UM6P College of Computing

Deep Learning, or Long Learning?

A Comparative Study

Group 6

Fjer Imane | **Hakkou Ilyas** | **Maftah Mahmoud**

Agenda

Introduction

Long ListOps Task Overview

Existing Models and Benchmarks

Proposed Approach

Experimental Results

Challenges and Lessons Learned

Conclusion

Introduction

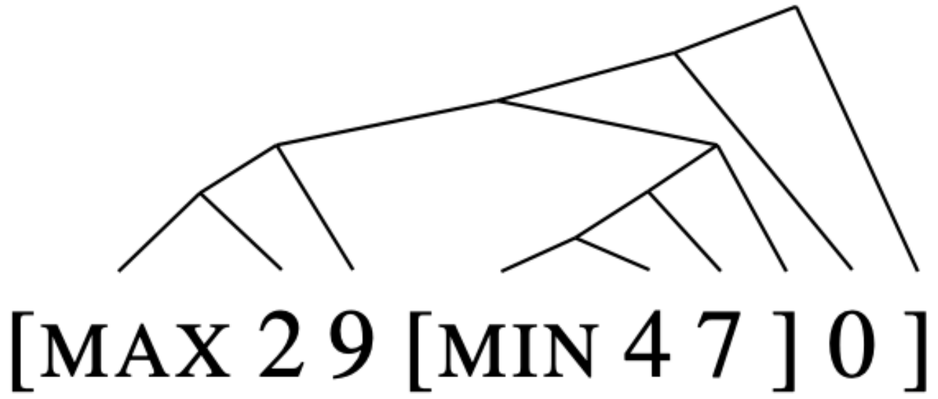
Background:

- Transformers have limitations in handling long sequences due to their quadratic self-attention complexity.
- Long Range Arena (LRA) is a benchmark designed to test models on tasks requiring long-range dependency handling, such as Long ListOps.

Objective:

Explore both transformer variants and RNN-based models to improve accuracy on the Long ListOps task.

Long ListOps
Task Overview



- Long ListOps involves parsing long sequences of nested mathematical operations to compute results.
- This task tests hierarchical understanding and the ability to capture long-range dependencies.

Challenges:

Standard transformers and RNNs often fail to capture dependencies across long sequences.

Existing Models and Benchmarks

Benchmark results show that even advanced transformers like Reformer, Linformer, and Longformer **struggle** with accuracy on Long ListOps, highlighting challenges in modeling hierarchical dependencies.

Model	ListOps
Chance	10.00
Transformer	36.37
Local Attention	15.82
Sparse Trans.	17.07
Longformer	35.63
Linformer	35.70
Reformer	**37.27**
Sinkhorn Trans.	33.67
Synthesizer	<u>36.99</u>
BigBird	36.05
Linear Trans.	16.13
Performer	18.01
Task Avg (Std)	29 (9.7)

Performance Challenges:

- Low accuracy due to the inability to model hierarchical dependencies effectively.
- High computational cost for very long sequences.

Proposed Approach

Model Selection:

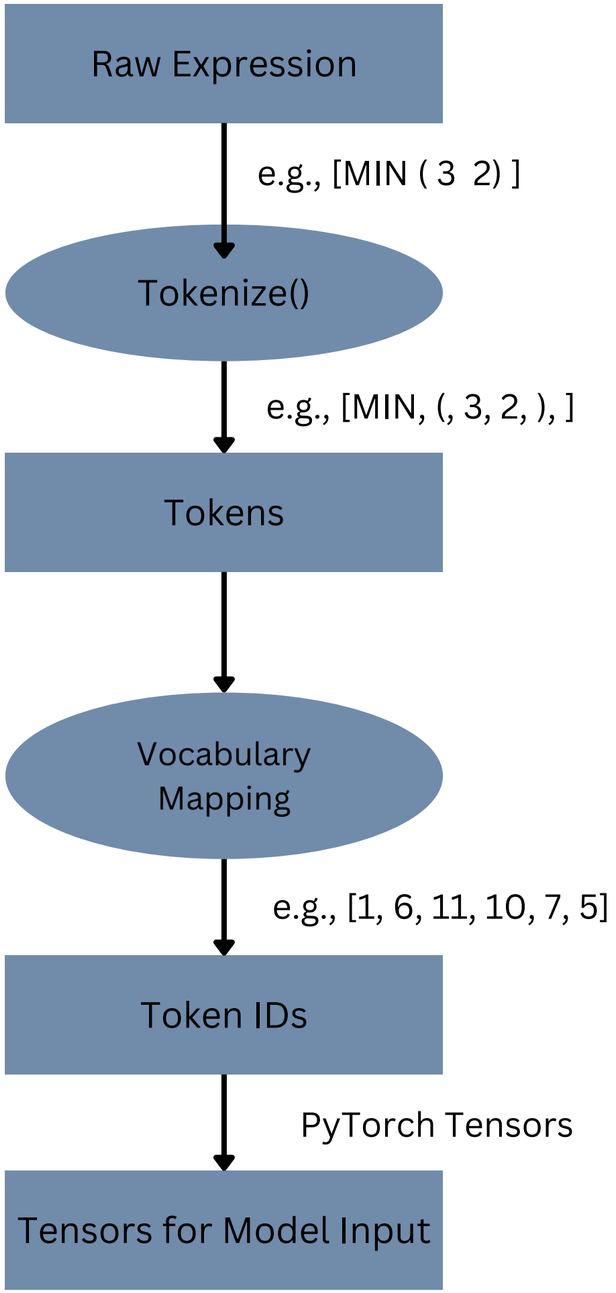
Implemented and tested the following:

- Transformer variants: Longformer, DistilBERT...
- RNN-based models: RNN, LSTM, and LSTM with attention.

Dataset Handling:

- For transformers, utilized existing dataset classes using the **“datasets”** Hugging Face library.
- For RNN, LSTM, and LSTM with attention, created a custom dataset class to tokenize sequences and preprocess data for training.

