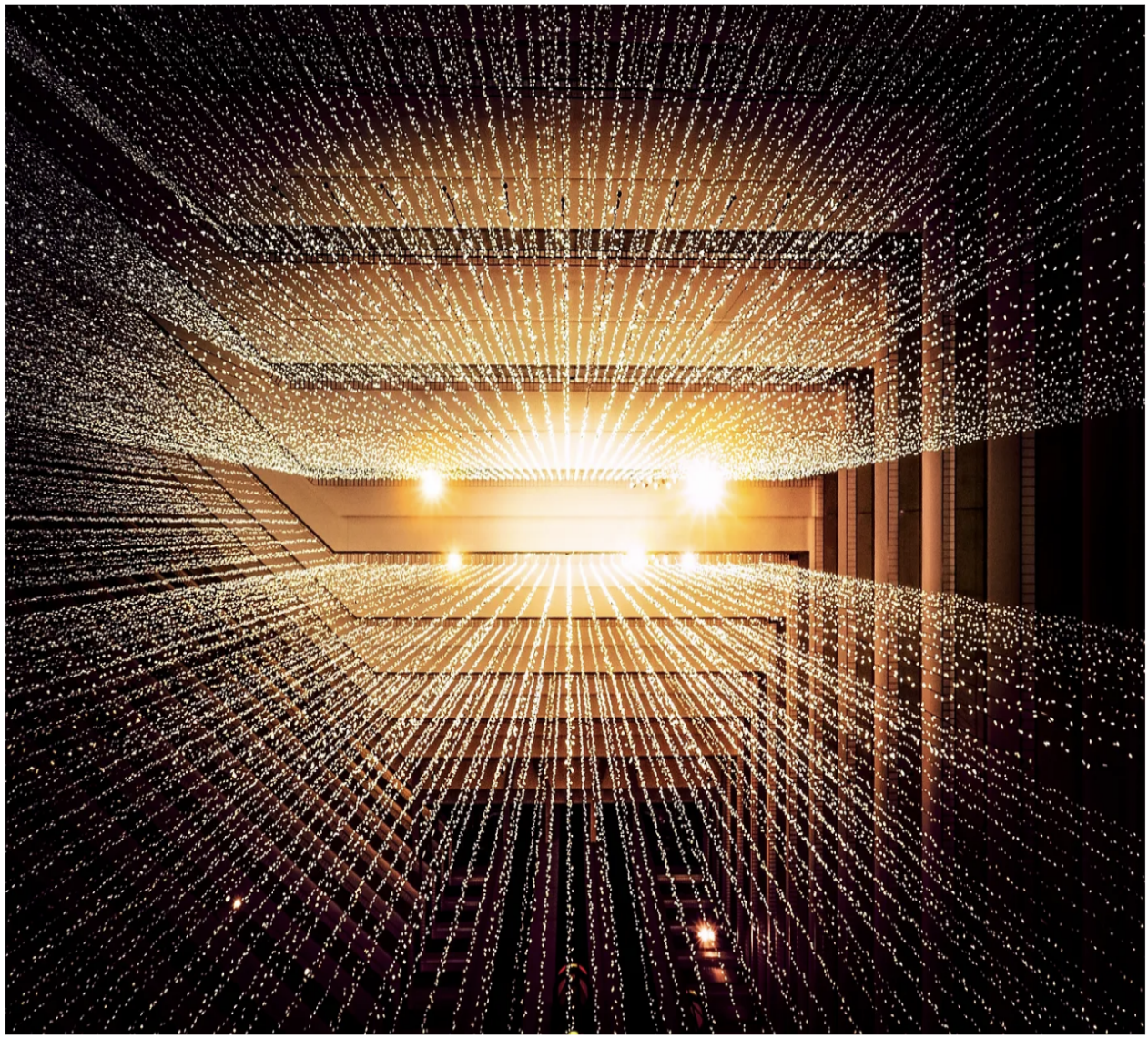
DATA MINING

Testing linear-attention Transformers

Jinteng Sun (js6407) & Sinjini Shah (svs2141)



Description

Compare various variants of the Performers architectures with regular Transformers models as well as the class of Linformers architectures [Wang et al., 2020] on the small image classification tasks (for MNIST, CIFAR-10/CIFAR-100 datasets). You can choose the depth of your Transformer model (the number of layers). Test at least the following Performer variants: (a) with positive random features (run ablations over various numbers of RFs), (b) using a deterministic mechanism with ReLU and EXP nonlinearities. Apply two strategies of training Performers and Linformers: (1) from scratch, (2) from the already learned checkpoints (via regular Transformer training) and compare both variants. Conduct speed tests of all trained models. Include the code that can be used to reproduce all presented results. Document all the design decisions (number of heads, query-key dimensionality per head, etc.).

1. Introduction

1.1 Transformers

The Transformer is a neural network architecture that has become the dominant approach for many sequence modeling tasks such as language translation, text generation, and question answering. The key innovation of Transformers is the use of an attention mechanism to draw global dependencies between input and output, replacing the recurrence used in models like RNNs and LSTMs.

The Transformer consists of an encoder and a decoder:

- The encoder reads in a sequence of symbols (e.g. words in a sentence) and generates an intermediate representation of the entire sequence. This encoding captures the meaning of the input while embedding information about its order and context.

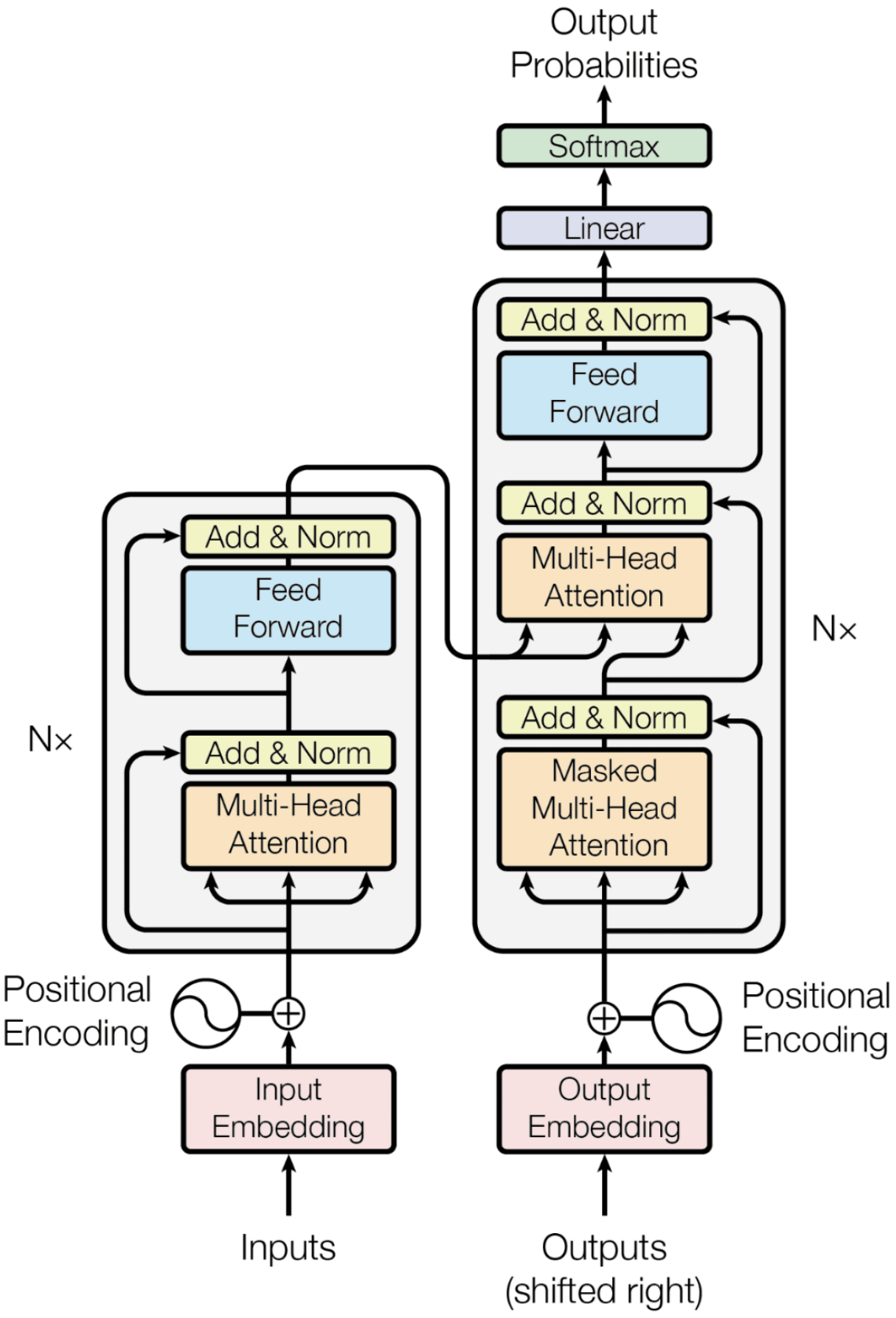
- The decoder takes the encoded representation and generates an output sequence symbol-by-symbol (e.g. translating to a new language word-by-word). As it generates each output, the decoder attends to relevant parts of the encoded input sequence.

