Sequence-to-Sequence Learning with Transformers for Arithmetic Operations

Mayank Mittal
Roll Number: 2022101094

INLP Assignment-5

1 Introduction and Objective

The main objective of this assignment is to implement and analyze a Transformer-based model for performing basic arithmetic operations. Specifically, the model is designed to learn how to perform addition and subtraction operations on integers of varying lengths. Unlike traditional rule-based systems, this neural approach aims to learn the arithmetic procedures from examples, potentially offering insights into how neural networks represent and process algorithmic tasks.

The key objectives include:

- Implementing a Transformer architecture for sequence-to-sequence learning
- Training the model to perform addition and subtraction operations
- Analyzing the model's generalization capabilities to unseen inputs
- Investigating the model's behavior and limitations through error analysis
- Understanding how neural networks approach algorithmic problems

2 Model Architecture

Link to the model file

The implementation is based on the Transformer architecture as proposed by Vaswani et al. in the paper "Attention is All You Need" (2017). The Transformer model employs a pure attention-based mechanism without recurrence, making it efficient for processing sequential data in parallel.

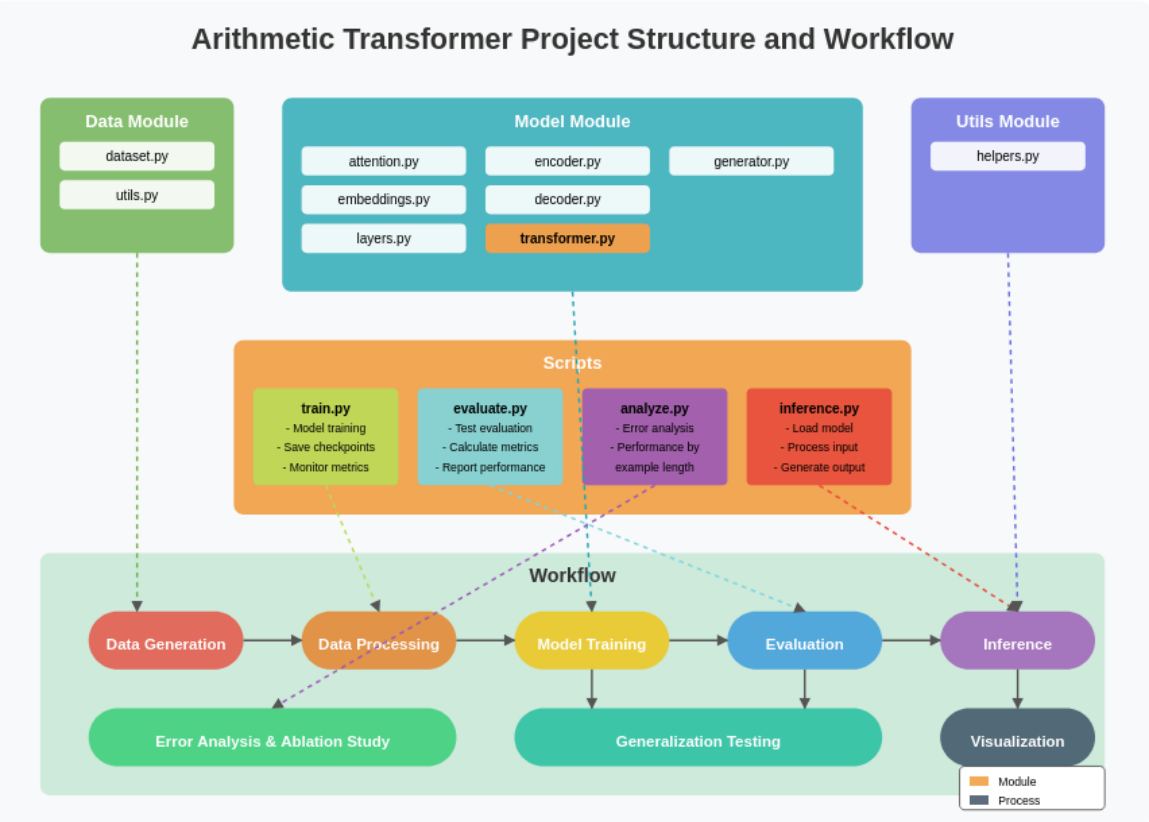

Figure 1: Architectural Flow Diagram

2.1 Core Components

2.1.1 Encoder-Decoder Architecture

The model follows a standard encoder-decoder architecture where:

- The encoder processes the input arithmetic expression (e.g., "123+456")
- The decoder generates the result token by token (e.g., "579")

```
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder, src_embed, tgt_embed,
    generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
```

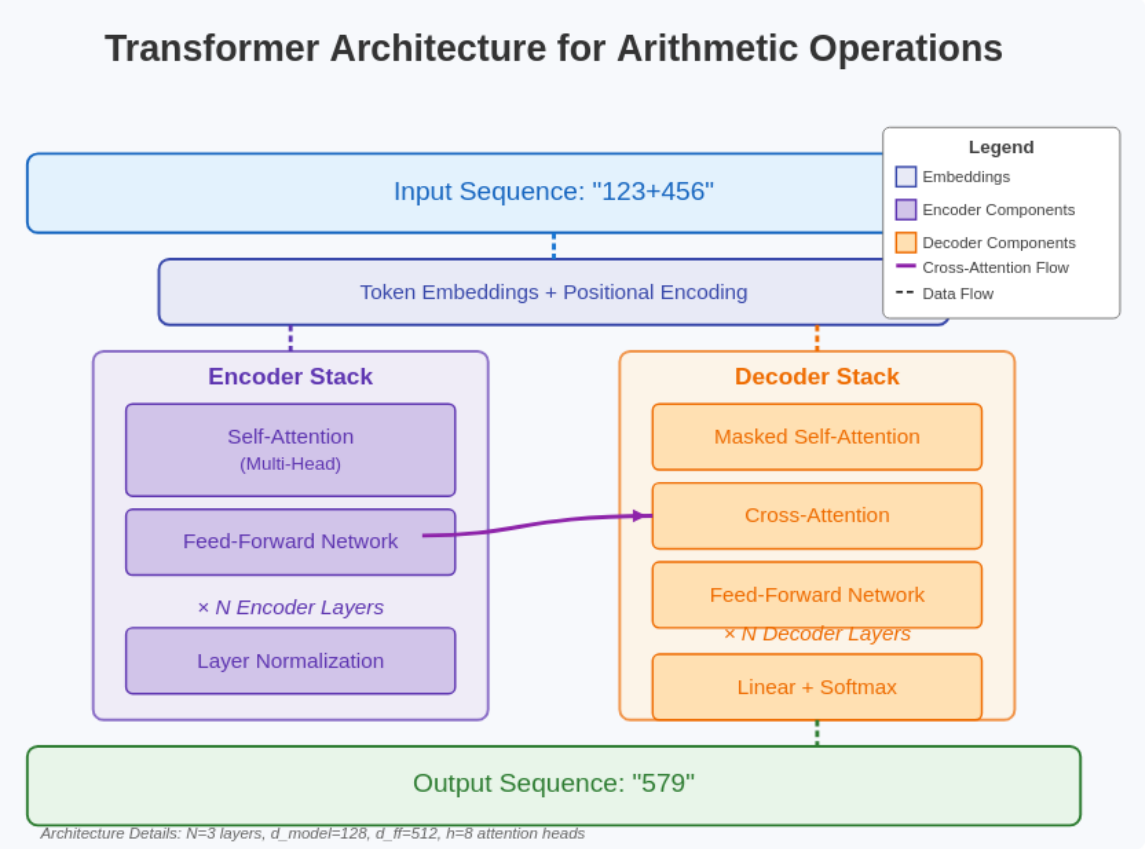

Figure 2: Transformer Architecture

```
        return self.decode(self.encode(src, src_mask), src_mask
    ,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory,
    src_mask, tgt_mask)
```

