CTL++: Evaluating Generalization on Never-Seen Compositional Patterns of Known Functions, and Compatibility of Neural Representations

Róbert Csordás[1] **Kazuki Irie**[1] **Jürgen Schmidhuber**[1,2]
[1]The Swiss AI Lab IDSIA, USI & SUPSI, Lugano, Switzerland
[2]AI Initiative, KAUST, Thuwal, Saudi Arabia
{robert, kazuki, juergen}@idsia.ch

Abstract

Well-designed diagnostic tasks have played a key role in studying the failure of neural nets (NNs) to generalize systematically. Famous examples include SCAN and Compositional Table Lookup (CTL). Here we introduce CTL++, a new diagnostic dataset based on compositions of unary symbolic functions. While the original CTL is used to test length generalization or *productivity*, CTL++ is designed to test *systematicity* of NNs, that is, their capability to generalize to unseen compositions of known functions. CTL++ splits functions into groups and tests performance on group elements composed in a way not seen during training. We show that recent CTL-solving Transformer variants fail on CTL++. The simplicity of the task design allows for fine-grained control of task difficulty, as well as many insightful analyses. For example, we measure how much overlap between groups is needed by tested NNs for learning to compose. We also visualize how learned symbol representations in outputs of functions from different groups are compatible in case of success but not in case of failure. These results provide insights into failure cases reported on more complex compositions in the natural language domain. Our code is public.[1]

1 Introduction

Neural networks (NNs) should ideally learn from training data to generalize *systematically* (Fodor et al., 1988), by learning generally applicable rules instead of pure pattern matching. Existing NNs, however, typically don't. For example, in the context of sequence processing NNs, superficial differences between training and test distributions, e.g., with respect to input sequence length or unseen input/word combinations, are enough to prevent current NNs from generalizing (Lake and Baroni, 2018). Training on a large amounts of data might alleviate the problem, but it is infeasible to cover all possible lengths and combinations.

Indeed, while large language models trained on a large amounts of data have obtained impressive results (Brown et al., 2020), they often fail on tasks requiring simple algorithmic reasoning, e.g., simple arithmetics (Rae et al., 2021). A promising way to achieve systematic generalization is to make NNs more compositional (Schmidhuber, 1990), by reflecting and exploiting the hierarchical structure of many problems either within some NN's learned weights, or through tailored NN architectures. For example, recent work by Csordás et al. (2022) proposes architectural modifications to the standard Transformer (Vaswani et al., 2017) motivated by the principles of compositionality. The resulting Neural Data Router (NDR) exhibits strong *length generalization* or *productivity* on representative datasets such as Compositional Table Lookup (CTL; Liska et al. (2018); Hupkes et al. (2019)).

The focus of the present work is on *systematicity*: the capability to generalize to unseen compositions of known functions/words. That is crucial for learning to process natural language or to reason on algorithmic problems without an excessive amount of training examples. Some of the existing benchmarks (such as COGS (Kim and Linzen, 2020) and PCFG (Hupkes et al., 2020)) are almost solvable by plain NNs with careful tuning (Csordás et al., 2021), while others, such as CFQ (Keysers et al., 2020), are much harder. A recent analysis of CFQ by Bogin et al. (2022) suggests that the difficult examples have a common characteristic: they contain some local structures (describable by parse trees) which are not present in the training examples. These findings provide hints for constructing both challenging and intuitive (simple to define and analyze) diagnostic tasks for testing systematicity. We propose CTL++, a new diagnostic dataset building upon CTL. CTL++ is basically as simple as the original CTL in terms of task definition, but adds the core challenge of compositional generalization

[1]https://github.com/robertcsordas/ctlpp

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 9758 - 9767

absent in CTL. Such simplicity allows for insightful analyses: one low-level reason for the failure to generalize compositionally appears to be the failure to learn functions whose outputs are symbol representations compatible with inputs of other learned neural functions. We will visualize this.

Well-designed diagnostic datasets have historically contributed to studies of systematic generalization in NNs. Our CTL++ strives to continue this tradition.

