Compositional Skills of Recurrent Neural Networks

Lukas Muttenthaler
University of Copenhagen
Department of Computer Science
mnd926@alumni.ku.dk

Abstract

Compositionality is the ability to represent complex concepts by combining simple parts. In this replication study, we investigated compositional skills of different recurrent sequence-to-sequence (seq2seq) neural architectures via zero-shot generalization tasks. We examined simple versions of those models, and versions that were equipped with an attention mechanism or an encoder that processes sequences forward and backward in time. In contrast to the original paper, we performed mini-batch instead of online training. This did not only lead to a significantly lower computational time but appears to help otherwise poorly-performing networks such as RNNs to solve the zero-shot generalization tasks. We can confirm that simple RNNs indeed fail even when differences between train and test sets are small, but we observe much better performance ($\sim 8.9\%$) than reported in the original paper ($< 1.5\%$). LSTMs and GRUs perform incredibly well on simple zero-shot generalization tasks but fail equally when compositional differences between train and test sets are too vast. This is in line with the results of the original paper.

1 Introduction

Recently, sequence-to-sequence (seq2seq) Recurrent Neural Networks (RNNs) have become widely used in Machine Translation (MT) (Sutskever et al., 2014; Bahdanau et al., 2014). They outperform rule-based SMT systems, sometimes even by a large margin, dependent on the MT task. Due to the fact that sequence-to-sequence models have become the default architecture for MT (Graves, 2013) and dominate the state-of-the-art (SOTA), it is crucial to examine their compositional skills, and better understand their behaviour. Recently, a number of studies shed light on compositionality in neural models to investigate their zero-shot generalization skills (e.g., Lake and Baroni, 2017) or representations of linguistic symbols (e.g., Andreas, 2019).

In this study, which can be seen as a slightly modified replication of (Lake and Baroni, 2017), we analyse not only compositional skills of seq2seq models but also conduct thorough error analyses, even for simple zero-shot generalization tasks where only vanilla RNNs appear to fail. We motivate to foster understanding of models that perform poorly, opposed to just discarding them as architectures that fail to understand the task. We suspect that even simple RNNs, although apparently worse than models with sophisticated cell states such as LSTMs (Hochreiter and Schmidhuber, 1997) or GRUs (Chung et al., 2014), understand the task better than reported by Lake and Baroni (2017) when trained differently (e.g., weight updates with larger gradients, TF curriculum that prepares the model for inference time), and might even perform on par when equipped with attention mechanisms or expanded in time (Schuster and Paliwal, 1997).

2 Method

In this section, we outline the modifications we applied to the general training procedure followed by an explanation of the models we trained and evaluated on the zero-shot generalization tasks.

2.1 Mini-batch training

Instead of online training, we performed mini-batch training. This was done to significantly reduce computational time, optimize our training procedure for GPUs, and update the weights of our models with larger gradients.

Since mini-batch training requires sequences within a batch to be of equal length, we padded each word sequence $x_i^T = [x_i^1, x_i^2, ..., x_i^T]$ with a special <PAD> token until the maximum sequence length in the respective training set. Hence, each x_i in a batch had the same number of timesteps T. Each mini-batch consisted of 32 word sequences (i.e., batch size $= 32$).

Furthermore, to neither compute the loss for <PAD> tokens nor update the weights for the corresponding vector representations, we were required to mask the loss. Since <PAD> tokens were indexed with 0, we computed the loss only for tokens that had a non-zero index.

2.2 Teacher Forcing scheduler

In the original paper (Lake and Baroni, 2017), TF was applied with a ratio of 0.5 per iteration. We did not deem this approach appropriate as a scheduler that applies TF by chance does not well prepare the model for inference time where any model f is forced to act as a completely auto-regressive network (i.e., uses its own predictions as inputs).

Hence, we implemented a TF scheduler that reduces the TF ratio linearly over time. We started off with a very high TF ratio of $p = .95$ per iteration, and linearly decreased the ratio towards the end to better prepare the model for inference time where no TF is applied.