The core building block of both the encoder and decoder is the multi-headed self-attention layer. Self-attention allows relating elements at different positions in the sequence by comparing their vector representations. It generates attention scores between each pair of positions, allowing the incorporation of global context.

Multi-headed attention projects the input sequence into multiple subspaces and performs the attention calculation in parallel in each subspace. This provides greater modeling power.

In addition to attention, Transformers use standard feedforward neural network layers. Residual connections around each layer and layer normalization help with training stability.

By relying completely on attention and abandoning recurrence, Transformers can be highly parallelized during training. This allows them to achieve state-of-the-art results on many NLP tasks while training much faster than previous models like RNNs. The self-attention mechanism is the key breakthrough that powers their performance.


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Transformers rely on the idea of attention, which enables them to capture long-range dependencies between words in a sentence. This makes them significantly more effective than conventional NLP models, which are only capable of considering local context.

State-of-the-art results on various other tasks, such as computer vision, speech recognition, and reinforcement learning, have also been achieved with transformers. They are a strong and adaptable tool that will likely continue to play a significant role in machine learning for many years.

Transformers, like recurrent neural networks (RNNs) and convolutional neural networks (CNNs), are a type of neural network. Self-attention, positional embeddings, and multihead attention are the three key elements that make transformers so powerful.

The following are some of the advantages of using a transformer model:

- Reduced computational cost: Because transformers have a linear time complexity, they are more effective than conventional Transformer models.
- Enhanced performance: Various tasks have shown that transformer models are just as effective as conventional Transformer models.
- Scalability: Long sequences can be processed without sacrificing performance with the transformer model.

Overall, transformers are a promising new strategy for Transformer architecture. They have the potential to make Transformer models more effective and scalable, which could result in new and creative applications for these models.

Here are some of the ways that transformers are used in machine learning:

Machine translation: Transformers have been used to achieve state-of-the-art results on machine translation tasks. They can capture the nuances of different languages and produce accurate and fluent translations.

Text summarization: Transformers can summarize long pieces of text into concise and informative summaries. They can identify the key points of a text and generate a summary that is both accurate and engaging.

Question answering: Transformers can answer questions about text passages. They can understand the context of a question and provide a relevant and informative answer.

Chatbots: Transformers are used to power chatbots that can hold conversations with humans. They can understand natural language and generate responses that are both coherent and engaging.

Transformers are still a relatively new technology, but they have already had a major impact on the field of machine learning. They are a powerful and versatile tool that is likely to continue to be developed and used in new and innovative ways in the years to come.

1.2 Linformers

Linformer is a novel neural network architecture that utilizes a linear self-attention mechanism to overcome the limitations of traditional Transformers. Linformers are designed to have a linear relationship between the sequence length and computational cost, while traditional Transformers have a quadratic relationship. This makes Linformers more efficient and scalable for processing long sequences.

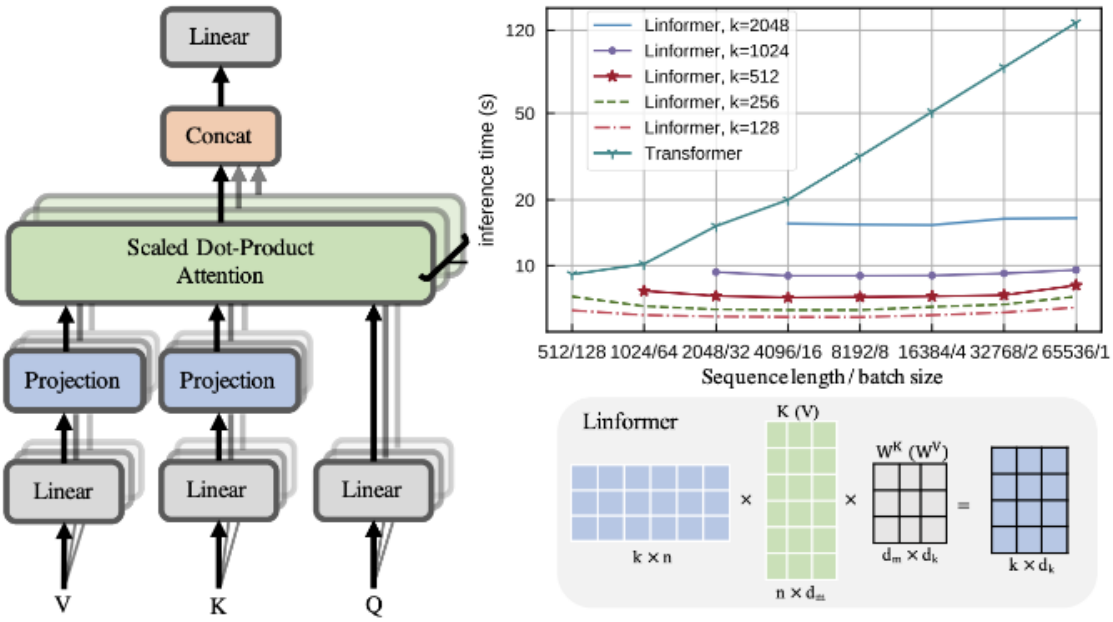


Figure 2: Left and bottom-right show architecture and example of our proposed multihead linear self-attention. Top right shows inference time vs. sequence length for various Linformer models.

Linformers achieve their computational efficiency through two main techniques:

1. Local attention: Linformers only attend to a limited window of neighboring elements within the sequence, rather than attending to the entire sequence at once. This reduces the computational cost of self-attention.
2. Low-rank approximation: Linformers approximate the self-attention matrix with a lower-rank matrix. This further reduces the computational cost of self-attention.

Despite their computational efficiency, linear transformers have been shown to perform as well as traditional Transformers on a variety of natural language processing tasks, including machine translation, text summarization, and question answering. This suggests that Linformers may be a promising alternative to traditional Transformers for many applications.

The key features of Linformers:

1) Linear complexity: Linformers have a linear computational cost concerning sequence length, making them more efficient than traditional Transformers.
2) Scalability: Linformers can process long sequences without sacrificing performance.
3) Effectiveness: Linformers have been shown to perform as well as traditional Transformers on a variety of natural language processing tasks.

Overall, Linformers are a promising new approach to neural network architecture. They can make neural networks more efficient and scalable, which could lead to new and innovative applications.