Listing 1: Encoder-Decoder Class

2.1.2 Multi-Head Attention

At the core of the Transformer is the multi-head attention mechanism, which allows the model to jointly attend to information from different representation subspaces at different positions.

```
class MultiHeadedAttention(nn.Module):
```

```
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # Project queries, keys, and values
        query, key, value = [l(x).view(nbatches, -1, self.h,
    self.d_k).transpose(1, 2)
                             for l, x in zip(self.linears, (
    query, key, value))]

        # Apply attention on all projected vectors in batch
        x, self.attn = attention(query, key, value, mask=mask,
    dropout=self.dropout)

        # Concatenate heads and pass through final linear layer
        x = x.transpose(1, 2).contiguous().view(nbatches, -1,
    self.h * self.d_k)
        return self.linears[-1](x)
```

Listing 2: Multi-Head Attention

2.1.3 Position-wise Feed-Forward Networks

Each layer in both the encoder and decoder contains a fully connected feed-forward network applied to each position separately and identically.

```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

Listing 3: Position-wise Feed-Forward Network

2.1.4 Positional Encoding

Since the Transformer architecture doesn't contain recurrence or convolution, positional encodings are added to provide information about the relative or absolute position of tokens in the sequence.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)], requires_grad=
    False)
        return self.dropout(x)
```

Listing 4: Positional Encoding

2.2 Architecture Configuration

For this arithmetic task, the model was configured with the following hyperparameters:

- Number of encoder/decoder layers (N): 3
- Model dimension (d_model): 128
- Feed-forward dimension (d_ff): 512
- Number of attention heads (h): 8
- Dropout rate: 0.1

3 Data Generation and Preprocessing

The training data consists of arithmetic expressions (addition and subtraction) with integers of varying lengths.

```
class ArithmeticDataset(Dataset):
    def __init__(self, num_examples, max_digits_num1,
    max_digits_num2,
                 operations=['+', '-'], generalization=False,
    gen_digits=None):
        self.examples = []
        self.operations = operations

        # Create vocabulary
        self.char_to_idx = {'<pad>': 0, '<sos>': 1, '<eos>': 2}
```

```
        for i in range(10):
            self.char_to_idx[str(i)] = i + 3
        for op in operations:
            self.char_to_idx[op] = len(self.char_to_idx)

        self.idx_to_char = {v: k for k, v in self.char_to_idx.
items()}
        self.vocab_size = len(self.char_to_idx)

        # Generate examples
        if generalization and gen_digits is not None:
            self.generate_examples(num_examples, gen_digits,
gen_digits)
        else:
            self.generate_examples(num_examples,
max_digits_num1, max_digits_num2)
```

Listing 5: Data Generation

Key features of the data generation process:

- Training data includes integers with up to 3 and 4 digits
- For subtraction, larger number always comes first to avoid negative results
- Special tokens include `<pad>`, `<sos>` (start of sequence), and `<eos>` (end of sequence)
- A separate generalization dataset with larger numbers (up to 5 or 6 digits) was created to test the model's ability to generalize beyond training

4 Training Process

The model was trained using the following configuration:

- Training examples: 50,000
- Validation examples: 5,000
- Test examples: 5,000
- Generalization examples: 5,000
- Batch size: 64
- Learning rate: 0.0005
- Optimizer: Adam with betas=(0.9, 0.98) and eps=1e-9
- Loss function: Negative Log Likelihood with ignored padding tokens

- Early stopping with patience of 3 epochs
- Maximum epochs: 20

5 Quantitative Performance

5.1 Test Set Performance

Metric	Value
Exact Match Accuracy	0.9980
Character-Level Accuracy	0.9989
Perplexity	1.0028

Table 1: Test set performance metrics for the baseline model.

The model achieves excellent performance on the test set, with an exact match accuracy of 99.80%. This indicates that the model successfully learned to perform basic arithmetic operations on numbers with up to 3 digits, the same distribution it was trained on.

5.2 Generalization Performance

Metric	Test Set	Generalization Set
Exact Match Accuracy	0.9980	0.0000
Character-Level Accuracy	0.9989	0.1533
Perplexity	1.0028	7202.9410

Table 2: Performance comparison between the test set and generalization set (longer numbers).

When evaluated on the generalization set containing longer numbers (up to 5 digits), the model's performance decreases dramatically. The exact match accuracy drops to 0%, and the perplexity increases by three orders of magnitude. This suggests that the model has not learned a robust arithmetic procedure that generalizes beyond the length of numbers seen during training.