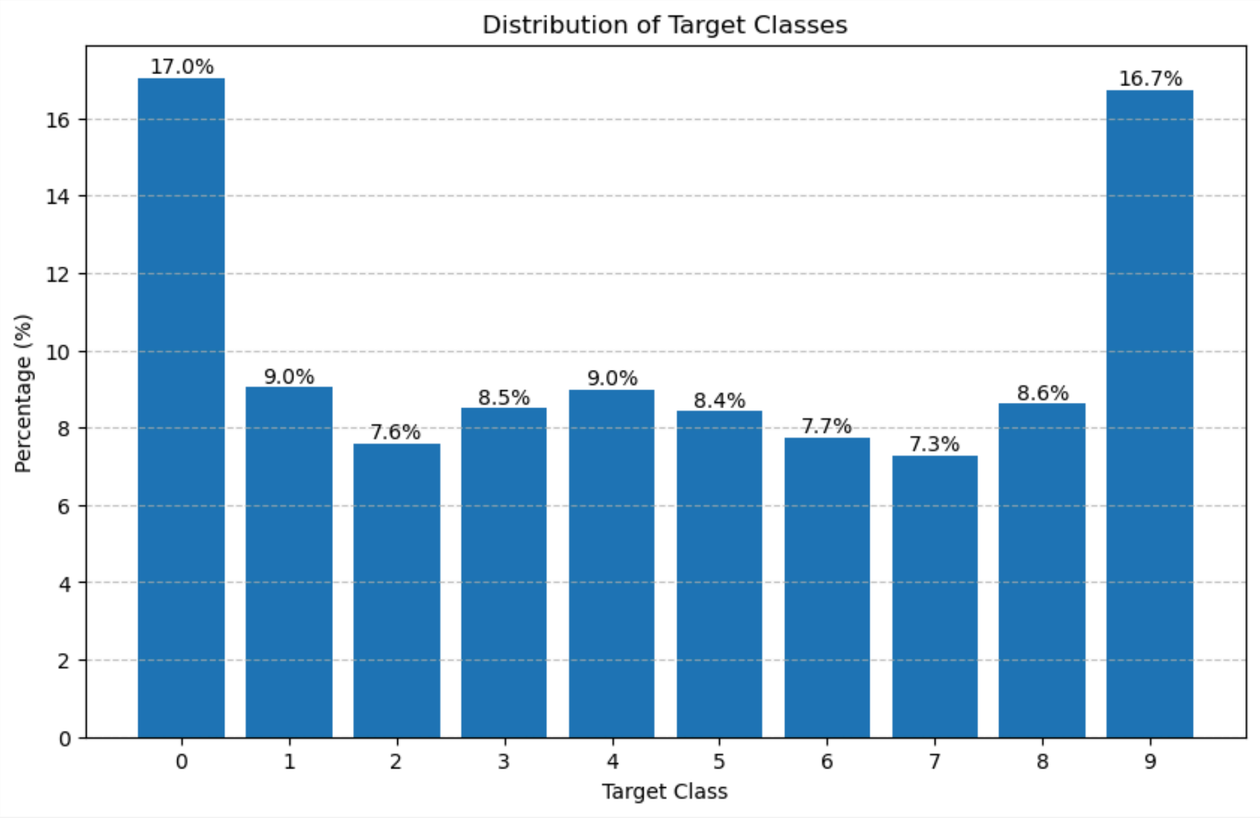
```python
class ListOpsDataset(Dataset):
    def __init__(self, X, y):
        """
        Args:
            X: Array of source expressions
            y: Array of target values
        """
        self.X = X
        self.y = y

        self.vocab = {
            'PAD': 0,
            '[MIN': 1,
            '[MAX': 2,
            '[MED': 3,
            '[SM': 4,
            ']': 5,
            '(': 6,
            ')': 7
        }
        # Add digits 0-9
        for i in range(10):
            self.vocab[str(i)] = i + 8

    def __len__(self):
        return len(self.X)

    def tokenize(self, expr):
        """Convert expression to token IDs."""
        tokens = tokenize(expr)
        return [self.vocab.get(token, 0) for token in tokens]

    def __getitem__(self, idx):
        expr = self.X[idx]
        target = self.y[idx]

        token_ids = self.tokenize(expr)

        return {
            'input_ids': torch.tensor(token_ids,
dtype=torch.long),
            'target': torch.tensor(target, dtype=torch.long)
        }
```



Dataset Preparation Pipeline

- Handles tokenization and vocabulary mapping.
- Prepares data in PyTorch tensor format.

Addressing Class Imbalance with Focal Loss



Impact:

- Imbalanced datasets can cause the model to favor dominant classes.
- This results in poor performance on minority classes, reducing overall accuracy and fairness.

Solution:

Implemented **focal loss** to address the imbalance.

Bidirectional RNN

```python
class SimpleRNNModel(nn.Module):
    def __init__(self,
                 vocab_size=18,
                 embedding_dim=128,
                 hidden_dim=256,
                 num_layers=2,
                 dropout=0.3):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size,
embedding_dim)
        self.rnn = nn.RNN(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0,
            bidirectional=True
        )
        self.fc = nn.Linear(hidden_dim * 2, 10)
        self.dropout = nn.Dropout(dropout)
```

Performance Metrics

- **Accuracy**: 45.20%
- **Training Time**: Low (fastest among tested models)
- **Strength**:
  - Effectively captures context from both directions in the sequence.
  - Computationally efficient compared to more complex models.
- **Weakness:**
  - Limited ability to model long-range dependencies due to the vanishing gradient problem.
  - Struggles with hierarchical tasks like Long ListOps.