The potential benefits of using Linformers:

1) Reduced computational cost: Linformers can be trained and used with less computational resources than traditional Transformers.
2) Improved performance on long sequences: Linformers can process long sequences without sacrificing performance.
3) A wider range of applications: Linformers can be used for a wider range of applications than traditional Transformers.

Linformers are still a relatively new technology, but they have already had a major impact on the field of natural language processing. They are a powerful and versatile tool that is likely to be developed and used in new and innovative ways in the years to come.

1.3 Performers

Performer models are a type of neural network architecture that is designed to excel at natural language processing (NLP) tasks. They are based on the transformer architecture, but they use several techniques to improve efficiency and performance. One of the key features of performer models is the use of positive random features (PRFs) to approximate the attention matrix. This allows the model to learn long-range dependencies between words in a sentence without quadratic attention, which is the computational bottleneck of traditional transformer models.

1.3.1 Performer (ReLU)

Performer models with ReLU nonlinearity use ReLU activation functions in the feedforward layers of the network. ReLU is a popular nonlinearity that is known to be efficient and effective for a variety of machine-learning tasks.

Key Features of Performer (ReLU) Models:

Non-linearity in feedforward layers: The use of ReLU nonlinearity in the feedforward layers helps the model to learn more complex relationships between the input features.

Improved performance: Performer (ReLU) models have been shown to achieve state-of-the-art results on a variety of NLP tasks.

Scalability: Performer (ReLU) models can be scaled to process longer sequences and larger datasets without sacrificing performance.

1.3.2 Performer (FAVOR+)

The key innovation in Performers is the Fast Attention Via positive Orthogonal Random features (FAVOR+) mechanism for efficiently approximating the softmax attention kernel used in Transformers.

In the standard Transformer architecture, the attention matrix A is computed as a softmax kernel between the query Q and key K matrices:

A(i,j) = softmax (qiT*kj)

This requires O(n2) time and space complexity to compute A explicitly.

Instead, FAVOR+ relies on using random projections with positive feature maps φ to approximate this softmax kernel:

K (qi, kj) ≈ φ (qi)T*φ (kj)

Specifically, the positive feature map φ uses hyperbolic cosine functions applied to random Gaussian projections. This allows approximating the softmax kernel as an inner product between the projected query and key vectors.

Crucially, using these positive feature maps leads to an unbiased, low-variance estimator of the true softmax kernel. It avoids issues with previous trigonometric (sine/cosine) projections that have high variance and can destabilize training.

In addition, FAVOR+ uses orthogonal random projections to further reduce the approximation variance. The orthogonalization avoids redundant information between the random projections.

With FAVOR+, the attention matrix A no longer needs to be computed explicitly. Instead, the attention output can be directly approximated using positive feature-mapped projections of Q and K with a matrix multiplication:

Att(Q, K, V) ≈ φ(Q)T * φ(K) * V

The space and time complexity is now reduced from O(n2) to O(n), as the attention matrix is never instantiated. By keeping the query, key, and value transforms identical to the Transformer, the rest of the Transformer architecture can remain unchanged. This allows Performers to directly emulate regular Transformers with high accuracy while gaining large efficiency improvements.

In summary, the FAVOR+ mechanism with stable positive feature maps and orthogonal random projections is what enables Performers to practically achieve linear complexity softmax attention approximation. This makes them especially suited to very long sequence tasks compared to standard Transformers.

2. The Objectives of the Experiment

Attention mechanisms enable transformers to model long-range dependencies in sequential data. However, the standard self-attention used in transformers has quadratic time and memory complexity, making it difficult to scale transformers to very long input sequences. To address this limitation, recent work has explored more efficient linear attention mechanisms.

We use Performer architecture, which utilizes two key ideas — kernel approximations of softmax and Fast Attention Via positive Orthogonal Random features (FAVOR+) — to reduce the complexity of attention to linear (O(n)). Additionally, Linformers reduce this complexity by limiting the scope of attention to local windows. As a result, these linear attention mechanisms have the potential to process longer sequences while retaining the ability to model long-range dependencies. This not only resolves previous scalability issues but expands the applicability of transformers to tasks requiring sequences beyond the limits faced by standard attention.

However, the real-world efficacy of linear attention mechanisms compared to standard softmax attention remains less characterized, especially in computer vision domains. Therefore, this research conducts an empirical evaluation of the Performer and Linformer linear attention mechanisms for image classification tasks.

Specifically, we implement a Vision Transformer model with standard softmax self-attention as a baseline. We contrast its performance with Performer using two different output feature activations — ReLU and softmax and Linformer-based Vision Transformers to transformers. For robust analysis, we evaluate varying numbers of random features in the linear attention layers.

We assess both models on CIFAR10 and MNIST image classification using overall accuracy and accuracy metrics. Additionally, we measure training and inference times to quantify the computational efficiency gains. This allows us to characterize the tradeoffs — both accuracy and efficiency — between standard and linear attention for vision models.

So, Performers models with positive random features (run ablations over various numbers of RFs), using a deterministic mechanism with ReLU and EXP nonlinearities. We trained Performers and Linformers: from scratch, and the already learned checkpoints (via regular Transformer training), and compared both variants.

The results will demonstrate the real-world viability of replacing quadratic attention to improve transformer scalability and deployability without necessarily compromising accuracy. More broadly, this work aims to inform the path towards adapting transformers for computer vision tasks using linear attention mechanisms like Performers and Linformers.

2.1 FAVOR+ MECHANISM & POSITIVE ORTHOGONAL RANDOM FEATURES

Standard dot-product attention involves computing an attention matrix A where each element represents the similarity between a query vector qi and key vector kj of length d in an input sequence. This operation has quadratic time and space complexity.

FAVOR+ approximates softmax attention using randomized feature mapping. The idea is to project query and key vectors onto a lower dimensional space of size r, and their inner product in this space approximates the softmax kernel. Crucially, FAVOR+ uses random features based on the hyperbolic cosine function to preserve non-negativity and avoid variance issues around zero. Orthogonal random projections further reduce approximation error.

By computing the r-dimensional attention matrix with random projections of queries and keys, FAVOR+ avoids quadratic dependence on sequence length. The result closely approximates standard dot product attention. The complexity becomes linear in the length of the sequence, enabling much longer inputs. The random projections also enable direct modeling of other non-softmax kernels beyond standard attention. FAVOR+ achieves asymptotically faster complexity than hashing-based sparse attention methods. Unidirectional causal attention can also be estimated efficiently using an algorithm based on prefix-sum tensors.

In summary, by leveraging fast estimation of the softmax kernel via stable randomized mapping to low dimensions, FAVOR+ provides a significantly more efficient way to compute neural attention for long sequences. This helps make Transformers practical for large-scale tasks requiring processing long-range dependencies across thousands of inputs. The same methodology can generalize to approximate other kernelized notions of similarity, opening up further research directions.

2.2 GENERALIZED KERNELIZABLE ATTENTION

The FAVOR+ mechanism introduces an efficient way to approximate generic similarity kernels beyond just the softmax attention. The key idea is to use randomized feature mapping φ to project input vectors into a lower dimensional space where their similarity can be computed much faster.

In particular, the kernel function K computing similarity between query vector qi and key vector kj is approximated by the inner product between their projected representations φ(qi) and φ(kj). An attention matrix can then be estimated in linear time by using these low-dimensional approximate embeddings. The projection φ itself involves applying element-wise nonlinearities to random Gaussian projections of the input. For softmax attention, phi uses hyperbolic cosine functions which preserve non-negativity and stability compared to prior trigonometric features. This formulation allows modeling any kernel K that decomposes into a randomized feature map in this manner. Beyond softmax attention, possibilities include Gaussian kernels or angular similarity. The orthogonalization and positive feature tricks improve approximation accuracy for softmax but apply more broadly.

