Differentiable Tree Operations Promote Compositional Generalization

Paul Soulos [1] [†]   **Edward Hu** [2] [†]   **Kate McCurdy** [3] [†]   **Yunmo Chen** [4] [†]   **Roland Fernandez** [5]   **Paul Smolensky** [1] [5]
Jianfeng Gao [5]

Abstract

In the context of structure-to-structure transformation tasks, learning sequences of discrete symbolic operations poses significant challenges due to their non-differentiability. To facilitate the learning of these symbolic sequences, we introduce a differentiable tree interpreter that compiles high-level symbolic tree operations into subsymbolic matrix operations on tensors. We present a novel Differentiable Tree Machine (*DTM*) architecture that integrates our interpreter with an external memory and an agent that learns to sequentially select tree operations to execute the target transformation in an end-to-end manner. With respect to out-of-distribution compositional generalization on synthetic semantic parsing and language generation tasks, *DTM* achieves 100% while existing baselines such as Transformer, Tree Transformer, LSTM, and Tree2Tree LSTM achieve less than 30%. *DTM* remains highly interpretable in addition to its perfect performance.

1. Introduction

Reasoning within the symbolic space through discrete symbolic operations can lead to improved out-of-distribution (OOD) generalization and enhanced interpretability. Despite the significant advances in representation learning made by modern deep learning, learning to directly manipulate discrete symbolic structures remains a challenge. One key issue is the non-differentiability of discrete symbolic operations, which makes them incompatible with gradient-based learning methods. Continuous representations offer greater learning capacity, but often at the expense of interpretability
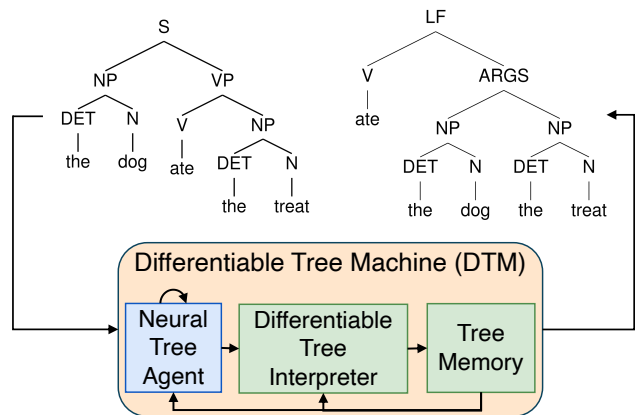

Figure 1. A high level overview of our model which consists of three modules. The Neural Tree Agent is a learnable component which, at each step of processing, selects the operation to perform and the arguments over which to operate. The Differentiable Tree Interpreter is a closed-form function precomputed at initialization which compiles high level symbolic operations into subsymbolic matrix operations on tensors. The output of the interpreter is a blended tree that is written to Tree Memory, which functions as a working memory to hold various partial and candidate trees. The final tree written to memory is the output tree. Blue represents the component with learnable parameters, and green represents components that operate in tree space.

and compositional generalization.

In this work, we focus on binary trees and three Lisp operators: `car`, `cdr`, and `cons` (Steele, 1990) (also known as left-child, right-child, and construct new tree). Examples of these operations are shown in Figure 3. Tensor Product Representation (TPR) provides a general encoding of structured symbolic objects in vector space (Smolensky, 1990). Crucially, within the TPR space, these symbolic operators on discrete objects become linear operators on continuous vectors (§2). Unlike normal symbolic structures, the vector space nature of TPRs allows blending multiple symbolic structures as interpolations between classic discrete structures. We restrict processing over our TPR encodings to the interpretable linear operations implementing the three Lisp operators and their interpolations, making the computation differentiable and accessible to backpropagation. Gradients can flow through our differentiable tree operations, allowing

---

[1] Department of Cognitive Science, Johns Hopkins University, Baltimore, MD, USA [2] Mila, Université de Montreal, Montreal, CA [3] School of Informatics, University of Edinburgh, Edinburgh, UK [4] Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA [5] Microsoft Research, Redmond, WA, USA. [†] Work partially carried out while at Microsoft Research. Correspondence to: Paul Soulos <psoulos1@jhu.edu>.

us to optimize the sequencing and blending of linear operations using nonlinear deep learning models to parameterize the decision space.

Employing TPRs to represent binary trees, we design a novel Differentiable Tree Machine architecture, *DTM*[1] (§3), capable of systematically manipulating binary trees (overview shown in Figure 1). The *DTM* architecture achieves near-perfect out-of-distribution generalization for the examined synthetic tree-transduction tasks, on which previous models such as Transformers, LSTMs, and their tree variants exhibit weak or no out-of-distribution generalization. A discussion of related work can be found in Appendix B.

2. Differentiable Tree Operations

In this work, we use a lossless encoding for structure in vector space. Given a tree depth limit of depth D, the total number of tree nodes is $N = (b^{D+1} - 1)/(b - 1)$ where b is the branching factor. We generate a set of N orthonormal role vectors of dimension $d_r = N$. For a particular position r_i in a tree, a filler f_i is assigned to this role by taking the outer product of the embedding vectors for the filler and the role: $f_i \otimes r_i$. The embedding of the entire structure is the sum over the individual filler-role combinations $T = \sum_{i=1}^{N} f_i \otimes r_i$. Since the role vectors are orthonormal, a filler f_i can be recovered from T by the inner product between T and r_i, $f_i = T r_i$.

Moving forward, we will focus on the case of binary trees ($b = 2$), which serve as the foundation for a substantial amount of symbolic AI research. From the orthonormal role set, we can generate matrices to perform the Lisp operators `car`, `cdr`, and `cons`. For a tree node reached from the root by following the path x, denote its role vector by r_x; e.g., r_{011} is the role vector reached by descending from the root to the left (0th) child, then the right (1st) child, then the right (1st) child. Let $P = \{r_x \| |x| < D\}$ be the roles for all paths from the root down to a depth less than D.

In order to extract the subtree which is the left child of the root (Lisp `car`), we need to zero out the root node and the right child subtree while moving each filler in the left subtree up one level. Extracting the right subtree (Lisp `cdr`) is a symmetrical process. This can be accomplished by:

$$\text{car}(T) = D_0 T; \; \text{cdr}(T) = D_1 T; \; D_c = I_F \otimes \sum_{x \in P} r_x r_{cx}^\top$$

where I is the identity matrix on filler space.

Lisp `cons` constructs a new binary tree given two trees to embed as the left- and right-child. In order to add a subtree as the cth child of a new root node, we define E_c to add c to the top of the path-from-the-root for each position:

$$\text{cons}(T_0, T_1) = E_0 T_0 + E_1 T_1; \; E_c = I_F \otimes \sum_{x \in P} r_{cx} r_x^\top$$

When performing `cons`, a new filler s can be placed at the parent node of the two subtrees T_0 and T_1 by adding $s \otimes r_{root}$ to the output of `cons`. Our model uses linear combination to blend the results of applying the three Lisp operations. The output of step $l \in 1 : L$, when operating on the arguments $\vec{T}^{(l)} = (T_{car}^{(l)}, T_{cdr}^{(l)}, T_{cons0}^{(l)}, T_{cons1}^{(l)})$, is

$$O^{(l)}(\vec{w}^{(l)}, \vec{T}^{(l)}, s^{(l)}) = w_{car}^{(l)} \text{car}(T_{car}^{(l)}) + w_{cdr}^{(l)} \text{cdr}(T_{cdr}^{(l)})$$
$$+ w_{cons}^{(l)} \left(\text{cons}(T_{cons0}^{(l)}, T_{cons1}^{(l)}) + s^{(l)} \otimes r_{root} \right) \quad (1)$$

The three operations are weighted by the level-specific weights $\vec{w}^{(l)} = (w_{car}^{(l)}, w_{car}^{(l)}, w_{cons}^{(l)})$, which sum to 1.

