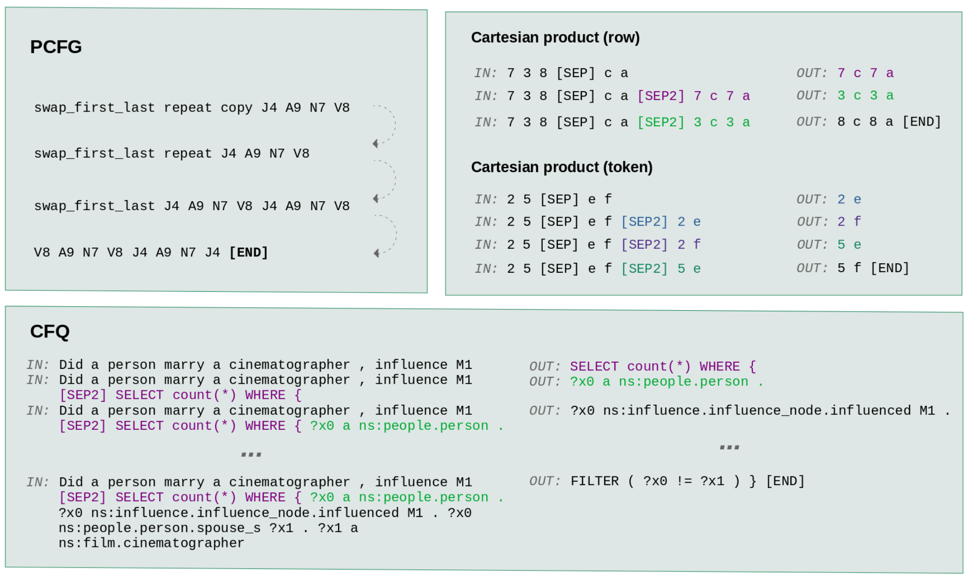
In this section, we introduce some background and related work on compositional generalization and transformer architectures.

2.1 Compositional generalization

Compositional generalization (or compositionality (Fodor, 2001)) refers to the ability of a model/agent that has learned to perform a set of basic operations—*primitives*—to generalize to more complex operations, i.e., operations consisting of *compositions* of the learned primitives (Lake and Baroni, 2018). Examples of operations requiring compositionality are shown in Fig. 1 for three datasets. For instance, the top-left corner shows an example from the PCFG (Hupkes et al., 2020) dataset. In some versions of this dataset, the model is trained to solve several atomic string editing operations (such as `copy` and `swap_first_last`), and how to compose them. The test data contains longer sequences with more operations than seen during training. Hence, the model's performance relies on compositional generalization. This string editing example can be seen as an instance of *productivity*, one of the five types of compositional generalization identified by Hupkes et al. (2020) which involves generalizing to longer examples. Another type of compositional generalization is *systematicity*, which is the ability to recombine known parts and rules in ways different than those seen during training.

Early works on compositionality have explored the limitations of machine learning models in generalizing compositionally. Liška et al. (2018) showed that, while it is theoretically possible for a recurrent neural network (RNN) to generalize in this way, only a small fraction of the models they trained behaved compositionally. Lake and Baroni (2018) proposed SCAN, a dataset consisting of navigation commands to be mapped to action sequences, and observed that while RNNs trained on it generalized well when the differences between the training and test sets were small, they failed when more systematic compositional skills were required. Other datasets created to measure compositionality include ListOps (Nangia and Bowman, 2018), where latent tree models perform worse than purely sequential RNNs, and PCFG (Hupkes et al., 2020) and CFQ (Keysers et al., 2019), where neither long-short term memory (LSTM) nor transformer-based architectures perform well.

More recently, a popular research direction is to try to endow these machine learning models with a "compositional generalization bias". Kim et al. (2021) saw benefits in converting CFQ into a classification task and using structural annotations (e.g., entity links) as attention masks in transformers. Ontañón et al. (2021) were able to improve transformer compositional generalization on a va-


PCFG

```
swap_first_last repeat copy J4 A9 N7 V8

swap_first_last repeat J4 A9 N7 V8

swap_first_last J4 A9 N7 V8 J4 A9 N7 V8

V8 A9 N7 V8 J4 A9 N7 J4 [END]
```

Cartesian product (row)

```
IN: 7 3 8 [SEP] c a                          OUT: 7 c 7 a
IN: 7 3 8 [SEP] c a [SEP2] 7 c 7 a           OUT: 3 c 3 a
IN: 7 3 8 [SEP] c a [SEP2] 3 c 3 a           OUT: 8 c 8 a [END]
```

Cartesian product (token)

```
IN: 2 5 [SEP] e f                            OUT: 2 e
IN: 2 5 [SEP] e f [SEP2] 2 e                 OUT: 2 f
IN: 2 5 [SEP] e f [SEP2] 2 f                 OUT: 5 e
IN: 2 5 [SEP] e f [SEP2] 5 e                 OUT: 5 f [END]
```