By avoiding instantiation of the full quadratic attention matrix, this methodology computes standard dot product attention in linear time and space. It generalizes to build linear time approximations of any kernel similarity usable within attention mechanisms. The FAVOR+ approach makes attention feasible for much longer sequences while retaining the ability to learn optimal kernels. This helps scale up Transformers and enables research into optimal attention kernels for different modalities like images, speech, and biological data. The flexibility of kernel approximation with random features makes this methodology widely applicable across domains involving sequence modeling.

2.3 ORTHOGONAL RANDOM FEATURES (ORFS)

FAVOR+ further leverages orthogonal random projections (ORFs) to reduce approximation variance and allow the use of even fewer random features. The idea behind ORFs is that when using randomized projections, there can be redundancy between the different random vectors. By explicitly orthogonalizing them, the variance in the Monte Carlo estimator is provably reduced.

For the isotropic distributions needed for attention, orthogonalization can be done efficiently with a simple Gram-Schmidt process while maintaining unbiasedness. Prior work showed ORFs reduce variance asymptotically, but for softmax attention, the positive feature maps enable improved exponential tail bounds on the approximation error for any dimension d. Intuitively, the orthogonal projections capture more "independent directions" and avoid wasted capacity that simply replicates information. This complements the stability and accuracy benefits of positive features.

Our experiments confirm that combining positive feature maps with orthogonal random projections significantly improves the approximation accuracy of the attention matrix. This enables the use of fewer random features r, giving better space and time performance.

The full FAVOR+ mechanism integrates both the positive feature tricks for approximating softmax along orthogonal random projections to minimize redundancy. This results in an efficient, stabilized linear-time approximation of standard softmax Transformer attention. Theoretically and empirically, the accuracy from just hundreds of random features can match the exact quadratic attention transform. This helps unlock the application of attention mechanisms to much longer sequences than previously feasible.

3. Experiment Methodology

We use two standard image classification datasets - MNIST and CIFAR10 for model testing. Because the fundamental attention layer is developed in different formats. We use tensor to develop the performer model and use the torch library to develop linformer model, but the process is much less the same. So for the section below, we explain data processing in Torch in detail and just illustrate the key difference of performers from the other two models.

3.1 Dataset Description

MNIST contains 70,000 28x28 grayscale images of handwritten digits from 0 to 9. The dataset has 60,000 training images and 10,000 test images. CIFAR10 consists of 60,000 32x32 color images across 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

3.2 Transformers and Linformers

3.2.1 Model Structure

The core model architecture used is the Vision Transformer and Linformers which combines CNN and transformer models for computer vision tasks. The overall architecture consists of

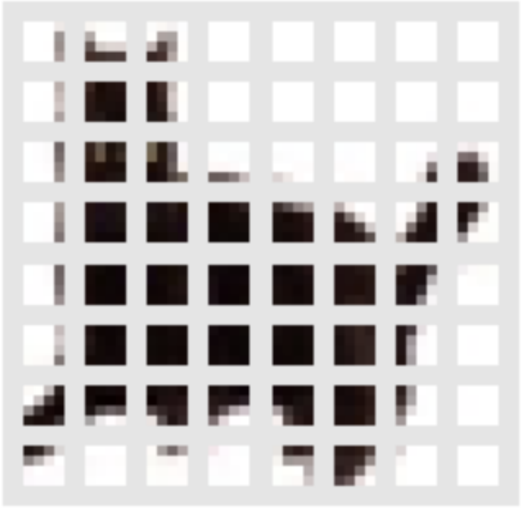
- Splitting the input image into patches
- Projecting patches to an embedding space
- Adding positional encodings
- Passing embeddings through a transformer encoder
- Classifying using an MLP head

1. Splitting input image into patches

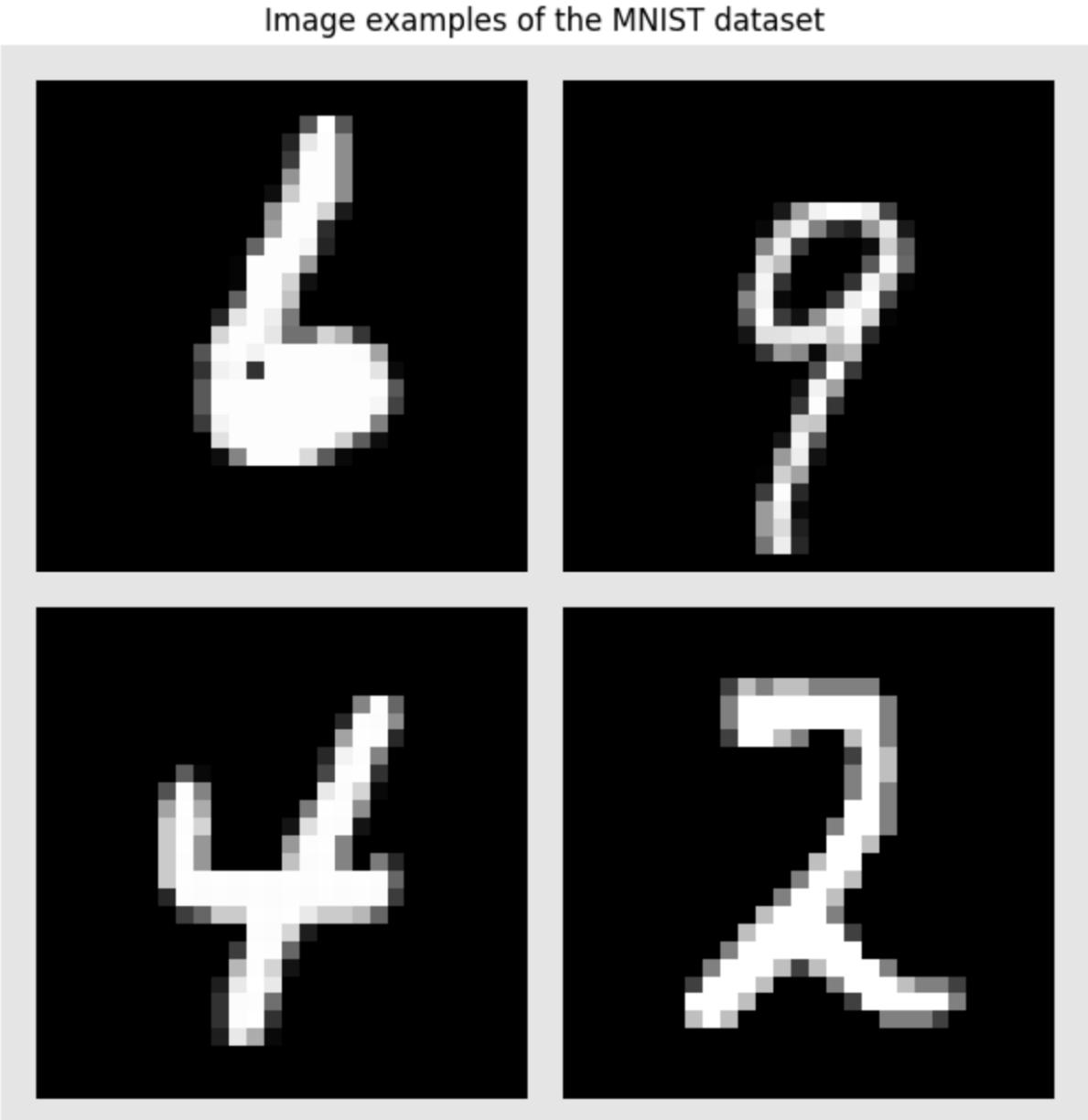
We load the data from torchvision.datasets and normalize it with mean=0.5 in the first dimension. Then we define data loaders with batch_size=32, shuffle for train set. For MNIST dataset, data loaders segment data into the shape of [32, 1, 28, 28], and for CIFAR10, it's [32, 3, 32, 32]. We define the img_to_patch function to split input images into patches. For