Moreover, due to the fact that we perform mini-batch training opposed to online training, it appeared reasonable to us to linearly decrease the TF ratio over epochs.

2.3 Models

We implemented the same models as Lake and Baroni (2017). That is, a simple Elman RNN (Elman, 1991), an LSTM (Hochreiter and Schmidhuber, 1997), and a GRU (Chung et al., 2014). We compared those rather simple models against models with future-aware sequence processing mechanisms, namely bidirectional encoders (Schuster and Paliwal, 1997). For many sequence modelling tasks, letting a neural network look into the future has proven to be highly successful (Graves and Schmidhuber, 2005). Hence, we deemed it appropriate to equip our recurrent encoders with bidirectional hidden layers. See Appendix (4) for our implementation of a bidirectional encoder.

Moreover, we implemented models with attention mechanisms. We compared the general attention mechanism developed by Bahdanau et al. (2014) against the multiplicative attention introduced by Luong et al. (2015), and observed in preliminary experiments that there is no notable difference in performance between the two. Hence, we proceeded with Luong-Attention for all experiments as it is widely adopted in the recent Machine Learning (ML) literature, and leveraged in the sequel of the paper we are meant to reproduce (Lake, 2019).

In addition, we performed gradient clipping during training whenever the norm of the gradient was < 10 to avoid potential exploding gradient problems, and initialized the encoder hidden states of our models with the uniform distribution introduced in (Glorot and Bengio, 2010). All models were developed in PyTorch (Paszke et al., 2017). Our code is publicly available. [1]

3 Experiments

We report results for experiments 1a, 1b, 2 and 3 following the implementations in (Lake and Baroni, 2017), with minor modifications if appropriate.

3.1 Experiment 1a

In this experiment, we exploited the main train-test split as used in the original paper. [2] Hence, 80% (16,728 source-target pairs), and 20% (4,182 source-target pairs) of the full dataset were used for train and test respectively. We scrutinized models of different complexities. Results of the respective top-performing models per neural architecture are reported in Table 1.

Model	Embedding	Hidden	Layers	Dropout	Accuracy
RNN	10	100	2	0.5	8.87%
LSTM	20	100	2	0.25	99.16%
GRU	20	100	2	0.5	98.7%
Attn-RNN	10	100	2	0.5	96.33%
Attn-LSTM	20	100	2	0.25	96.78%
Attn-GRU	20	100	2	0.5	97.58%
Bi-RNN	10	100	2	0.5	18.4%
Bi-LSTM	20	100	2	0.25	**99.5%**
Bi-GRU	20	100	2	0.5	99.4%

Table 1: Exact-match test accuracy reported for the respective best models per neural network architecture for experiment 1.

Similarly to Lake and Baroni (2017), we observe that, among unidirectional architectures, a two-layer LSTM with 100-dimensional hidden units, 20-dimensional embeddings and a layer dropout rate of 0.25 achieves the highest exact-match accuracy during inference (99.16%). Moreover, we can confirm that attention does not help LSTMs and GRUs. This is most likely due to two reasons. Firstly, both LSTMs and GRUs consist of sophisticated memory and cell states (Hochreiter and Schmidhuber, 1997; Chung et al., 2014), and

[1] https://github.com/LukasMut/ATNLP
[2] https://github.com/brendenlake/SCAN

therefore might not benefit greatly from an additional memory mechanism for such a simple task. Secondly, simple versions of LSTMs and GRUs achieved almost perfect performance during inference ($\sim$ 99%). This means, that performance can simply not be improved much further (i.e., `ceiling effect`).

On the other hand, letting the model look into the future, that is adding additional backward passes per layer, enhances performance and improves exact-match accuracy to 99.5% and 99.4% for Bi-LSTMs and Bi-GRUs respectively.

What's particularly striking is that our simple Elman RNNs did not perform as poorly as reported in (Lake and Baroni, 2017). Vanilla RNNs show an $\sim$ 800% relative and $>$ 7% absolute improvement over the RNN's scrutinized in the original paper. RNNs with a bidirectional encoder even achieve an exact-match accuracy of 18.4%. Those results are still poor compared to LSTMs and GRUs but indicate that RNNs might learn more useful representations than outlined in the original paper, if trained with the appropriate approach and informed deeply in space and time about the input sequence.