3. Differentiable Tree Machine (*DTM*) Architecture for Binary Tree Transformation

In order to actualize the theory described in Section 2, we introduce the Differentiable Tree Machine (*DTM*), a model that is capable of learning how to perform operations over binary trees. Since the primitive functions `car`, `cdr`, and `cons` are precomputed at initialization from the orthogonally generated role vectors, this learning problem reduces to learning which operations to perform on which trees in Tree Memory to arrive at a correct output. A high-level overview of our model is given in Figure 1. *DTM* consists of a learned component (Neural Tree Agent), a differentiable pre-designed tree interpreter described in Equation 1, and an external Tree Memory for storing trees.

At a given timestep l, our agent selects the inputs to Equation 1: the tree arguments for the operations ($\vec{T}^{(l)}$), the new root symbol for `cons` ($s^{(l)}$) and how much to weight the output of each operation ($\vec{w}^{(l)}$). To select $\vec{T}^{(l)}$, *DTM* produces coefficients over the trees in Tree Memory, where the coefficients across trees in $\vec{T}^{(l)}$ sum to 1. For example, if Tree Memory contains only T_0 & T_1, weights $\vec{a}_{car}^{(l)} = (a_{car,0}^{(l)}, a_{car,1}^{(l)})$ are computed to define the argument to `car`: $T_{car}^{(l)} = a_{car,0}^{(l)} T_0 + a_{car,1}^{(l)} T_1$, and similarly for `cdr` and the two arguments of `cons`. $\vec{a}_T^{(l)} = (\vec{a}_{car}^{(l)}; \vec{a}_{cdr}^{(l)}; \vec{a}_{cons0}^{(l)}; \vec{a}_{cons1}^{(l)})$ denotes all such weights.

These decisions are computed within the Neural Tree Agent module of *DTM* using a standard Transformer layer (Vaswani et al., 2017) consisting of multiheaded self-attention, a feedforward network, residual connections, and layer norm. Figure 4 shows the computation in a single step of *DTM*. When a binary tree is read from Tree Memory, it is compressed from the TPR dimension d_{tpr} to the Transformer input dimension d_{model} using a linear transformation $W_{shrink} \in \mathbb{R}^{d_{tpr} \times d_{model}}$. We also feed in two special tokens to encode the operation-weighting coeffi-

DATA SET	*DTM*	TRANSFORMER	LSTM	TREE2TREE	TREE TRANSFORMER
CAR_CDR_SEQ					
-TEST OOD LEXICAL	**.94 ± .04**	.66 ± .00	.66 ± .00	.66 ± .00	.66 ± .00
-TEST OOD STRUCTURAL	**.93 ± .04**	.68 ± .01	.47 ± .03	.74 ± .02	.64 ± .01
ACTIVE↔LOGICAL					
-TEST OOD LEXICAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.00 ± .00	.00 ± .00
-TEST OOD STRUCTURAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.10 ± .03	.03 ± .01
PASSIVE↔LOGICAL					
-TEST OOD LEXICAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.00 ± .00	.00 ± .00
-TEST OOD STRUCTURAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.19 ± .02	.00 ± .00
ACTIVE & PASSIVE→LOGICAL					
-TEST OOD LEXICAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.00 ± .00	.00 ± .00
-TEST OOD STRUCTURAL	**1.0 ± .00**	.00 ± .00	.00 ± .00	.10 ± .01	.01 ± .00

Table 1. Mean accuracy and standard deviation across five random initializations on synthetic tree-to-tree transduction tasks using different model architectures. All architectures achieve near perfect accuracy (¿95%) on the train and test IID splits across all four tasks. Test sets include in-distribution and out-of-distribution splits.

cients and the new root-symbol prediction. In addition to the standard parameters in a Transformer layer, our model includes three additional weight matrices $W_{op} \in \mathbb{R}^{d_{model} \times 3}$, $W_{root} \in \mathbb{R}^{d_{model} \times d_{symbol}}$, and $W_{arg} \in \mathbb{R}^{d_{model} \times 4}$. W_{op} projects the operation token encoding into logits for the three operations which are then normalized via softmax. W_{root} projects the root symbol token encoding into the new root symbol. W_{arg} projects the encoding of each TPR in memory to logits for the four tree arguments, the input to car, cdr, and cons left and right. The arguments for each operator are a linear combination of all the TPRs in memory, weighted by the softmax of the computed logits. These values are used to create the output for this step as described in equation 1 and the output TPR is written into Tree Memory at the next sequential slot. For the beginning of the next step, the contents of the Tree Memory are encoded to model dimension by W_{shrink} and appended to the Neural Tree Agent Transformer input sequence. The input to the Neural Tree Agent grows by one compressed tree encoding at each time step to incorporate the newly produced tree, as shown in Figure 5.

The tree produced by the final step of our network is used as the output (predicted tree). We minimize the mean-squared error between the predicted symbol at each node in the predicted tree and the target tree for all non-empty nodes in the target tree. We penalize the norm of filled nodes in the predicted tree that are empty in the target tree. Additional training details can be found in Appendix D.1.

4. Empirical Validation

4.1. Datasets and Baselines

We introduce four tree-to-tree transformation tasks inspired by semantic parsing and language generation with active

and passive voice: **CAR_CDR_SEQ**, **ACTIVE↔LOGICAL**, **PASSIVE↔LOGICAL**, and **ACTIVE & PASSIVE→LOGICAL**. Each task has two out-of-distribution splits to test lexical and structural generalization. The train split has 10,000 samples; all of the other splits have 1250 samples. Additional details about the three tasks are available in Appendix E.

We compare against standard seq2seq (Sutskever et al., 2014) models and tree2tree models as our baselines. For seq2seq models, we linearize our trees by coding them as left-to-right sequences with parentheses to mark the tree structure. Our seq2seq models are an Encoder-Decoder Transformer (Vaswani et al., 2017) and an LSTM (Hochreiter & Schmidhuber, 1997). We test two tree2tree models: Tree2Tree LSTM (Chen et al., 2018) and Tree Transformer Shiv & Quirk (2019). Training details for the baselines can be found in Appendix D.

4.2. Results

The results for *DTM* and the baselines can be seen in Table 1. All architectures achieve near perfect accuracy (¿95%) on the train and test IID splits across all four tasks. While the baselines perform similarly to *DTM* when compared on train and test IID, the results are drastically different when comparing the results across OOD splits. Across all tasks, *DTM* generalizes similarly regardless of the split, whereas the baselines struggle with lexical generalization and fail completely at structural generalization.