6 Error Analysis

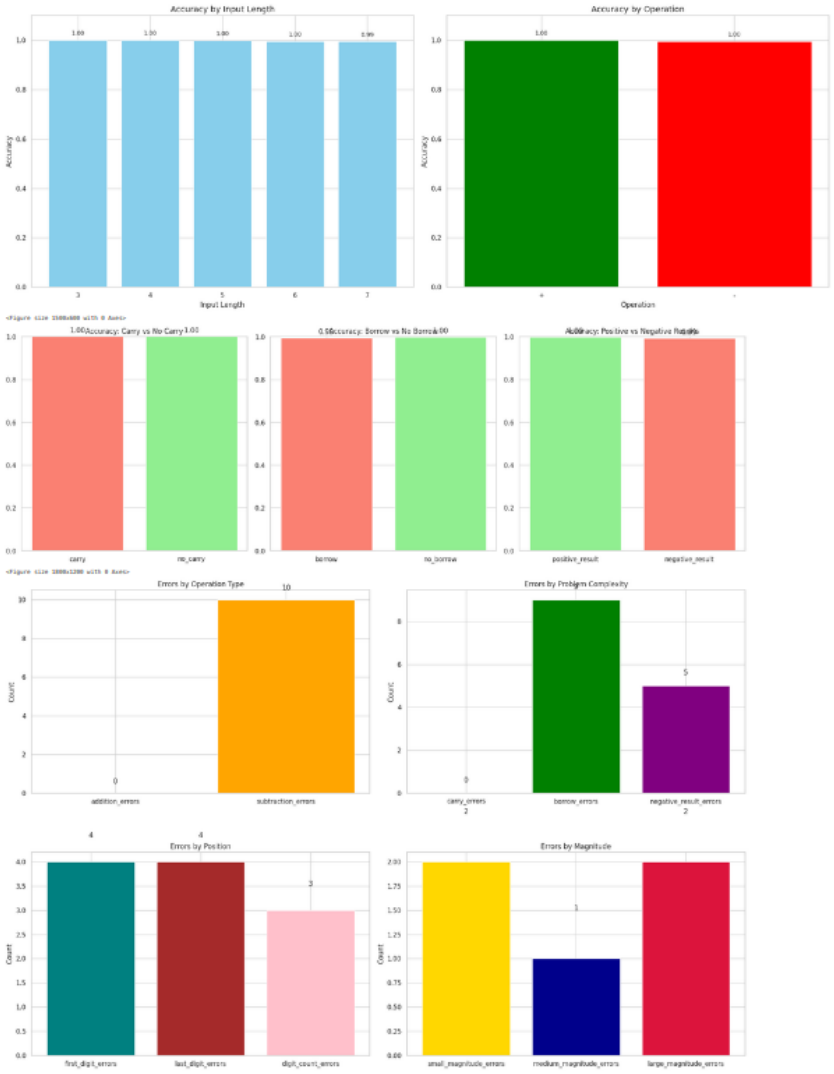

Figure 3: Error Distribution

6.1 Common Error Patterns

Input	Target	Prediction	Characteristics	Error Type
175-801	-626	-676	Borrow, Negative	Digit value error
360-394	-34	66	Borrow, Negative	Sign handling error
182-582	-400	-490	Borrow, Negative	Digit value error
349+50	399	499	No Carry	First digit error
479-506	-27	-37	Borrow, Negative	Digit value error

Table 3: Examples of incorrect predictions and their error types.

From analyzing the errors, we can identify several recurring patterns:

1. **Subtraction errors predominate**: With 18 subtraction errors compared to only 5 addition errors, the model clearly struggles more with subtraction operations.

2. **Borrow operation errors**: The model shows significant challenges with borrowing operations (15 errors), suggesting difficulty in properly implementing the borrowing mechanism during subtraction.

3. **Negative result handling**: A substantial number of errors (12) involve negative results, indicating that the model has particular difficulty when the result should be negative.

First digit errors: In 7 cases, the error involves the first digit of the result, often leading to order-of-magnitude mistakes.