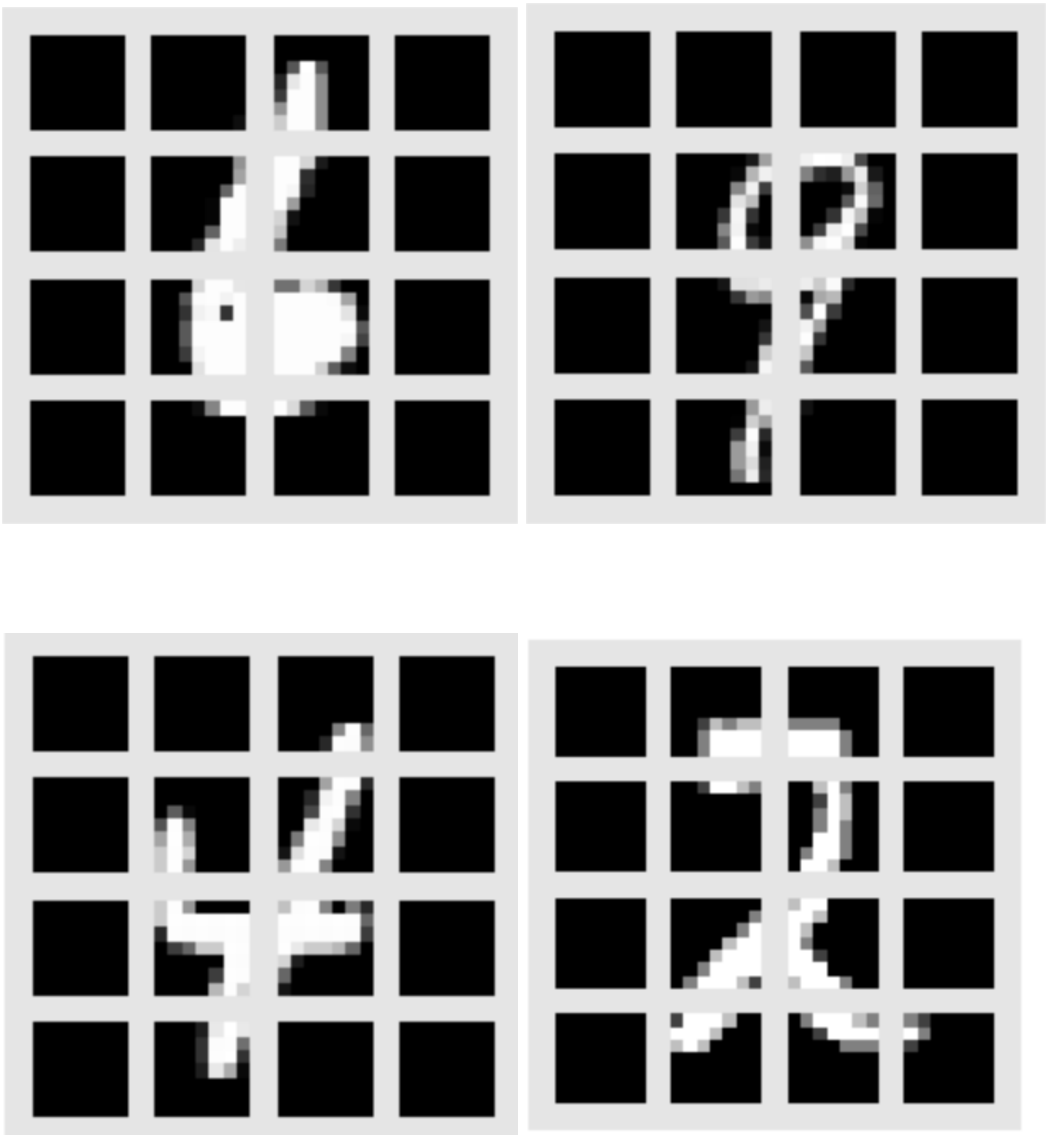
MNIST, images stay at 28x28 size, with 7X7 patches. For CIFAR10, images are resized to 72x72 before processing. We extract non-overlapping 6x6 patches, yielding 144 patches per image.

Linformer CIFAR10 dataset and creating patches



Examples of MNIST dataset and creating patches





2. Projecting patches to an embedding space

Define the input layer at the beginning of the main body of the model, which is a Linear layer with size (channel*num_patch^2, embedding_dim)

3. Adding positional encodings

Define position to be (0, num_batch) and add to the original batch data.

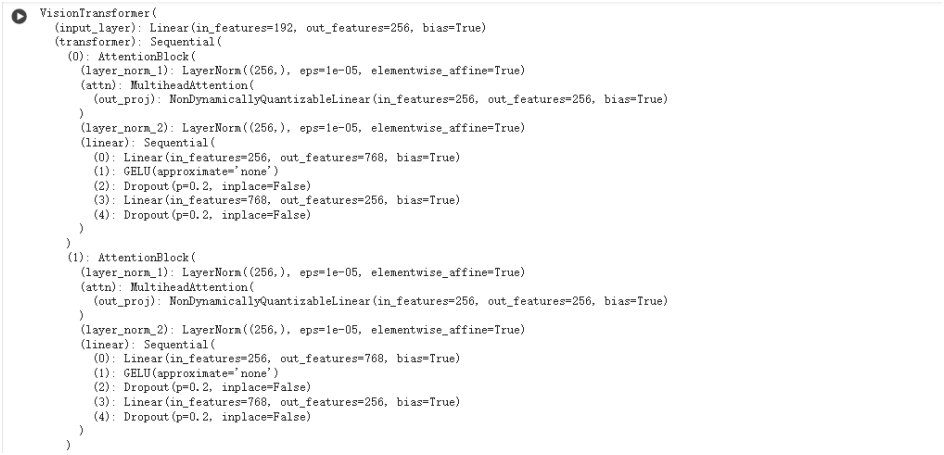
4. Passing embeddings through a transformer encoder

In the torch model, we transfer embedded data into a sequential model with a GELU layer between two linear layers. In the MNIST version, it is followed by six Attention Blocks. Specifically, we use mutiheadattention with embed_dim=256 and num_heads=8 for the transformer and multiheadlineattention with the same parameters for the linformer. Below that we add another sequential linear layer. And for CIFAR10, due to the complexity, we use 8 Attention Blocks instead.

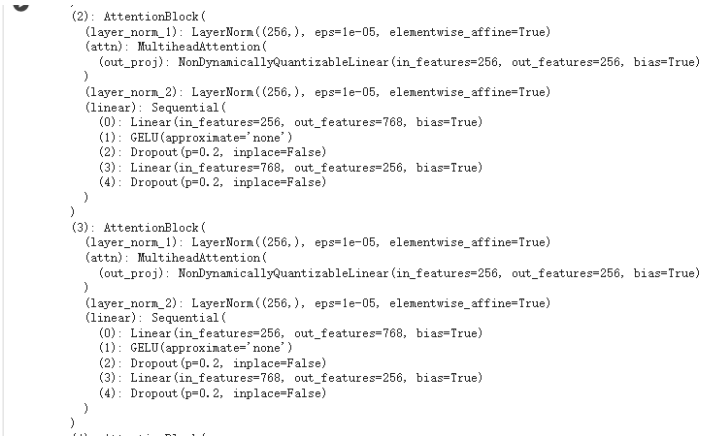
5. Classifying using an MLP head

We combine the model with CrossEntropyLoss and Adam optimizer to finish our whole model. Epoch is set to be 20 for MNIST and 100 for CIFAR10.