The baseline models perform the best on **CAR_CDR_SEQ**, whereas this is the most difficult task for *DTM*. We suspect that tuning the hyperparameters for *DTM* directly on this task would alleviate the less-than-perfect performance. Despite performing less than perfectly, *DTM* performance

on the OOD splits of **CAR_CDR_SEQ** outperforms all of the baselines. Whereas **CAR_CDR_SEQ** involves identifying a subtree (or subsequence for the baselines) within the input, the other four tasks involve reorganizing the input and potentially adding additional tokens in the case of **Passive↔Logical**. On these linguistically-motivated tasks, the baselines mostly achieve 0% OOD generalization, with a maximum of 19%.

DTM can be compared against the other tree models to see the effects of structured processing in vector space. While the Tree2Tree LSTM and Tree Transformer are both capable of representing trees, the processing that occurs over these trees is still black-box nonlinear transformations. *DTM* isolates black-box nonlinear transformations to the Neural Tree Agent, while the processing over trees is factorized into interpretable operations over tree structures with excellent OOD generalization. This suggests that it is not the tree encoding scheme itself that is critical, but rather the processing that occurs over the trees. Ablation experiments are shown in Appendix C.1.

4.3. Interpreting Inference as Programs

The output of the Neural Tree Agent at each timestep can be interpreted as routing data and performing a predefined operation. At convergence, we find that the path from the input tree to the output tree is defined by interpretable one-hot softmax distributions. An example of our model's behavior over 28 steps on Logical→Passive can be seen in Figure 2. In particular, we were excited to find an emergent operation in our model's behavior. Transducing from Logical→Passive not only requires rearranging nodes but also inserting new words into the tree, "was" and "by". At first glance, car, cdr, and cons do not appear to support adding a new node to memory. The model learns that taking cdr of a tree with only a single child returns an empty tree (Step 2 in Figure 2); the empty tree can then be used as the inputs to cons in order to write a new word as the root node with no children on the left or right (Step 3). The programmatic nature of our network at convergence — the fact that the weighting coefficients $\vec{w}, \vec{a}$ become 1-hot — makes it trivial to discover how an undefined operation emerged during training.

5. Conclusions, Limitations, and Future Work

We introduce *DTM*, an architecture for leveraging differentiable tree operations and an external memory to achieve compositional generalization. *DTM* achieves 100% out-of-distribution generalization for both lexical and structural distributional shifts across a variety of synthetic tree-to-tree tasks and is highly interpretable. Future work will focus on allowing *DTM* to work with unstructured data which will allow it to be evaluated on more datasets.
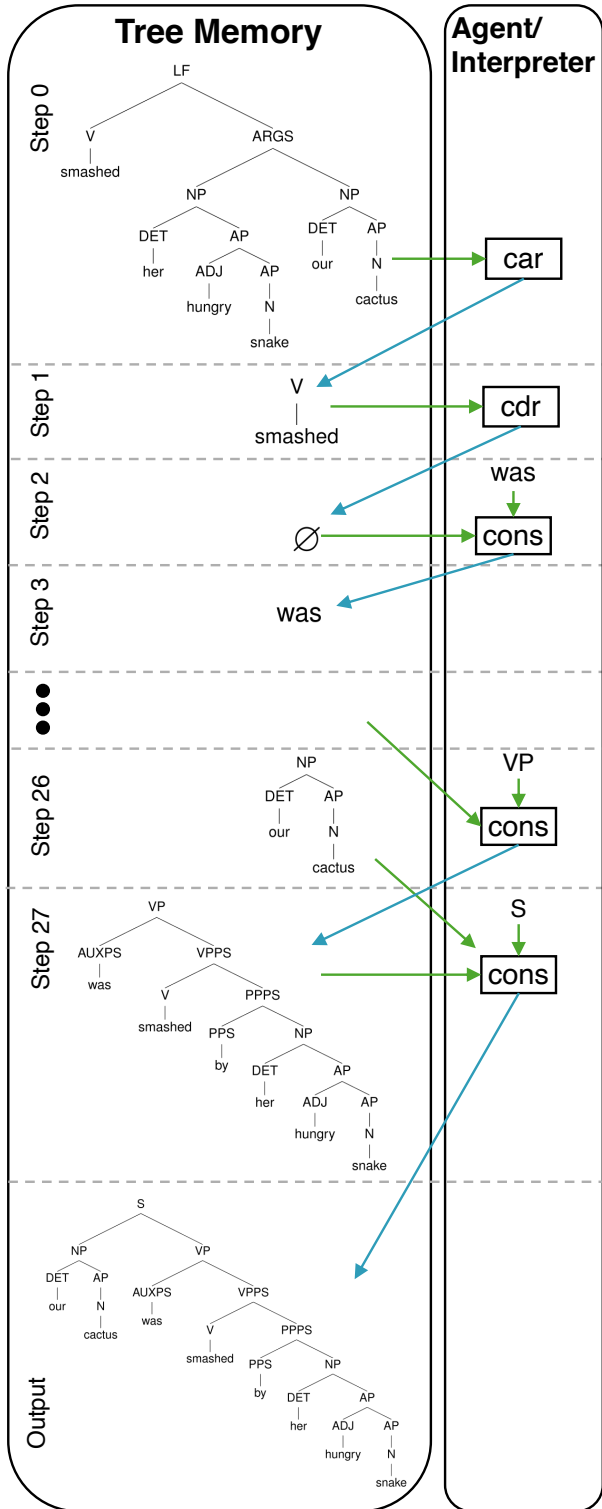

Figure 2. An interpretable transformation from logical form to passive. For readability, trees are shown here symbolically, but Tree Memory contains the vector embeddings (TPRs) of these trees. At each step, all of the items in memory from previous steps are available to the agent/interpreter. Reads are shown in green and writes in blue. The interpretation is discussed in Section 4.3.

Acknowledgements

References

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39–48, 2015.

Bošnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable forth interpreter. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 547–556. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/bosnjak17a.html.

Chen, K., Huang, Q., Palangi, H., Smolensky, P., Forbus, K., and Gao, J. Mapping natural-language problems to formal-language solutions using structured neural representations. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 1566–1575. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/chen20g.html.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

Csordás, R., Irie, K., and Schmidhuber, J. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 619–634, 2021.

Dong, L. and Lapata, M. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33–43, 2016.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., and Rasmussen, D. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012. doi: 10.1126/science.1225266. URL https://www.science.org/doi/abs/10.1126/science.1225266.

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

Gayler, R. W. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. In Slezak, P. (ed.), *Proceedings of the ICCS/ASCS Joint International Conference on Cognitive Science (ICCS/ASCS 2003)*, pp. 133–138, Sydney, NSW, AU, jul 2003. University of New South Wales. URL http://arxiv.org/abs/cs/0412059.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *ArXiv*, abs/1410.5401, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwinska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J. P., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476, 2016.

Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. Learning to transduce with unbounded memory. *ArXiv*, abs/1506.02516, 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Hupkes, D., Dankers, V., Mul, M., and Bruni, E. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, 67: 757–795, 2020.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rkE3y85ee.