CFQ

```
IN: Did a person marry a cinematographer , influence M1       OUT: SELECT count(*) WHERE {
IN: Did a person marry a cinematographer , influence M1       OUT: ?x0 a ns:people.person .
    [SEP2] SELECT count(*) WHERE {
IN: Did a person marry a cinematographer , influence M1       OUT: ?x0 ns:influence.influence_node.influenced M1 .
    [SEP2] SELECT count(*) WHERE { ?x0 a ns:people.person .

                      ...                                                    ...

IN: Did a person marry a cinematographer , influence M1       OUT: FILTER ( ?x0 != ?x1 ) } [END]
    [SEP2] SELECT count(*) WHERE { ?x0 a ns:people.person .
    ?x0 ns:influence.influence_node.influenced M1 . ?x0
    ns:people.person.spouse_s ?x1 . ?x1 a
    ns:film.cinematographer
```

Figure 2: Examples of intermediate input-output pairs in the PCFG and Cartesian product datasets.

riety of compositionally hard datasets by making architectural modifications such as relative attention, copy decoders, and weight sharing. Taking a similar approach, Csordás et al. (2021) observed performance improvements from relative position encodings and scaled embeddings. Other strategies to improve compositional generalization include increased pretraining (Furrer et al., 2020), data augmentation (Andreas, 2019) and differentiable neurocomputers (Graves et al., 2016).

Closely related to our work, PonderNet (Banino et al., 2021) trains a model that iterates internally to achieve a better compromise between training accuracy and generalization. It predicts both an output and a halting probability at each step, operating recurrently. Iterative decoding also operates recurrently, but with two important differences: (i) the intermediate steps are supervised, and (ii) the model is trained to produce a special token to indicate the end of the iteration rather than predicting a halting probability at each step.

2.2 Transformer model

In this paper we focus on transformer models. Despite struggling to generalize compositionally, transformer-based architectures such as BERT (Devlin et al., 2018) and T5 (Raffel et al., 2019) were popularized by their remarkable performance in machine translation (Zhu et al., 2020), question answering (Ainslie et al., 2020), summarization (Zhang et al., 2019) and other natural language processing (NLP) tasks.

Introduced by Vaswani et al. (2017), the basic transformer model is composed of an encoder and a decoder. The encoder is made up of layers consisting of a self-attention sublayer and a feedforward sublayer. The decoder has the same structure, but with an additional attention sublayer to compute the decoder-to-encoder attention. The input to the transformer is a sequence of token embeddings. Since these embeddings do not carry information about the position of each token in the sequence, a position encoding is typically added to the input embeddings. These are then fed to the encoder, which encodes all tokens at once and forwards the result to the decoder. From the encoded input and the decoded output tokens generated so far, the decoder generates the distribution of the next output token, one token at a time.

We experiment with two extensions of the original transformer architecture: **relative position encodings** (Shaw et al., 2018) and **copy decoding** (Gu et al., 2016). We chose these techniques because they have been shown to improve compositionality in seq2seq learning (Ontañón et al., 2021) and require less parameters than larger transformer models. To each pair of tokens in the input, relative position encodings assign a label that equal to the minimum between their relative distance and a relative attention radius. Relative position encodings are thus position invariant, which means that two tokens that are k positions apart will attend to each other in the same way regardless of their absolute positions in the sequence. This also makes them in-

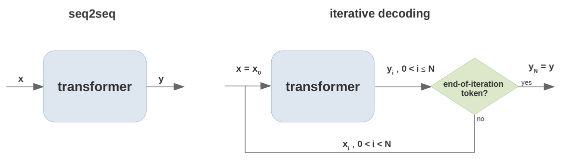


Figure 3: Prediction routines for the seq2seq transformer (left) and iterative decoding transformer (right).

variant to length of the sequence which, intuitively, should improve compositional generalization.

Copy decoding involves adding a learnable parameter that allows to switch between the decoder and a copy decoder which produces an independent embedding that can be interpreted as a "copy" from the input sequence. This helps with compositional generalization because many tasks, e.g., PCFG, have a type of input-output symmetry that requires producing parts of the input at the output. Copy decoding can also be seen as "pointing" to tokens in the input sequence. This pointing mechanism was proposed in (Gulcehre et al., 2016), and Oriol et al. (2015) showed that it allows models to generalize beyond the lengths they are trained on.

3 Iterative Decoding

To improve compositional generalization in transformers, we introduce iterative decoding. As illustrated on the right hand side of Fig. 3, iterative decoding consists of predicting a series of intermediate outputs $y_1, y_2, y_3, \ldots$ from an input $x = x_0$, and then adapting these outputs into intermediate inputs $x_i = y_i, i > 0$, that are fed back to the model until the final output $y_N = y$ is predicted. This can be visualized by considering the PCFG example $x =$ "`swap_first_last repeat copy J4 A9 N7 V8`" on the left hand side of Fig. 2. A seq2seq transformer trained on the PCFG dataset is expected to output $y =$ "`V8 A9 N7 V8 J4 A9 N7 J4 [END]`" in one forward pass (i.e, to go from top to bottom in the Fig.). However, in iterative decoding, the transformer's output to the input $x_0 = x$ would be $y_1 =$ "`swap_first_last repeat J4 A9 N7 V8`", which is the first intermediate output of iterative decoding (the second string from the top), and corresponds to just executing one of the operations in the input, `copy`. Setting $x_1 = y_1$ and feeding this instruction back to the transformer, we obtain y_2= "`swap_first_last J4 A9 N7 V8 J4 A9 N7 V8`" (the third string from the top). The intermediate output y_2 then becomes the intermediate input x_2, which the transformer processes to produce the final output $y_3 =$ "`V8 A9 N7 V8 J4 A9 N7 J4 [END]`".