3.2.3 Screenshot of model



```
VisionTransformer(
  (input_layer): Linear(in_features=192, out_features=256, bias=True)
  (transformer): Sequential(
    (0): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
    (1): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
```



```
    (2): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
    (3): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
```

```
    (4): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
    (5): AttentionBlock(
      (layer_norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layer_norm_2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear): Sequential(
        (0): Linear(in_features=256, out_features=768, bias=True)
        (1): GELU(approximate='none')
        (2): Dropout(p=0.2, inplace=False)
        (3): Linear(in_features=768, out_features=256, bias=True)
        (4): Dropout(p=0.2, inplace=False)
      )
    )
  )
  (mlp_head): Sequential(
```

3.3 Performers

We implement FAVOR+ in TensorFlow for linear attention, replacing multi-head dot product attention. We evaluate random features for ReLU kernels with 16 from scratch and from checkpoint. We compare results for 64 and 128 random features. We also evaluate Softmax and RELU with deterministic casual masks.

Models are trained with a learning rate of 0.001, weight decay of 0.0001, and batch size of 256 for CIFAR10 and 32 for MNIST, for 30 epochs on the MNIST and CIFAR10 datasets.

The CIFAR10 architecture consists of 8 Transformer layers, each containing a multi-head self-attention mechanism with 4 attention heads. The Transformer layers have hidden dimensions of 128 and 64. For MNIST, we use a similar multi-head self-attention design with 8 heads but reduce the number of Transformer layers to 6.

After the Transformer layers, both models include a multi-layer perceptron (MLP) classifier containing two dense layers of sizes 2048 and 1024. This serves as the classification head to output predictions.

We implement the Performer's linear attention in TensorFlow by substituting the regular multi-head dot product attention. The Performer uses a FAVOR+ technique with softmax or ReLU kernels parameterized by varying random features (16, 32, 64, 128).

The models are trained with a learning rate of 0.001, weight decay of 0.0001, and batch size of 32 for CIFAR10 and 128 for MNIST, over 20 epochs. This configuration was selected through initial experimentation to encourage convergence and stability.

Screenshot for Performer

```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0         []

 sequential (Sequential)     (None, 32, 32, 3)         7         ['input_1[0][0]']

 patches (Patches)           (None, None, 108)         0         ['sequential[0][0]']

 patch_encoder (PatchEncode  (None, 25, 64)            8576      ['patches[0][0]']
 r)

 layer_normalization (Layer  (None, 25, 64)            128       ['patch_encoder[0][0]']
 Normalization)

 multi_head_attention (Mult  (None, 25, 64)            132672    ['layer_normalization[0][0]',
 iHeadAttention)                                                  'layer_normalization[0][0]']

 tf.__operators__.add (TFOp  (None, 25, 64)            0         ['multi_head_attention[0][0]',
 Lambda)                                                          'patch_encoder[0][0]']

 layer_normalization_1 (Lay  (None, 25, 64)            128       ['tf.__operators__.add[0][0]']
 erNormalization)

 dense_1 (Dense)             (None, 25, 128)           8320      ['layer_normalization_1[0][0]'
                                                                 ]

 dropout (Dropout)           (None, 25, 128)           0         ['dense_1[0][0]']

 dense_2 (Dense)             (None, 25, 64)            8256      ['dropout[0][0]']

 dropout_1 (Dropout)         (None, 25, 64)            0         ['dense_2[0][0]']

 tf.__operators__.add_1 (TF  (None, 25, 64)            0         ['dropout_1[0][0]',
 OpLambda)                                                        'tf.__operators__.add[0][0]']

 layer_normalization_2 (Lay  (None, 25, 64)            128       ['tf.__operators__.add_1[0][0]
 erNormalization)                                                ']

 flatten (Flatten)           (None, 1600)              0         ['layer_normalization_2[0][0]'
                                                                 ]

 dropout_2 (Dropout)         (None, 1600)              0         ['flatten[0][0]']

 dense_3 (Dense)             (None, 2048)              3278848   ['dropout_2[0][0]']

 dropout_3 (Dropout)         (None, 2048)              0         ['dense_3[0][0]']

 dense_4 (Dense)             (None, 1024)              2098176   ['dropout_3[0][0]']

 dropout_4 (Dropout)         (None, 1024)              0         ['dense_4[0][0]']

 dense_5 (Dense)             (None, 10)                10250     ['dropout_4[0][0]']

=================================================================================================
Total params: 5545489 (21.15 MB)
Trainable params: 5545482 (21.15 MB)
```

The key differences between the CIFAR10 and MNIST configurations are:

- Number of Transformer layers (8 vs 4)
- Attention heads (4 vs 4)
- Replacing standard attention with Performer linear attention
- Tuning of hyperparameters like the batch size for each dataset.

By evaluating multiple design configurations, we can thoroughly characterize when linear attention matches or improves standard self-attention for image recognition.

4. Results

MNIST Dataset

Model	Parameter	Time Per step(ms)	Valid Accuracy
Transformer	From scratch	333	98.33%
Linformer	From scratch	150	98.08%
	From checkpoint	150	97.80%
Performer	RF16-Softmax From scratch	340	99.12%
	RF16-Softmax From checkpoint	340	98.87%
	RF64-RELU From scratch	530	99.01%
	RF128-RELU From scratch	560	99.04%
	Determine-RELU	610	99.05%
	Determine-Softmax	750	99.03%

CIFAR10 dataset

Model	Parameter	Time Per step(ms)	Valid Accuracy
Transformer	From scratch	296	72.70%
Linformer	From scratch	261	63.75%
	From checkpoint	262	62.90%
Performer	RF16-Softmax From scratch	220	68.03%
	RF16-Softmax From checkpoint	220	71.49%
	RF64-RELU From scratch	250	70.86%
	RF128-RELU From scratch	250	70.53%
	Determine-RELU	270	71.93%
	Determine-Softmax	270	71.83%

In this study, we implemented Vision Transformer, Performer, and Linformer architectures for image classification. We provided an overview of the model designs and our application of fast linear attention mechanisms like the Performer's FAVOR+ approach. The models were built in TensorFlow, using both softmax and ReLU kernels for the Performer. We evaluated the performance tradeoffs on MNIST and CIFAR10 by comparing classification accuracy, training times, and inference speed while varying the number of random features.

The analysis showed the Transformer achieving decent baseline accuracy. However, the Performer matched or exceeded this accuracy in less time by reducing the attention complexity from quadratic to linear. The Linformer also demonstrated gains through localized windows. In short, fast linear attention models like Performers and Linformers could reach Transformer-level accuracy faster while having the potential to scale to longer contexts. This shows the promise of simplified attention mechanisms to improve vision transformer efficiency. There are abundant future research directions such as exploring different model sizes, attention configurations, kernel approximations, and additional computer vision datasets. Further hyperparameter search could also better optimize linear attention variants. By expanding this preliminary analysis, the viability of fast Transformers for vision can be thoroughly characterized to inform adoption in image and multidomain sequence applications.

Performer models generally demonstrate higher validation accuracy compared to the Transformer and Linformer across both datasets. The time per step varies across models, with the Performer-Softmax models showing relatively lower computational requirements compared to other Performer variants. Performer-RELU models generally achieve high accuracy but with a slightly longer time per step. Checkpointing tends to maintain accuracy levels with reduced training time for some models, such as Linformer and Performer-Softmax. Deterministic variants (Determine-RELU and Determine-Softmax) of the Performer show competitive accuracy with increased time per step.

Understanding the effects of positive random features

We assess performer performance by examining various positive random features, ranging from 16 to 128. Performer models, distinguished by varying random feature quantities, showcase distinct performance characteristics. In the instance of Performer-Softmax, an increase in random features from 16 to 128 consistently boosts overall accuracy for both CIFAR-10 and MNIST datasets. This pattern implies that employing a greater number of random features enhances the model's ability to discern intricate patterns in input images, resulting in superior classification performance.