Jiang, Y., Celikyilmaz, A., Smolensky, P., Soulos, P., Rao, S., Palangi, H., Fernandez, R., Smith, C., Bansal, M., and Gao, J. Enriching transformers with structured tensor-product representations for abstractive summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4780–4793, 2021.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, 2015.

Kanerva, P. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1: 139–159, 2009.

Kim, N. and Linzen, T. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 9087–9105, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.731. URL https://aclanthology.org/2020.emnlp-main.731.

Kim, N., Linzen, T., and Smolensky, P. Uncontrolled lexical exposure leads to overestimation of compositional generalization in pretrained models, 2022. URL https://arxiv.org/abs/2212.10769.

Kleyko, D., Rachkovskij, D. A., Osipov, E., and Rahimi, A. A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations. *ACM Comput. Surv.*, 55(6), dec 2022. ISSN 0360-0300. doi: 10.1145/3538531. URL https://doi.org/10.1145/3538531.

Kurach, K., Andrychowicz, M., and Sutskever, I. Neural random access machines. *ICLR*, 2016. URL http://arxiv.org/abs/1511.06392.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1jE5L5gl.

Marcus, G. F. *The algebraic mind: Integrating connectionism and cognitive science*. MIT press, 2003.

McCoy, R. T., Linzen, T., Dunbar, E., and Smolensky, P. RNNs implicitly implement tensor-product representations. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJx0sjC5FX.

Palangi, H., Smolensky, P., He, X., and Deng, L. Question-answering with grammatically-interpretable representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Plate, T. A. *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. CSLI Publications, USA, 2003. ISBN 1575864290.

Reed, S. E. and de Freitas, N. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

Sartran, L., Barrett, S., Kuncoro, A., Stanojević, M., Blunsom, P., and Dyer, C. Transformer grammars: Augmenting transformer language models with syntactic inductive biases at scale. *Transactions of the Association for Computational Linguistics*, 10:1423–1439, 2022.

Schlag, I. and Schmidhuber, J. Learning to reason with third order tensor products. *Advances in neural information processing systems*, 31, 2018.

Schlag, I., Smolensky, P., Fernandez, R., Jojic, N., Schmidhuber, J., and Gao, J. Enhancing the transformer with explicit relational encoding for math problem solving. *CoRR*, abs/1910.06611, 2019. URL http://arxiv.org/abs/1910.06611.

Shiv, V. and Quirk, C. Novel positional encodings to enable tree-based transformers. *Advances in neural information processing systems*, 32, 2019.

Smolensky, P. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artif. Intell.*, 46:159–216, 1990.

Soulos, P., McCoy, R. T., Linzen, T., and Smolensky, P. Discovering the compositional structure of vector representations with role learning networks. In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 238–254, 2020.

Soulos, P., Rao, S., Smith, C., Rosen, E., Celikyilmaz, A., McCoy, R. T., Jiang, Y., Haley, C., Fernandez, R., Palangi, H., et al. Structural biases for improving transformers on translation into morphologically rich languages. *Proceedings of Machine Translation Summit XVIII*, 2021.

Steele, G. *Common LISP: the language*. Elsevier, 1990.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1556–1566, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1150. URL https://aclanthology.org/P15-1150.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural*

Information Processing Systems, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

Wang, Y., Lee, H.-Y., and Chen, Y.-N. Tree transformer: Integrating tree structures into self-attention. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 1061–1070, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1098. URL `https://aclanthology.org/D19-1098`.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *CoRR*, abs/1410.3916, 2014.

A. *DTM* Figures
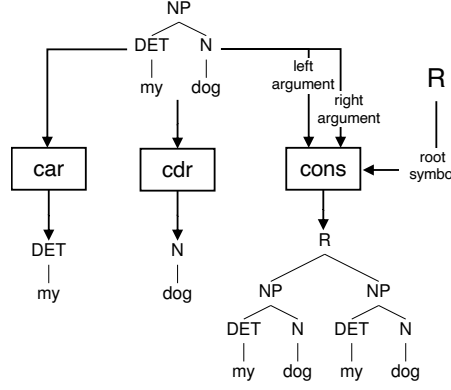
Figure 3. An example showing the output of our three operations.
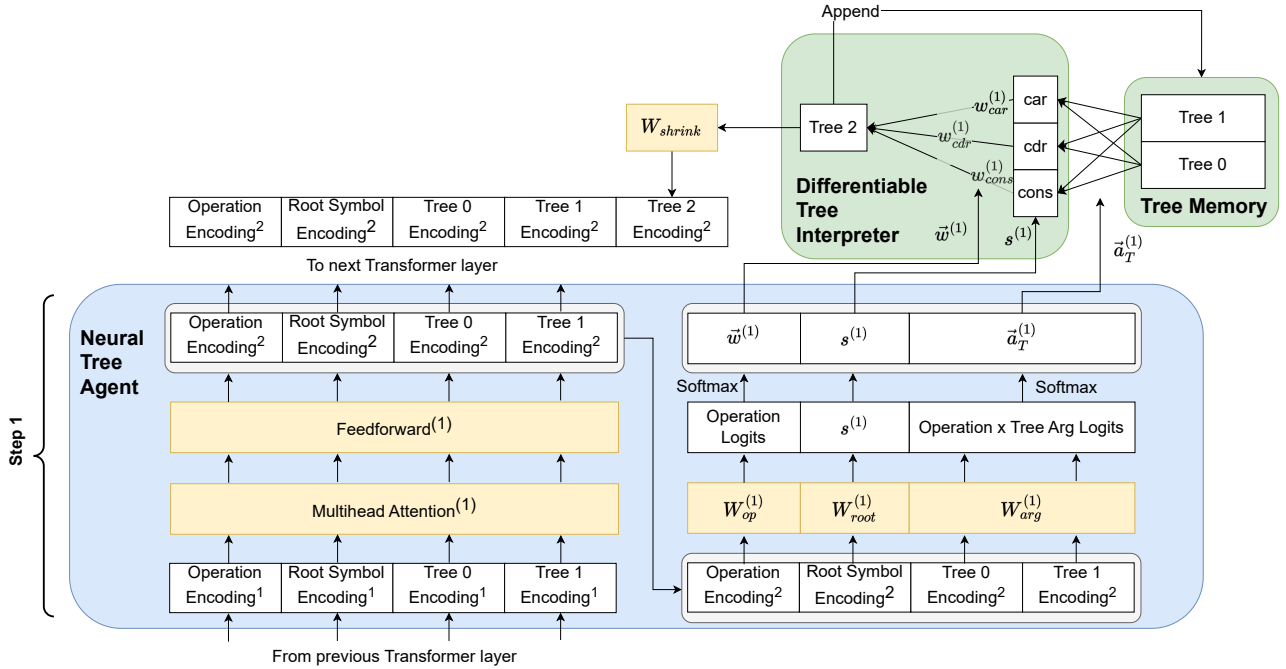
Figure 4. Step 1 of the *DTM* architecture is expanded to show the information flow. The yellow boxes identify the parameters that are learnable. The blue box highlights the Neural Tree Agent, and the green boxes highlight components in tree space: the Differentiable Tree Interpreter (Eq 1) and Tree Memory. The left side of the Neural Tree Agent is a standard transformer layer with self-attention and a feedforward network. Residual connections and layer norm are not shown.
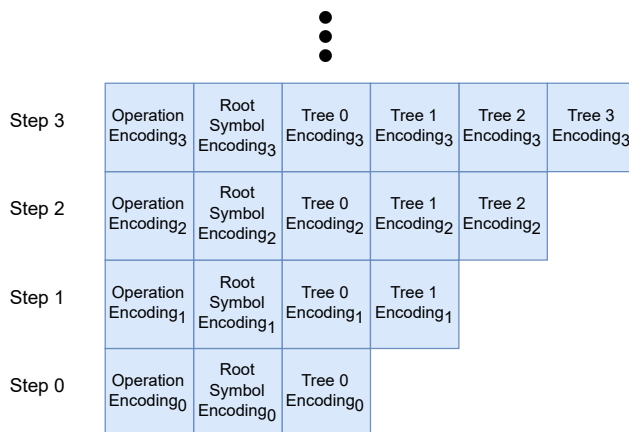
Figure 5. Inputs to the Neural Tree Agent at each step of processing. The length of the input grows by one token each step to incorporate the output of the previous step.