2 Original CTL

Our new task (Sec. 3) is based on the CTL task (Liska et al., 2018; Hupkes et al., 2019; Dubois et al., 2020) whose examples consist of compositions of bijective unary functions defined over a set of symbols. Each example in the original CTL is defined by one input symbol and a list of functions to be applied sequentially, i.e., the first function is applied to the input symbol and the resulting output becomes the input to the second function, and so forth. The functions are bijective and randomly generated. The original CTL uses eight different symbols. We represent each symbol by a natural number, and each function by a letter. For example, '`d a b 3`' corresponds to the expression $d(a(b(3)))$. The model has to predict the corresponding output symbol (this can be viewed as a sequence classification task). When the train/test distributions are independent and identically distributed (IID), even the basic Transformer achieves perfect test accuracy (Csordás et al., 2022). The task becomes more interesting when test examples are longer than training examples. In such a *productivity* split, which is the common setting of the original CTL (Dubois et al., 2020; Csordás et al., 2022), standard Transformers fail, while NDR and bi-directional LSTM work perfectly.

3 Extensions for Systematicity: CTL++

To introduce a *systematicity* split to the CTL framework, we divide the set of functions into disjoint *groups* and restrict the sampling process such that some patterns of compositions between group elements are never sampled for training, only for testing. Based on this simple principle, we derive three variations of CTL++. They differ from each other in terms of compositional patterns used for testing (excluded from training) as described below. We'll also visualize the difference using *sampling graphs* in which the nodes represent the groups, and the edges specify possible compositional patterns. The colors of the edges reflect when the edges are used: black for both training and testing, blue for training, and red only for testing.

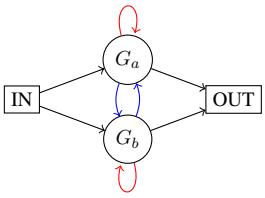


Figure 1: Sampling graph for variant 'A.'

Variation 'A' (as in 'Alternating'). Here functions are divided in groups G_a and G_b. During training, successively composed functions are sampled from different groups in an alternating way—i.e., successive functions cannot be from the same group. During testing, however, only functions from the same group can be composed. The sampling graph is shown in Fig. 1. Importantly, the single function applications are part of the training set, to allow the model to learn common input/output symbol representations for the interface between different groups.

Variation 'R' (as in 'Repeating'). This variant is the complement of variation 'A' above. To get a training example, either G_a or G_b is sampled, and all functions in that example are sampled from that same group for the whole sequence. In test examples, functions are sampled in an alternating way. There is thus no exchange of information between the groups, except for the shared input embeddings and the output classification weight matrix. The sampling graph is like in Fig. 1 for 'A' except that blue edges should become red and vice versa (see Fig. 5 in the appendix).

Variation 'S' (as in 'Staged'). In this variant, functions are divided into five disjoint groups: G_{a1}, G_{a2}, G_{b1}, G_{b2} and G_o. As indicated by the indices, each group belongs to one of the two *paths* ('a' or 'b') and one of the two *stages* ('1' or '2'), except for G_o which only belongs to stage '2' shared between paths 'a' and 'b' during training. The corresponding sampling graph is shown in Fig. 2. To get a training example, we sample an integer K which defines the sequence length as $2K+1$, and iterate the following process for $k \in [0, .., K]$ and $i = 2k$: we first sample a path $p \in \{a, b\}$ and then a function f_i from G_{p1} and a function f_{i+1}

from $G_{p2} \cup G_o$. Each example always contains an even number of functions, and no isolated single function application is part of training, unlike in the previous two variants. For testing, we sample a path $p \in \{a, b\}$ and a function f_i from G_{p1}, but then sample a function f_{i+1} from $G_{\{a,b\}\setminus\{p\}2}$, which results in a compositional pattern never seen during training.

The unique feature of this variant is the use of two stages: as can be seen in Fig. 2, during training, given a path $p \in \{a, b\}$, outputs of any functions belonging to G_{p1} are only observed by the functions belonging to G_{p2}, i.e., the stage '2' group belonging to the same path p, or G_o. Hence, if $G_o = \emptyset$, the model has no incentive to learn common representations for the interface between G_{a1} and G_{b1}: to solve the training examples, it suffices to learn output representations of G_{a1} which are 'compatible' with the input representations of G_{a2}; similarly for G_{b1} and G_{b2}. There is no reason for outputs of G_{a1} to be compatible with the inputs of G_{b2} (analogously for G_{b1} and G_{a2}) which is required at test time. The size of G_o is our first parameter for controlling task difficulty (the y-axis of Fig. 4 which we will present later).

