AI SCIENTIST TEAM REVIEW

Paper Summary This paper investigates the impact of a temporal consistency regularization term on the compositional generalization of sequence models. The regularizer penalizes large changes in the embedding representation between successive time steps. The experiments consider simple arithmetic tasks and provide evidence that such a regularizer does not improve performance when training the sequence model on multiple tasks. Furthermore, the paper provides small sweeps across different settings including embedding dimension, regularization strength and architectures.

Strengths

- Although the reasoning behind the design of the proposed regularization is not immediately clear, a simple approach–such as encouraging successive token embeddings to be closer together–presents an interesting avenue for exploring compositional representations.
- The chosen arithmetic task is simple but suitable for testing the hypothesis for varying degrees of difficulty. The chosen experiments provide insights into the impact on various aspects and limitations of the regularization impact.

Weaknesses

- The description of the regularization term is vague and can be misleading. Intuitively, the reader can think that it is applied to the LSTM hidden state. Inspecting the code reveals that the regularizer refers to the input embedding hidden state. The text could be enhanced by being more explicit about this detail, adding a code appendix or providing ablations that apply the regularizer to the LSTM hidden state.
- The paper lacks several references and for example does not cite Hochreiter and Schmidhuber (1997) but instead opts for the textbook by Goodfellow et al (2016).
- The caption of Figure 3 is wrong. The validation loss increases as task complexity increases. Furthermore, the self-attention based version discussed in Figure 5 performs significantly better than the LSTM version, while the text argues that they perform on par.
- The experimental evaluation could benefit from more depth. The considered sequence lengths are very short and the considered task is only synthetic. Some of the claims could require more rigorous evidence, including real world tasks, larger networks and in-depth mechanistic analysis.

Scores

- Soundness: 3/5 good. ⇒ Interesting idea with targeted experiments.
- Presentation: 2/5 fair ⇒ Citations, imprecise description, too confident interpretation.
- Contribution: 3/5 good ⇒ Regularizer, analysis, ablations
- Overall - Workshop: 5/10 (Borderline accept): Technically solid paper where reasons to accept outweigh reasons to reject, e.g., limited evaluation.
- Overall - Conference: 4/10: (Borderline reject): Technically solid paper where reasons to reject, e.g., limited evaluation, outweigh reasons to accept, e.g., good evaluation.
- Confidence: 4/5. You are confident in your assessment, but not absolutely certain. It is unlikely, but not impossible, that you did not understand some parts of the submission or that you are unfamiliar with some pieces of related work.

Additional Comments

- To strengthen the analysis, several different compositional regularizers should be compared across different tasks. Additionally, it needs to be more explicitly tested whether the regularizer actually induces compositional representations. This could be done for example via linear probes trained on the embedding representations or by visualizing low-dimensional embeddings.

Potential Violation of Code of Ethics: No.

AI SCIENTIST TEAM CODE REVIEW

E.1 INSPECTING THE DATASET GENERATION PROCESS

The data-generating function, which uses a single-digit expression as shown in Figure 10, generates at most $81 * k$ possible combinations, where k is the number of operators. This suggests that the training and test datasets can have significant overlap, depending on the number of samples and the choice of operators.

As a sanity check, we generated the dataset 10 times using addition and multiplication operators, with [0-9] as the available numbers, and 1,000 training samples and 200 test samples. On average, we found that about 57% of the test set overlapped with the training set.

```python
# Generate synthetic data with varying operator complexity
def generate_expression_data(n_samples, operator_set):
    numbers = list(range(1, 10))
    expressions = []
    results = []

    for _ in range(n_samples):
        num1, num2 = np.random.choice(numbers, 2)
        op = np.random.choice(operator_set)
        expr = f"{num1}{op}{num2}"

        # Handle division by zero
        if op == "/" and num2 == 0:
            num2 = np.random.choice([n for n in numbers if n != 0])
            expr = f"{num1}{op}{num2}"

        result = eval(expr)
        expressions.append(expr)
        results.append(result)

    return expressions, results


# Create vocabulary including all possible operators
vocab = list("0123456789+-*/")
char2idx = {c: i for i, c in enumerate(vocab)}
idx2char = {i: c for c, i in char2idx.items()}
vocab_size = len(vocab)
```

Figure 10: Example of the data generating function used in the experiments.

E.2 MODEL ARCHITECTURE, LOSS FUNCTION, AND EVALUATION FUNCTION

While the model architecture is simple, its implementation appears to be correct, as shown in Figure 11.

In the training loop, presented in Figure 12, compositional regularization is computed using the embedding states. Therefore, the main paper should use the notation e_t to represent embeddings instead of h_t, and explicitly refer to these as embeddings rather than hidden states. Although embeddings are technically a hidden layer, the term 'hidden states' in this context usually refers to LSTM hidden states, which could be confusing.

The accuracy calculation function (Figure 13) indicates that the model performs regression on the output to match the ground truth digits. This approach makes sense, as it allows the model to handle arbitrary values, including those outside the range [0-9].

E.3 ATTENTION-AUGMENTED LSTM

In Appendix B, a 100% test accuracy was reported for the attention-augmented LSTM. To verify this, we re-ran the same experiment using the generated code for two cases: the first with the available numbers [1-9] (as in the original setup), and the second with the available numbers modified to [10-19]. In the first case, the attention-augmented LSTM achieved 100% test accuracy, while in the second case, it achieved 56% test accuracy. For the baseline LSTM, the first case resulted in 85%

```python
# Model
class CompositionalModel(nn.Module):
    def __init__(self, vocab_size, hidden_size=64):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        hidden = lstm_out[:, -1, :]
        return self.fc(hidden)

    def get_compositional_loss(self, hidden_states):
        return torch.mean((hidden_states[:, 1:] - hidden_states[:, :-1]).pow(2))
```

Figure 11: The generated model class shows an embedding layer, a single LSTM layer, and a linear layer head.

```python
for epoch in range(n_epochs):
    model.train()
    train_loss = 0
    comp_loss = 0

    for batch_idx, (expr, result) in enumerate(train_loader):
        expr, result = expr.to(device), result.to(device)

        optimizer.zero_grad()
        output = model(expr)

        # Calculate main loss
        loss = criterion(output.squeeze(), result)

        # Add compositional regularization
        hidden_states = model.embedding(expr)
        comp_reg = model.get_compositional_loss(hidden_states)
        total_loss = loss + compositional_weight * comp_reg

        total_loss.backward()
        optimizer.step()

        train_loss += loss.item()
        comp_loss += comp_reg.item()
```

Figure 12: The generated training loop shows the loss function as well as the proposed regularization.

```python
with torch.no_grad():
    for expr, result in test_loader:
        expr, result = expr.to(device), result.to(device)
        output = model(expr)
        test_loss += criterion(output.squeeze(), result).item()

        # Calculate accuracy within a tolerance
        correct += torch.sum(torch.abs(output.squeeze() - result) < 0.5).item()
        total += result.size(0)
```

Figure 13: The generated accuracy calculation function uses regression to match an output with a ground truth.

test accuracy, and the second case yielded 0% test accuracy. We concluded that the first case was too simple for the attention-augmented LSTM, and as the task complexity increased (e.g., the first case involved a length of 3, such as 3 + 5, while the second case involved a length of 5, such as 14 * 19, with a larger output space), the test accuracy deviated from the initial 100%.