A Reproduction of Sequence-to-Sequence Recurrent Networks for SCAN Tasks

Xuemin Duan
University of Copenhagen
npm838@alumni.ku.dk

Abstract

Systematic compositionality is a basic and essential capability of humans to understand and produce new utterances based on prior knowledge. This paper reimplemented and reproduced several experiments and results that were introduced by Lake and Baroni. We discussed the compositional generalization of sequence to sequence recurrent networks and proposed a novel method to improve model performance. We found that the networks perform poorly on the dataset which is constrained from the perspective of compositionality, which may indicates that current recurrent networks have no systematic compositionality.

1 Introduction

In recent years, deep learning models have achieved great success in various tasks. However, unlike humans can learn quickly from a small number of samples, the success of deep learning models still relies on large amounts of training data. There is a question that whether the deep learning models have the capability to learn as humans. The key point is whether deep learning models can have compositional generalization capability. Systematic compositionality is a basic and essential capability of humans to understand the infinite world of natural language with limited linguistic elements by understanding unknown combinations of known elements (Liu et al., 2020).

(Lake and Baroni, 2018) proposed the **S**implified version of the **C**omm**AI** Navigation (SCAN) tasks, which is to translate natural language commands such as "turn left twice" into navigation actions such as "LTURN LTURN". They found that the seq2seq networks have good zero-shot generalization capability when the difference between the commands of the training set and test set is small, but performed particularly poorly on this dataset when systematic compositionality was required (Lake and Baroni, 2018).

It seems like this is a very simple generation task. If we trained the seq2seq model on a randomly split SCAN dataset (80% training set and 20% test set), we can get the accuracy of 99.8% on the test set. However, once the split of the training set and test set is constrained from the perspective of compositionality, the netwrok is no longer successful. For example, when the network is exposed to primitive command "jump" but no composed commands of "jump" and tested on the set including composed commands of "jump", the results show that the network can achieve only 1.2% correct.

We reimplemented experiment 1, 2, and 3 of Lake and Baroni and presented the results. This paper also proposed a novel method to improve the model performance in experiment 3.

2 Dataset and DataLoader

This paper implemented several seq2seq networks on the SCAN dataset [1]. We also implemented a data loader to insert the '<EOS>' and '<SOS>' token into the sequence and transfer the words to tokens as the input of the model.

3 Methodology

This paper implemented three different encoder-decoder architectures: LSTM with no attention, LSTM with attention, GRU with attention. The encoder and decoder does not share parameters. The models were implemented in PyTorch and based on the publicly available code [2] [3].

3.1 Encoder

3.1.1 LSTM

This paper initialized the hidden state of encoder to all zero. The encoder takes strings of T command

[1] https://github.com/brendenlake/SCAN
[2] https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
[3] https://pytorch.org/tutorials/beginner/chatbot_tutorial.html

tokens as input. In the first layer, we used an embedding layer to transform the input to a sequence of vectors $\{w_1, \ldots, w_T\}$. The second layer leveraged the nn.LSTM to use the embedding vector and the hidden state from the previous time step, followed $h_t = f_E(h_{t-1}, w_t)$ (Lake and Baroni, 2018), to process each token where h_{t-1} is the hidden state from the previous token and w_t is the embedding vector of t-th token. After processing all time steps of a command, we got the final hidden state h_T as the initial input hidden state of decoder. In the encoder part, the processing of each time step is automatically employed by the LSTM layer without the need to feed token one by one.

3.1.2 GRU

The GRU encoder architecture is identical to the LSTM encoder we described above. The only difference is that we use nn.GRU for the second layer instead of LSTM.

3.2 Decoder

3.2.1 LSTM without attention

The decoder takes the previous token and previous hidden state as input to select the next token, followed $h_t = f_D(h_{t-1}, w_{t-1})$ where h_{t-1} is the hidden state from the previous token and w_t is the embedding vector of t-th token. The initial hidden state is the final hidden state h_T of encoder. In each time step, the first layer is still a embedding layer and then the embedding vector is mapped to a ReLU. Then the hidden state h_{t-1} and the embedding vector w_{t-1} pass a second LSTM layer to generate a output and a hidden state. Finally the output go through a single linear layer and mapped to a softmax to select the next token.

3.2.2 LSTM with attention

The decoder with attention can see all of final hidden states of the encoder. At each time step, we first embedded the token and compute $e_{it} = v_a^T \tanh(W_a O_E + U_a h_{t-1})$ where O_E is the outputs of encoder and h_{t-1} is the previous decoder hidden state. v_a, W_a, U_a are learnable parameters. Then softmax the e_{it} to compute the attention weights α_{it} and performs a matrix multiplication of the attention weights α_{it} and encoder outputs O_E to compute the context vector c_i. The embedding vector is concatenated with the context vector c_i to pass to a LSTM layer. Then the outputs of LSTM layer is concatenated with context vector c_i to go through a out linear layer. Finally softmax the out to generate a probability distribution to select next token.