We introduce further restrictions: for each function $f \in G_o$, we define a set of symbols S_a^f for G_{a1} (and S_b^f for G_{b1}), and we only allow for sampling f if the output symbol of function from G_{a1} (or G_{b1}) belongs to S_a^f (or S_b^f). This allows for defining another control parameter: the number of overlapping symbols between S_a^f and S_b^f (same for all f; the x-axis of Fig. 4). Note that we ensure that the union of shared symbols defined for functions in G_o cover all possible symbols. This might not be the case in a more realistic scenario, but as we'll see, the standard models already struggle in this setting. By controlling these two parameters, we precisely control the degree of overlap offered by G_o in terms of both the number of functions and symbols. Ideal models should be "sample efficient" in terms of this overlap, since we cannot expect the training set to contain all combinations of such overlaps in a practical scenario with semantically rich domains such as natural language.

4 Results

We evaluate standard CTL-tested models on the new CTL++ task, including: the Transformer (Vaswani et al., 2017) with shared layers (Dehghani et al., 2019), the neural data router (NDR) (Csordás et al., 2022), and the bi-directional LSTM (Hochreiter and Schmidhuber, 1997; Schuster and Paliwal, 1997; Graves et al., 2005). Recall that both NDR and bi-directional LSTM are reported to perfectly solve the original CTL's length generalization split (Csordás et al., 2022), unlike the Transformer. Further experimental details can be found in Appendix A.

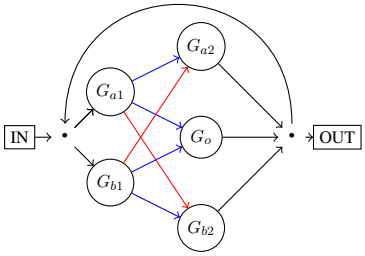


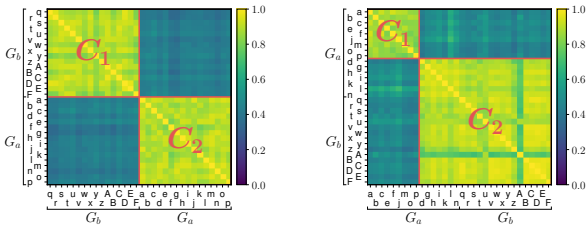
Figure 2: Sampling graph for Variant 'S'

Table 1: Results on the task variants 'A' and 'R.' Mean and standard deviation are computed using 25 seeds.

Model	Dataset	Accuracy	
		IID	OOD
Bi-LSTM	A	1.00 ± 0.00	0.95 ± 0.03
	R	1.00 ± 0.00	1.00 ± 0.00
Transformer	A	1.00 ± 0.00	0.21 ± 0.09
	R	1.00 ± 0.00	0.75 ± 0.25
NDR	A	1.00 ± 0.00	0.34 ± 0.26
	R	1.00 ± 0.01	0.75 ± 0.27

4.1 Results on Variants 'A' and 'R'

Table 1 shows the performance overview for 'A' and 'R.' The OOD (out-of-distribution) column indicates the train/test data sampling processes described above. As a reference, we also report the IID cases where the training example sampling graph is also used for testing. Our initial expectation was that the pressure from shared input/output embeddings is sufficient for these models to learn common symbol representations for all functions. However, we observe that only the bi-LSTM solves these tasks consistently across seeds. Interestingly, the NDR, which perfectly performs on the length generalization split of CTL and beyond (Csordás et al., 2022), performs poorly on both the 'A' and 'R' variants of CTL++. Tested with 25 seeds, only 32% of the seeds (out of 25) achieve over 95% accuracy for NDR on 'R' (20% for the standard Transformer). The success rate is 0% for the 'A'



(a) Example for symbol '6'. Perfect clustering w.r.t. the function groups is observed.

(b) Example for symbol '3'. Partial clustering w.r.t. the function groups is observed.

Figure 3: Cosine similarity of output representations of different functions representing the same symbol for the NDR with a seed that fails on 'R.'