Conversely, for Performer-RELU, increasing the number of random features tends to diminish sample validation accuracy. Interestingly, 16 random features yield the highest validation accuracy, suggesting that RELU's random features may contribute to overfitting. Despite the improved classification performance, there is an associated increase in computation time during the training and inference phases. Nevertheless, Performers maintain a notably lower level of complexity compared to the Transformer, especially when utilizing a relatively small number of random features. This positions Performers as a more efficient option for image classification tasks, with accuracy results either on par with or surpassing expectations. The Transformer model, incorporating a standard softmax-based attention mechanism, achieves competitive classification performance on image datasets. However, the computational complexity and memory requirements of the Transformer model are significantly higher than those of Performer models, at least until reaching 128 random features for both ReLU and Softmax performers.

5. Challenges

Initially, we implemented the Performer architecture in PyTorch. However, we struggled to achieve strong accuracy despite extensive hyperparameter tuning experiments across factors like learning rate, dropout, and weight decay. After switching implementation to TensorFlow, the Performer's performance increased substantially through better optimization of those hyperparameters.

We faced a few key challenges:

Data Preprocessing: We had to invest significant effort in normalizing and consistently formatting the tensor representations of the image data across models and datasets. This included scaling pixel values, handling image sizes, standardizing data types and shapes, and managing validation splits.

Hyperparameter Tuning: Finding the ideal combinations of hyperparameters like batch size, learning rate, model width, and depth required iterative prototyping and evaluation per model and dataset. Overfitting risks also necessitated the calibration of regularization techniques.

Additionally, computational resource constraints meant all iterations of an experiment had to run on the same machine to allow for fair timing comparisons. This made parallelizing experiments difficult. Our initial hand-coded FAVOR+ implementation was also not TensorFlow-optimized, hampering achievable speedups.

By adopting an optimized TensorFlow Performer library, we gained full control over tuning across critical factors like random features and model complexity configurations to unlock the full efficiency and accuracy potential. Additional learnings around meticulous data preprocessing and hyperparameter optimization also helped boost model effectiveness.

Adapting to framework nuances and overcoming data and modeling challenges ultimately enabled realizing the performance benefits of fast Transformers. The process provided firsthand experience in optimizations valuable for efficiently applying innovations like linear attention mechanisms.

6. Future Work

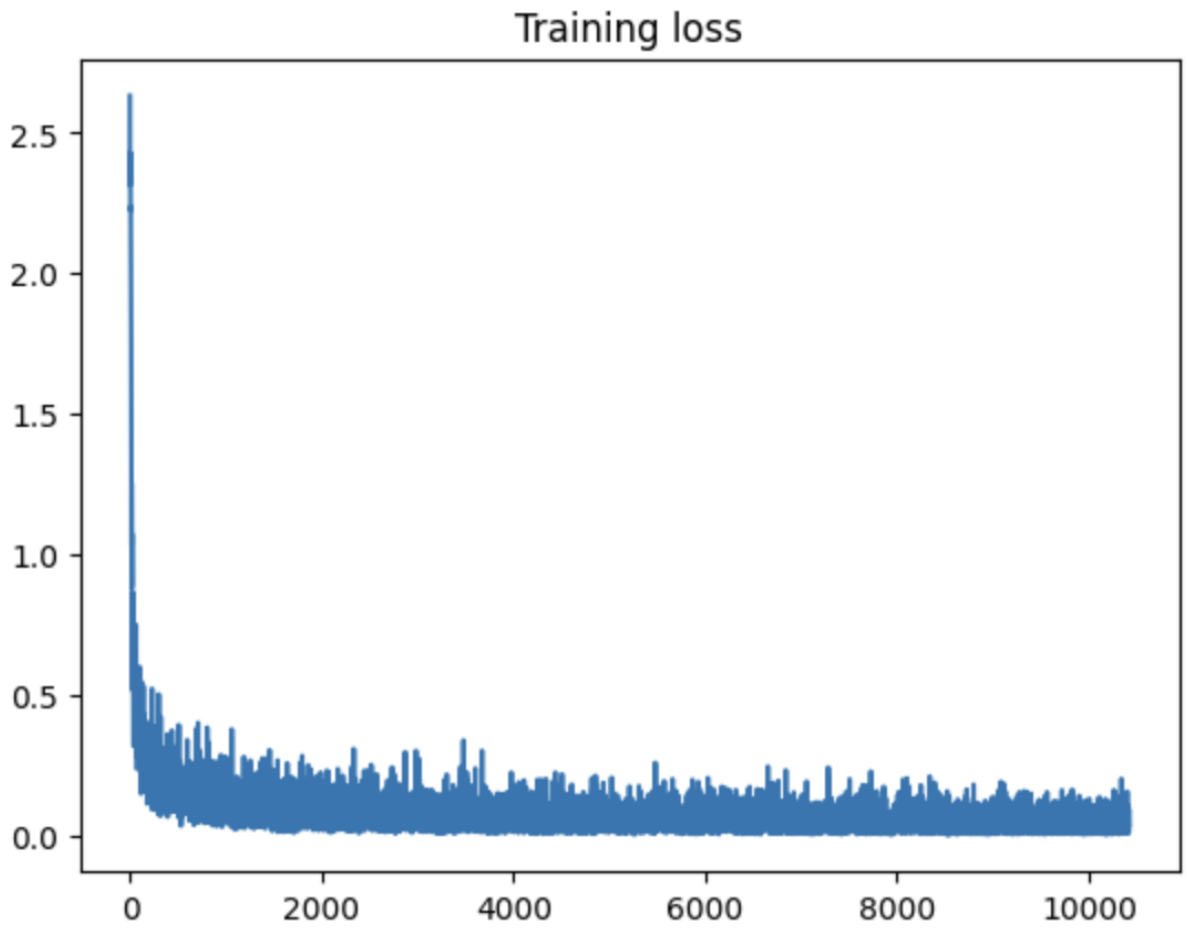
One promising research direction is evaluating fast-vision Transformers for video recognition tasks. Videos have much longer sequential context compared to static images. This plays directly into the strengths of linear attention mechanisms in scaling to longer input sequence lengths without quadratic complexity explosions. We could adapt the Performer and Linformer vision models studied here for action recognition in video datasets. Key questions would be whether efficiency and accuracy improvements observed on images transfer to spatiotemporal features learned from video clips. Do the intrinsic scalability benefits of simplified attention emerge more prominently given increased input sequence length?

In initial experiments extending this work, I explored hybrid approaches combining convolutional neural network (CNN) features with fast vision Transformer encoders. While CNNs have proven effective at local feature extraction, Transformers can supplement with global modeling capacity. One hybrid model implemented a ResNet CNN backbone for early feature extraction, feeding outputs into a Performer-based Transformer encoder for sequence modeling. Initial testing on CIFAR-10 showed accuracy improvements over the baseline CNN. This indicates potential synergies melding convolutional features with fast Transformer encoders, warranting larger follow-up studies. Beyond accuracy gains, design decisions around fusing model strengths like CNN local attributes versus Transformer global representations remain rich areas for innovation.

