Differentiable Tree Operations Promote Compositional Generalization

Authors

Author #1, Author #2, Author #3, Author #4, Author #5, Author #6, and Author #7

Proceedings of the Society for Computation in Linguistics (SCiL) 2023, pages 364-374.
Amherst, Massachusetts, June 15-17, 2023

Differentiable Tree Operations Promote Compositional Generalization

Paul Soulos
Johns Hopkins University
psoulos1@jhu.edu

Edward Hu
Université de Montreal

Kate McCurdy
University of Edinburgh

Yunmo Chen
Johns Hopkins University

Roland Fernandez
Microsoft Research

Paul Smolensky
Johns Hopkins University
Microsoft Research

Jianfeng Gao
Microsoft Research

1 Introduction

Reasoning within the symbolic space through discrete symbolic operations can lead to improved out-of-distribution generalization and enhanced interpretability. Despite the significant advances in representation learning made by modern deep learning, learning to directly manipulating 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 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). Tensor Product Representation (TPR) provides a general encoding of structured symbolic objects in vector space (Smolensky, 1990). Crucially, within the TPR space, the three Lisp operators on discrete objects become linear operators on continuous vectors. 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 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. The *DTM* architecture achieves perfect out-of-distribution generalization for the examined synthetic tree-transduction tasks,

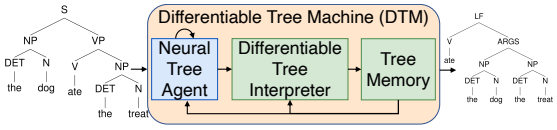

Figure 1: A high level overview of our model which consists of three modules. The Neural Tree Agent is a learnable neural network which, at each step of processing, selects the operation to perform and the arguments over which to operate. The Differentiable Tree Interpreter is an analytical function which compiles high level symbolic operations into subsymbolic matrix operations on tensors.

on which previous models exhibit partial or no out-of-distribution generalization. A discussion of related work can be found in Appendix A.

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 in a tree r_i, a filler f_i is assigned to this role by taking the outer product of the filler vector and role vector $f_i \otimes r_i$. The value of the entire structure is the sum over the individual filler-role combinations $T = \sum_i^N f_i \otimes r_i$. Since the role vectors are orthonormal, a filler f_i can be recovered from T by the inner product between T and r_i, $f_i = Tr_i$.

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

[1]Code available at https://github.com/psoulos/dtm.

(1st) child, then the right (1st) child. Let $P = \{r_x \| \, |x| < D\}$ be the set of 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:

$\texttt{car}(T) = D_0 T; \texttt{cdr}(T) = D_1 T; D_c = I_F \otimes \sum r_x r_{cx}^\top$

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:

$\texttt{cons}(T_0, T_1) = E_0 T_0 + E_1 T_1; E_c = I_F \otimes \sum r_{cx} r_x^\top$

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 $\vec{T}^{(l)} = (T^{(l)}_{\texttt{car}}, T^{(l)}_{\texttt{cdr}}, T^{(l)}_{\texttt{cons0}}, T^{(l)}_{\texttt{cons1}})$, is

$$O^{(l)}(\vec{w}^{(l)}, \vec{T}^{(l)}, s^{(l)}) = w^{(l)}_{\texttt{car}} \texttt{car}(T^{(l)}_{\texttt{car}}) + w^{(l)}_{\texttt{cdr}} \texttt{cdr}(T^{(l)}_{\texttt{cdr}}) + w^{(l)}_{\texttt{cons}} \left(\texttt{cons}(T^{(l)}_{\texttt{cons0}}, T^{(l)}_{\texttt{cons1}}) + s^{(l)} r_{root}^\top \right) \quad (1)$$

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

3 Differentiable Tree Machine (*DTM*) Architecture for Binary Tree Translation

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 $\vec{a}^{(l)}_{\texttt{car}} = (a^{(l)}_{\texttt{car},0}, a^{(l)}_{\texttt{car},1})$ are computed to define the argument to `car`: $T^{(l)}_{\texttt{car}} = a^{(l)}_{\texttt{car},0} T_0 + a^{(l)}_{\texttt{car},1} T_1$, and similarly for `cdr` and the two arguments of `cons`. $\vec{a}^{(l)}_T = (\vec{a}^{(l)}_{\texttt{car}}; \vec{a}^{(l)}_{\texttt{cdr}}; \vec{a}^{(l)}_{\texttt{cons0}}; \vec{a}^{(l)}_{\texttt{cons1}})$ 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. Appendix Figure 3 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 coefficients 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. 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 Transformer grows by one compressed tree encoding at each time step to incorporate the newly produced tree, as shown in Appendix Figure 4.