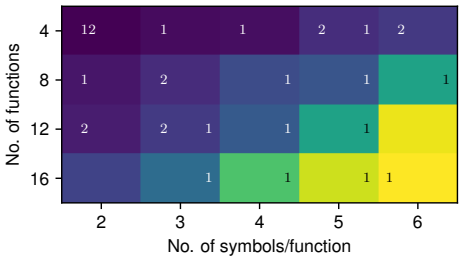


Figure 4: Test accuracy of NDR on the 'S' variant. The total number of symbols is 8, and the number of functions is 32. The y-axis shows the number of overlapping functions, while the x-axis shows the number of symbols shared between two groups for each function in G_o during training (Sec. 3). Results for the Transformer and LSTM are reported in the appendix (Figs. 7 and 8).

variant[2]. These simple tasks thus turn out to be good first diagnostic tasks for testing systematicity.

Analysis. The small input/output space of this task allows for an exhaustive analysis of the learned symbol representations. Specifically, given an *output* symbol s', for each function $f \in G_a \cup G_b$, we can find a unique input symbol s such that $s' = f(s)$ (because all functions defined for this task are bijective). Hence, for a fixed symbol s', for all functions f, we can extract the learned vector representation of this symbol s' at the output of f as the vector of the layer beneath the final classification layer when we feed 'f s' to the network. Then we can compare the extracted representations (of a fixed symbol for different functions) by computing their cosine similarities.

Here we compare representations learned with successful/failed seeds for NDR in variation 'R.' Fig. 3a and 3b show the results for two different (output) symbols '6' and '3' from the same *failed* seed. In both cases we observe two clusters (C_1 and C_2): two separate/different representations are learned for the same symbol (by abuse of notation, we also refer to the corresponding representations as C_1 and C_2). In the case of symbol '6' in Fig. 3a, we observe perfect/strict clustering in line with the group of the applied function; C_1 and C_2 are representations of symbol '6' learned by functions belonging to G_a and G_b respectively. This is problematic since functions in G_a never see symbol '6' represented as C_2 during training (analogously for functions in G_b with representation C_1). As a consequence, during testing, when a function $f_a \in G_a$ is applied after a function $f_b \in G_b$, f_b may output symbol '6' represented as C_2, and pass it to f_a, but in principle, f_a can not "understand/interpret" C_2 as representing symbol '6.' This naturally prevents cross-group generalization. In the case of symbol '3' shown in Fig. 3b, some of the functions yield the same symbol representations as certain functions from the other group (see the cluster C_2 at the lower right: a good trend), but we still have a small cluster (C_1 at the upper left) consistent only among elements of G_a. Hence, cross-group generalization can still fail because the functions in G_b never see symbol '3' represented as C_1 during training but only during testing. In contrast, for *successful* seeds, we do not observe any of these clusters for any symbols (see Fig. 9 in the appendix). A single representation shared across all functions is learned for each symbol. Further quantitative analysis can be found in Appendix B.

4.2 Results of Staged Variant 'S'

As described in Sec. 3, variant 'S' is designed to evaluate models at different task difficulty levels determined by the number of overlapping functions and symbols during training. Fig. 4 shows the corresponding performance overview for NDR. The overall picture is similar for bi-LSTM and Transformer (see Figs. 7 and 8 in the appendix). We observe that to achieve 100% accuracy, half of the possible functions should overlap (16/32), as well as most of the possible symbols seen for each function (6/8). This implies an unrealistically large amount of data for real world scenarios, where the

[2] 'A' turns out to be harder than 'R.' We speculate that in both 'A' and 'R', given that the training set contains single function applications with shared input/ouput embeddings, the learned symbol representation of all functions should be compatible with each other to some extent, but with some "deviation" from perfect compatibility in case of failure. Such "deviations" might accumulate in case of 'A' where we sample all functions in each sequence from a single group at test time.

"functions" might correspond to more complex operations with multiple input arguments (as in the CFQ case). This calls for developing approaches that achieve higher accuracy in the upper left part of Fig. 4.

5 Conclusion

Motivated by the historically crucial role of diagnostic datasets for research on systematic generalization of NNs, we propose a new dataset called CTL++. Unlike the classic CTL dataset, typically used for testing productivity, CTL++ is designed for testing systematicity. We propose three variants, 'A,' 'R,' and 'S.' Despite their simplicity, even the CTL-solving Transformer variant fails on 'A' and 'R.' Using 'S,' we show that existing approaches require impractically large amounts of examples to achieve perfect compositional generalization. The small task size allows for conducting exhaustive visualizations of (in)compatibility of learned symbol representations in outputs of functions with inputs of subsequent functions. Of course, the ultimate goal is to go beyond just solving CTL++. Nevertheless, we hope CTL++ will become one of the standard diagnostic datasets for testing systematicity of NNs.

