Compositional Attention with Intermediate Step Supervision

Jiaqi Shang
jiaqi.shang@uwaterloo.ca
Report due: April 15, 2025

Abstract

Transformers have demonstrated remarkable success in various tasks. However, recent studies highlight the limitations of the Transformer in compositional tasks, as it struggles to generalize beyond memorizing patterns. In this paper, we propose the Compositional Attention Mechanism, an enhancement to the standard attention mechanism of Transformer designed to capture compositional structures and long-range dependencies while remaining computationally practical. [1]

1 Introduction

The Transformer architecture (Vaswani et al. 2017) has been widely used in model training across various domains, including Natural Language Processing (NLP), computer vision, and mathematical reasoning. However, these models often face challenges in handling tasks that require compositional reasoning. Compositional reasoning involves understanding how individual functions or sub-tasks can be combined to achieve more complex objectives (Sinha et al. 2024). Recent studies suggest that transformer-based models tend to reduce the multi-step compositional reasoning into linearized pattern matching, indicating their inability to perform compositional reasoning and hierarchical structure recognition (Dziri et al. 2023).

Addressing this limitation requires an explicit design of models to learn the nature of compositional reasoning. From previous studies, Chain-of-Thought (CoT) prompting (Wei et al. 2023) and Tree Transformer (Wang et al. 2019) have shown potential for hierarchical reasoning. However, some practical challenges are presented: CoT prompting requires training or fine-tuning large language models with extensive computational resources, whereas Tree Transformer requires specialized parsing modules only applied in NLP contexts.

Due to these practical constraints, this project proposes and evaluates a novel compositional attention mechanism explicitly designed for function composition tasks without relying on language models or large-scale textual datasets. Instead, we introduce a simulated numeric dataset inspired by existing compositional reasoning benchmarks but simplified to numeric operations and list operations, thus avoiding the complexity associated with NLP tasks. This dataset includes numeric arithmetic and list manipulation functions, and also intermediate computational steps to enable intermediate supervision.

Our approach incorporates compositional embeddings into the self-attention mechanism, explicitly encoding function compositions with input data. In addition, we introduce intermediate step supervision to explicitly reinforce the model's understanding of each compositional step. We analyze the effect of the levels of intermediate supervision on the model's ability to generalize to novel composition patterns.

[1] https://github.com/jqshang/compositional_reasoning

From our experiments, the results reveal critical insights into the trade-off between intermediate-step supervision and generalization capability. While lower intermediate supervision benefits arithmetic function compositions, we observed that excessive intermediate supervision can reduce generalization performance; especially when more complex list operations such as sorting and reversing are involved.

Overall, this project contributes both theoretically and practically to understanding compositional reasoning within transformer architectures. Even though we are not able to evaluate LLM baselines such as CoT and Tree Transformers, our findings provide valuable guidance on designing transformer-based models in true compositional reasoning without relying on memorization or extensive computational resources required for traditional NLP-based compositional benchmarks.

2 Problem Definition

2.1 Function Composition

Any task can be modeled as a composition of tasks or functions. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ be a set of basic functions such as arithmetic operations and sorting that a model can apply. Given the input data $\mathbf{x} \in \mathbb{R}^d$, each basic function is defined as a transformation

$$f_j : \mathcal{Z}_{j-1} \to \mathcal{Z}_j, \quad \forall j = 1, \ldots, k$$

where $\mathcal{Z}_0 = \mathbf{x}$ and $\mathcal{Z}_m = \mathbf{y}$ for $m \in \{1, \ldots, k\}$ indicating the number of total transformations applied.

A complex task can be formulated as applying a sequence of functions from $\mathcal{F}$ to the input $\mathbf{x}$. This is formally denoted as

$$f_{i_m} \circ f_{i_{m-1}} \circ \cdots \circ f_{i_2} \circ f_{i_1}(\mathbf{x})$$

which is a composition of m functions. In addition, denote such a composition function as $f_{\mathcal{F}_P}(\mathbf{x})$, where $\mathcal{F}_P \subseteq \mathcal{F}$, and $P = (i_1, i_2, \ldots, i_m)$ is the composition pattern for $m \in \{1, \ldots, k\}$ (the indices of functions in the order are applied).