B. Related Work

B.1. Compositional Generalization

Research on compositional generalization has been one of the core issues in Machine Learning since its inception. Despite improvements in architectures and scalability (Csordás et al., 2021), neural network models still struggle with out-of-distribution generalization (Kim et al., 2022). The lack of robust compositional generalization has been a central argument against neural networks as models of cognition for almost half a century by proponents of GOFAI systems that leverage symbolic structures (e.g., Fodor & Pylyshyn, 1988; Marcus, 2003). These symbolic systems are brittle and face scalability problems because their nondifferentiability makes them incompatible with gradient learning methods. Our work attempts to bridge the neural network-symbolic divide by situating symbolic systems in vector space, where a first-order gradient can be derived as a learning signal.

In practice, the term "compositional generalization" has been associated with a range of different tasks (Hupkes et al., 2020). Kim & Linzen (2020) identify a key distinction relevant to natural language: lexical versus structural generalization. *Lexical* generalization is required when a model encounters a primitive (e.g., a word) in a structural environment (e.g., a position in a tree) where it has not been seen during training. *Structural* generalization is required when a model encounters a structure that was not seen during training, such as a longer sentence or a syntactic tree with new nodes. Kim et al. (2022) demonstrate that structural and lexical generalization remain unsolved: pretrained language models still do not consistently generalize fully to novel lexical items and structural positions. The tasks we study below explicitly test both types of compositional generalization (§4.1).

Our proposed *DTM* model encodes and manipulates data exclusively in the form of Tensor Product Representations (TPRs; §B.2). This formalism inherently supports composition and decomposition through symbol-role bindings, creating an inductive bias toward symbolic operations. *Lexical* generalization is straightforward when syntactic trees are encoded as TPRs: a novel symbol can easily bind to any role. *Structural* generalization is possible through our linear representation of the `car`, `cdr`, and `cons` functions, as these operators are not sensitive to the size or structure of the trees they take as arguments. We evaluate *DTM*'s capacity for both types of compositional generalization in §4.2.

B.2. Tensor Product Representations (TPRs)

Tensor Product Representations have been used to enhance performance and interpretability across textual question-answering (Schlag & Schmidhuber, 2018; Palangi et al., 2018), natural-language-to-program-generation (Chen et al., 2020), math problem solving (Schlag et al., 2019), synthetic sequence tasks (McCoy et al., 2019; Soulos et al., 2020), summarization (Jiang et al., 2021), and translation (Soulos et al., 2021). While previous work has focused on using TPRs to structure and interpret representations, the processing over these representations was done using black-box neural networks. In this work, we predefine structural operations to process TPRs and use black-box neural networks to parameterize the information flow

and decision making in our network.

B.3. Vector Symbolic Architectures

Vector Symbolic Architecture (VSA) (Gayler, 2003; Plate, 2003; Kanerva, 2009) is a computing framework for realizing classic symbolic algorithms in vector space. Our work bridges VSAs and Deep Learning by using black-box neural networks to write differentiable vector-symbolic programs. For a recent survey on VSAs, see Kleyko et al. (2022), and for VSAs with spiking neurons see Eliasmith et al. (2012).

B.4. Differentiable Computing

One approach to integrating neural computation and GOFAI systems is Differentiable Computing. In this approach, components of symbolic computing are re-derived in a continuous and fully differentiable manner to facilitate learning with backpropagation. In particular, neural networks that utilize an external memory have received considerable attention (Graves et al., 2014; 2016; Weston et al., 2014; Kurach et al., 2016).

Another significant aspect of Differentiable Computing involves integrating structured computation graphs into neural networks. Tree-LSTMs (Tai et al., 2015; Dong & Lapata, 2016; Chen et al., 2018) use parse trees to encode parent nodes in a tree from their children's representations or decode child nodes from their parent's representations. Some Transformer architectures modify standard multi-headed attention to integrate tree information (Wang et al., 2019; Sartran et al., 2022), while other Transformer architectures integrate tree information in the positional embeddings (Shiv & Quirk, 2019). Neural Module Networks (Andreas et al., 2015) represent a separate differentiable computing paradigm, where functions in a symbolic program are replaced with black-box neural networks.

A few works have explored using differentiable interpreters to learn subfunctions from program sketches and datasets (Bošnjak et al., 2017; Reed & de Freitas, 2015). Most similar to our work, Joulin & Mikolov (2015) and Grefenstette et al. (2015) learn RNNs capable of leveraging a stack with discrete push and pop operations in a differentiable manner. While they use a structured object to aid computation, the operations they perform involve read/write operations over unstructured vectors, whereas the operations we deploy in this work consist of structured operations over vectors with embedded structure.

C. Ablation Experiments

C.1. Ablations

In order to examine how the components of our model come together to achieve compositional generalization, we run several ablation experiments on **Active↔Logical**.

C.1.1. PRE-DEFINED STRUCTURAL OPERATIONS

What purpose do the `car`, `cdr`, and `cons` equations defined in Section 2 play in our network's success? Instead of defining the transformations with the equations, we can randomly initialize the D and E matrices and allow them to be learned during training. The results of learning the D and E matrices are shown in Table 2. Since the D and E matrices, whether predefined or learned, operate only on the role space, it is unsurprising that our model continues to achieve perfect lexical generalization without the predefined equations for D and E. However, structural generalization suffers dramatically when the D and E matrices are learned. This result indicates that the predefined tree operations are essential for our model to achieve structural generalization.

	PREDEFINED TRANSFORMATIONS	LEARNED TRANSFORMATIONS
-TRAIN	$1.0 \pm .00$	$1.0 \pm .00$
-TEST IID	$1.0 \pm .00$	$.99 \pm .02$
-LEXICAL	$1.0 \pm .00$	$.99 \pm .01$
-STRUCTURAL	$1.0 \pm .00$	$.35 \pm .08$

Table 2. Accuracy on Active↔Logical across five random initializations for models with predefined `car`, `cdr` and `cons` operations versus learned transformations. Lexical and structural are test OOD splits.