Limitations

Achieving 100% on this dataset may be a necessary condition for NNs capable of systematic generalization, but certainly not a sufficient one. In practice, there may be many reasons which prevent NNs from generalizing systematically in other tasks or more generally on real world data. Compare the original CTL dataset for evaluating productivity: Csordás et al. (2022) show that some models that achieve 100% on CTL still fail in other tasks such as ListOps. This is why we refer to CTL++ as a simple *diagnostic* dataset for testing systematicity of NNs. Nevertheless, it allows for uncovering certain important failure modes of NNs.

Acknowledgments

This research was partially funded by ERC Advanced grant no: 742870, project AlgoRNN, and by Swiss National Science Foundation grant no: 200021_192356, project NEUSYM. We are thankful for hardware donations from NVIDIA and IBM. The resources used for this work were partially provided by Swiss National Supercomputing Centre (CSCS) project s1154.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *Preprint arXiv:1607.06450*.

Ben Bogin, Shivanshu Gupta, and Jonathan Berant. 2022. Unobserved local structures make compositional generalization hard. *Preprint arXiv:2201.05899*.

Tom B Brown et al. 2020. Language models are few-shot learners. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. 2021. The devil is in the detail: Simple tricks improve systematic generalization of Transformers. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Punta Cana, Dominican Republic.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. 2022. The neural data router: Adaptive control flow in transformers improves systematic generalization. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proc. Association for Computational Linguistics (ACL)*, pages 2978–2988, Florence, Italy.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal Transformers. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA.

Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. 2020. Location attention for extrapolation to longer sequences. In *Proc. Association for Computational Linguistics (ACL)*, pages 403–413, Virtual only.

Jerry A Fodor, Zenon W Pylyshyn, et al. 1988. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. 2005. Bidirectional LSTM networks for improved phoneme classification and recognition. In *Proc. Int. Conf. on Artificial Neural Networks (ICANN)*, pages 799–804, Warsaw, Poland.

Stephen José Hanson. 1990. A stochastic version of the delta rule. *Physica D: Nonlinear Phenomena*, 42(1-3):265–272.

Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (GELUs). *Preprint arXiv:1606.08415*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, pages 1735–1780.

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

Dieuwke Hupkes, Anand Singh, Kris Korrel, German Kruszewski, and Elia Bruni. 2019. Learning compositionally through attentive guidance. In *Proc. Int. Conf. on Computational Linguistics and Intelligent Text Processing*, La Rochelle, France.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. 2020. Measuring compositional generalization: A comprehensive method on realistic data. In *Int. Conf. on Learning Representations (ICLR)*, Addis Ababa, Ethiopia.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Virtual only.

Brenden M. Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 2873–2882, Stockholm, Sweden.

Adam Liska, Germán Kruszewski, and Marco Baroni. 2018. Memorize or generalize? Searching for a compositional RNN in a haystack. In *AEGAP Workshop ICML*, Stockholm, Sweden.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, Vancouver, Canada.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *Preprint arXiv:2112.11446*.

Jürgen Schmidhuber. 1990. Towards compositional learning in dynamic networks. Technical Report FKI-129-90, Institut für Informatik, Technische Universität München.

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

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, Long Beach, CA, USA.

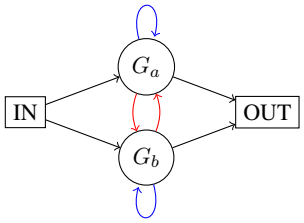


Figure 5: Sampling graph for variant 'R.'

A Experimental Details

A.1 Modified NDR architecture

We experimentally found some architectural modifications to the original NDR (Csordás et al., 2022) that yield faster convergence and produce more stable results than the original architecture. Here we describe our modifications. We use the GELU activation function (Hendrycks and Gimpel, 2016) instead of ReLU, and a residual connection in the feedforward data path. Concretely, Eq. 5 in Csordás et al. (2022) is replaced by Eq. 1 below, while Eq. 2 is replaced by Eq. 2 below. We do not use any dropout in Eq. 1.

$$\mathrm{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2\,\mathrm{GELU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \quad (1)$$

$$\boldsymbol{u}^{(i,t+1)} = \mathrm{LN}(\mathrm{FFN}^{\mathrm{data}}(\boldsymbol{a}^{(i,t+1)}) + \boldsymbol{a}^{(i,t+1)}) \quad (2)$$

where LN denotes layer normalization (Ba et al., 2016).