We aim to learn the mapping $\mathscr{F} : \mathbf{x} \mapsto f_{\mathcal{F}_P}(\mathbf{x})$ for any composition pattern P. A true compositional reasoning of a model allows for the execution of any novel composition pattern that was never seen during training.

Further, to explicitly make inferences on compositional reasoning, the model ideally can carry out intermediate steps correctly for each $f_j \in \mathcal{F}_P$. That is

$$\begin{aligned}
\mathbf{x}^{(1)} &= f_{i_1}(\mathbf{x}) \\
\mathbf{x}^{(j)} &= f_{i_j}(\mathbf{x}^{(j-1)}) \qquad \forall j = 2, \ldots, m-1 \\
\hat{\mathbf{y}} &= f_{i_m}(\mathbf{x}^{(m-1)})
\end{aligned}$$

where the model should produce the correct output (i.e., $\hat{\mathbf{y}} = \mathbf{y}$) by implicitly computing each $\mathbf{x}^{(j)}$ in the right order for all $j = 1, \ldots, m-1$.

2.2 Compositional Reasoning vs. Memorization

A memorizing approach might treat each function in the sequence as a separate task, and simply memorize the input-output pairs for the compositions seen during training. Such non-compositional models would fail on a new composition pattern P_{new}.

In contrast, a model with a true ability in compositional reasoning can understand each $f_j \in f_{\mathcal{F}_{P_{\mathsf{new}}}}$ and how to chain them. Rather than memorizing the complete mapping for each possible composition function, we would see such models to learn the effect of each f_j and the order in which the compositions are applied. Essentially, the model is learning

$$\mathbb{F} = \{f_{\mathcal{F}_P} : \mathcal{F}_P \subseteq \mathcal{F}, P \text{ is any composition pattern}\}$$

This requires capturing the compositional structure of any given function composition task, and the ability to reuse the learning building blocks (each f_j component).

3 Methodology

Standard Self-Attention

Standard self-attention computes attention scores based only on input token embeddings (Vaswani et al. 2017). Consider an input vector $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T \in \mathbb{R}^{n \times d_x}$, transformer projects $\mathbf{X}$ to a d-dimensional hidden-state matrix by

$$\mathbf{H} = \mathbf{X}\mathbf{W}_{in} + \mathbf{b}_{in} \in \mathbb{R}^{n \times d}$$

where $\mathbf{W}_{in}$ and $\mathbf{b}_{in}$ are learnable weights and bias respectively. We compute the attention scores by

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

where

- $\mathbf{Q} = \mathbf{H}\mathbf{W}_Q \in \mathbb{R}^{n \times d_k}$
- $\mathbf{K} = \mathbf{H}\mathbf{W}_K \in \mathbb{R}^{n \times d_k}$
- $\mathbf{V} = \mathbf{H}\mathbf{W}_V \in \mathbb{R}^{n \times d_v}$

with learnable weights $\mathbf{W}_Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_K \in \mathbb{R}^{d \times d_k}$, and $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$.

Compositional Self-Attention

In contrast, our proposed Compositional Attention modifies the query, key, and value by explicitly combining compositional embeddings. Consider a composition task $f_{\mathcal{F}_P} \in \mathbb{F}$ with $\mathcal{F}_P \subseteq \mathcal{F} = \{f_1, f_2, \ldots, f_b\}$ on input $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T \in \mathbb{R}^{n \times d_x}$. Let the learnable parameter $e_{f_j} \in \mathbb{R}^{d_f}$ denote the corresponding embedding for the function $f_j \in \mathcal{F}$. Given a specific composition pattern $P = (i_1, i_2, \ldots, i_m)$, the function embedding sequence is denoted as

$$\mathbf{E}_{f_{\mathcal{F}_P}} = [e_{f_{i_1}}, e_{f_{i_2}}, \ldots, e_{f_{i_m}}]^T \in \mathbb{R}^{m \times d_f}$$

We project $\mathbf{E}_{f_{\mathcal{F}_P}}$ to a d-dimensional hidden-state matrix by

$$\mathbf{E}'_{f_{\mathcal{F}_P}} = \mathbf{E}_{f_{\mathcal{F}_P}}\mathbf{U}_f \in \mathbb{R}^{m \times d}$$

where $\mathbf{U}_f \in \mathbb{R}^{d_f \times d}$ is a learnable projection matrix.