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 ground-truth tree. Additional training details

Data set	*DTM*	Transformer	LSTM	Tree2Tree	Tree Transformer
Active↔Logical					
-train	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00
-test IID	1.0 ± .00	1.0 ± .00	1.0 ± .00	.99 ± .00	1.0 ± .00
-test OOD lexical	**1.0 ± .00**	.31 ± .00	.31 ± .00	.00 ± .00	.00 ± .00
-test OOD structural	**1.0 ± .00**	.00 ± .00	.00 ± .00	.10 ± .03	.00 ± .00
Passive↔Logical					
-train	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00
-test IID	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00
-test OOD lexical	**1.0 ± .00**	.36 ± .00	.37 ± .00	.00 ± .00	.00 ± .00
-test OOD structural	**1.0 ± .00**	.00 ± .00	.00 ± .00	.19 ± .02	.00 ± .00
Active & Passive→Logical					
-train	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00	1.0 ± .00
-test IID	1.0 ± .00	1.0 ± .00	1.0 ± .00	.99 ± .00	1.0 ± .00
-test OOD lexical	**1.0 ± .00**	.32 ± .00	.32 ± .00	.00 ± .00	.00 ± .00
-test OOD structural	**1.0 ± .00**	.00 ± .00	.00 ± .00	.10 ± .02	.00 ± .00

Table 1: Mean accuracy and standard deviation across 5x random initializations on synthetic tree-to-tree transduction tasks using different model architectures. Test sets include independent in-distribution (IID) and out-of-distribution splits (OOD).

can be found in Section C.1.

4 Empirical Validation

4.1 Datasets and Baselines

We introduce three tree-to-tree transformation tasks inspired by semantic parsing and language generation with active and passive voice: **Active↔Logical**, **Passive↔Logical**, and **Active & Passive→Logical** [2]. 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 D.

We compare *DTM* against seq2seq and tree2tree models as our baselines. For seq2seq models, we linearize our trees by coding them as a left-to-right sequences with parentheses to mark the tree structure. Our seq2seq models are Transformer (Vaswani et al., 2017) and LSTM (Hochreiter and Schmidhuber, 1997). Our tree2tree models are Tree2Tree LSTM (Chen et al., 2018) and Tree Transformer (Shiv and Quirk, 2019). Model details are provided in Appendix E.

4.2 Results

The results for *DTM* and the baselines can be seen in Table 1. *DTM* achieves 100% accuracy across all splits for the three 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. This is in line with research showing that current models struggle much more with structural generalization than lexical generalization (Kim et al., 2022).

DTM can be compared against the other tree models to see the effects of learning 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.

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

[2]Data available at https://huggingface.co/datasets/rfernand/nc_pat.

distributions. We can trace the program execution to see how the input tree is transformed into the output tree. 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 (the third step 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 (the fourth step). 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 Conclusion

We introduce *DTM*, an architecture for leveraging differentiable tree operations and an external memory to achieve compositional generalization. *DTM* outperforms baselines across a variety of synthetic tree-to-tree tasks. Future work will focus on allowing *DTM* to work with unstructured data which will allow it to be evaluated on more datasets.