A.2 Hyperparameters

Dataset. The train set in all of our experiments consists of 300k examples. The maximum number of composed functions is 6. We make sure that we obtain an equal number of samples for different lengths whenever possible (in some cases this is impossible because by construction, there are fewer short examples than long ones). In 'A' and 'R' variants, single functions are always part of the training set with all possible symbols. There are in total 8 symbols and 32 functions. All samples are presented in a right-to-left manner (e.g. "c b a 3"). The IID and OOD test sets contain 1000 examples in all cases. Our code generates the data for given dataset specifications (number of functions etc). The seed for data generation is fixed.

Training. Unless noted otherwise, for all of our models, we use a batch size of 512, a learning rate of 0.00015, and a dropout rate (Hanson, 1990; Srivastava et al., 2014) of 0.5. We also use a linear learning rate warmup for the first 500 iterations. We use PyTorch's (Paszke et al., 2019) adaptive mixed precision and bin the batches by length for greater efficiency. We use the AdamW optimizer (Loshchilov and Hutter, 2019). The NDR and bi-LSTM is trained for 80k and the Transformers for 300k iterations. We find the standard Transformer to be very unstable even in the IID setting. In fact, for Table 1, unlike other models trained for 25 seeds, we train the Transformer for 50 seeds: the 25 seeds used to report mean and std in Table 1 are those among 50 which converged within 300k training iterations. For Figs. 4, 7 and 8, five seeds were used for each configuration. All of our models are trained on a single P100 GPU. The corresponding number of parameters, training steps and average wall-clock time is shown in Table 2.

Table 2: Training details for different models. Runtime is in hour:min.

Model	Num. params	Num. steps	Runtime
Bi-LSTM	408k	80k	0:18
Transformer	672k	300k	3:37
NDR	679k	80k	1:22

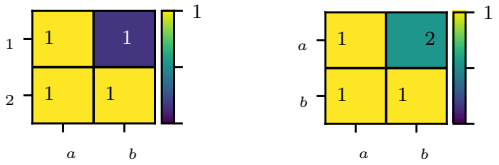
Models. For Transformer and NDR, we use 8 layers, and 4 heads. NDR uses a gate dropout of 0.1, state size of 256, feedforward size of 1024. Transformers use a state size of 128 and feedforward size of 512, layer sharing (Dehghani et al., 2019), and Transformer-XL-style (Dai et al., 2019) relative positional encoding. For bidirectional LSTM, we use 1 layer with 256 units (128 per direction). The gradient is clipped to max norm of 1 for NDR and 5 for Transformer and LSTM. Transformers use a weight decay of 0.0025.

B More Analyses and Plots

B.1 Quantitative Analysis of Incompatibility

Here we provide additional results on the analysis of Sec. 4.1 conducted for variant 'R.' To quantify the correlation between the clusters (C_1 and C_2) identified in Fig. 3b and the compatibility of representations, we measured the proportion of correct output classifications, by taking the first function from a given cluster and the second one from a given group, for all pairs of functions for each pair of the form (cluster, group). The results are shown

in Fig. 6. Fig. 6a shows that the first cluster C_1 is effectively compatible only with functions from G_a, while the second one C_2 works with both G_a and G_b, as predicted by Fig. 3b. Fig. 6b shows the same analysis, but uses the groups to define the cluster. As predicted by Fig. 3b, only roughly half of the functions from G_a generate representations compatible with G_b, while all representations generated by functions in G_b are compatible with all in G_a.



(a) Performance of each cluster vs. each group

(b) Performance for each pair of groups

Figure 6: Accuracy measured after two successive function applications, for the symbol corresponding to Fig. 3b. (a) shows the proportion of correct outputs when the first function is taken from a given cluster (y-axis), and the second from a given group (x-axis). Clusters are shown in the main diagonal of Fig. 3b. (b) is analogous to (a) but using groups as clusters.

B.2 Representative Cosine Similarities

Here we show additional visualizations similar to those of Fig. 3. In Fig. 9 and 10, we plot cosine similarities of functional outputs for all possible symbols for successful and failed seeds of NDR on variant 'R.' The symbol representations are taken from the layer right below the final classification layer. They are representative examples; the observation holds over all seeds we inspected. Fig. 11 shows a similar example for variant 'A.'