Similar to standard self-attention, input $\mathbf{X}$ of Compositional Attention will be projected to $\mathbf{H} = \mathbf{X}\mathbf{W}_{in} + \mathbf{b}_{in} \in \mathbb{R}^{n \times d}$. Then, we incorporate token embeddings $\mathbf{H}$ with compositional embedding $\mathbf{E}'_{f_{\mathcal{F}_P}}$ by concatenation, i.e.,

$$\mathbf{H}_{\textsf{comp}} = \begin{bmatrix}\mathbf{H} & \mathbf{E}'_{f_{\mathcal{F}_P}}\end{bmatrix} \in \mathbb{R}^{(n+m) \times d}$$

We compute the compositional attention scores by

$$\text{Attention}_{\textsf{comp}}(\mathbf{Q}_{\textsf{comp}}, \mathbf{K}_{\textsf{comp}}, \mathbf{V}_{\textsf{comp}}) = \text{softmax}\left(\frac{\mathbf{Q}_{\textsf{comp}}\mathbf{K}_{\textsf{comp}}^T}{\sqrt{d_k}}\right)\mathbf{V}_{\textsf{comp}}$$

where

- $\mathbf{Q}_{\textsf{comp}} = \mathbf{H}_{\textsf{comp}}\mathbf{W}_{Q_{\textsf{comp}}} \in \mathbb{R}^{(n+m) \times d_k}$
- $\mathbf{K}_{\textsf{comp}} = \mathbf{H}_{\textsf{comp}}\mathbf{W}_{K_{\textsf{comp}}} \in \mathbb{R}^{(n+m) \times d_k}$
- $\mathbf{V}_{\textsf{comp}} = \mathbf{H}_{\textsf{comp}}\mathbf{W}_{V_{\textsf{comp}}} \in \mathbb{R}^{(n+m) \times d_v}$

with learnable weights $\mathbf{W}_{Q_{\textsf{comp}}} \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_{K_{\textsf{comp}}} \in \mathbb{R}^{d \times d_k}$, and $\mathbf{W}_{V_{\textsf{comp}}} \in \mathbb{R}^{d \times d_v}$.

This concatenation allows the model to learn compositional patterns explicitly, enabling it to capture hierarchical dependencies related to specific functional compositions.

Intermediate Step Supervision

To explicitly teach the model compositional reasoning, we leverage intermediate step outputs provided in our training data (See Section 4.1 for more details). Particularly, given a function composition pattern $P = (i_1, i_2, \ldots, i_m)$, the dataset contains intermediate results:

$$\mathbf{x} \xrightarrow{f_{i_1}} \mathbf{x}^{(1)} \xrightarrow{f_{i_2}} \mathbf{x}^{(2)} \xrightarrow{f_{i_3}} \ldots \xrightarrow{f_{i_{m-1}}} \mathbf{x}^{(m-1)} \xrightarrow{f_{i_m}} \mathbf{y}$$

During training, we introduce auxiliary intermediate supervision losses. Let $\mathbf{x}^{(j)}$ represent the model's predicted intermediate output at step $j = 1, \ldots, m$. We define the intermediate step loss as:

$$\mathcal{L}_{\text{intermediate}} = \sum_{j=1}^{m-1} \|\hat{\mathbf{x}}^{(j)} - \mathbf{x}^{(j)}\|^2$$

The total loss function thus becomes:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{final}}(\mathbf{y}, \hat{\mathbf{y}}) + \alpha\mathcal{L}_{\text{intermediate}}$$

where α is a hyper weighting parameter for the relative importance of intermediate supervision. This explicit intermediate step supervision improves the model's ability to reason about compositional transformations, as it directly learns how inputs are transformed at each intermediate step.

