Chebychev Positional Encoding in Transformers and Long Term Dependencies

June 16, 2024

Word Count: 3016 Github repo: https://github.com/JeffHastingsUCSD/Chebychev-Replication

1 Section 1: Abstract

Long-term dependencies are essential in understanding relationships within text, images, and videos, where contextual references span over long distances. Transformers often struggle with these dependencies, which are vital for interpreting pronouns in text or actions in video frames. Traditional absolute positional encoding, utilizing fixed sine and cosine functions, often fail to adequatley model longer sequences due to periodic repetition and loss of positional information. Chebyshev polynomials, known for their efficient function approximation and oscillatory nature, provide a potentially compact and precise positional representation, capturing long-range patterns effectively.

This project explores the use of Chebyshev Positional Encoding into transformer models to enhance long-term dependency modeling. The Long Range Arena (LRA) benchmark, particularly the ListOps task, evaluates this integration. ListOps involves parsing nested list expressions, necessitating the handling of hierarchical data and extended dependencies. Traditional encodings have underperformed on such tasks, highlighting the potential of Chebyshev encoding.

A synthetic ListOps dataset was generated to simulate tree-like equations with various operations. The ChebyshevPositionalEncoding was incorporated into a transformer model that includes a pretrained BERT and a classifier for binary classification. Training and evaluation on the ListOps dataset showed that the Chebyshev Positional Encoder outperforms other transformer models. However, despite positive results, further validation across different tasks and datasets is required and the results produced by this experiment should be viewed with caution. Future research should consider the influence of the pretrained BERT model, specific dataset characteristics, and the computational efficiency of the Chebyshev approach.

2 Section 2: Introduction

Long-term dependencies consist of relationships between components in a sequence over a long distance. Transformers have traditionally struggled to represent these relationships effectively (Zimerman and Wolf, 2023). Yet, long-term dependencies are crucial in various data types, including text, images, and video. For example, in text, understanding a pronoun often requires reference to a noun mentioned several sentences earlier. In a novel, understanding a character's actions may depend on some event or choice taking place well before the character's present dilemma. Similarly, in video analysis, an action might be best understood in the context of events that occurred several frames earlier. The ability to make connections between current and past events is a central component of intelligence. Therefore, to effectively process increasingly complex data, transformers will need to incorporate mechanisms for successfully modeling long-term dependencies.

In a transformer-based architecture, the modelling of dependencies typically begins with positional encoding where each token in a sequence is assigned a specific number corresponding to its position. For instance, in this sentence, the "For" could be encoded with a 0 or a 1 to indicate that it occupies the first position in the sentence. These encodings are typically appended to the numerical vector representing the token (Vaswani et al, 2017). Such a process is necessary for attention units to function properly. Without positional encodings, the attention mechanism would be unaware of the token positions, losing the sense of sequence order, which is crucial for understanding the context.

2.0.1 Absolute Positional Encoding

In their paper 'Attention is All You Need', the authors utlize a positional encoding mechanism known as absolute encoding where each token's position remains fixed and unaltered during training (Vaswani et al, 2017). Specifically, these absolute encodings are implemented in a straightforward way using sinusoidal functions which assign each position in the sequence a unique representation, thus allowing the model to learn and attend to different positions effectively.

Sine Component (for even dimensions):

$$[PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)]$$

Cosine Component (for odd dimensions):

$$[PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)]$$

Where:

- $$(pos)$$
  : The position of the token in the sequence (e.g., 0 for the first token, 1 for the second token, etc.).
- $$(i)$$
  : The dimension index (e.g., 0, 1, 2, ...).
- $$(d_{model})$$
  : The dimensionality of the model's embeddings.

2.0.2 Absolute Encoding Issues

While the method of relying on differing sine and cosine wavelengths works well for shorter sequences, it becomes less effective for very long sequences (Ruoss et al, 2023; Delétang et al., 2023).