The main motivation for iterative decoding comes from the very idea of compositional generalization: by decomposing complex instructions into intermediate steps, iterative decoding essentially teaches models how to compose. Another motivation, related to the first, is that iterative decoding mimics how humans are taught how to perform many compositional tasks. For example, when teaching how to solve the arithmetic expression $2 \times (1 + 1)$, first we demonstrate how to solve the inner sum, then how to eliminate the parentheses, and finally how to compute the product. A third motivation for iterative decoding is that learning step by step can potentially prevent the model from learning shortcuts, i.e., from overfitting to specific examples instead of learning compositional symmetries applicable to examples requiring the same type of compositional generalization to be solved.

3.1 Implementation

To implement iterative decoding, we modify both how models are trained and how they predict.

Training. Instead of being trained on the original inputs and outputs, iterative decoding transformers are trained on the "intermediate input-intermediate output" pairs (x_{i-1}, y_i) for $1 \leq i \leq N$. These inputs and outputs are pre-generated, and their form is specific to each task (see Sec. 4 for examples for PCFG, Cartesian product, and CFQ). Naturally, the addition of the "intermediate input-intermediate output" pairs to the training data increases the number of training samples, which leads to an increase of the computational cost per epoch proportional to the average number of intermediate steps. In our experiments, we avoid this by training the iterative decoding and the seq2seq transformer not for the same number of epochs, but for the same number of steps. Another difference with seq2seq training is that in iterative decoding an EOI token has to be added to the training outputs so that the model learns when to stop. Intuitively, the intermediate outputs correspond to providing supervision on the steps that the transformer should execute to solve the task. This is analogous to how, rather than learning to solve arithmetic expressions by looking at input-output mappings, humans are taught to solve the intermediate steps involved in their computation. Further note that there is no recurrence

Table 1: Sentence accuracy on the training and test sets for the random, productivity and systematicity splits of the PCFG dataset, for seq2seq and iterative decoding.

Model	**Random**		**Productivity**		**Systematicity**	
	train	test	train	test	train	test
Seq2seq	87.2%	85.9%	90.6%	34.2%	89.5%	64.8%
+ Relative attention	98.1%	97.4%	98.8%	65.1%	98.2%	85.7%
+ Copy decoder	97.7%	97.0%	98.3%	63.9%	98.2%	85.1%
Iterative decoding	96.5%	93.2%	96.6%	45.7%	94.6%	82.9%
+ Relative attention	99.8%	99.2%	100%	91.9%	99.7%	97.0%
+ Copy decoder	99.7%	**99.4**%	100%	**93.3**%	99.8%	**97.8**%

involved in the training of the iterative decoding transformer. An intermediate output produced by the model during training does not need to be fed back to the model as an intermediate input, because the subsequent intermediate input is already an input sample of the training set. Moreover, "intermediate input-intermediate output pairs" corresponding to the same example do not need to be presented to the transformer in any particular order, and can be shuffled at random in the training set.

Prediction. Depending on the number of intermediate steps necessary to iteratively decode an example, the prediction requires multiple forward passes of the transformer. Hence, it is implemented as a while loop where the stopping condition is finding the EOI token. This is illustrated on Fig. 3, which compares seq2seq (left) with iterative decoding predictions (right). After each intermediate prediction, a data processing step may be needed to adapt the intermediate outputs into the following intermediate inputs in some datasets. As we could see from the example above, this is not necessary in PCFG, because it has a built-in recursive structure. But it is necessary in Cartesian product and CFQ as we detail in Sec. 4. While ideally this processing step should be learned, in this paper we provide it manually to develop a preliminary understanding of the advantages of iterative decoding.

Remark 1 (Construction of intermediate steps.) The construction of the intermediate steps is dataset-specific. In PCFG, Cartesian and CFQ, this construction is detailed in Secs. 4.1 through 4.3. While in principle the intermediate steps do not have to fit any specific requirements, their definition will typically rely on a heuristic method to decompose the input into "intermediate input-intermediate output" pairs, which push the model to learn the basic primitives operations in a dataset, which can then be composed to perform more complex operations. In some datasets, e.g., PCFG, this is straightforward. In others, e.g., Cartesian, there are multiple admissible options to decompose the input into intermediate steps. In more complex problems such as CFQ and other semantic parsing tasks, this decomposition is less obvious and requires some engineering. The definition of heuristics for different datasets is out of the scope of this paper as our goal is not to propose iterative decoding as a "one-size-fits-all" solution, but rather to show that, when a heuristic is available, iterative decoding can improve the ability of transformers to generalize in a compositional way.