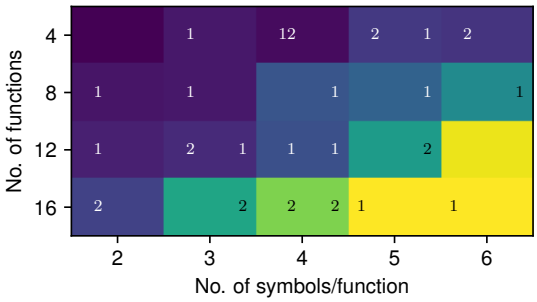


Figure 7: Final test performance of the Transformer on variant 'S.' The behavior is similar to the one shown in Fig. 4. For lower numbers of shared functions, performance is worse. Interestingly, however, with 16 shared functions, it outperforms NDR.

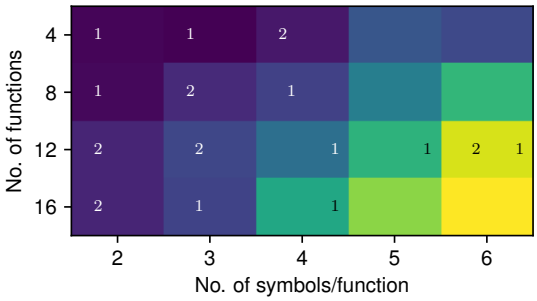


Figure 8: Final test performance of bi-directional LSTM on variant 'S.' The behavior is similar to the one shown in Fig. 4. For lower numbers of shared functions, the performance is worse. Interestingly, however, with 16 shared functions, it significantly underperforms NDR.

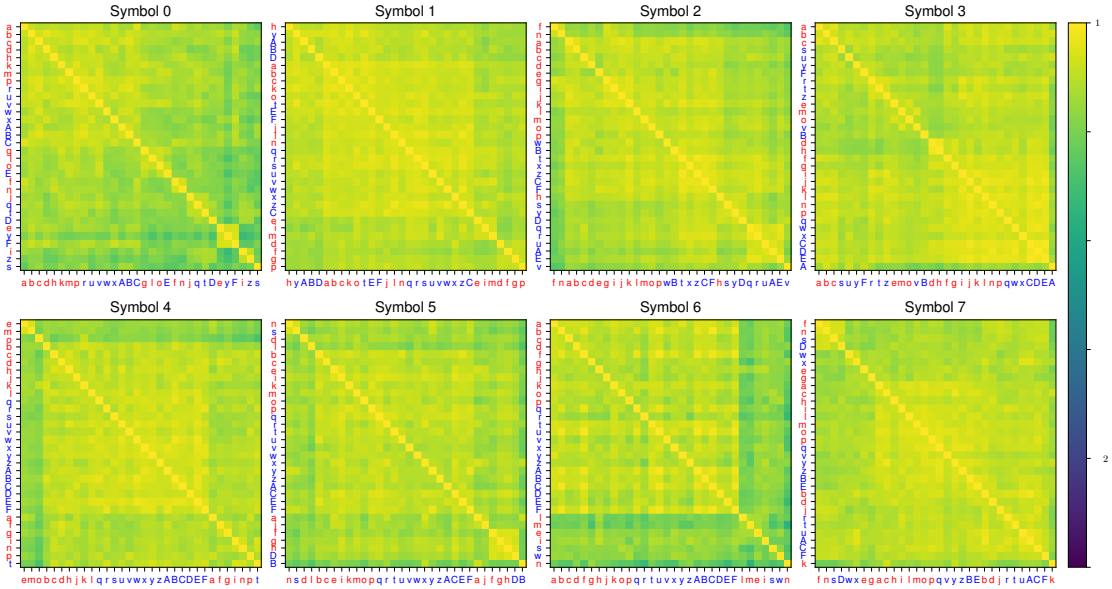


Figure 9: Symbol cosine similarity between different functions for NDR on variant 'R.' A representative example from a seed that performs **perfectly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.