The fixed wavelengths mean that the encoding patterns repeat periodically, which can lead to ambiguity and loss of positional information for longer sequences. This can make it challenging for the model to distinguish between different positions in a long sequence.

Absolute positional encoding uses a fixed set of sine and cosine functions. This rigidity limits the model's ability to adapt to different types of sequences or capture complex patterns in the data. The encoding scheme may not be able to capture positional relationships or adapt to varying sequence lengths, leading to suboptimal performance in tasks requiring long-term dependency capture.

The sine and cosine functions provide a smooth, continuous encoding of positions. While this can be beneficial for capturing local dependencies, it might not be powerful enough to capture more complex positional relationships or long-range dependencies effectively. The model might struggle to learn and represent more complex positional patterns. This could lead to a potential loss of information and accuracy, especially in tasks involving hierarchical or structured data.

As sequence length increases, the computation of sine and cosine values for each position and dimension can become computationally expensive. This can lead to inefficiencies and slower training times, particularly for models dealing with very long sequences or large-scale datasets.

The positional encodings generated by sine and cosine functions might not be able to capture long-term dependencies effectively due to their periodic nature and limited range of representable patterns. For tasks requiring the model to understand relationships between distant elements in a sequence, absolute positional encoding may fall short, leading to poorer performance in tasks like document summarization, long-form text generation, or sequence-based predictions.

2.0.3 Mathematical Advantages of Chebychev Polynomials

Chebychev polynomials offer several desirable properties that make them prime candidates for addressing the long-term dependencies deficiencies of transformers. First, Chebyshev polynomials are known for their efficiency in approximating functions. They can provide a more compact representation of positional information compared to other polynomial bases due to their minimization of the maximum error in approximations.

Second, Chebyshev polynomials oscillate between -1 and 1 over the interval [-1, 1], which helps in capturing periodic patterns in the data. This oscillatory nature can help in capturing repeating patterns over long sequences more effectively than absolute positional encodings.

Third, Chebyshev polynomials are orthogonal with respect to the weight function (1-$x^{2)}$(-1/2) on the interval [-1, 1]. This orthogonality can lead to better numerical stability and precision, especially when dealing with long sequences where positional values can become very large.

Fourth, Absolute positional encodings (like those used in the original Transformer) can become less effective as the sequence length increases because they are not designed to capture repeating patterns. Chebyshev polynomials, on the other hand, can represent functions over a wide range with high accuracy, making them suitable for capturing dependencies over long sequences.

Fifth, Chebyshev polynomials provide a compact representation of positional information, which can help in reducing the complexity of the model. This compactness can be particularly useful when dealing with very long sequences, where the sheer amount of positional data can become overwhelming and inefficient to process

2.0.4 LRA Benchmark

The Long Range Arena is a series of benchmarks established for the purpose of evaluating transformer model peformance on a variety of long-range intensive tasks (it can be accessed here: https://github.com/google-research/long-range-arena). As Zimerman and Wolf (2023) note, "The LRA benchmark has emerged as a sought-after dataset tailored for evaluating these models across a variety of long-context scenarios, tasks, and data types. By offering a common ground for comparison, LRA scrutinizes model capabilities with sequences ranging from 1K to 16K tokens, encompassing text, visual data, and mathematical expressions". The table below shows transformer performance on these tasks using accuracy as its major metric.