4 Results and Discussion

In this section we describe the iterative decoding schemes for PCFG, Cartesian product and CFQ, and present and discuss numerical results obtained for seq2seq and iterative decoding transformers on these datasets. All transformers have $\ell = 6$ encoder/decoder layers, embedding dimension $d = 64$, feedforward dimension $f = 256$ and $h = 4$ attention heads. For each dataset, the seq2seq and the iterative decoding transformer are trained for the same number of training steps and each experiment is repeated 3 times. Implementation details can be found in Appendix B.

4.1 PCFG

PCFG is an artificial translation dataset proposed by Hupkes et al. (2020) and generated by a probabilistic context free grammar. A more detailed description can be found in Appendix A, and an example of input-output pair is shown on the top left corner of Fig. 1. There are six training-test splits of PCFG. The first is a random split which we use as a baseline. The other five are compositionally hard splits used to measure the five different types compositional generalization. We focus on two of them: productivity and systematicity. In the productivity split, the training samples have up to 8 string operations, while the test samples have 9 or more. In the systematicity split, the operations in the test set they are combined in different ways than in the training set.

We apply iterative decoding to PCFG by breaking down each example into a number of intermediate steps equal to the number of string editing operations present in the original input. Each intermediate step solves the rightmost instruction in the current intermediate input. Up until the last step, all of the intermediate outputs are themselves string editing instructions. Hence, the intermediate outputs do not need to be adapted and serve as the intermediate inputs to the next step. Hence, the only additional processing of the dataset needed for iterative decoding is the addition of the EOI token `[END]` at the end of the final output. An iterative decoding PCFG example with three intermediate steps is shown on the left hand side of Fig. 2.

To compare iterative decoding with seq2seq, we start with a basic transformer model with absolute position encodings as in the original architecture by Vaswani et al. (2017). This is so we can observe the advantages of iterative decoding in the absence of other compositional generalization biases. The results are reported in Table 1, where we see that in the random split of the data, the test accuracy of the seq2seq model is close to its training accuracy. This indicates that the model generalizes well to in-distribution samples. In contrast, in the productivity and systematicity splits there is a dramatic drop in test accuracy. Hence, the basic seq2seq transformer struggles to generalize compositionally. In the fourth row of Table 1, we see that for the iterative decoding transformer the gap between training and test accuracy is much smaller. This indicates that iterative decoding increases the ability of transformers to generalize compositionally.

Although iterative decoding helps with compositionality, the iterative decoding test accuracy is still low compared to their training accuracy. This implies that composing individual operations into complex instructions is only one facet of compositionality, which makes sense as decomposing complex instructions into individual operations only helps if the model can execute each operation correctly (see Appendix C for more details). Therefore, we repeat our experiments with transformers including modifications shown to increase compositional generalization in seq2seq learning (Ontañón et al., 2021): *relative attention* (Shaw et al., 2018) and *copy decoding* (Gu et al., 2016).

The results for seq2seq and iterative decoding with relative attention are shown in the second and fifth rows of Table 1. The relative attention radius is $r = 8$. As expected, relative attention helps both models with compositionality, particularly in the productivity split. Moreover, iterative decoding achieves a much better test accuracy, of over 90%, on both the productivity and systematicity splits. Results for transformers with relative attention and copy decoders are shown in the third and sixth rows of Table 1. Additionally, examples of errors made by both transformers can be found in Appendix D. While adding a copy decoder does not improve compositional generalization in the seq2seq transformer, it helps in iterative decoding, nearly closing the gap between training and test accuracy on the systematicity split, and leading to a 2% increase in test accuracy on the productivity split—which can be attributed to the ability to copy the longer strings in this split.

4.2 Cartesian product

In the Cartesian product dataset (Ontañón et al., 2021), the inputs are two vectors and the outputs are their Cartesian product. A more detailed description of this dataset can be found in Appendix A and an example of input-output pair is shown on the bottom left corner of Fig. 1. We consider four train-test splits. In the first split, both the training and test set consists of samples with up to five numbers and letters drawn i.i.d and split at random. This is the “easy split”. In the other splits, the training set is the same as in the first split, but the test set consists of examples with either more number, more letters, or both. These are “hard splits”, which we use to measure productivity.

To iteratively decode a Cartesian product, we first need to define what are going to be the iterative decoding intermediate steps. We consider two options as illustrated on the right corner of Fig. 2. The first is decoding one *row* at a time., which entails decoding the Cartesian product between one element from the first vector and all elements of the second vector at each intermediate step. The second is decoding one *token pair* at a time, which entails decoding only the product between one element of the first vector and one element of the second vector at each intermediate step. If the lengths of the input vectors are ℓ_1 and ℓ_2 respectively, decoding row by row requires ℓ_1 and token by token $\ell_1 \times \ell_2$ intermediate steps.

When decoding rows, the intermediate output

Table 2: Sentence accuracy achieved by the seq2seq and iterative decoding transformers with relative attention ($r = 8$) on the training set and on multiple test sets of the Cartesian product dataset.