3.2.3 GRU with attention

The GRU decoder with attention architecture is identical to the LSTM decoder with attention we described above. The only difference is that the LSTM layer is replaced by GRU layer.

4 Experiments

In addition to the top-performing network for each experiment, we also employed a overall-best network (2-layer LSTM, 200 hidden units per layer, no attention, and 0.5 dropout (Lake and Baroni, 2018)) on each experiment. Every networks in this paper were all trained on 100,000 iters for 5 trials, used ADAM optimization algorithm with learning rate of 0.001 to optimize, and used Gradient clipping of 5.0 to tackle exploding gradients. For overall-best network, we used embedding size of 20 which is different from (Lake and Baroni, 2018) since we didn't notice the supplementary materials at the beginning. The embedding size of top-performing networks are all the same with the hidden units.

4.1 Experiment 1 and discussion

This experiment explored the zero-shot generalization ability of the seq2seq network under the different percent of commands used for training.

We started with datasets that were randomly split into a training set (80%) and a test set (20%). According to (Lake and Baroni, 2018), the **top-performing** network of this experiment is a LSTM without attention, 2 layers of 200 hidden units, and no dropout (Lake and Baroni, 2018). Finally, we got an average accuracy of **99.8%** for 5 trials which is the same as the result of Lake and Baroni. The **overall-best** architecture achieves the average accuracy of **99.6%** which is slightly lower than the accuracy of **99.7%** of Lake and Baroni. The results showed that training the network on the training set with 80% of commands is sufficient to achieve a good performance.

We were also interested in the effect of different percent of commands used for training on accuracy. So we then employed the **overall-best** network on the training set of different percent of commands, and the results are shown in Figure 1. From this figure, we can see that the network trained on the 1% of the commands got the low accuracy of **6.0%**. The network trained on the 2% of the commands performed much better than 1%,

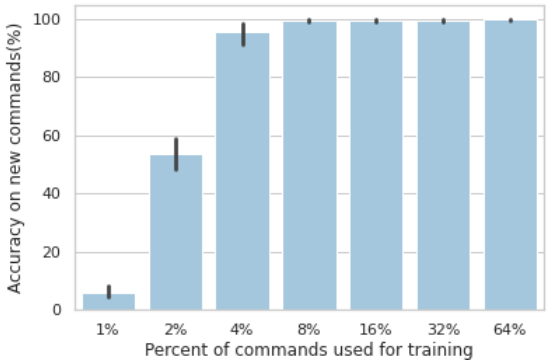

Figure 1: Zero-shot generalization of the overall-best network after training on a random subset of the SCAN tasks. Each bar shows the mean over 5 training runs with corresponding ± 1 SEM.

getting about the **53.7%** accuracy. With the 4% of the commands, the network achieved the accuracy of **95.7%**. When the percent of commands used for training is higher than 8%, the accuracy remained stable at about 99.6%. The results of different percent of commands show that the recurrent network has a great zero-shot generalization ability even trained on a sparse coverage training set.

4.2 Experiment 2 and discussion

This experiment explored the ability of generalizing to the commands demanding longer action sequences. In this experiment, the SCAN dataset is split according to the sequence length. The network is trained on shorter sequences which are 80% of the full set and tested on longer sequences. Length is defined as the number of output actions. Under this split, the network is required to be familiar with the verbs, modifiers, and conjunctions to generate an action of a length never seen before.

The **top-performing** network of this experiment is GRU with attention, 1 layer of 50 hidden units, and dropout 0.5. We finally got the average accuracy of **15.7%** which is lower than the 20.8% of Lake and Baroni article. The **overall-best** network provided a **13.7%** average accuracy over 5 runs, which is slightly lower than 13.8% of Lake and Baroni. Then we analyzed the prediction accuracy of ground-truth actions and commands of different lengths in detail, which is shown in Figure 2. The analysis is based on the results of the 5 runs of the **overall-best** model. We slightly modified the test function in our code to record the length of commands, the length of actions, and whether generate correctly per sequence. By analyzing the results according to actions, we found that the length of the ground-truth action sequence in the test set included $\{24, 25, 26, 27, 28, 30, 32, 33, 36, 40, 48\}$. Figure 2 shows that the shorter the actions, the higher accuracy it is. There is a sharp drop in model performance between action lengths of 25 and 26.