C.1.2. BLENDING VS. DISCRETE SELECTIONS

While our model converges to one-hot solutions where it chooses a single operation over a single tree in memory, it is not constrained to do so, and it deploys heavy blending prior to final convergence. There are two sources of blending: the input arguments to each operation can be a blend of trees in memory, and the output written to memory is a weighted blend of the three operations. We can explore the importance of blending by restricting our model to make discrete decisions using the Gumbel-Softmax distribution (Jang et al., 2017; Maddison et al., 2017). Table 3 shows the results of models trained with (blend-producing) softmax or (discrete) Gumbel-Softmax for argument and operation selection. We observe that the use of Gumbel-Softmax in either operation or argument sampling leads to a complete breakdown in performance. This demonstrates that blending is an essential component of our model, and that our network is not capable of learning the task without it.

ACTIVE↔LOGICAL	OP (SOFTMAX) ARG (SOFTMAX)	OP (SOFTMAX) ARG (GUMBEL)	OP (GUMBEL) ARG (SOFTMAX)	OP (GUMBEL) ARG (GUMBEL)
-TRAIN	1.00 ± 0.00	$.086 \pm .172$	0.00 ± 0.00	0.00 ± 0.00
-TEST IID	1.00 ± 0.00	$.088 \pm .176$	0.00 ± 0.00	0.00 ± 0.00
-TEST OOD LEXICAL	1.00 ± 0.00	$.094 \pm .188$	0.00 ± 0.00	0.00 ± 0.00
-TEST OOD STRUCTURAL	1.00 ± 0.00	$.068 \pm .136$	0.00 ± 0.00	0.00 ± 0.00

Table 3. Accuracy on Active↔Logical across five random initializations for models which use varying combinations of softmax and Gumbel-Softax for operation and argument selection.

D. Model Hyperparameter Selection

For all of the models we evaluated, the HP searching and training was done in 3 steps:

1. An optional exploratory random search over a wide range of HP values (using the Active↔Logical task)

2. A grid search (repeat factor=3) over the most promising HP values from step 1 (using the Active↔Logical task)

3. Training on the target tasks (repeat factor=5)

All of our models were trained on 1x V100 (16GB) virtual machines.

D.1. *DTM*

For the *DTM* models, we ran a 3x hyperparameter grid search over the following ranges. The best performing hyperparameter values are marked in bold.

Computation Steps:	[X+2, **(X+2)*2**] where X is the minimum number of steps required to complete a task
weight_decay:	[**.1**, .01]
Transformer model dimension:	[32, **64**]
Adam β_2:	[.98, **.95**]
Transformer dropout:	[0, **.1**]

The following hyperparameters were set for all models

lr_warmup:	[10000]
lr_decay:	[cosine]
training steps:	[20000]
Transformer encoder layers per computation step:	[1]
Transformer # of heads:	[4]
Batch size:	[16]
d_symbol:	# symbols in the dataset
d_role:	$2^{D+1} - 1$ where D is the max depth in the dataset
Transformer non-linearity:	gelu
Optimizer:	Adam
Adam β_1:	.9
Gradient clipping:	1
Transformer hidden dimension:	4x Transformer model dimension

Notes:

- For the Passive↔Logical task, a batch size of 8 was used to reduce memory requirements.

- Training runs that didn't achieve 90% training accuracy were excluded from evaluation

D.2. Baselines

We search over model and training hyperparameters and choose the combination that has the highest (and in the case of ties, quickest to train) mean validation accuracy on Active & Passive→Logical. The best hyperparameter setting for each model was then used to train that model on all four of our tasks.

D.2.1. TRANSFORMER

The Transformer 1x exploratory random search operated on the following HP values:

lr:	[.0001, .00005]
lr_warmup:	[0, 1000, 3000, 6000, 9000]
lr_decay:	[none, linear, factor, noam]
lr_decay_factor:	[.9, .95, .99]
lr_patience:	[0, 5000]
stop_patience:	[0]
weight_decay:	[0, .001, .01]
hidden:	[64, 96, 128, 256, 512, 768, 1024]
n_encoder_layers:	[1, 2, 3, 4, 5, 6, 7, 8]
n_decoder_layers:	[1, 2, 3, 4, 5, 6, 7, 8]
dropout:	[0, .1, .2, .3, .4]
filter:	[256, 512, 768, 1024, 2048, 3096, 4096]
n_heads:	[1, 2, 4, 8, 16]

The Transformer 3x grid search operated on the following HP values:

hidden:	[768, 1024]
n_encoder_layers:	[1, 4]
n_decoder_layers:	[3, 4]
dropout:	[0]
filter:	[768, 1024]
n_heads:	[2, 4]

The Transformer 5x training on the target tasks was done with these final HP values:

n_steps:	30_000
log_every:	100
eval_every:	1000
batch_size_per_gpu:	256
max_tokens_per_gpu:	20_000
lr:	.0001
lr_warmup:	1000
lr_decay:	linear
lr_decay_factor:	.95
lr_patience:	5000
stop_patience:	0
optimizer:	adam
weight_decay:	0
max_abs_grad_norm:	1
grad_accum_steps:	1
greedy_must_match_tf:	0
early_stop_perfect_eval:	0
hidden:	1024
n_encoder_layers:	1
n_decoder_layers:	3
dropout:	0
filter:	1024
n_heads:	2

D.2.2. LSTM

The LSTM 1x exploratory random search operated on the following HP values:

weight_decay:	[0, .001, .01]
lr:	[.0001, .00005]
lr_warmup:	[0, 1000, 3000, 6000, 9000]
lr_decay:	[none, linear, factor, noam]
lr_decay_factor:	[.9, .95, .99]
lr_patience:	[0, 5000]
hidden:	[64, 96, 128, 256, 512, 768, 1024]
n_encoder_layers:	[1, 2, 3, 4, 5, 6]
n_decoder_layers:	[1, 2, 3, 4, 5, 6]
dropout:	[0, .05, .1, .15, .2]
bidir:	[0, 1]
use_attn:	[0, 1]
rnn_fold:	[min, max, mean, sum, hadamard]
attn_inputs:	[0, 1]

The LSTM 3x grid search operated on the following HP values:

lr_decay:	[linear, noam]
hidden:	[512, 768, 1024]
n_encoder_layers:	[1, 6]
n_decoder_layers:	[1, 2, 3]
attn_inputs:	[0, 1]

The LSTM 5x training on the target tasks was done with these final HP values:

n_steps:	30_000
log_every:	200
eval_every:	1000
stop_patience:	0
optimizer:	adam
max_abs_grad_norm:	1
grad_accum_steps:	1
greedy_must_match_tf:	0
early_stop_perfect_eval:	0
batch_size_per_gpu:	256
max_tokens_per_gpu:	20_000
weight_decay:	0
lr:	.0001
lr_warmup:	1000
lr_decay:	noam
lr_decay_factor:	.95
lr_patience:	5000
hidden:	512
n_encoder_layers:	6
n_decoder_layers:	1
dropout:	.1
bidir:	0
use_attn:	1
rnn_fold:	max
attn_inputs:	1