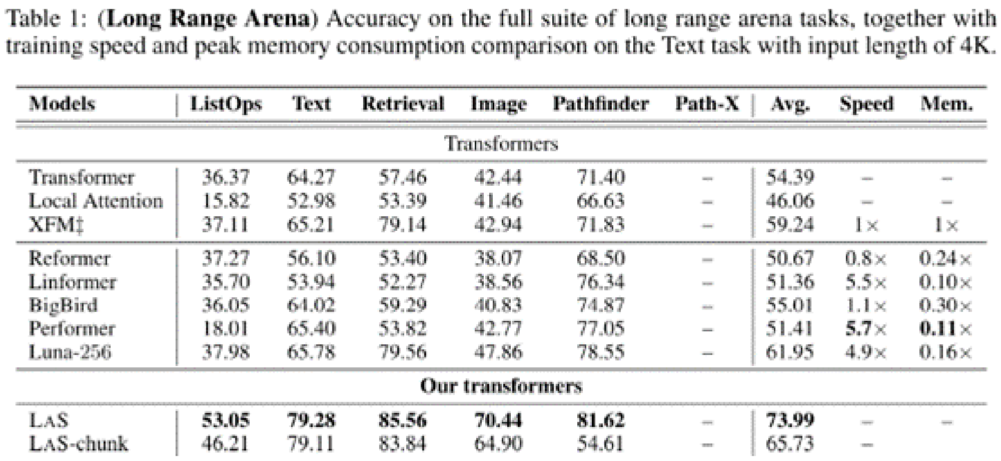

Table 1: (**Long Range Arena**) Accuracy on the full suite of long range arena tasks, together with training speed and peak memory consumption comparison on the Text task with input length of 4K.

Models	ListOps	Text	Retrieval	Image	Pathfinder	Path-X	Avg.	Speed	Mem.
				Transformers					
Transformer	36.37	64.27	57.46	42.44	71.40	–	54.39	–	–
Local Attention	15.82	52.98	53.39	41.46	66.63	–	46.06	–	–
XFM‡	37.11	65.21	79.14	42.94	71.83	–	59.24	1×	1×
Reformer	37.27	56.10	53.40	38.07	68.50	–	50.67	0.8×	0.24×
Linformer	35.70	53.94	52.27	38.56	76.34	–	51.36	5.5×	0.10×
BigBird	36.05	64.02	59.29	40.83	74.87	–	55.01	1.1×	0.30×
Performer	18.01	65.40	53.82	42.77	77.05	–	51.41	**5.7**×	**0.11**×
Luna-256	37.98	65.78	79.56	47.86	78.55	–	61.95	4.9×	0.16×
				Our transformers					
LAS	**53.05**	**79.28**	**85.56**	**70.44**	**81.62**	–	**73.99**	–	–
LAS-chunk	46.21	79.11	83.84	64.90	54.61	–	65.73	–	

Source: Zimerman and Wolf, 2023

3 Section 3: Methods

3.0.1 Listops Dataset

ListOps is a benchmark task within the LRA designed to evaluate the capability of models to handle hierarchical data and capture long-range dependencies. It involves parsing and evaluating expressions represented as nested lists, with operations such as minimum (MIN), maximum (MAX), median (MED), and sum modulo (SUM_MOD). The task asks models to process sequences of varying lengths and complexities, requiring them to maintain and manipulate hierarchical structures accurately. Performance on ListOps is often used to assess the effectiveness of different models and positional encoding methods in capturing and processing long-term dependencies within sequences. Traditional absolute and relative positional encodings have not fared well on such tasks (Zimerman and Wolf, 2023). Consequently, ListOps is one suitable test for evaluating the performance of Chebyshev positional encoding.

3.0.2 ListOps Dataset Generation

In this part of the project, I generate a synthetic dataset for the ListOps task, based on the LRA repo, by constructing tree-like equations using a recursive function, where each node can be an operator (e.g., MIN, MAX, MED, SUM_MOD) or a value from a predefined set. The tree's depth and the number of arguments for each operator are controlled by specified parameters such as the max-depth (length of each individual sequence) and max-legnth (the length of each total sequence).

The dataset is divided into training, validation, and test sets based on the desired number of samples for each set. The generated equations and their corresponding values are then saved to TSV files for use in training machine learning models.