RNNs with an additional attention mechanism clearly outperform vanilla RNNs and achieve an exact-match accuracy (96.33%) comparable to the performance of LSTMs and GRUs. This performance is again stronger than the performance of RNNs with attention in (Lake and Baroni, 2017).

Error analysis We investigated whether RNNs fail across all sequences, and are not capable of solving the zero-shot generalization task at all, as reported in the original paper, or whether the length of a sequence x_i plays a crucial determinant for the RNN's poor performance. To better understand what causes the RNN's erroneous predictions on this simple task, we analysed exact-match accuracy as a function of command or action sequence length for both simple and bidirectional RNNs. Furthermore, we examined the learning curve until model convergence for command and action sequences dependent on their respective length. The results of the analyses are depicted in Figure 1 and Figure 2 in the Appendix.

What can be inferred from both Figure 1 and Figure 2 is that RNNs clearly fail at longer sequences. They learn much slower and notably less about longer compared to shorter sequences (see Figure 2), and perform poorly on the former at inference time (see Figure 1). This indicates that RNNs suffer from a lack of memory for long-term dependencies. Furthermore, adding recurrent layers that process sequences backwards in time clearly helps simple RNNs, without memory or cell states, to process and translate any word sequence x_i. This hold in particular for longer sequences. Interestingly, bidirectionality enhances exact-match accuracy more notably for shorter ($4-15$ tokens) and longer (>21 tokens) action sequences compared to average long ($16-20$ tokens) action sequences respectively (see Figure 1a). For command sequences, bidirectionality enhances performance significantly for all sequences that contain >4 tokens (see Figure 1b).

Ablations To justify our choices with regard to the training procedure, we conducted two crucial ablation experiments. The results can be found in the Appendix (4).

3.2 Experiment 1b

In this experiment, we reduced the number of distinct samples shown during training, and hence provided splits with notably less training data than in the main split. We trained the model with $n \in \{1\%, 2\%, 4\%, 8\%, 16\%, 32\%, 64\%\}$ distinct samples of the original train split. For each of the seven different splits, the training presentations were set to 100k for online training and to 20k per epoch for mini-batch training. Contrary to the original paper (Lake and Baroni, 2017), we did not sample distinct pairs from the entire dataset but from the train set ($16,728$ source-target pairs) only. This was done to make sure that the model has never seen the test set at any given point in time prior to inference. Hence, our results differ slightly from the results reported in the original paper. Results are depicted in Figure 3 in the Appendix.

Similarly to Lake and Baroni (2017), we observe that our model performs poorly ($\sim$ 4.4% for online and $\sim$ 2.6% for mini-batch training respectively) when it has only seen 1% of distinct commands during training (see Figure 3 in the Appendix). However, we cannot report vast improvements of our model when 2% of distinct commands were sampled during training. Our model achieves $\sim$ 20.1% for online and $\sim$ 17% for mini-batch training, compared to $\sim$ 54% as reported in the original paper. This difference might have multiple reasons (e.g., due to limited computational resources we did not run our model over 5 different random seeds as Lake and Baroni (2017)). The most likely reason for the worse performance, however, is that we did not sample distinct commands from the entire dataset but only from the train set.

For online training, it seems as if $\sim 4\%$ of distinct commands shown during training suffice to let the model generalize on new commands ($\sim 77.4\%$ on test set). Mini-batch training, however, requires $\sim 8\%$ of distinct commands shown during training to achieve similar performance on the test set (~ 83.5). Interestingly, mini-batch training improves zero-shot generalization more than online training when the model was provided with more distinct commands during training. When 32% of distinct commands were used during training, our best-performing model achieved $\sim 97.3\%$ exact-match accuracy on the test set for online training and $\sim 98.4\%$ for mini-batch training.