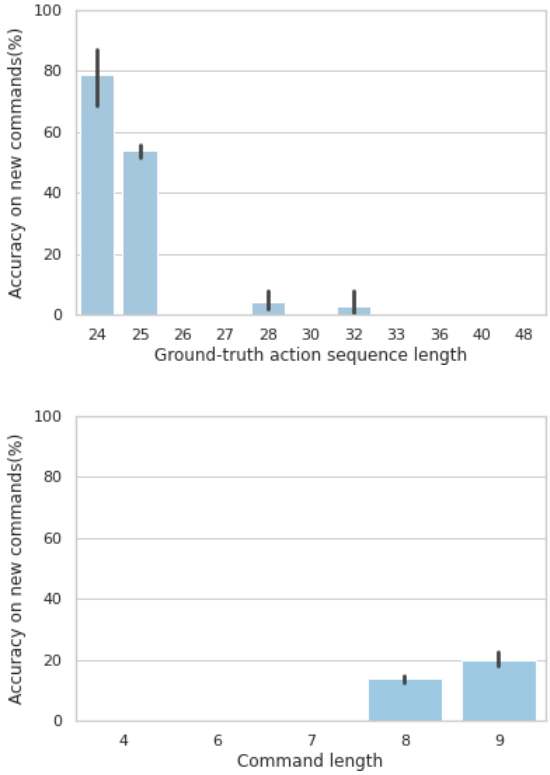

Figure 2: Zero-shot generalization of the overall-best model to commands with action sequence lengths not seen in training.

By analyzing the results according to command length, we found that the command length is distributed in $\{4, 6, 7, 8, 9\}$. Figure 2 shows that the model performed better on command lengths of 8 and 9. The possible reason is that most of the command lengths in the training set are also 8 or 9, which indicates that the model generalizes better when the training set and test set are similar.

4.3 Experiment 3 and discussion

This experiment explored the compositionality of seq2seq networks. The model is trained on a training set that includes a primitive command of an action a and the primitive and composed commands of other actions. Then evaluate the network on the test set which includes the composed commands of a. There are two different datasets, "jump" and

"turn left". For the "jump" set, the training set includes the composed commands that do not include "jump" and a primitive "jump" command in isolation. The test set includes the composed commands of "jump". The 'turn left' set is similar to "jump" set.

For the "turn left" dataset, the **top-performing** network is the GRU with attention, 1 layer with 100 hidden units, and dropout of 0.1. We got the average accuracy of **88.5%** which is slightly lower than the 90.3% of Lake and Baroni. The **overall-best** network achieved the average **84.2%** correct which is lower than the 90% of Lake and Baroni. For "jump" dataset, the **top-performing** network is LSTM with attention, 1 layer with 100 hidden units, and dropout of 0.1. We got the average accuracy of **0.106%**. The **overall-best** network achieved the average accuracy of **0.013%**. The huge difference between "jump" and "turn left" results maybe since there are many other composed commands that also denote "LTURN", while the "JUMP" is only denoted by "jump". "turn left" is more easier to generalization.

Then we computed 5 nearest neighbors for a sample of commands. The command similarity is calculated by the cosine similarity between the final hidden states of the two commands of the encoder. The results are shown in Table 1. Our 5 nearest neighbors are different from those of the Lake and Baroni, so I also calculated the similarity scores based on the 5 nearest neighbors in the article which are shown in the appendix.

Since the network performed extremely poorly on a training set containing only one primitive command of "jump", then we explored how increasing the number of composed commands of "jump" in the training set affected the results. Specifically, we trained the top-performing model of "jump" on the training sets which include $\{1, 2, 4, 8, 16, 32\}$ composed commands of "jump", and the results are shown in Figure 3. From this figure, we can see that even when 1 composed command of "jump" appears in the training set, the network still performs poorly. The presence of 16 or 32 composed commands in the training set will bring significant improvements to the network.

5 Task D

5.1 Motivation

The poor performance of the networks in experiment 3 seems to indicate that the seq2seq networks

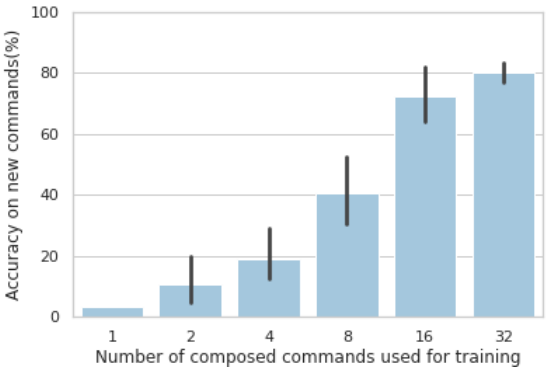

Figure 3: Zero-shot generalization of top-performing model after adding the primitive "jump" and some compositional "jump" commands.

lack the systematic compositionality so that they can not recombine the primitive command into a new action. Humans are born with systematic compositionality. If a person knows about the meaning of "run after walk" and "jump", then he can understand the "jump after walk". However, the model failed to translate the composed commands of "jump" in the test set, which may indicate poor systematic compositionality.