D.2.3. TREE2TREE

The Tree2Tree 1x exploratory random search operated on the following HP values:

lr:	[.01, .005, .001, .0005]
lr_decay_factor:	[.8, .9, .95]
max_abs_grad_norm:	[1, 5]
hidden:	[64, 128, 256, 512, 768]
dropout:	[0, .1, .2, .4, .5, .6]
lr_decay:	[none, linear, factor, patience, noam]

The Tree2Tree 3x grid search operated on the following HP values:

dropout:	[0, .6]
hidden:	[512, 768]

The Tree2Tree 5x training on the target tasks was done with these final HP values:

n_steps:	[10_000]
stop_patience:	[0]
early_stop_perfect_eval:	[0]
lr:	[.0005]
lr_decay:	[patience]
lr_warmup:	[1000]
lr_patience:	[500]
lr_decay_factor:	[.95]
batch_size_per_gpu:	[256]
max_tokens_per_gpu:	null
optimizer:	[adam]
weight_decay:	[0]
max_abs_grad_norm:	[1]
grad_accum_steps:	[1]
n_encoder_layers:	[1]
n_decoder_layers:	[1]
dropout:	[0]
hidden:	[512]

D.2.4. TREETRANSFORMER

The TreeTransformer 1x exploratory random search operated on the following HP values:

dropout_rate:	[0, .05, .1, .2]
batch_size:	[64, 128, 256]
learning_rate:	[.0001, .0005, .001, .00001]
optimizer:	[adam, sgd, momentum, adagrad, adadelta, rmsprop]
max_gradient_norm:	[0.0, .05, .1]
momentum:	[0.0, .1, .5, .9]
d_model:	[128, 256]
d_ff:	[128, 256, 512, 1024]
encoder_depth:	[1, 2, 3, 4]
decoder_depth:	[2, 4, 6, 8]

The TreeTransformer 3x grid search operated on the following HP values:

optimizer:	[adagrad, adadelta]
max_gradient_norm:	[.05, .1]
momentum:	[0.0, .5]
d_model:	[256]
d_ff:	[256]
encoder_depth:	[1, 2]
decoder_depth:	[2, 4]

The TreeTransformer 5x training on the target tasks was done with these final HP values:

train_batches:	[30_000]
max_eval_steps:	[2000]
dropout_rate:	[0]
batch_size:	[256]
learning_rate:	[.0001]
optimizer:	[adagrad]
max_gradient_norm:	[.05]
momentum:	[.5]
num_heads:	[2]
d_model:	[256]
d_ff:	[256]
encoder_depth:	[1]
decoder_depth:	[2]

E. Basic Sentence Transforms

We introduce the **Basic Sentence Transforms** dataset for testing tree-to-tree transformations. It contains various synthetic tree-transform tasks, including a Lisp function interpreter task and several natural-language tasks inspired by semantic parsing and language generation. This dataset is designed to test compositional generalization in structure transformations, as opposed to most existing compositionality-related datasets, which focus on linear sequence transformations.

Each task in the dataset has five splits: train, validation, test, out-of-distribution lexical (OOD-lexical), and out-of-distribution structural (OOD-structural). The OOD-lexical split tests a model's ability to perform zero-shot lexical generalization to new adjectives not seen during training. The OOD-structural split tests a model's structural generalization by using longer adjective sequences and new tree positions not encountered during training. The train split has 10,000 samples, while the other splits have 1,250 samples each. Samples of these tasks are shown in Appendix E.2 and additional information about the construction of the dataset is in Appendix E.1. We focus our evaluation on the following four tasks:

CAR-CDR-SEQ is a tree transformation task where the source tree represents a template-generated English sentence, and the target tree represents a subset of the source tree. The target tree is formed from a sequence of Lisp `car` and `cdr` operations on the source tree. The desired sequence of operations is encoded into a single token in the source tree root, and the transformation requires learning how to interpret this root token and execute the associated sequence of actions. Although its internal structure is not accessible to the model, the token is formed according to the Lisp convention for combining these operations into a single symbol (starting with a `c`, followed by the second letter of each operation, and terminated by an `r`, e.g., `cdaadr` denotes the operation sequence: `cdr`, `car`, `car`, `cdr`). This task uses sequences of 1-4 `car`/`cdr` operations (resulting in 30 unique functions).

Active↔Logical contains syntax trees in active voice and logical form. Transforming from active voice into logical form is similar to semantic parsing, and transducing from logical form to active voice is common in natural language generation. An example from this task is shown in Figure 1.

Passive↔Logical contains syntax trees in passive voice and logical form. This task is similar to the one above but is more difficult and requires more operations. The passive form also contains words that are not present in logical form, so unlike Active↔Logical, the network needs to insert additional nodes. At first glance, this does not seem possible with `car`, `cdr`, and `cons`; we will show how our network manages to solve this problem in an interpretable manner in §4.3. An example from this task is shown in Figure 2.

Active & Passive→Logical contains input trees in either active or passive voice and output trees in logical form. This tests whether a model can learn to simultaneously parse different types of trees, distinguished by their structures, into a shared logical form.

E.1. Dataset Construction

The Basic Sentence Transforms vocabulary size and tree depth are available below. The lexical splits are constructed by using 1 set of adjectives for the Train, Dev, Test IID, and OOD Structural splits, and a disjoint set for the OOD Lexical split. The structural splits are constructed by using 0-2 nested adjectives distributed randomly to the two noun phrases for train, dev, and OOD Lexical splits, and 3-4 nested adjectives to the two noun phrases for the OOD Structural split. Adjective phrases are nested within each other within a noun phrase, so each additional adjective increases the overall tree depth by 1.

Vocabulary Size

DATASET	TRAIN/DEV/TEST	TEST OOD LEXICAL	TEST OOD STRUCTURAL
CAR_CDR_SEQ	142	153	142
ACTIVE↔LOGICAL	101	112	101
PASSIVE↔LOGICAL	107	118	107
ACTIVE & PASSIVE→LOGICAL	105	116	105

Tree Depth

DATASET	TRAIN/DEV/TEST	TEST OOD LEXICAL	TEST OOD STRUCTURAL
CAR_CDR_SEQ	10	10	12
ACTIVE↔LOGICAL	8	8	10
PASSIVE↔LOGICAL	10	10	12
ACTIVE & PASSIVE→LOGICAL	10	10	12

E.2. Dataset Samples

This appendix contains samples of the 4 tasks that we used from the Basic Sentence Transforms Dataset.