4 Result

4.1 Datasets

Simulated Dataset for function composition

For this study, we developed a simulated numeric function composition algorithm explicitly designed to generate comprehensive datasets. The generated dataset structure was inspired by the mathematical dataset with questions on composition (Saxton et al. 2019), but a simplified version contains only numeric function compositions.

Each input $\mathbf{x}_i$ is represented by a fixed-length vector containing numeric values uniformly sampled within a predefined range, ensuring sufficient variability. Basic numeric functions are selected randomly with replacement from a defined function pool for diverse composition patterns. The datasets also include accurate intermediate computational steps, supporting 3 Intermediate Step Supervision.

Each data instance in the simulated dataset consists:

- **input**: The uniformly sampled numeric values as initial input.
- **output**: The true computed final output values resulting from applying the defined functions in **composition** in order.
- **composition**: The ordered set of sampled basic numeric functions applied to the input values.
- **intermediate_steps**: The true intermediate results following each compositional step.

4.2 Experiments

In our experimental setup, we selected a set of arithmetic and list manipulation functions as our basic function collection:

- Increment $(x + 1)$
- Decrement $(x - 1)$
- Multiplication by two $(x \times 2)$
- Division by two $(x/2)$
- Sorting in ascending order

• Reversing

Each experiment runs on a dataset with 2000 data instances, where each input vector has a fixed length of 8 numeric elements ranging from 1 to 100. The composition patterns are also randomly sampled so that each composition has 2 to 5 functions.

Note that the testing dataset only contains novel composition patterns not encountered during training, allowing for rigorous assessment of a model's compositional generalization and robustness.

Finally, the model was trained over 50 epochs with varying levels of intermediate supervision by controlling the hyperparameter alpha (α), ranging from 0 (no intermediate supervision) to 1 (full intermediate supervision).

Mean squared error (MSE) is the primary metric used to assess the model's performance. Figure 1 and in Figure 2 visualize the training and testing loss curves across epochs.

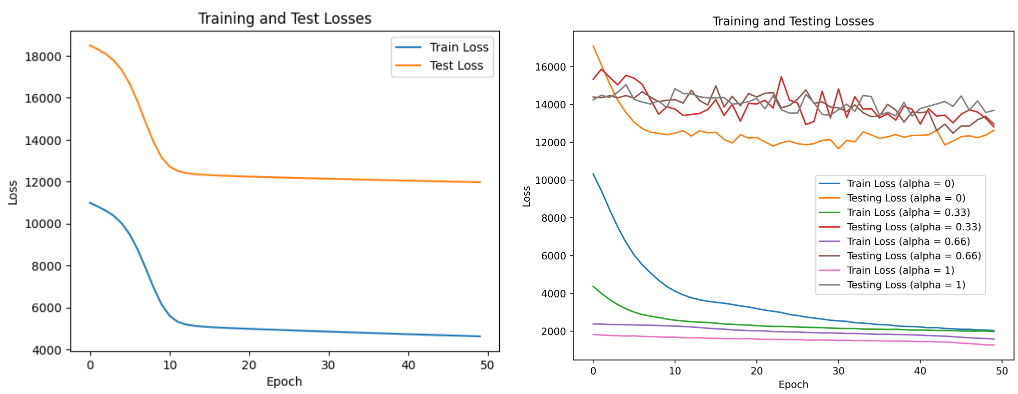

Figure 1: Left: Performance of baseline Self-Attention involving only arithmetic functions (increment, decrement, multiply, and divide). Right: Performance of Composition Attention with intermediate step supervision involving only arithmetic functions (increment, decrement, multiply, and divide). Multiple levels of intermediate supervision by the hyperparameter alpha (α) are applied.