However, I argue that such expositions are unsatisfactory since there are inequities in the learning process between machines and humans. The human can understand "jump after walk" because humans don't just see the four letters when they learn "jump". They know "jump" is a verb so that they can understand it based on "run after walk". Different from how humans learn "jump", the network is exposed to only four letters and has no other relevant information. So it is difficult for networks to generalize composed commands after learning a primitive command. It is unfair to compare systematic compositionality directly since humans and machines are not on the same starting line.

Therefore, this paper proposed a method to incorporate the parts of speech information into the training and employed it on the "jump" dataset.

5.2 Implementation detail

I slightly modified our Dataloader to add the parts of speech information into commands. I classified the words of commands into three categories, verb, modifier and conjunction and inserted it after each word of command. For example, this paper mapped the 'run twice' to 'run <verb> twice <modifier>'. The actions keep the same with before.

run		jump		run twice		jump twice	
run after run left	1.0	*look opposite right thrice and walk left*	*0.17*	run after run left	1.0	*look opposite right thrice and walk left*	*0.17*
run twice and look left thrice	1.0	*look opposite left thrice after walk around left*	*0.17*	run twice and look left thrice	1.0	*look opposite left thrice after walk around left*	*0.17*
run opposite left twice and turn opposite left	1.0	*look around left after run around left*	*0.17*	run opposite left twice and turn opposite left	1.0	*look around left after run around left*	*0.17*
run around left thrice after run opposite left twice	1.0	*look twice after look twice*	*0.17*	run around left thrice after run opposite left twice	1.0	*look twice after look twice*	*0.17*
run opposite right twice after look right thrice	1.0	*look around right twice after run opposite left thrice*	*0.17*	run opposite right twice after look right thrice	1.0	*look around right twice after run opposite left thrice*	*0.17*

Table 1: Nearest training commands for representative commands based on the top-performing model of "jump" dataset, with the respective cosines. Italics mark low similarities (cosine < 0.2).

5.3 Results and discussion

I trained the top-performing model of the "jump" dataset on the transformed "jump" dataset with the same settings as experiment 3. Finally, I got an average accuracy of 0.71% which is higher than the 0.11% accuracy we got in experiment 3. However, such a slight improvement does not provide sufficient evidence that this method can help improve the compositionality of the network. I took a closer look at the results and observeed that the network could only correctly decode the commands beginning with "jump" such as "jump after twice". And the network fails to decode correctly when "jump" is in the middle of a command such like "walk right after jump twice", which may caused by the training set that doesn't contain composed command of "jump" and no other tokens can be connected to "jump" when training even through I inserted '<verb>' after "jump". So the network still performs poor on the test set that "jump" is in the middle of a command.

6 Challenges

I found that the loss value of some iterations were NaN in the training process after completed the encoder-decoder LSTM network.

At first, I thought the problem might be caused by gradient explosion, but we have used gradient clipping to prevent exploding gradients, so this should not be the cause of the problem. And then I found the crux is that some iterations incorrectly uses 0 as a divisor due to the calculation method of Loss. Although (Lake and Baroni, 2018) feed data one by one into the model (batch size=1), we still want to support data of different batch size for training when designing the training function. Since the truth lengths of different actions are different, we used the maskNLLLoss to calculate the loss of a batch. Because the padding of sequence is based on the whole dataset, the last n bits of each sequence in some epochs may all be 0 and the mask may all be false. We used the "mask.sum()" to calculate the number of loss values and then used the "totalLoss/numLoss" to calculate the loss value of one epoch. Therefore, the case that masks are all false results in "numLoss" being 0 and loss being NaN. Finally, I solved this problem by adding a decision-making statement in loss fucntion to skip the case that masks are all false.

References

Brenden M. Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks.

Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. 2020. Compositional generalization by learning analytical expressions.

run		**jump**		**run twice**		**jump twice**	
look	0.53	*run*	*0.14*	look twice	0.77	*walk and walk*	*0.07*
walk	0.56	*walk*	*0.13*	run twice and look opposite right thrice	0.32	*run and walk*	*0.09*
walk after run	0.57	*turn right*	*0.05*	run twice and run right twice	0.76	*walk opposite right and walk*	*0.02*
run thrice after run	0.55	*look right twice after walk twice*	*0.11*	run twice and look opposite right twice	0.49	*look right and walk*	*0.02*
run twice after run	0.55	*turn right after turn right*	*0.10*	walk twice and run twice	0.86	*walk right and walk*	*0.02*

Table A.1: The cosine similarities of 5 nearest training commands and representative commands in (Lake and Baroni, 2018) calculated through our work.

A Appendix