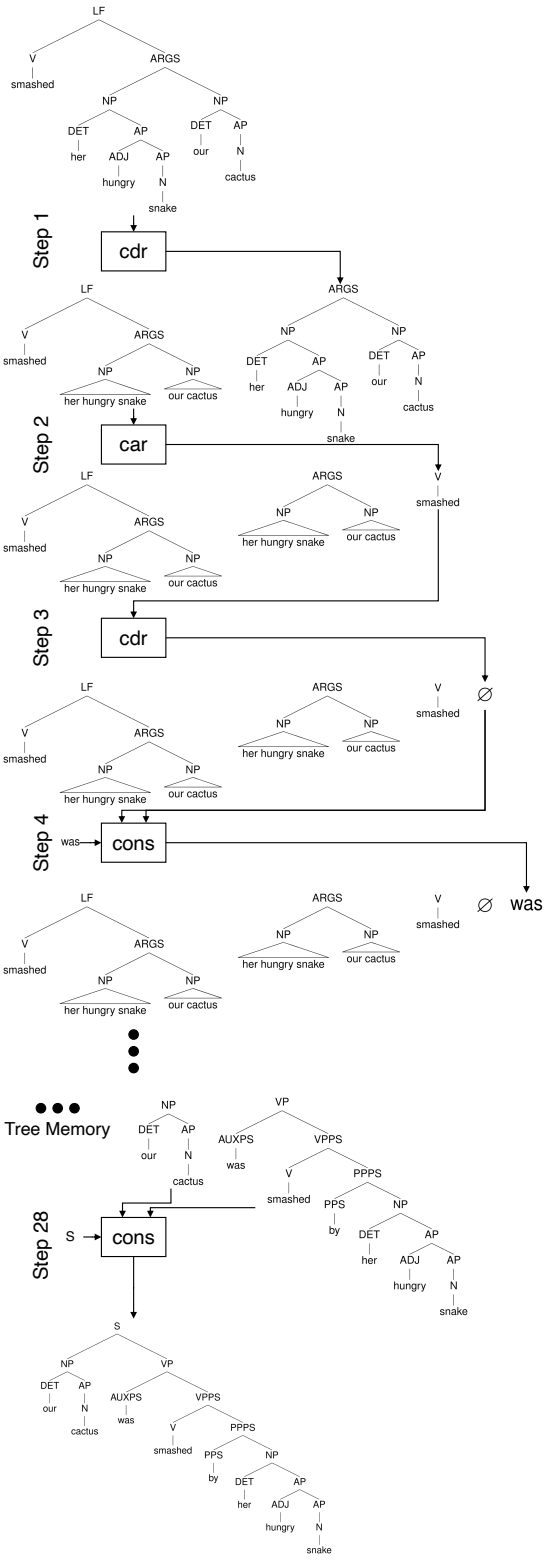

Figure 2: An interpretable transformation from logical form to passive. The interpretation is discussed in Section 4.3.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2015. Neural module networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 39–48.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a differentiable forth interpreter. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 547–556. PMLR.

Kezhen Chen, Qiuyuan Huang, Hamid Palangi, Paul Smolensky, Ken Forbus, and Jianfeng Gao. 2020. Mapping natural-language problems to formal-language solutions using structured neural representations. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1566–1575. PMLR.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31.

Róbert Csordás, Kazuki Irie, and Juergen Schmidhuber. 2021. 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*, pages 619–634.

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

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

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *ArXiv*, abs/1410.5401.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John P. Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. *ArXiv*, abs/1506.02516.

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

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

Yichen Jiang, Asli Celikyilmaz, Paul Smolensky, Paul Soulos, Sudha Rao, Hamid Palangi, Roland Fernandez, Caitlin Smith, Mohit Bansal, and Jianfeng Gao. 2021. 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*, pages 4780–4793.

Armand Joulin and Tomas Mikolov. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

Najoung Kim, Tal Linzen, and Paul Smolensky. 2022. Uncontrolled lexical exposure leads to overestimation of compositional generalization in pretrained models.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. 2016. Neural random access machines. *ICLR*.

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

R. Thomas McCoy, Tal Linzen, Ewan Dunbar, and Paul Smolensky. 2019. RNNs implicitly implement tensor-product representations. In *International Conference on Learning Representations*.

Hamid Palangi, Paul Smolensky, Xiaodong He, and Li Deng. 2018. Question-answering with grammatically-interpretable representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

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

Laurent Sartran, Samuel Barrett, Adhiguna Kuncoro, Miloš Stanojević, Phil Blunsom, and Chris Dyer. 2022. Transformer grammars: Augmenting transformer language models with syntactic inductive biases at scale. *Transactions of the Association for Computational Linguistics*, 10:1423–1439.

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

Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. 2019. Enhancing the transformer with explicit relational encoding for math problem solving. *CoRR*, abs/1910.06611.

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

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

Paul Soulos, R Thomas McCoy, Tal Linzen, and Paul Smolensky. 2020. 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*, pages 238–254.

Paul Soulos, Sudha Rao, Caitlin Smith, Eric Rosen, Asli Celikyilmaz, R Thomas McCoy, Yichen Jiang, Coleman Haley, Roland Fernandez, Hamid Palangi, et al. 2021. Structural biases for improving transformers on translation into morphologically rich languages. *Proceedings of Machine Translation Summit XVIII*.

Guy Steele. 1990. *Common LISP: the language*. Elsevier.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. 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)*, pages 1556–1566, Beijing, China. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Yaushian Wang, Hung-Yi Lee, and Yun-Nung Chen. 2019. 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)*, pages 1061–1070, Hong Kong, China. Association for Computational Linguistics.

Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory networks. *CoRR*, abs/1410.3916.

A Related Work

A.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 and Pylyshyn, 1988; Marcus, 2003). These symbolic systems are brittle and face scalability problems due to their incompatibility with differentiable 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 and 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. Kim et al. (2022) demonstrate that lexical generalization remains unsolved: pretrained language models still do not consistently generalize fully novel lexical items. *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. 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; §A.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.

A.2 Tensor Product Representations (TPRs)

Tensor Product Representations have been used to enhance performance and interpretability across textual question-answering (Schlag and 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.

A.3 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 faciliate 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 and 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 and 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 and de Freitas, 2015). Most similar to our work, Joulin and Mikolov (2015); Grefenstette et al. (2015) learn an RNN 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.