3.3 Experiment 2

This experiment was significantly more difficult than the previous one. The models were trained on short sequences (commands requiring sequences of $<= 24$ actions), and must generalize to notably longer sequences (commands requiring actions sequences between 26-50) at inference time. The number of tokens is shifted by $+2$ since we, contrary to Lake and Baroni (2017), count the special <SOS> and <EOS> tokens. With this experiment, we investigated a more `systematic` form of generalization, and examine compositionality under both more challenging and realistic conditions. We used the train-test split as provided by the authors. [3] Hence, all models were trained on 16,990 source-target pairs and tested on 3,920.

As can be inferred from Table 2, even complex models had apparent difficulties to generalize to longer action sequences. This time, attention did not help at all. For some models, it even decreased performance, and thus was harmful. Bidirectionality, however, enhanced the performance of every model. This might be due to the fact, that (additionally) reading sequences in reverse minimizes time lags between input and output sequences (Schuster and Paliwal, 1997; Sutskever et al., 2014). Our top-performing model achieved an exact-match accuracy score of 15.3%.

A more fine-grained analysis is displayed in Figure 4. Depicted is the performance of our top-performing model according to Table 2. The plots clearly show that the model failed spectacularly on

[3] `https://github.com/brendenlake/SCAN`

Model	Embedding	Hidden	Layers	Dropout	Accuracy
RNN	10	100	2	0.5	0.38%
LSTM	20	100	1	0.5	11.96%
GRU	20	100	1	0.25	12.24%
Attn-RNN	10	100	2	0.5	3.88%
Attn-LSTM	20	100	2	0.5	11.4%
Attn-GRU	20	100	1	0.5	12.63%
Bi-RNN	10	100	2	0.5	3.67%
Bi-LSTM	20	100	1	0.25	**15.3%**
Bi-GRU	20	100	1	0.5	13.04%

Table 2: Exact-match test accuracy reported for the respective best models per neural network architecture for experiment 2.

commands that required longer action sequences. Almost the entire performance can be explained through the model's generalization to sequences with 26 or 27 tokens respectively. If this was just due to the fact that the model was trained only on short sequences, then doubling the length of each training example would resolve the problem.

Is it sequence length? Hence, we removed the conjunctions `and` and `after`, and extended each training sequence with a copy of itself. Prior to conducting this experiment, we investigated on experiment 1a whether the `and` and `after` conjunctions are indispensable to learn the semantic mapping, or can be dropped. To inspect this, we were required to reverse the order of action sequences, whenever an `after` appeared in the corresponding command. Thus, instead of $[x_1 \text{ after } x_2] = [x_2][x_1]$, the model had to learn $[x_1\ x_2] = [x_1][x_2]$. For `and`, we simply removed the conjunction. Performance did not drop for experiment 1a, when removing the conjunctions from the sequences. However, doubling the length of word sequences did not improve performance whatsoever. Thus, we must assume that poor performance in this task is not due to lower sequence lengths in the train set but rather due to a lack of compositional skills. Again, we did not want to modify anything at inference time. Therefore, we only applied changes to the training procedure.

3.4 Experiment 3

In this experiment, the model is exposed to primitive commands indicating a specific action. The model is trained on two different datasets, where it is required to either generalize from `turn left` or `jump`. Due to the fact that both `turn left` and `turn right` as well as `walk`, `run`, `look`, and `jump` are distributionally equivalent, it is not necessary to test the remaining commands.

The decisive difference between the two experiments lies in the composition of the primitive commands in the training examples. Whereas `turn left` is presented in context with other actions, `jump` is displayed to the model only in isolation.

This difference in compositionality between the two training distributions is strongly reflected in the performance of our models on the two variants. Similarly to Lake and Baroni (2017), many models generalized well to the test set when trained on the `turn left` variant of the experiment. Our best model, a Bi-GRU with two layers, 100 hidden units, and a layer dropout of 0.5, achieved remarkable 93.28%. In contrast, the same model did not generalize at all to the test set when trained on the `jump` variant (0.0%). The best model for this experiment, a GRU with attention, performed with 0.07% not notably better.