Split	**Seq2seq**	**Iterative decoding**			
		short inputs		long inputs	
		row	token	row	token
Train (up to 5 numbers/letters)	100%	100%	100%	100%	100%
Test (up to 5 numbers/letters)	97.8%	100%	100%	100%	100%
Test (6 numbers, 5 letters)	14.3%	89.2%	**100%**	**100%**	**100%**
Test (5 numbers, 6 letters)	12.2%	0%	99.5%	0%	**100%**
Test (6 numbers/letters)	1.1%	0%	98.7%	0%	**100%**

at a given step is the current row. When decoding token pairs, the intermediate outputs are the current token pairs. Since these intermediate outputs do not carry any information about the next row or pair of tokens, they cannot be used as intermediate inputs. To construct intermediate inputs, we concatenate a copy of the original input—the two vectors separated by the `[SEP]` token—with a second separation token `[SEP2]` followed by either (i) the last intermediate output, or (ii) all the intermediate outputs so far. Scenario (i), which is illustrated on the right hand side of Fig. 2, yields *short intermediate inputs* where the last intermediate output acts as a "pointer" to where the decoding process stopped in the previous step. Scenario (ii) produces *long intermediate inputs*. While in both scenarios the intermediate outputs need to be concatenated to produce the final prediction, their prediction routines are different because scenario (i) only needs to append the current intermediate output to the input vectors to produce the next intermediate output, but scenario (ii) needs to append the current intermediate output to the last intermediate input to produce the next intermediate input.

To analyze the compositional generalization of seq2seq and iterative decoding transformers on the Cartesian product dataset, we consider the following experimental setup. Both transformers are trained on samples with up to five numbers and letters. Then, they are tested on the four different test sets described above: up to five numbers and letters; six numbers and five letters; five numbers and six letters; and six numbers and letters. The second, third and fourth test sets can be seen productivity tests. Additionally, we only report results for transformers with relative attention (relative radius $r = 8$) as they were the best performing architecture in our experiments.

The average training and test accuracies achieved by the seq2seq model, as well as by the iterative decoding model in the short/long intermediate input and row/token scenarios, are reported in Table 2. The seq2seq model (first column) achieves 100% accuracy on the training set and close to that on the "easy" test set with up to five numbers and letters. However, it pretty much fails in all of the "hard" test sets, implying that it cannot generalize compositionally when even one extra token is added to the input. The models that decode one row at a time (second and fourth columns) do slightly better as they achieve accuracy closer to the training accuracy in the test set with six numbers and five letters. However, they still fail at the test sets with six letters, which means that the iterative decoding transformer trained to decode one row at a time only generalizes well to calculating Cartesian products with a larger number of numbers or, equivalently, of rows.

In contrast, the iterative decoding models that decode one pair of tokens at a time (third and fifth columns) achieve close to 100% accuracy in all compositionally hard splits. This means that the iterative decoding transformer can only generalize to longer iterations when these iterations are the ones that were unrolled during training via iterative decoding. If we take a transformer that has been trained to decode rows, and add one more letter to the input—resulting in one more pair of tokens in each row—it will not be able to predict the extra pair because it has learned how to unroll more rows through iterative decoding but not more token pairs within a row. We thus conclude that in the Cartesian product dataset transformers struggle to learn iteration by themselves, i.e., without the help of iterative decoding. This is an important result because, despite being universal function approximators in theory (Yun et al., 2019), it sheds light onto what transformers can actually learn in practice. Finally, the difference between long and short inputs is not substantial, but longer inputs seem to be better, probably because they provide the transformer with more memory (i.e., more information

Table 3: Sentence accuracy achieved by the seq2seq and iterative decoding transformers with relative attention ($r = 8$) on the training and test sets of the MCD1 split of the CFQ dataset.

Split	Seq2seq	It. decoding
Train	99.8%	99.7%
Test	37.1%	32.5%

about the previous intermediate steps).

4.3 CFQ

Introduced by Keysers et al. (2019), the CFQ dataset consists of natural language questions and their corresponding SPARQL queries against the Freebase knowledge base. Keysers et al. (2019) introduces a number of compositionally hard splits of the CFQ dataset. In this paper, we focus on the MCD1 split. Additional details are described in Appendix A and an example of question and query are shown on the right hand side of Fig. 1.

Both PCFG and Cartesian product were well adapted for iterative decoding because, in PCFG, the recursive structure of the inputs makes it easy to define the intermediate steps, and in Cartesian product we have the flexibility to choose their granularity. On the CFQ dataset, defining the intermediate steps is less obvious. The natural choice is to define each intermediate output as a clause of the query as illustrated in Fig. 2. However, unlike in PCFG—where the order of the intermediate steps was defined by the recursion—and in Cartesian product—where the order of the tokens in the input determines the order of the intermediate steps—this ordering of the intermediate steps is not very "natural" because it is alphabetic. Hence, on CFQ transformers also have to learn how to sort.