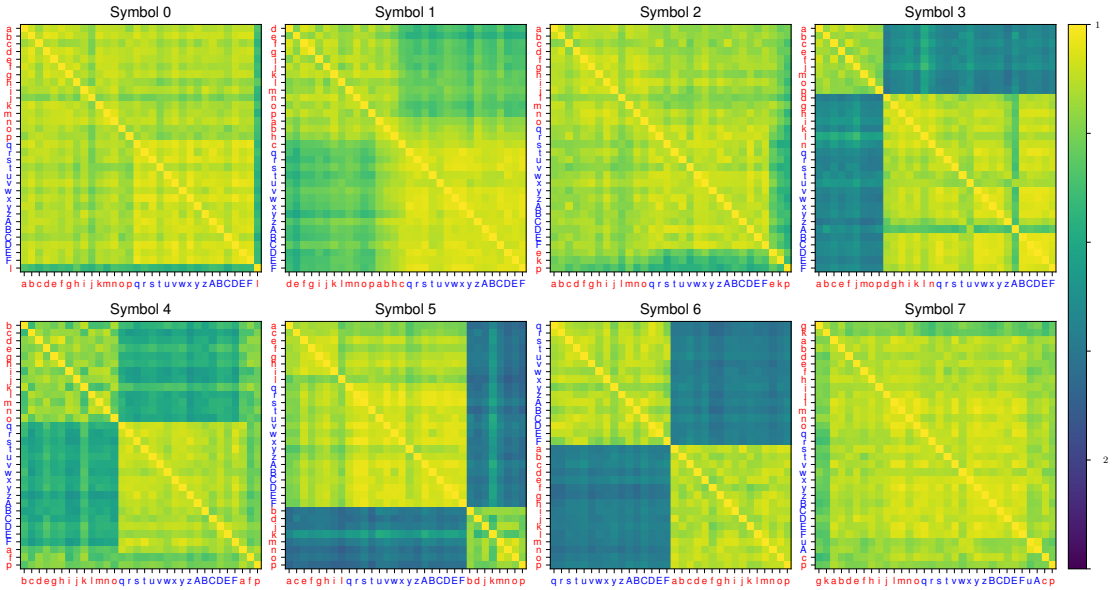


Figure 10: Symbol cosine similarity between different functions for NDR on variant 'R.' A representative example from a seed that performs **poorly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.

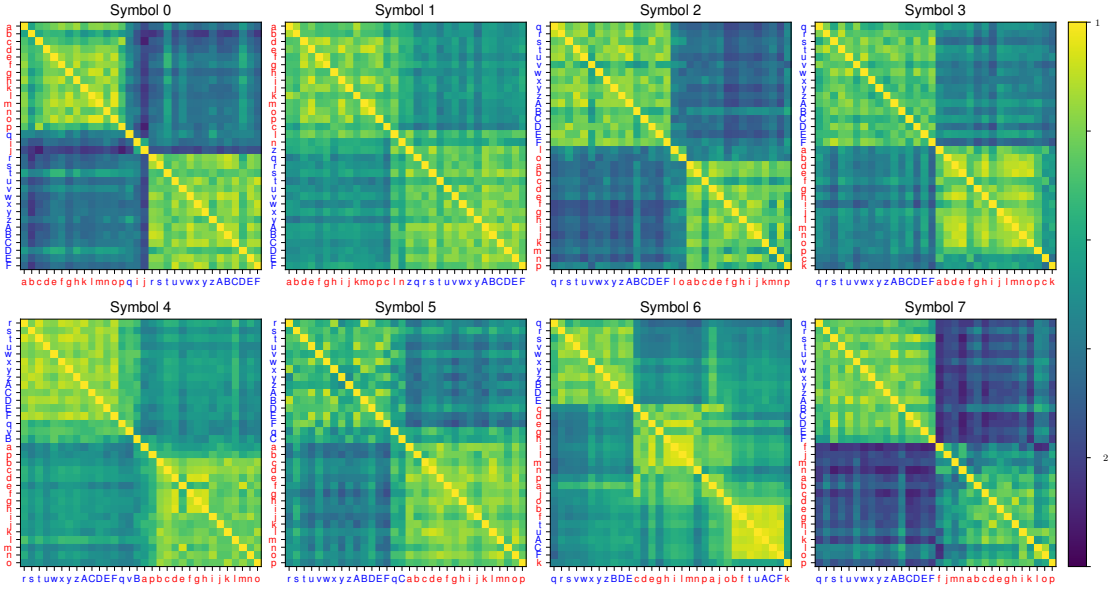


Figure 11: Symbol cosine similarity between different functions for NDR on variant 'A.' A representative example from a seed that performs **poorly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.