To understand the source of the errors in more detail, we examined both the exact-match accuracy and total number of errors dependent on the individual command components. Figure 5 clearly shows that the model generalized well to components that required a translation of the `turn left` command but performed poorly on components that included an action u[4]. Interestingly and almost surprisingly, most errors by the model are made for `turn left` or a conjunction of the latter. This is in line with (Lake and Baroni, 2017).

Furthermore, we calculated cosine similarities between the final encoder hidden states of the four primitives `run`, `jump`, `run twice` and `jump twice` trained compositionally and in isolation respectively, and all other training commands (see Table 4). It seems as if `jump` was represented in isolated vector space as there is no command that is particularly close ($> .7$) according to its cosine. A similar picture can be drawn for `jump twice` which is close to the primitive command `jump` but not to any other action present in the set u. Rather surprisingly, its representation is similar to commands that contain `turn left` or `turn right`. However, as expected the hidden vectors of `run` and `run twice` have particularly close neighbours (cosine $> .9$), and thus do not have isolated mappings. This is most likely due to the fact that `run` was trained compositionally.

Additionally, we computed cosine similarities between the final encoder hidden states of the four representative commands (see above) and all other training commands for the `turn left` dataset, where all commands were trained compositionally. As depicted in Table 5, both `jump` and `jump twice` have close neighbours (cosine $> .9$) of which all contain actions in u. Contrary to the vector similarities in the former dataset, both cosines and neighbours are highly similar across all four representative commands. This reflects a better understanding of the model for compositionality, when trained on the `turn left` version.

Furthermore, similar to Lake and Baroni (2017), we added $n \in \{1, 2, 4, 8, 16, 32\}$ different composed commands to the train set in the `jump`-generalization task. As expected, model performance scales with the number of added commands n (see Figure 6). The top-performing model, a two-layer Bi-GRU with Attention, even achieved almost perfect exact-match accuracy ($\sim 99.1\%$) when 32 composed commands were added to the train set in combination with the primitive `jump`. Note that model performance varied notably with different composed commands used as additions.

4 Conclusions

Although we did not strictly follow the implementations in the original paper and made modifications to the models (e.g., bidirectionality) or the training procedure (e.g., mini-batch training) where appropriate and helpful, we could reproduce or even improve upon the results. Similarly to Lake and Baroni (2017), we observed that models fail spectacularly, if the data distributions between train and test sets differ too notably (see Section 3.3 and 3.4).

However, if a model is exposed to just a few composed commands that resemble the test data distribution, performance increases significantly. This shows, that recurrent neural networks - augmented with memory states - are capable of quickly learning a semantic mapping function when distributional differences are not too large, even when only little training data is available.

[4]$u = \{jump, look, run, walk\}$

References

Jacob Andreas. 2019. Measuring compositionality in representation learning. *CoRR*, abs/1902.07181.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Jeffrey L Elman. 1991. Distributed representations, simple recurrent networks, and grammatical structure. *Machine learning*, 7(2-3):195–225.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

A. Graves and J. Schmidhuber. 2005. Framewise phoneme classification with bidirectional lstm networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2047–2052 vol. 4.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Lstm can solve hard long time lag problems. In *Advances in neural information processing systems*, pages 473–479.

Brenden M. Lake. 2019. Compositional generalization through meta sequence-to-sequence learning. *CoRR*, abs/1906.05381.

Brenden M. Lake and Marco Baroni. 2017. Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks. *CoRR*, abs/1711.00350.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch.