Bidirectional LSTM

```
class LSTMModel(nn.Module):
    def __init__(self,
                 vocab_size=18,
                 embedding_dim=128,
                 hidden_dim=256,
                 num_layers=2,
                 dropout=0.3):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size,
                                      embedding_dim)
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else
0,            bidirectional=True
        )
        self.fc = nn.Linear(hidden_dim * 2, 10)
        self.dropout = nn.Dropout(dropout)
```

Performance Metrics

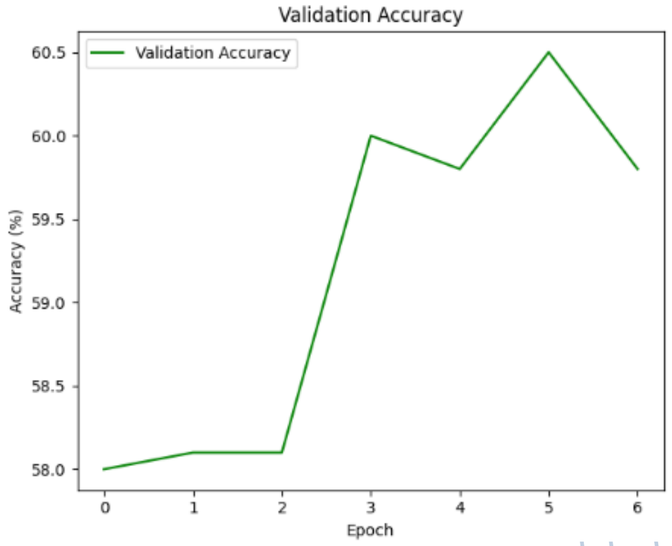
- **Accuracy**: 59.50%
- **Training Time**: Medium
- **Strength**:
  - Effectively retains long-range dependencies.
  - Captures bidirectional context for enhanced sequence understanding.
- **Weakness:**
  - Requires more computational resources compared to RNNs.

LSTM with Attention

```python
class AttentionLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim=128, hidden_dim=256, num_layers=2, dropout=0.3,
num_classes=10):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0,
            bidirectional=True
        )

        # Attention layer
        self.attention = nn.Linear(hidden_dim * 2, 1)

        # Final classification layers
        self.fc = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, num_classes)
        )
```