E.2.1. **CAR-CDR-SEQ** SAMPLES

Source Tree:

(CDDDDR (NP (DET the) (AP (N goat))) (VP (AUXPS was) (VPPS (V bought) (PPPS (PPS by) (NP (DET the) (AP (ADJ round) (AP (N rose))))))))
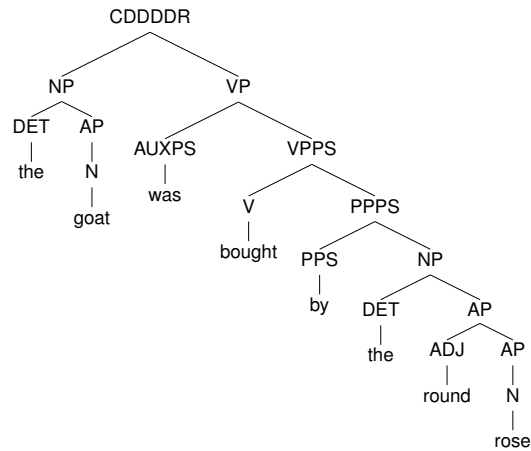
Target (Gold) Tree:

(NP (DET (the)) (AP (ADJ (round)) (AP (N (rose)))))
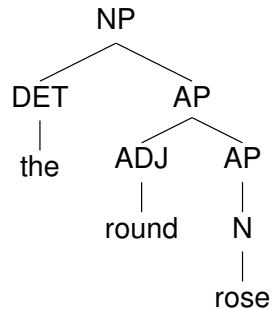
E.2.2. **ACTIVE↔LOGICAL** SAMPLES

Source Tree:

(S (NP (DET some) (AP (N crocodile)))) (VP (V washed) (NP (DET our) (AP (ADJ happy) (AP (ADJ thin)
(AP (N donkey))))))))

```
                                      S
                          ┌───────────┴───────────┐
                         NP                        VP
                    ┌─────┴─────┐            ┌──────┴──────┐
                   DET          AP           V             NP
                    │           │            │        ┌─────┴─────┐
                  some          N          washed     DET          AP
                                │                      │      ┌─────┴─────┐
                             crocodile                our    ADJ          AP
                                                             │       ┌─────┴─────┐
                                                           happy    ADJ          AP
                                                                     │       ┌────┴────┐
                                                                    thin     N
                                                                             │
                                                                          donkey
```
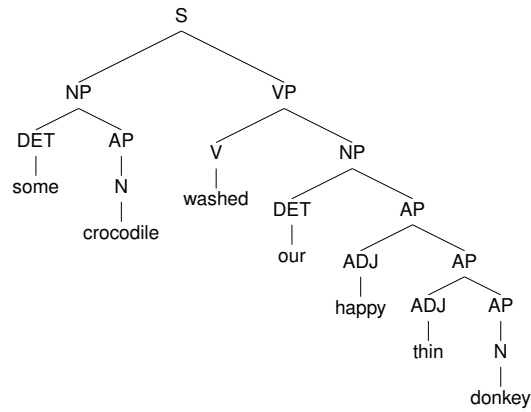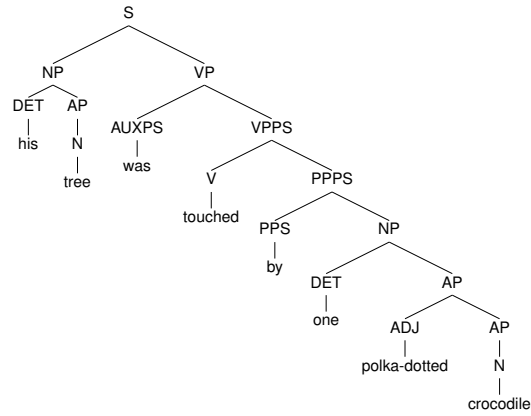
Target (Gold) Tree:

(LF (V washed) (ARGS (NP (DET some) (AP (N crocodile))) (NP (DET our) (AP (ADJ happy) (AP (ADJ
thin) (AP (N donkey))))))))

```
                                     LF
                          ┌───────────┴───────────┐
                          V                       ARGS
                          │              ┌──────────┴──────────┐
                        washed          NP                     NP
                                   ┌─────┴─────┐          ┌─────┴─────┐
                                  DET          AP        DET          AP
                                   │           │          │      ┌─────┴─────┐
                                 some          N         our    ADJ          AP
                                               │                 │       ┌─────┴─────┐
                                            crocodile          happy    ADJ          AP
                                                                         │       ┌────┴────┐
                                                                        thin     N
                                                                                 │
                                                                              donkey
```
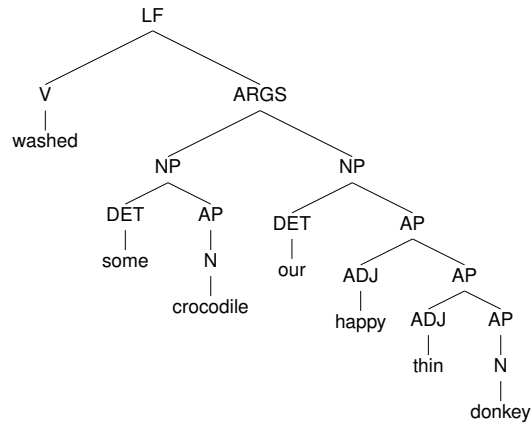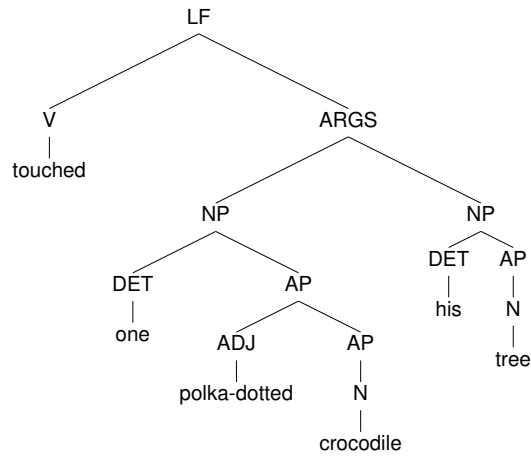
E.2.3. **PASSIVE↔LOGICAL** SAMPLES

Source Tree: (S (NP (DET his) (AP (N tree))) (VP (AUXPS was) (VPPS (V touched) (PPPS (PPS (PPS by) (NP (
DET one) (AP (ADJ polka-dotted) (AP (N crocodile)))))))))

Target (Gold) Tree: (LF (V touched) (ARGS (NP (DET one) (AP (ADJ polka-dotted) (AP (N crocodile)))) (NP (DET his) (AP (N tree)))))

E.2.4. ACTIVE & PASSIVE→LOGICAL SAMPLES

Source Tree: (S (NP (DET a) (AP (N fox))) (VP (AUXPS was) (VPPS (V kissed) (PPPS (PPS by) (NP (DET my) (AP (ADJ blue) (AP (N giraffe))))))))
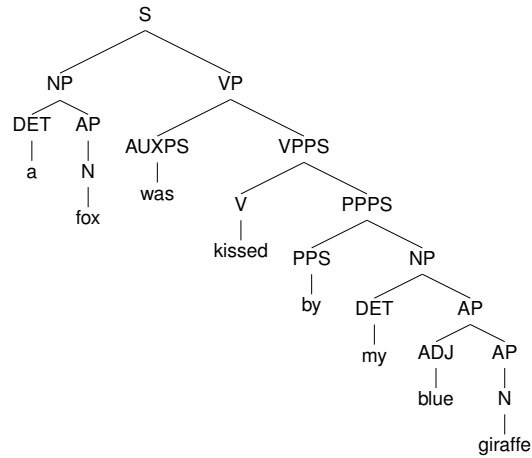
Target (Gold) Tree: (LF (V kissed) (ARGS (NP (DET my) (AP (ADJ blue) (AP (N giraffe)))) (NP (DET a) (AP (N fox)))))