To make learning this ordering easier for the transformer, we define long intermediate inputs for the intermediate steps. These intermediate inputs are constructed by concatenating the question with all of the previous intermediate outputs so far. As such, on CFQ the iterative decoding prediction routine is the same as for Cartesian product with long inputs: we append the current intermediate output to the previous intermediate input to obtain the next intermediate input, and, once the EOI token is predicted, concatenate all of the intermediate outputs to obtain the query prediction.

The average training and test accuracies are shown in Table 3, where we only report results for the best performing model in our experiments—relative attention with relative radius $r = 8$. While the accuracies achieved by our models are lower than the SOTA results reported in (Furrer et al., 2020) with pretraining, note that our goal is not to achieve the best possible performance for CFQ, but rather to compare transformers with the same architecture trained via seq2seq and iterative decoding. Both the seq2seq and the iterative decoding transformer exhibit low compositionality, however, iterative decoding performs worse than seq2seq. This reinforces the limitations of iterative decoding that we observed in Cartesian product, namely, that iterative decoding performance is largely dependent on how we define the intermediate steps.

In CFQ specifically, we also hypothesize that the worse performance of iterative decoding is tied to the alphabetical ordering of the clauses, as it does not follow naturally from the grammatical structure of the input. Even though sorting these clauses is something that both the seq2seq and the iterative decoding transformer have to learn how to do, in iterative decoding the transformer has to sort at all intermediate steps, so there are more opportunities to make mistakes. In other words, the error probability is larger in iterative decoding, because it compounds with each intermediate step.

5 Conclusions

This paper introduces iterative decoding as an alternative to seq2seq learning. Through numerical experiments on PCFG and Cartesian product, we demonstrate that, in general, seq2seq transformers do not learn iterations that are not unrolled. By unrolling them, iterative decoding improves transformer compositional generalization. However, iterative decoding has a limitation, which is that it depends on how the intermediate steps are defined. We hypothesize that their ordering is the reason why the seq2seq transformer outperforms iterative decoding on CFQ, as unnatural orderings require transformers to learn how to sort and iterative decoding may increase the overall sorting error probability. In our future work, we aim to apply iterative decoding strategies to more datasets and understand whether the number of iterative steps can be traded for transformer depth. We will further use iterative decoding to investigate the aspects of compositional generalization that transformers can and cannot learn. A next step is understanding the effect of the order of the intermediate steps and what that says about transformers' ability to sort.

References

Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. 2020. Etc: Encoding long and structured inputs in transformers. *arXiv preprint arXiv:2004.08483*.

Jacob Andreas. 2019. Good-enough compositional data augmentation. *arXiv preprint arXiv:1904.09545*.

Andrea Banino, Jan Balaguer, and Charles Blundell. 2021. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. 2021. The devil is in the detail: Simple tricks improve systematic generalization of transformers. *arXiv preprint arXiv:2108.12284*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Jerry Fodor. 2001. Language, thought and compositionality. *Royal Institute of Philosophy Supplements*, 48:227–242.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. 2020. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *arXiv preprint arXiv:2007.08970*.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the unknown words. *arXiv preprint arXiv:1603.08148*.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. 2020. Compositionality decomposed: how do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, et al. 2019. Measuring compositional generalization: A comprehensive method on realistic data. *arXiv preprint arXiv:1912.09713*.

Juyong Kim, Pradeep Ravikumar, Joshua Ainslie, and Santiago Ontañón. 2021. Improving compositional generalization in classification tasks via structure annotations. *arXiv preprint arXiv:2106.10434*.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature*, 521(7553):436–444.

Adam Liška, Germán Kruszewski, and Marco Baroni. 2018. Memorize or generalize? searching for a compositional rnn in a haystack. *arXiv preprint arXiv:1802.06467*.

Nikita Nangia and Samuel R Bowman. 2018. Listops: A diagnostic dataset for latent tree learning. *arXiv preprint arXiv:1804.06028*.

Benjamin Newman, John Hewitt, Percy Liang, and Christopher D Manning. 2020. The eos decision and length extrapolation. *arXiv preprint arXiv:2010.07174*.

Santiago Ontañón, Joshua Ainslie, Vaclav Cvicek, and Zachary Fisher. 2021. Making transformers solve compositional tasks. *arXiv preprint arXiv:2108.04378*.

Vinyals Oriol, Fortunato Meire, Jaitly Navdeep, C Cortes, ND Lawrence, DD Lee, M Sugiyama, and R Garnett. 2015. Pointer networks. *Advances in neural information processing systems*, 28:2692–2700.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. 2019. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077*.

Xingxing Zhang, Furu Wei, and Ming Zhou. 2019. Hibert: Document level pre-training of hierarchical bidirectional transformers for document summarization. *arXiv preprint arXiv:1905.06566*.

Jinhua Zhu, Yingce Xia, Lijun Wu, Di He, Tao Qin, Wengang Zhou, Houqiang Li, and Tie-Yan Liu. 2020. Incorporating bert into neural machine translation. *arXiv preprint arXiv:2002.06823*.