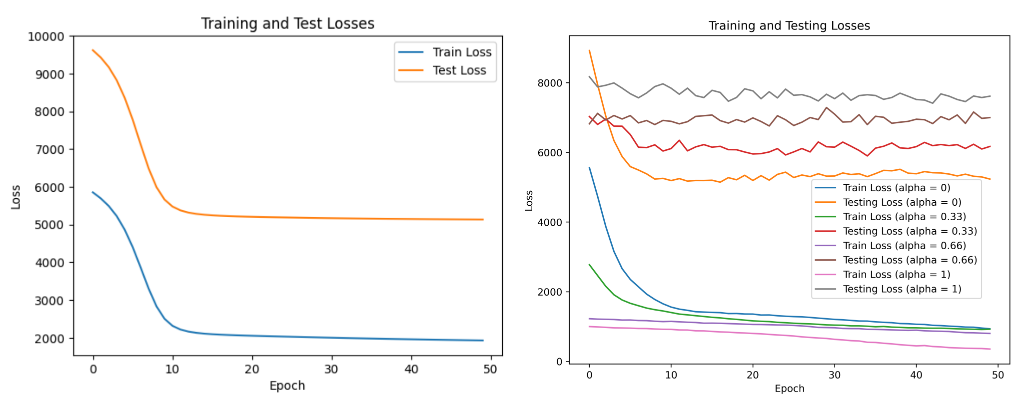

Figure 2: Left: Performance of baseline Self-Attention involving all basic functions (increment, decrement, multiply, divide, sort, and reverse). Right: Performance of Composition Attention with intermediate step supervision involving all basic functions (increment, decrement, multiply, divide, sort, and reverse). Multiple levels of intermediate supervision by the hyperparameter alpha (α) are applied.

In both experimental settings, we observed a consistent trend that increasing intermediate step supervision generally reduces the training loss, but simultaneously causes a higher testing loss. For the dataset containing only arithmetic operations, moderate intermediate supervision ($\alpha = 0.33$) yields the optimal balance.

Interestingly, when more complex list manipulation operations such as sorting and reversing are introduced, removing intermediate supervision entirely becomes favourable. This unexpected outcome suggests that excessive intermediate step supervision in complex tasks hinders the model's performance, causing overfitting and hence reducing the generalization capability.

5 Discussion

In this project, our aim is to study the ability of a machine learning model to perform compositional reasoning effectively. Our results show an improvement over the standard self-attention baseline method, but we acknowledge several limitations and areas for further investigation.

First, although our proposed model performed better than the baseline, we are not confident to state that it outperforms the existing LLMs such as GPT-4 and Llama 3. It remains unclear whether our proposed approach provides meaningful compositional reasoning capabilities or simply outperforms the baseline.

Secondly, the high Mean Squared Error (MSE) observed in our predictions is another major issue. Upon close examination of the predicted values, we find that our model appears to capture some compositional structure, but the numerical predictions remain significantly inaccurate. This indicates that the model fails to precisely represent these compositions.

Furthermore, due to the limited time for this study, our work did not comprehensively explore the originally proposed NLP compositional tasks. In particular, we did not utilize the three datasets identified in the initial project proposal: the Compositional Freebase Questions (CFQ) (Keysers et al. 2020), the mathematical compositional dataset (Saxton et al. 2019 and Ontanon et al. 2021), and the simplified version of CommAI Navigation tasks (SCAN) (Lake and Baroni 2018). A robust compositional model should ideally demonstrate strong performance across these diverse datasets, thus providing evidence of true compositional reasoning capabilities. In this paper, we limit the scope to focus on the simulated compositional datasets.

To robustly validate compositional reasoning, future research should consider a rigorous benchmarking against established LLMs with a variety of compositional datasets. Moreover, explicit qualitative analyses such as attention visualization or intermediate-step verification could clarify how internal model mechanisms support theoretical compositional reasoning.