M. Schuster and K. K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

Appendix

Method

Bidirectional RNN Bidirectional recurrent neural networks recursively process a word sequence $x_i^T = [x_i^1, x_i^2, ..., x_i^T]$ forward and backward in time. As such, both past-aware $\overrightarrow{h_i} = [h_i^1, h_i^2, ..., h_i^T]$ and future-aware $\overleftarrow{h_i} = [h_i^T, h_i^{T\text{-}1}, ..., h_i^1]$ hidden representations of an input sequence x_i^T were computed in every recurrent hidden layer h_i. To compute the final hidden state h_i, we summed the forward $\overrightarrow{h_i}$ and backward $\overleftarrow{h_i}$ hidden states.

$$h_i = \overrightarrow{h_i} + \overleftarrow{h_i} \tag{1}$$

For LSTMs, the same computation as outlined above was performed for cell states c_i. We did not concatenate the forward and backward hidden states to make sure that encoder and decoder networks consist of the same number of parameters, namely hidden units. For every training iteration, the decoder was initialized with the encoder's summed forward and backward hidden representations h_i.

Ablations

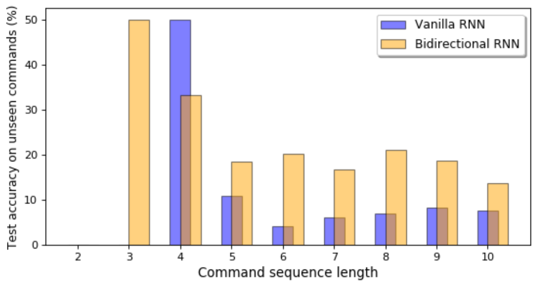
Firstly, we compared mini-batch vs. online training against each other. To appropriately scrutinize, we held all other variables constant. Results are depicted in Table 3. Mini-batch clearly outperforms online training for an RNN with attention. We assume this is due to weight updates with larger, and hence more informative gradients, combined with training over more epochs.

Seed	Online	Mini-batch
1	22.4%	97.7%
2	33.0%	93.4%
3	21.8%	98.6%
4	14.7%	96.3%
5	21.6%	93.9%

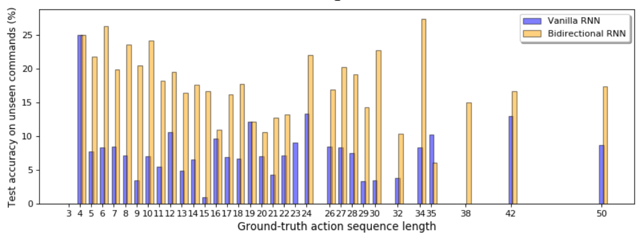
Table 3: Accuracies for Experiment 1a with 5 different seeds comparing mini-batch and online training for RNN with Attention model, TF ratio maintained at 0.5

Secondly, we compared our TF scheduler (see Section 2.2) against a constant TF ratio of $p = .5$ across iterations as was utilized in the original paper. We could not observe any notable differences in performance between the two TF approaches for mini-batch training. Hence, we must conclude that our TF scheduler did not contribute to the enhanced performance as we hypothesized previously.

Experiment 1a

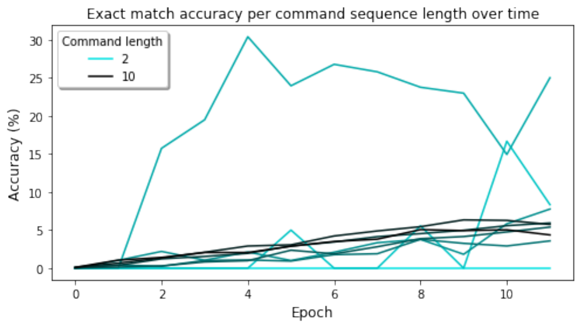


(a) Command sequences

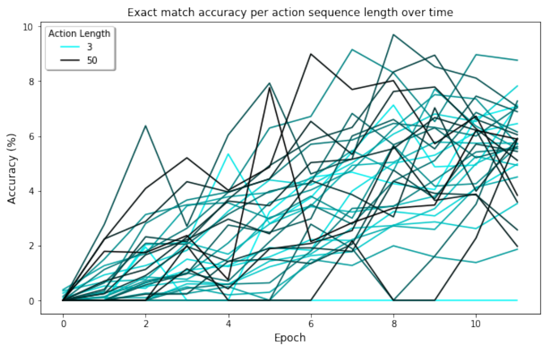


(b) Action sequences

Figure 1: Exact-match accuracy at inference time as a function of command (a) or action (b) sequence length for vanilla RNNs and bidirectional RNNs.