A Dataset Details

A.1 PCFG

In all splits of the PCFG dataset, the input data consists of string editing instructions with four types of tokens: unary operation tokens (e.g., `reverse`), binary operation tokens (e.g., `append`), a string separation token "`,`" (to separate arguments of binary operations), and string elements (e.g., `B10`, `D2`). The output data consists of the strings resulting from the application of the operations; see the top left corner of Fig. 1 for an example.

The main challenge of the PCFG dataset is that it requires learning ten string editing operations, some of which are very similar. The unary operation `echo`, for instance, only differs from `copy` by repeating the last element of the string. While transformers generally achieve good performance in the random split of the PCFG dataset, the productivity and systematicity splits are harder because transformers tend to learn mappings which do not exploit compositional symmetries. In particular, a key difficulty of the productivity split is that the model needs to learn to do "recursion" and apply an arbitrary number of operations when input examples grow in length. In some cases, the strings to modify can also be very long, which places an additional capacity burden on transformers by requiring them to learn how to copy strings.

A.2 Cartesian Product

In the Cartesian product dataset, the input consists of two vectors. The first is a vector of numbers. The second is a vector of letters, separated from the numbers by the special token `[SEP]`. Both numbers and letters are picked at random, without repetition, from the decimal digits and the first ten letters of the alphabet respectively. The output is then the Cartesian product between the first and second input vectors. An example of input-output pair is shown on the bottom left corner of Fig. 1.

The productivity splits of the Cartesian product dataset are remarkably hard; even transformers with some compositional generalization ability in other mathematical datasets have been seen to fail (Ontañón et al., 2021). This is due to the fact that, in order to solve Cartesian products, models need to learn to execute two nested loops. Moreover, the output is quadratic on the size of the inputs. For models that have to learn to predict an end-of-sequence token, extrapolating to longer sequences than those seen during training has been shown to be difficult (Newman et al., 2020).

A.3 CFQ

Introduced by Keysers et al. (2019), the CFQ dataset consists of natural language questions and their corresponding SPARQL queries against the Freebase knowledge base. Hence, it can be used to perform semantic parsing by taking the questions as the inputs and the queries as the outputs. One of the difficulties of CFQ is that some of its examples require solving Cartesian products. As such, in CFQ transformers may face similar challenges to the ones faced in Cartesian product. Another difficulty relates to the ordering of the clauses in the SPARQL query, which are ordered alphabetically by convention. Not only is this ordering different than the one implied by the order of the tokens in the question, it also requires transformers to learn how to sort.

B Implementation Details

Across all experiments, the transformer parameters were the same as in the original implementation in (Vaswani et al., 2017), including the learning rate schedule. All experiments were run on machines with a single CPU and four Tesla V100 GPUs with batch size 64 per device.

For each dataset, and for both the seq2seq and iterative decoding splits, the vocabulary size, the size of the training and test sets, and the total number of training steps is shown in Table 4. The iterative decoding vocabularies are larger due to the addition of special start, end and separation tokens. The number of training samples is larger for the iterative decoding splits because they include all intermediate steps. To make for a fair comparison, the number of training steps is the same for iterative decoding and seq2seq.

C Additional Iterative Decoding Results for PCFG

To assess the advantages of iterative decoding under no other sources of compositional generalization, we consider the base transformer (i.e., without relative attention and copy decoder) and analyze its performance on PCFG per number of string editing

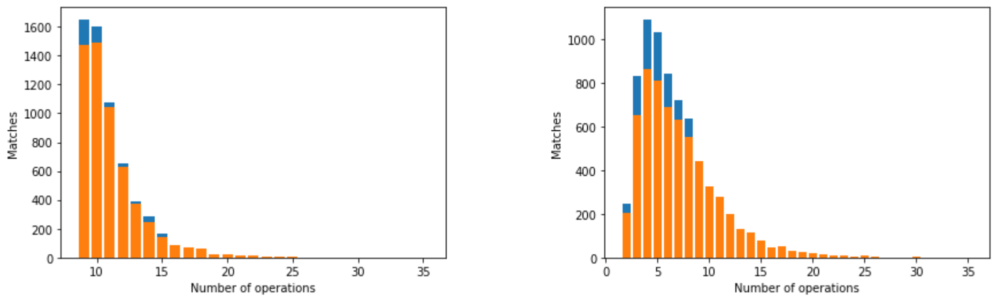


Figure 4: Number of correct predictions versus number of string operations in the input for seq2seq (orange) and iterative decoding (blue), on the productivity (left) and systematicity (right) splits of PCFG.

Table 4: Vocabulary size, training and test samples, and number of training steps for all seq2seq and iterative decoding datasets.