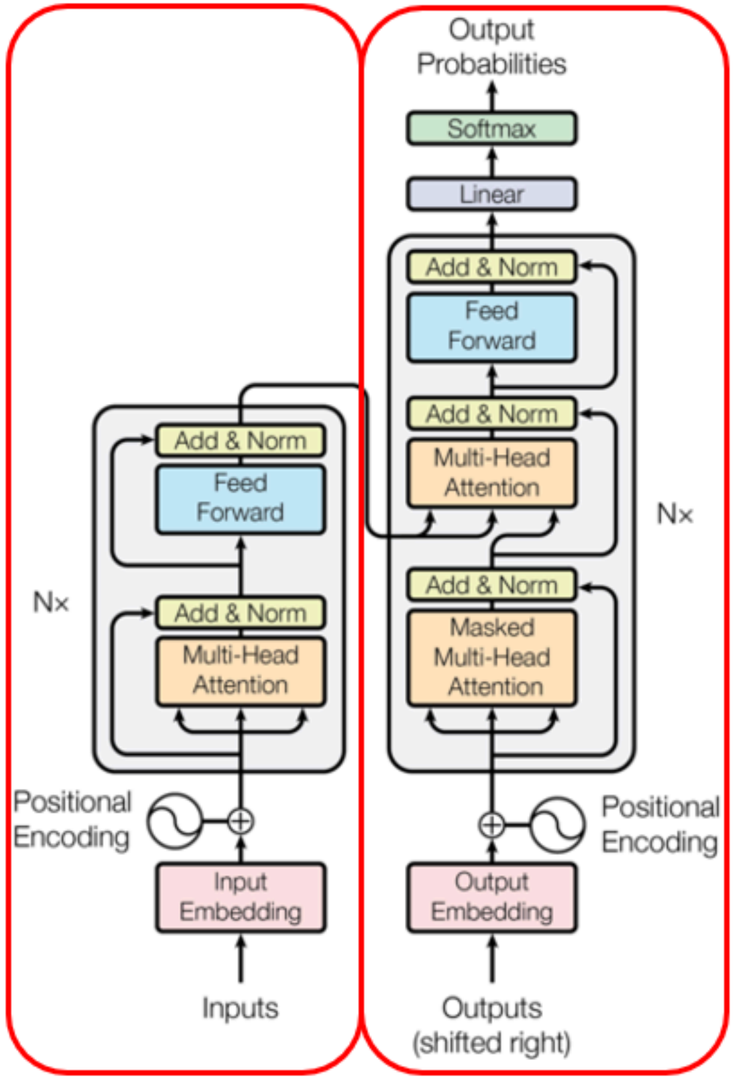
Performance Metrics

- **Accuracy**: 62.25%
- **Training Time**: Medium-High
- **Strength**:
  - Focuses on critical sequence components for better results.
- **Weakness:**
  - Computationally heavier than plain LSTMs.



Transformers

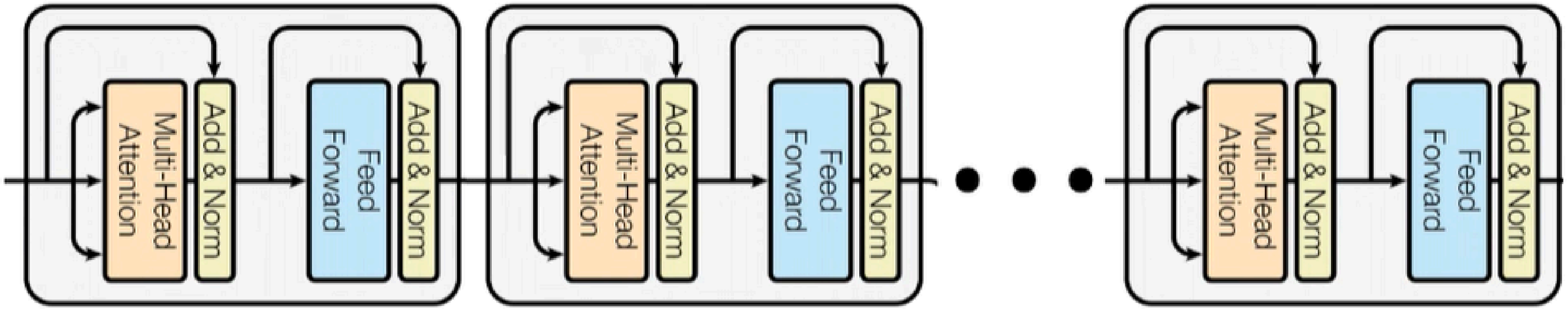
Encoder



Decoder

BERT

Bidirectional **E**ncoder **R**epresentation from **T**ransformers



Stacked Encoders

BERT

Bidirectional **E**ncoder **R**epresentation from **T**ransformers

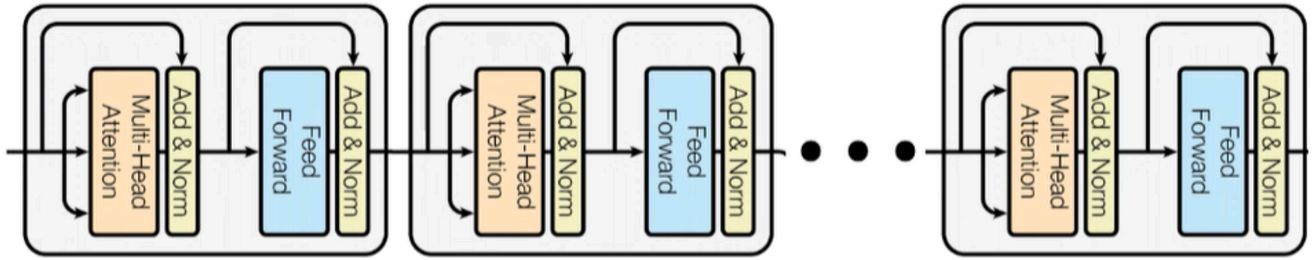
Capabilities

- Neural Machine Translation
- Question Answering
- Token Classification (NER)
- Sequence Classification (Sentiment Analysis...)