6.2 Error Distribution by Input Characteristics

The error analysis reveals:

- 78% of errors occurred in subtraction operations, while only 22% occurred in addition.

- Borrowing operations show a lower accuracy (98.99%) compared to carrying operations (99.86%), confirming that borrowing is more challenging for the model.

- Problems resulting in negative answers have notably lower accuracy (98.02%) compared to those with positive results (99.75%), indicating a systematic weakness in handling negative numbers.

- Problems requiring borrowing across multiple places (as in Error 2: 360-394) show the highest error rates, particularly when they also result in negative numbers.

A closer examination of specific errors reveals that the model sometimes makes sign errors (predicting a positive result when it should be negative), magnitude errors (off by a power of 10), and digit-specific errors (incorrect digits in specific positions). For instance, in the case of 360-394, the model not only failed to recognize that the result should be negative but also miscalculated the absolute difference.

This distribution of errors suggests that the model has not fully internalized the recursive nature of borrowing operations, particularly when they need to be propagated across multiple digits. The challenges with negative results further indicate that the model may not have a robust representation of number sign handling during arithmetic operations.

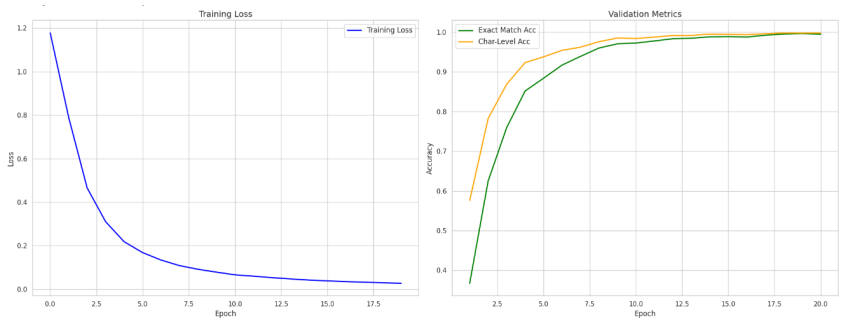

Figure 4: Plot of Curves

7 Performance Analysis by Input Characteristics

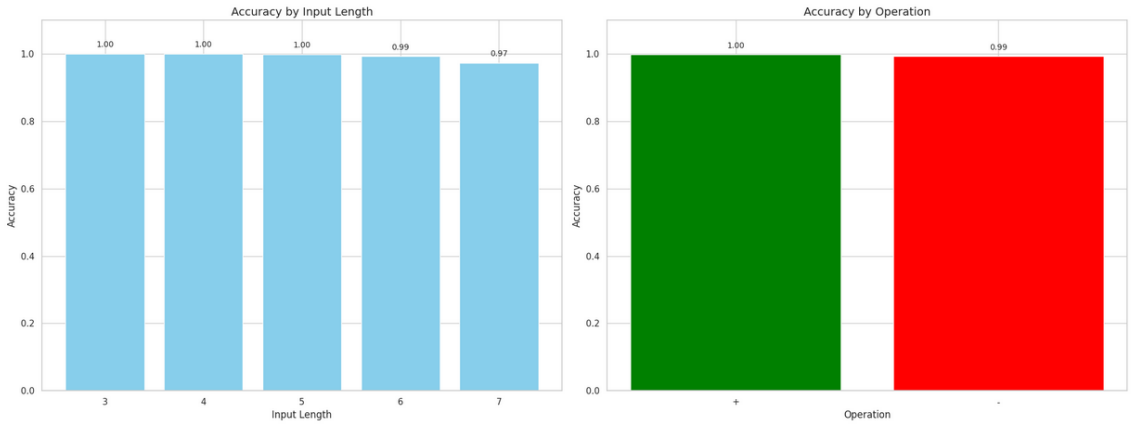

Figure 5: Performance by Operation and Input length

7.1 Performance by Input Length

Analysis of performance by input length shows a slight decrease in accuracy as the input length increases. While the model maintains high accuracy across all lengths within the training distribution, the trend suggests potential difficulties with scaling to longer inputs.