	Seq2seq			**Iterative decoding**			
	vocabulary	train	test	vocabulary	train	test	steps
PCFG-i.i.d.	534	82662	9721	535	426558	9721	33325
PCFG-prod.	534	81010	11333	535	346222	11333	27049
PCFG-syst.	534	82168	10175	535	403808	10175	31458
Cartesian-row	26	200000	1024	28	600036	1024	4688
Cartesian-token	26	200000	1024	28	1801869	1024	14077
CFQ	181	95743	11968	186	682470	11968	53318

operations. Namely, in Fig. 4 we plot the number of correct predictions achieved by seq2seq (orange) and iterative decoding (blue) on the productivity and systematicity splits of PCFG. We observe that, in the productivity split, the performance improvement comes mostly from samples with a small number of string editing instructions. Consistent with Table 1, without any other form of compositional generalization bias iterative decoding is more helpful with systematicity.

We draw a similar conclusion from Fig. 5, which plots the error per intermediate step ((test error)$^{1/N}$, where N is the number of operations) versus the number of operations in the input for both splits. The error per intermediate step can be seen as the probability of making a mistake at any given intermediate step. On the left, this error approaches one for a smaller number of operations than on the right, indicating that errors compound faster in the productivity split. Interestingly, this Fig. also corroborates our claim from Sec. 4.1 that decomposing complex instructions into individual operations only helps if the model can execute each operation correctly. In other words, composing individual operations into complex instructions is only one facet of compositionality, but one with which iterative decoding helps.

D Error Examples for PCFG

Tables 5 and 6 show examples of wrong predictions made by the seq2seq and iterative decoding transformers with relative attention and copy decoding.

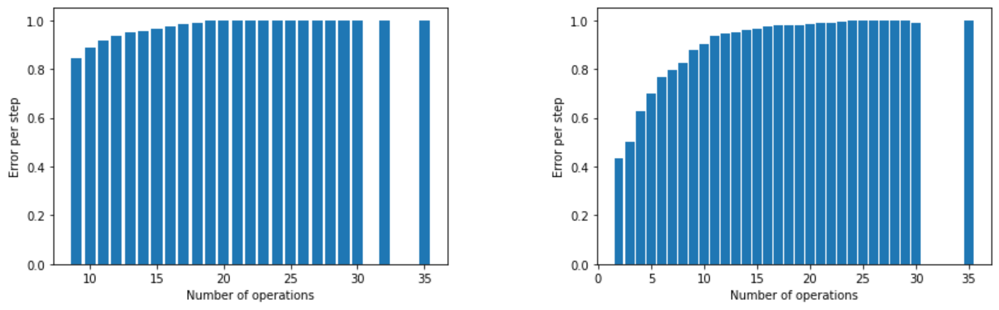


Figure 5: Error per intermediate step versus number of string operations in the input for the iterative decoding transformer on the productivity (left) and systematicity (right) splits of PCFG.

Table 5: Productivity error examples for seq2seq and iterative decoding.

Seq2seq

Input	`shift repeat prepend append Z6 A8 C12 U1 T5 , repeat repeat prepend N8 K15 , S18 B4 , repeat reverse shift echo I2 V2 F5`
True output	`F5 F5 V2 I2 F5 F5 V2 Z6 A8 C12 U1 T5 S18 B4 N8 K15 S18 B4 N8 K15 S18 B4 N8 K15 I2 F5 F5 V2 I2 F5 F5 V2 Z6 A8 C12 U1 T5 S18 B4 N8 K15 S18 B4 N8 K15 S18 B4 N8 K15 S18 B4 N8 K15 I2`
Prediction	`F5 F5 V2 I2 F5 F5 V2 I2 F5 F5 V2 Z6 A8 C12 U1 T5 S18 B4 N8 K15 S18 B4 I2 F5 V2 I2 F5 F5 V2 S18 B4 N8 K15 S18 B4 I2 F5 V2 I2 F5 F5 V2 Z6 A8 C12 U1 T5 S18 B4 N8 K15 S18 B4`

Iterative decoding

Input	`remove_first repeat repeat swap_first_last swap_first_last R9 Q20 N10 , shift repeat echo repeat V17 V14 E4 A7`
True output	`V14 E4 A7 V17 V14 E4 A7 A7 V17 V14 E4 A7 V17 V14 E4 A7 A7 V17 END`
Prediction	`V14 E4 A7 V17 V14 E4 A7 A7 V17 V14 E4 A7 V17 V14 E4 E4 A7 V17 END`

Table 6: Systematicity error examples for seq2seq and iterative decoding.

Seq2seq

Input	`swap_first_last remove_first F10 E6 T18 , echo append reverse J18 H10 K12 X11 , swap_first_last repeat remove_second copy U4 E15 I2 , X11 C6 W3`
True output	`U4 K12 H10 J18 I2 E15 I2 U4 E15 U4 X11`
Prediction	`U4 K12 H10 J18 I2 E15 I2 U4 E15 U4 U4 E15 X11`

Iterative decoding

Input	`repeat remove_second repeat prepend echo reverse echo prepend Q7 C15 I14 H13 , P9 O5 A12 K19 , remove_second copy copy G4 W3 U10 S4 , swap_first_last echo repeat shift swap_first_last I7 S5 Z16 K13 Q9 , copy T16 X18 E15`
True output	`G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 END`
Prediction	`G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 P9 G4 W3 U10 S4 H13 H13 I14 C15 Q7 K19 A12 O5 P9 K19 END`