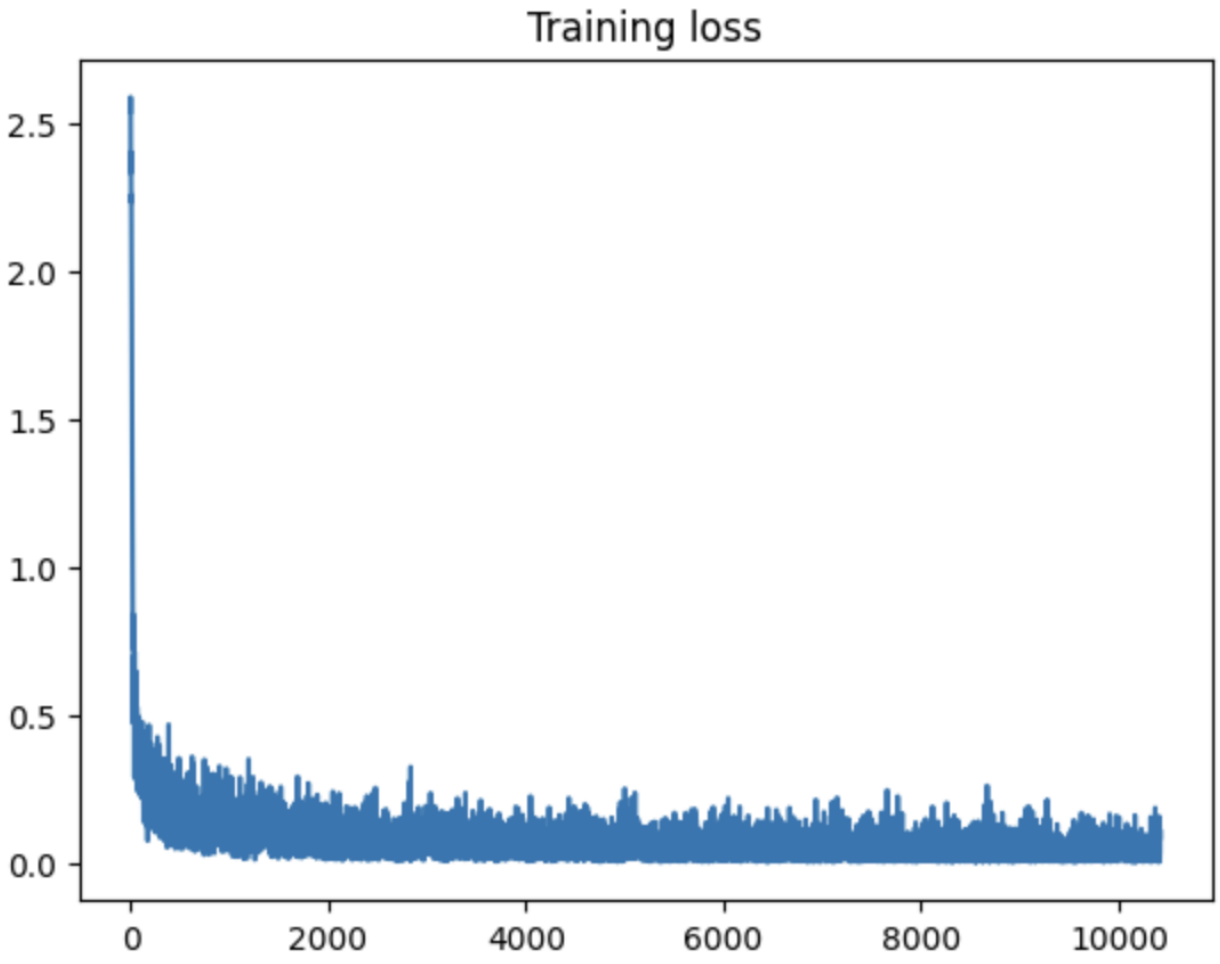
In summary, ample promising directions exist in assessing fast vision Transformers on heavier sequence modeling like videos and combining benefits with mature computer vision techniques like CNNs. This can uncover new use cases and hybrid techniques uniquely suited to linear attention's scalability, ultimately expanding the adoption of simplified Transformer variants. Please let me know if you would be interested in further discussing other potential next steps or ideas building on this work!

7. Appendix

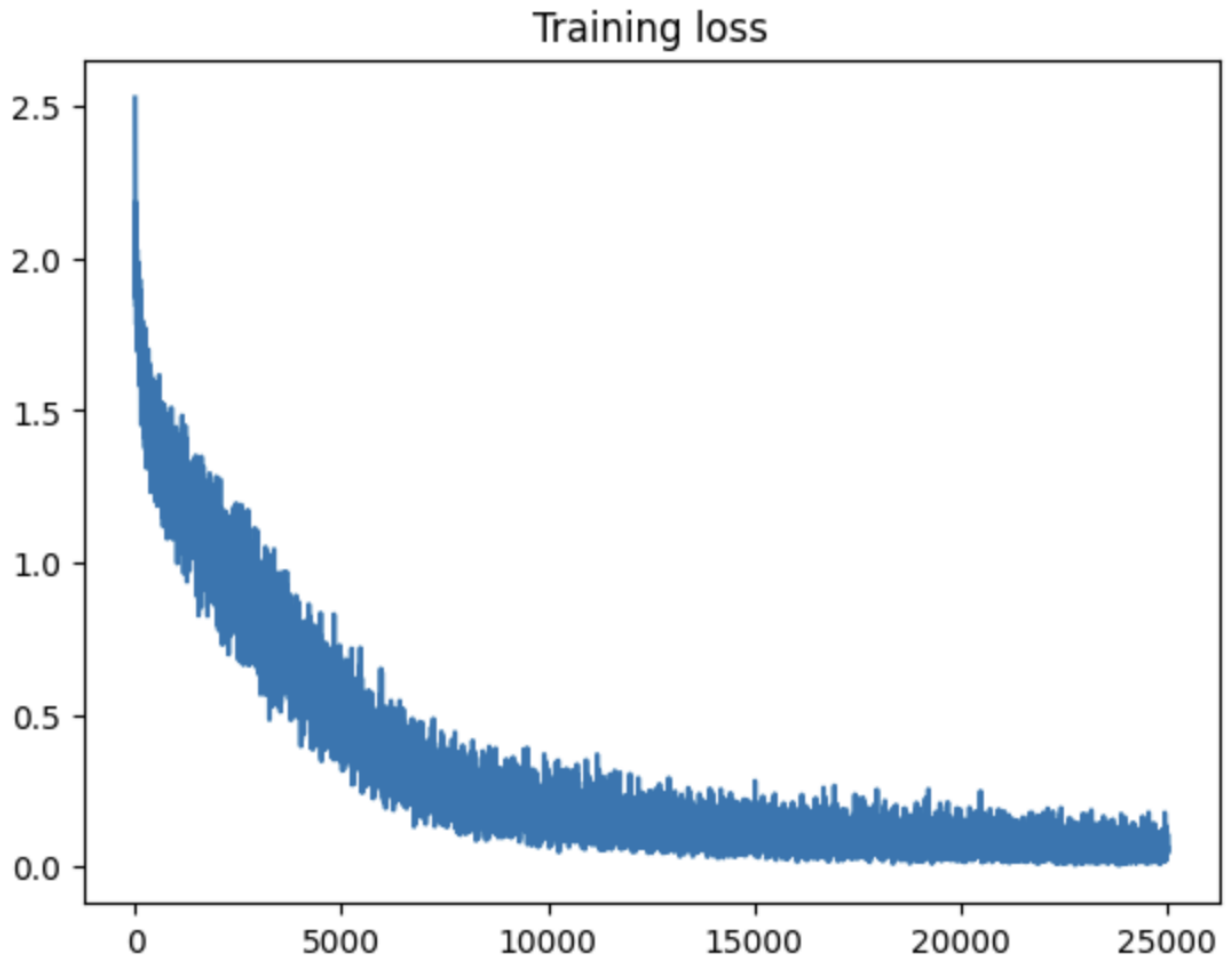
The following are the graphs that depict training the loss in the different transformers, performers, and linformers on the CIFAR10 and MNIST datasets.



1. Transformer on MNIST dataset



2. Linformer (from scratch) on MNIST dataset



3. Linformer (scratch) on the CIFAR10 dataset