6 Conclusion

Our study examined compositional reasoning in machine learning models theoretically and empirically, particularly with transformer-based models. We have presented results that demonstrate an improvement over standard self-attention approaches. However, we also explored limitations, including high prediction errors and insufficient testing against advanced LLMs. Future work could aim to address these limitations through extensive benchmarking, thereby contributing to the understanding and practical development of true compositional reasoning in machine learning models.

References

Alkaissi, H. and S. I. McFarlane (2023). "Artificial hallucinations in ChatGPT: implications in scientific writing". *Cureus*, vol. 15, no. 2.

Dziri, N. et al. (2023). "Faith and fate: Limits of transformers on compositionality". *Advances in Neural Information Processing Systems*, vol. 36, pp. 70293–70332.

Guan, X., Y. Liu, H. Lin, Y. Lu, B. He, X. Han, and L. Sun (Mar. 2024). "Mitigating Large Language Model Hallucinations via Autonomous Knowledge Graph-Based Retrofitting". *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, pp. 18126–18134.

Huang, L. et al. (2025). "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions". *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55.

Keysers, D. et al. (2020). "Measuring Compositional Generalization: A Comprehensive Method on Realistic Data". arXiv: `1912.09713 [cs.LG]`.

Lake, B. M. and M. Baroni (2018). "Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks". arXiv: `1711.00350 [cs.CL]`.

Ontanon, S., J. Ainslie, V. Cvicek, and Z. Fisher (2021). "Making transformers solve compositional tasks". *arXiv preprint arXiv:2108.04378*.

Peng, B., S. Narayanan, and C. Papadimitriou (2024). "On Limitations of the Transformer Architecture". arXiv: `2402.08164 [stat.ML]`.

Saxton, D., E. Grefenstette, F. Hill, and P. Kohli (2019). "Analysing Mathematical Reasoning Abilities of Neural Models". arXiv: `1904.01557 [cs.LG]`.

Sinha, S., T. Premsri, and P. Kordjamshidi (2024). "A Survey on Compositional Learning of AI Models: Theoretical and Experimental Practices". arXiv: `2406.08787 [cs.AI]`.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). "Attention is all you need". *Advances in neural information processing systems*, vol. 30.

Wang, Y.-S., H.-Y. Lee, and Y.-N. Chen (2019). "Tree Transformer: Integrating Tree Structures into Self-Attention". *arXiv preprint arXiv:1909.06639*. arXiv: `1909.06639 [cs.CL]`.

Wei, J., X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou (2023). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". arXiv: `2201.11903 [cs.CL]`.

Appendix

Python code for generating simulated datasets:

```python
import numpy as np
import random


class CompositionalDataset:

    def __init__(self,
                 num_samples=1000,
                 min_seq_len=2,
                 max_seq_len=5,
                 input_dim=5,
                 train_ratio=0.8,
                 seed=42):

        np.random.seed(seed)
        random.seed(seed)

        self.num_samples = num_samples
        self.min_seq_len = min_seq_len
        self.max_seq_len = max_seq_len
        self.input_dim = input_dim
        self.train_ratio = train_ratio

        self.basic_functions = {
            "inc": lambda x: x + 1,
            "dec": lambda x: x - 1,
            "mul2": lambda x: x * 2,
            "div2": lambda x: x / 2,
            "sort": lambda x: np.sort(x),
            "rev": lambda x: x[::-1],
        }
        self.function_names = list(self.basic_functions.keys())

    def generate_sample(self, composition):
        """Generate a single sample given a composition of functions."""
        x = np.random.randint(1, 100, self.input_dim).astype(float)
        intermediate_steps = [x.copy()]

        for func_name in composition:
            func = self.basic_functions[func_name]
            x = func(x)
            intermediate_steps.append(x.copy())

        return {
            "input": intermediate_steps[0],
            "output": intermediate_steps[-1],
            "composition": composition,
            "intermediate_steps": intermediate_steps[1:-1]
        }

    def generate_dataset(self, num_samples, seen_compositions=None):
        dataset = []

        for _ in range(num_samples):
            if seen_compositions:
                composition = random.choice(seen_compositions)
```

```
            else:
                seq_len = random.randint(self.min_seq_len, self.max_seq_len)
                composition = [
                    random.choice(self.function_names) for _ in range(seq_len)
                ]
            sample = self.generate_sample(composition)
            dataset.append(sample)

        return dataset

    def prepare_datasets(self):
        num_train_samples = int(self.num_samples * self.train_ratio)
        num_test_samples = self.num_samples - num_train_samples

        seen_compositions = []
        for _ in range(10):
            seq_len = random.randint(self.min_seq_len, self.max_seq_len)
            composition = [
                random.choice(self.function_names) for _ in range(seq_len)
            ]
            seen_compositions.append(composition)

        train_dataset = self.generate_dataset(
            num_train_samples, seen_compositions=seen_compositions)
        test_dataset = self.generate_dataset(num_test_samples)
        return {
            "train_dataset": train_dataset,
            "test_dataset": test_dataset,
            "seen_compositions": seen_compositions,
            "basic_functions": self.function_names
        }
```