3.1 Examples of ListOps Sequences

Here are some examples of ListOps sequences generated by the provided code:

1. **Sequence:** `( [SUM_MOD 2 5 3 9 ] )`
   - **Value:** `9`
2. **Sequence:** `( [MAX ( [MED 8 4 5 7 ] ) 3 1 0 ] )`
   - **Value:** `8`
3. **Sequence:** `( [MIN ( [SUM_MOD 1 3 4 ] ) 2 0 ( [MAX 6 8 7 ] ) ] )`
   - **Value:** `0`
4. **Sequence:** `( [MED 3 8 1 4 9 2 5 ] )`
   - **Value:** `4`
5. **Sequence:** `( [MAX 1 ( [MIN 6 4 2 ] ) ( [SUM_MOD 9 7 3 ] ) ] )`
   - **Value:** `9`

These sequences demonstrate various operators such as `SUM_MOD`, `MAX`, `MED`, and `MIN` applied to a set of random values, illustrating the tree-like structure and the operations performed to generate the corresponding values.

```
[ ]: # This code is derived from LRA repo ListOps generator code:
     # https://github.com/google-research/long-range-arena/blob/main/lra_benchmarks/
      ↪data/listops.py

     import os
     import csv
     import random
     import numpy as np
     import tensorflow.compat.v1 as tf

     # Define constants and flags
     MIN = '[MIN'
     MAX = '[MAX'
     MED = '[MED'
     FIRST = '[FIRST'
     LAST = '[LAST'
     SUM_MOD = '[SM'
     END = ']'

     OPERATORS = [MIN, MAX, MED, SUM_MOD]
     VALUES = range(10)
     VALUE_P = 0.25

     FLAGS = tf.app.flags.FLAGS

     tf.app.flags.DEFINE_string('task', 'basic', 'Name of task to create.')
tf.app.flags.DEFINE_integer('num_train_samples', 56000, 'Number of train␣
 ↪samples.')
tf.app.flags.DEFINE_integer('num_valid_samples', 2000, 'Number of validation␣
 ↪samples.')
tf.app.flags.DEFINE_integer('num_test_samples', 2000, 'Number of test samples.')
tf.app.flags.DEFINE_integer('max_depth', 7, 'Maximum tree depth of training␣
 ↪sequences.')
tf.app.flags.DEFINE_integer('max_args', 7, 'Maximum number of arguments per␣
 ↪operator in training sequences.')
tf.app.flags.DEFINE_integer('max_length', 1000, 'Maximum length per sequence in␣
 ↪training sequences.')
tf.app.flags.DEFINE_integer('min_length', 250, 'Minimum length per sequence in␣
 ↪training sequences.')
tf.app.flags.DEFINE_string('output_dir', 'output_dir', 'Directory to output␣
 ↪files.')

def generate_tree(depth, max_depth, max_args):
    """Generate tree-like equations."""
    if depth < max_depth:
        r = random.random()
    else:
        r = 1

    if r > VALUE_P:
        value = random.choice(VALUES)
        return value, 1
    else:
        length = 2
        num_values = random.randint(2, max_args)
        values = []
        for _ in range(num_values):
            sub_t, sub_l = generate_tree(depth + 1, max_depth, max_args)
            values.append(sub_t)
            length += sub_l

        op = random.choice(OPERATORS)
        t = (op, values[0])
        for value in values[1:]:
            t = (t, value)
        t = (t, END)
    return t, length

def to_string(t, parens=True):
    if isinstance(t, str):
        return t
    elif isinstance(t, int):
        return str(t)
    else:
        if parens:
            return '( ' + to_string(t[0]) + ' ' + to_string(t[1]) + ' )'

def to_value(t):
    """Compute the output of equation t."""
    if not isinstance(t, tuple):
        return t
    l = to_value(t[0])
    r = to_value(t[1])
    if l in OPERATORS:  # Create an unsaturated function.
        return (l, [r])
    elif r == END:  # l must be an unsaturated function.
        if l[0] == MIN:
            return min(l[1])
        elif l[0] == MAX:
            return max(l[1])
        elif l[0] == FIRST:
            return l[1][0]
        elif l[0] == LAST:
            return l[1][-1]
        elif l[0] == MED:
            return int(np.median(l[1]))
        elif l[0] == SUM_MOD:
            return np.sum(l[1]) % 10
    elif isinstance(l, tuple):
        # We've hit an unsaturated function and an argument.
        return (l[0], l[1] + [r])

def write_to_file(data, fp):
    """Write to file output."""
    print(f'Writing {len(data)} samples to {fp}.tsv')
    os.makedirs(os.path.dirname(fp), exist_ok=True)  # Ensure directory exists
    with open(fp + '.tsv', 'w+', newline='') as f:
        writer = csv.writer(f, delimiter='\t')
        writer.writerow(['Source', 'Target'])
        writer.writerows(data)

def main(_):
    print('Start dataset construction')

    data = set()
    num_samples = FLAGS.num_train_samples + FLAGS.num_test_samples + FLAGS.
 ↪num_valid_samples
    while len(data) < num_samples:
        tree, length = generate_tree(1, FLAGS.max_depth, FLAGS.max_args)
        if length > FLAGS.min_length and length < FLAGS.max_length:
            data.add(tree)
            if len(data) % 1000 == 0:
                print(f'Processed {len(data)}')

    train = []
    for example in data:
        train.append([to_string(example), to_value(example)])

    print('Finished running dataset construction')

    val = train[FLAGS.num_train_samples:]
    test = val[FLAGS.num_valid_samples:]
    val = val[:FLAGS.num_valid_samples]
    train = train[:FLAGS.num_train_samples]

    print(f'Dataset size: {len(train)}/{len(val)}/{len(test)}')

    write_to_file(train, f'{FLAGS.output_dir}/{FLAGS.task}_train')
    write_to_file(val, f'{FLAGS.output_dir}/{FLAGS.task}_val')
    write_to_file(test, f'{FLAGS.output_dir}/{FLAGS.task}_test')
    print('Finished writing all to file')

if __name__ == '__main__':
    tf.app.run(main)
```