B DTM Figures

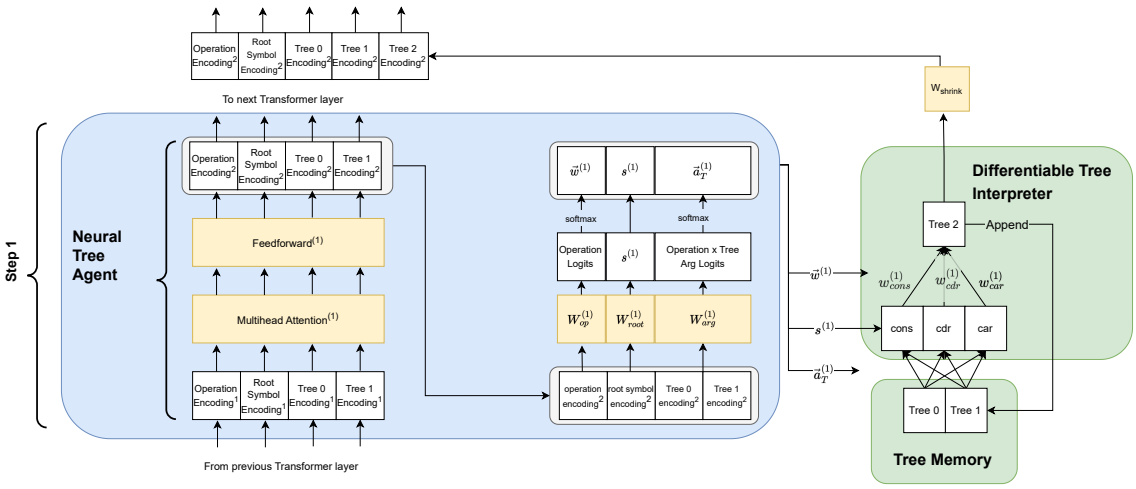

Figure 3: Step 1 of the *DTM* architecture is expanded to show the information flow in *DTM*. The yellow boxes show which parameters are learnable. The blue box highlights the Neural Tree Agent, and the green boxes highlight components in tree space: the Differentiable Tree Interpreter (Eq 1) and Tree 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.

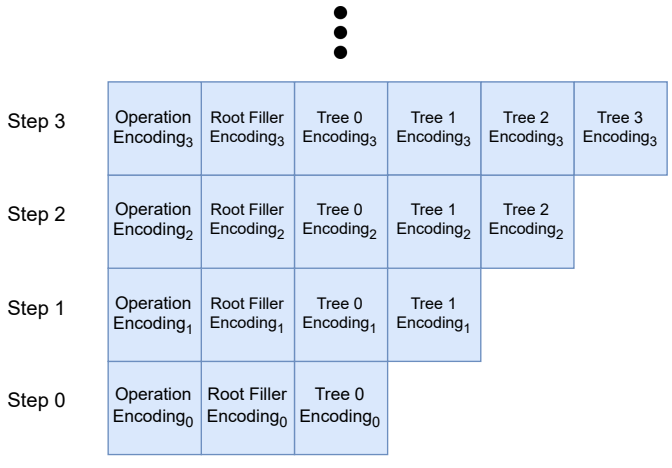

Figure 4: Inputs to the Neural Tree Agent at each step of processing.

C 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.

C.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 Dataset Details

We introduce the **Basic Sentence Transforms** dataset for testing tree-to-tree transformations. It contains various synthetic tree-transform 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 are 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. We focus our evaluation on the following three tasks:

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.

Source Tree:

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

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. The input and output of Figure 1 show a transformation from logical form to passive form.

Source Tree: (S (NP (DET his) (AP (N tree))) (VP (AUXPS was) (VPPS (V touched) (PPPS (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))))))

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 trees into a shared logical form.

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))))))

E Baseline Details

Transformer For our Transformer model, we use the PyTorch library implementation of the Encoder-Decoder Transformer (Vaswani et al., 2017). 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 the Active & Passive→Logical task. The best hyperparameter setting was then used to train models on all four of our tasks. More training details can be found in Appendix A.

LSTM We use LSTM (Hochreiter and Schmidhuber, 1997) seq2seq (Sutskever et al., 2014) implementations that support hyperparameters for attention (on/off), bidirectionality (on/off), and the number of encoder and decoder layers. 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 was then used to train models on all four of our tasks. More training details can be found in Appendix A.

Tree2Tree LSTM Tree2Tree LSTM (Chen et al., 2018) combines a Tree-LSTM encoder (Tai et al., 2015) and a Tree-LSTM decoder (Dong and Lapata, 2016) to create a tree2tree models.

Tree Transformer We select the Tree Transformer model from Shiv and Quirk (2019) since it has a bias to encode and decode trees. Tree information is encoded in relative positional embeddings as the path from one node to another..