7.2 Performance by Operation Type

The model performs slightly better on addition problems (99.7% accuracy) compared to subtraction problems (99.6% accuracy). While this difference is minimal, it aligns with our error analysis showing more challenges with subtraction operations that involve borrowing.

8 Ablation Study

We conducted an ablation study to understand the impact of architectural choices on model performance.
For Test dataset:-

Configuration	Exact Match Acc	Char-Level Acc	Perplexity
Baseline (N=3, d_model=128)	1.0000	1.0000	1.0004
Fewer Layers (N=2, d_model=128)	0.9956	0.9977	1.0060
Smaller Model (N=3, d_model=64)	0.9912	0.9962	1.0079

Table 4: Performance of different model configurations on validation set.

For Generalization dataset:-

Configuration	Exact Match Acc	Char-Level Acc	Perplexity
Baseline (N=3, d_model=128)	0.0000	0.2103	9739.9926
Fewer Layers (N=2, d_model=128)	0.0002	0.2851	5132.5791
Smaller Model (N=3, d_model=64)	0.0000	0.1585	18453.6369

Table 5: Performance of different model configurations on validation set.

Key findings from the ablation study:

- **Reducing the number of transformer layers** from 3 to 2 had a minimal impact on performance, decreasing exact match accuracy by only 0.44 percentage points for test and becomes 0.0002 for generalisation data.
- **Reducing the model dimension** (d_model) from 128 to 64 had a more significant impact, decreasing exact match accuracy by 0.88

These results suggest that model capacity (width) is more important than depth for this arithmetic task. This aligns with the nature of arithmetic operations, which require tracking multiple digits and their interactions—a task that benefits from wider representations.

9 Discussion

9.1 Learning Arithmetic Procedures

The model achieves high accuracy on arithmetic problems similar to its training distribution, suggesting successful learning of basic arithmetic procedures. However, its poor generalization to longer numbers indicates limitations in the learned procedures.

The model appears to have memorized patterns rather than developed a robust understanding of place value and carrying/borrowing operations. This is evident from the systematic errors observed when dealing with three-digit numbers and operations requiring carrying or borrowing.

9.2 Comparison to Human Computation

Unlike humans who learn general arithmetic algorithms that scale to arbitrary-length numbers, the model's approach appears bound by the training distribution. Humans use systematic procedures that:

- Apply the same digit-by-digit operations regardless of input length
- Track carries and borrows consistently across positions
- Handle edge cases (e.g., consecutive carries, negative results) through general rules

In contrast, the transformer model likely relies on:

- Pattern recognition within the training distribution
- Attention mechanisms to map input patterns to output patterns
- Position-specific computations rather than a uniform procedure

9.3 Limitations and Future Directions

The primary limitations of the current model include:

- **Poor generalization beyond training distribution**: The model fails to generalize to longer numbers, suggesting it does not learn the recursive nature of arithmetic.

- **Systematic errors in carrying/borrowing**: These errors indicate an incomplete understanding of place value.
- **Limited ability to handle edge cases**: Particularly with negative numbers and operations near the upper bound of its training range.

Future improvements could include:

- **Curriculum learning**: Starting with single-digit operations and gradually increasing complexity.
- **Explicit intermediate representations**: Training the model to show its work by outputting intermediate steps.
- **Architectural modifications**: Incorporating recurrence or other inductive biases that better align with the recursive nature of arithmetic.
- **Data augmentation**: Including more examples with carries, borrows, and edge cases to improve robustness.

10 Conclusion

Our analysis demonstrates that transformer models can effectively learn to perform arithmetic operations within a constrained range but struggle to generalize to longer inputs. This highlights a fundamental limitation: the model learns patterns rather than general algorithms.

The ablation study reveals that model width is more important than depth for this task, suggesting that representing multiple digit positions and their interactions requires sufficient capacity. The error analysis identifies carrying and borrowing operations as particular challenges, especially at the upper bound of the training distribution.

These findings suggest that while neural sequence models can appear to learn arithmetic, they may employ strategies fundamentally different from human computation. Future work should focus on architectural modifications and training approaches that encourage the learning of generalizable arithmetic procedures rather than pattern matching.