Python code for the Compositional Attention Module:

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class CompositionalTransformer(nn.Module):

    def __init__(self, input_dim, hidden_dim, num_layers, num_heads,
                 basic_functions):
        super().__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.basic_functions = basic_functions
        self.func_embeddings = nn.Embedding(len(basic_functions), hidden_dim)

        encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
                                                   nhead=num_heads,
                                                   batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                                         num_layers=num_layers)

        self.input_proj = nn.Linear(input_dim, hidden_dim)
        self.output_proj = nn.Linear(hidden_dim, input_dim)

    def forward(self, x, composition):
        if x.dim() == 1:
```

```
            x = x.unsqueeze(0)

        batch_size = x.size(0)

        h = self.input_proj(x)
        h = h.unsqueeze(1)

        comp_indices = torch.tensor(
            [self.basic_functions.index(f) for f in composition]).to(x.device)
        comp_embed = self.func_embeddings(comp_indices)
        comp_embed = comp_embed.unsqueeze(0).repeat(batch_size, 1, 1)

        h_seq = torch.cat([h, comp_embed], dim=1)
        h_encoded = self.transformer_encoder(h_seq)

        output = self.output_proj(h_encoded[:, -1, :])

        return output


class ISCompositionalTransformer(nn.Module):

    def __init__(self, input_dim, hidden_dim, num_layers, num_heads,
                 basic_functions, max_comp_len):
        super().__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.basic_functions = basic_functions

        self.func_embeddings = nn.Embedding(len(basic_functions), hidden_dim)
        self.input_proj = nn.Linear(input_dim, hidden_dim)
        self.positional_enc = nn.Parameter(
            torch.randn(1, max_comp_len + 1, hidden_dim))

        encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
                                                   nhead=num_heads,
                                                   batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                                         num_layers=num_layers)

        self.intermediate_proj = nn.Linear(hidden_dim, input_dim)
        self.output_proj = nn.Linear(hidden_dim, input_dim)

    def forward(self, x, compositions):
        batch_size = x.size(0)
        device = x.device

        h = self.input_proj(x).unsqueeze(1)

        max_comp_len = max(len(comp) for comp in compositions)
        comp_embed_tensor = torch.zeros(batch_size,
                                        max_comp_len,
                                        self.hidden_dim,
                                        device=device)

        for i, comp in enumerate(compositions):
            indices = torch.tensor(
                [self.basic_functions.index(f) for f in comp], device=device)
            comp_embed_tensor[i, :len(comp), :] = self.func_embeddings(indices)
```

```
        h_seq = torch.cat([h, comp_embed_tensor], dim=1)
        h_seq += self.positional_enc[:, :h_seq.size(1), :]
        h_encoded = self.transformer_encoder(h_seq)

        intermediate_preds = self.intermediate_proj(h_encoded[:, 1:-1, :])

        final_pred = self.output_proj(h_encoded[:, -1, :])

        return intermediate_preds, final_pred
```