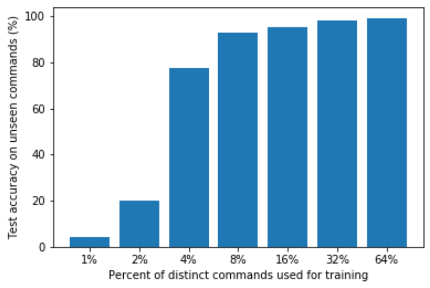
(a) Command sequences



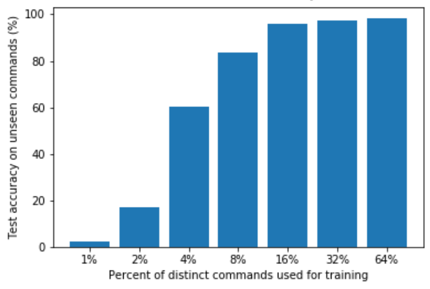
(b) Action sequences

Figure 2: Exact-match accuracy per command (a) or action (b) sequence length during training as a function of time. The model that was trained in this experiment was a simple Elman RNN. The darker the color of a line, the longer was the respective sequence.

Experiment 1b



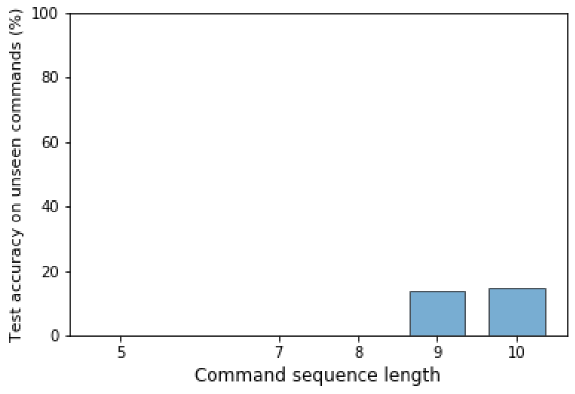
(a) Online training



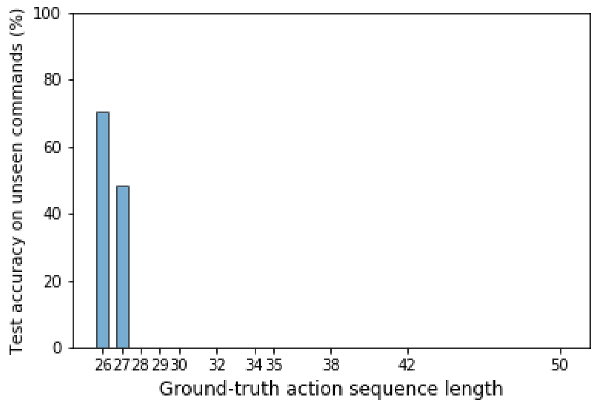
(b) Mini-batch training

Figure 3: Exact-match accuracy at inference time for a one-layer LSTM with hidden size 100 and no attention after online and mini-batch training respectively.

Experiment 2



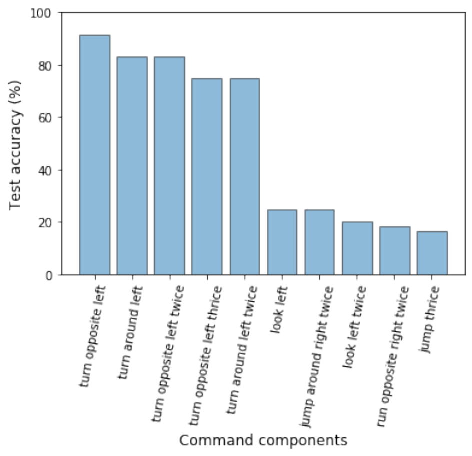
(a) Command sequences



(b) Action sequences

Figure 4: Exact-match accuracy at inference time as a function of command (a) or action (b) sequence length for a bidirectional LSTM.