How?

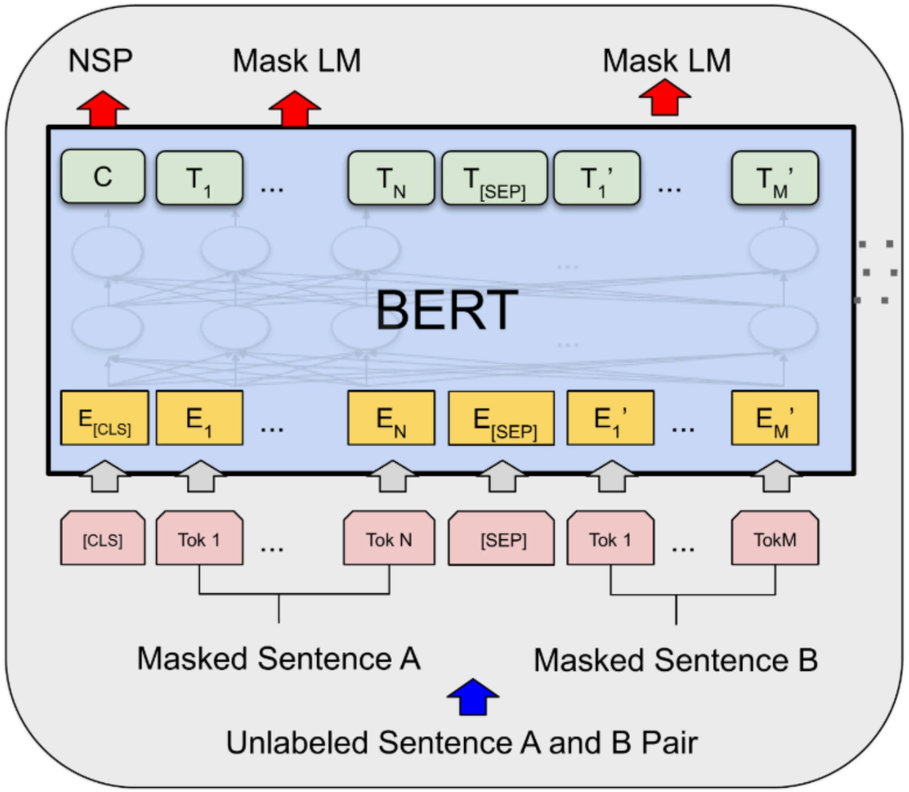
1 - Pretraining => Understand language

2 - Fine-tuning => Learn specific task



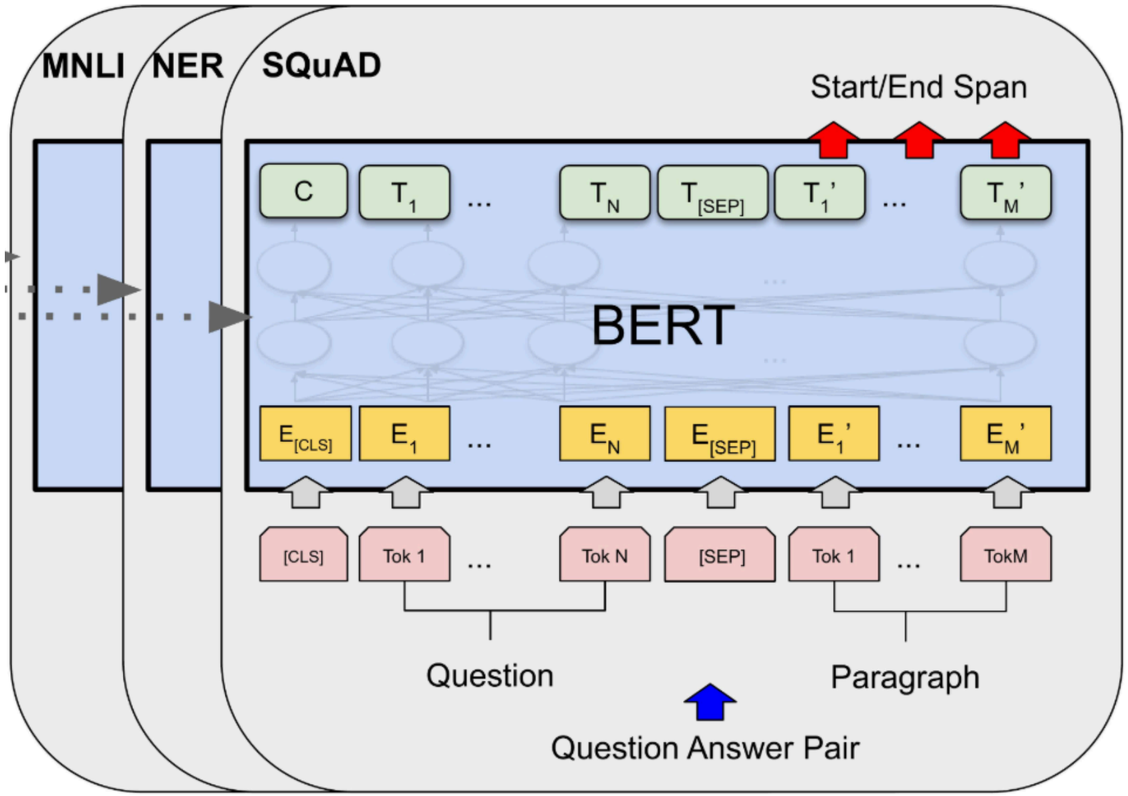
BERT

Bidirectional **E**ncoder **R**epresentation from **T**ransformers



Pre-training

(Self-supervised)



Fine-Tuning

(Supervised)

DistilBERT

Distilled version of BERT

- Smaller, faster, cheaper, and lighter
- Retains 97% of BERT’s performance 🚀
- 40% smaller
- 60% faster inference
- Knowledge distillation (Teacher - Student)

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
    num_labels=10,
)
```

BERT, DistilBERT

Key points to consider:

- Max sequence length
  - BERT: **512**
  - DistilBERT: **384**
  - **Solution?** Truncation

Accuracy: **43.75%**

- Overkill for ListOps ! Unused big vocabulary

Experimental Results

Model	Accuracy (%)	Training Time
RNN	**45.20**	Low
LSTM	**59.50**	Medium
LSTM with attention	**62.25**	Medium-High
AllenAI LongFormer	**17.25**	High
DistilBERT	**43.75**	High
Transformer Classifier Model	**38.65**	High
Transformer (Baseline)	**37.27**	High

Evaluation Metrics:

- **Accuracy**: Correctness of mathematical operations.
- **Training Time**: Efficiency of models.

Insights

LSTM with attention significantly **outperformed** baseline transformers in accuracy.

Challenges and Lessons Learned

Challenges

- Dependency hell (official LRA repository)
- Training transformers on long sequences was computationally expensive.
- Gradient issues in RNNs required careful tuning

Lessons Learned:

- Attention mechanisms are crucial for hierarchical dependency modeling.
- Simpler models like LSTM with attention can sometimes outperform transformers for specific tasks.

Conclusion

Summary

- The project demonstrated that transformer limitations can be mitigated by using variants or alternative architectures like LSTM with attention.
- LSTM with attention showed competitive performance while being computationally efficient.

Takeaways

- The importance of architecture design tailored to specific tasks.

References

- https://github.com/google-research/long-range-arena
- CodeEmporium - “BERT Neural Network - EXPLAINED!”
- https://arxiv.org/pdf/2011.04006
- https://discuss.pytorch.org/t/focal-loss-for-imbalanced-multi-class-classification-in-pytorch/61289
- https://medium.com/@preeti.rana.ai/distilbert-multiclass-text-classification-using-transformers-hugging-face-7c072656525d

Thank you for your attention

Check Out the Project on GitHub

UM6P College of Computing