3.1.1 How the Chebychev Encoder is Built

The ChebyshevPositionalEncoding functionality generates positional encodings based on Chebyshev polynomials. The TransformerModel combines a pretrained BERT model with Chebyshev Positional Encoding and includes a classifier for binary classification because the ListOps problem is a binary classification problem in my code. The ListOps code takes the value of each sequence and performs the modulus 2 operation to reduce the value down to a 0 if it's even or a 1 if it's odd.

In the code below, the ChebyshevPositionalEncoding class initializes with the model dimension (d_model) and the maximum sequence length (max_len). The constructor calls the generate_chebyshev_encodings method to create and store the positional encodings as a tensor.

The generate_chebyshev_encodings method creates a zero-initialized NumPy array of shape (max_len, d_model) to store the positional encodings. For each position in the sequence, it normalizes the position to the range [-1, 1]. Then, for each dimension in the model, it calculates the corresponding Chebyshev polynomial value using the chebyshev_polynomial method and stores it in the encoding array. Finally, it converts the encoding array to a PyTorch tensor.

The chebyshev_polynomial method calculates the value of the n-th Chebyshev polynomial at a given position `x`. It initializes the first two Chebyshev polynomials

$$(T_0(x) = 1)$$

and

$$(T_1(x) = x)$$

For higher-order polynomials, it iteratively computes the values using the recurrence relation

$$(T_k(x) = 2x \cdot T_{k-1}(x) - T_{k-2}(x))$$

and returns the n-th polynomial value.