Experiment 3



(a) Online training

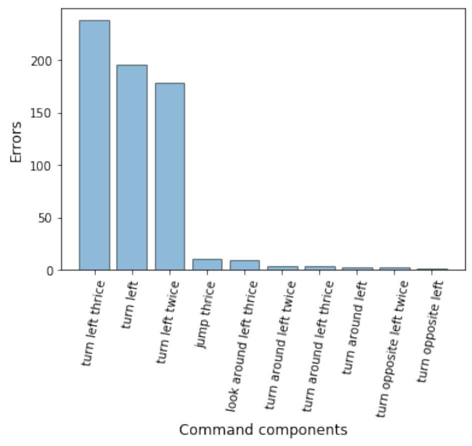


Figure 5: Exact-match accuracy a) and total number of errors b) per command-component at inference time for a one-layer bidirectional GRU with 100 hidden units. In both bar graphs the top and bottom five commands with respect to either exact-match accuracy a) or total number of errors b) are depicted. For this investigation, we did not inspect our best but a somewhat decently performing model to get a more thorough picture of model errors.

RUN		JUMP		RUN TWICE		JUMP TWICE	
command	cosine	command	cosine	command	cosine	command	cosine
run and run	.924	*turn right*	.691	run thrice	.944	jump	.851
run after run	.923	*turn left*	.650	run and run twice	.940	turn right twice	.763
run right and run	.910	*run after turn right*	.623	walk right and run twice	.934	turn opposite right	.755
run left and run	.907	*run after turn left*	.622	run left and run twice	.933	turn left twice	.754
walk right and run	.904	*turn opposite left after turn right*	.615	walk left and run twice	.931	turn opposite left	.754

Table 4: Final encoder hidden state vectors from the dataset where models were required to generalize from the command `jump`. Representative commands and nearest training commands according to the cosines between respective final Encoder hidden states. The primitive ”jump” was trained in isolation, whereas ”run” was trained compositionally. Italics denote medium large similarities (cosine $< .7$).

RUN		JUMP		RUN TWICE		JUMP TWICE	
command	cosine	command	cosine	command	cosine	command	cosine
run after run	.929	run after jump	.912	run after run twice	.938	jump twice and jump twice	.926
walk twice and run	.924	jump twice and jump	.912	walk twice and run twice	.936	walk right and jump twice	.925
look twice and run	.920	walk twice and jump	.907	walk left and run twice	.932	walk left and jump twice	.924
run twice and run	.914	look twice and jump	.905	look after run twice	.931	walk twice and jump twice	.924
look after run	.911	run twice and jump	.898	look twice and run twice	.930	look twice and jump twice	.923

Table 5: Final encoder hidden state vectors from the dataset where models were required to generalize from the command `turn left`. Representative commands and nearest training commands according to the cosines between respective final Encoder hidden states. All commands were trained compositionally.

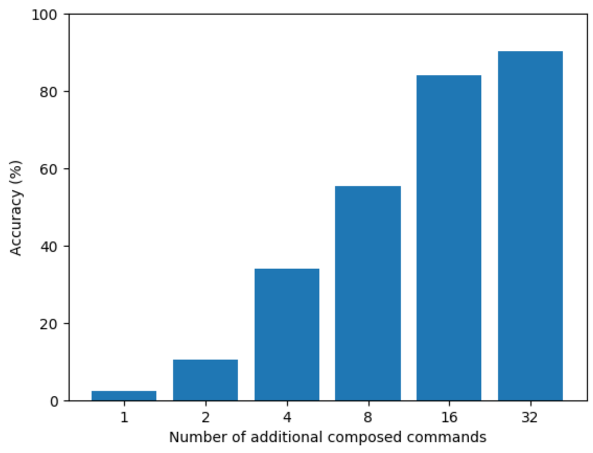


Figure 6: Exact-match accuracy at inference time as a function of added compositional commands in combination with the primitive `jump`. The evaluated model was a two-layer Bi-LSTM with Attention and a hidden size of 100.