The forward method takes a tensor positions as input and returns the corresponding Chebyshev positional encodings. It ensures the encodings are repeated across the batch size.

```
[ ]: class ChebyshevPositionalEncoding(nn.Module):
         def __init__(self, d_model, max_len=512):
             super(ChebyshevPositionalEncoding, self).__init__()
             self.d_model = d_model
             self.max_len = max_len
             self.encoding = self.generate_chebyshev_encodings()

         def generate_chebyshev_encodings(self):
             encodings = np.zeros((self.max_len, self.d_model))
             for pos in range(self.max_len):
                 normalized_pos = 2 * pos / (self.max_len - 1) - 1  # Normalize to␣
      ↪range [-1, 1]
                 for i in range(self.d_model):
                     encodings[pos, i] = self.chebyshev_polynomial(normalized_pos, i)
             return torch.tensor(encodings, dtype=torch.float32)

         def chebyshev_polynomial(self, x, n):
             T = [1, x]  # T_0(x) = 1, T_1(x) = x
             for k in range(2, n + 1):
                 T.append(2 * x * T[-1] - T[-2])
             return T[n]

         def forward(self, positions):
             batch_size, seq_len = positions.size()
             encodings = self.encoding[:seq_len, :].unsqueeze(0).repeat(batch_size,␣
      ↪1, 1)
             return encodings
```

4 Section 4: Experiment

4.0.1 Training

The training process for the model involves several key steps. First, the data preparation phase initializes the BERT tokenizer from the bert-base-uncased pretrained model and sets up the ListOpsDataset class to read and prepare datasets for training, validation, and testing. PyTorch DataLoader batches are then created with a batch size of 8. Next, the TransformerModel class is defined, incorporating a pretrained BERT model and Chebyshev Positional Encoding for handling long-term dependencies. The model can be converted to a GPU if available. The training setup includes defining the cross-entropy loss function, the Adam optimizer with L2 regularization, and a StepLR scheduler to decay the learning rate every 3 epochs. The training loop runs for 7 epochs, and processes batches by transferring data to the GPU, computing model outputs, calculating loss, performing backpropagation, clipping gradients, and updating parameters. After each epoch, validation loss and accuracy are calculated, with early stopping employed to prevent overfitting. Finally, the model is evaluated on the test dataset, and test loss and accuracy are reported.

4.0.2 Results

Models	ListOps % Accuracy
LAS	53.05
Chebyshev	49.51
LAS-chunk	46.21
Luna-256	37.98
Reformer	37.27
XFM	37.11
Transformer	36.37
BigBird	36.05
Linformer	35.70
Performer	18.01
Local Attention	15.82

Table 2: Model Accuracy As the Table 2 shows, the Chebychev Positional Encoder, in terms of accuracy, excels at the ListOps task compared with most other models.

5 Section 5: Conclusion

Despite the high accuracy compared to most of the benchmarks, these results should be viewed with a high degree of skepticism. In fact, there is good reason to believe these results are false positives. For one thing, this is only one test. There are numerous other tests in the 'Long Range Arena' to which the Chebychev would need to be subjected, such as the retrieval, image, and pathfinder tasks, in order to conclude it performs well on long range tasks.

The extent of the influence of the Bert base is unclear. The pretrained nature of the Bert model may contribute significantly to the performance, overshadowing the actual impact of the Chebychev Positional Encoding. To truly test if Chebychev is superior to positional encoding, it may be necessary to use a bare-bones transformer, as opposed to the Bert, that can be used to assess the differences in accuracy. Furthermore, the dataset used for training and evaluation might have specific characteristics that favor the Chebychev approach but may not generalize well to other datasets or real-world scenarios.

Finally, the computational efficiency and scalability of the Chebychev Positional Encoder in larger and more complex models remain untested, which could limit its practical application. Therefore, further investigations and testing across various tasks, datasets, and models are necessary to validate the effectiveness and reliability of the Chebychev Positional Encoder

6 Section 6: References

Anian Ruoss, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Róbert Csordás, Mehdi Bennani, Shane Legg, and Joel Veness. Randomized positional encodings boost length generalization of transformers. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pp. 1889–1903, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.161. URL https://aclanthology.org/2023.acl-short.161.

Delétang, Grégoire, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural networks and the chomsky hierarchy. In The Eleventh International Conference on Learning Representations.

Vaswani, Ashish , Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. arXiv preprint arXiv:2011.04006, 2020

Zimerman, Itamar, Lior Wolf. On the Long Range Abilities of Transformers. 2023. https://doi.org/10.48550/arXiv.2311.16620

```
[ ]:
```