Data Mining 1 Report

Fall 2024

Optimizing Image Classification on CIFAR-10 with Transformer Architectures

Columbia University - Data Mining

Authors:
Daniel Clepper, [Anonymous Student] (x2)

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Problem Statement . . . 3
- 1.3 Objectives . . . 3

2 Theory and Architecture **4**
- 2.1 Intuition Behind Attention . . . 4
- 2.2 Standard Attention Mechanism in Image Processing . . . 4
  - 2.2.1 Embedding . . . 4
  - 2.2.2 Query, Key and Value . . . 5
  - 2.2.3 Self Attention . . . 6
- 2.3 Multi-Head Attention Mechanism in Image Processing . . . 7
  - 2.3.1 Independent Attention Heads . . . 7
  - 2.3.2 Attention Calculation for Each Head . . . 7
  - 2.3.3 Concatenating and Projecting Heads . . . 7
  - 2.3.4 Advantages of Multi-Head Attention . . . 8
- 2.4 Transformer Architecture . . . 8
  - 2.4.1 Input Embedding Layer . . . 8
  - 2.4.2 Multi-Head Self-Attention Mechanism . . . 8
  - 2.4.3 Feedforward Neural Network (FFN) . . . 8
  - 2.4.4 Residual Connections and Layer Normalization . . . 8
- 2.5 Performers, Fast Attention Mechanisms and Kernel-based Approximation . . . 9
  - 2.5.1 Introduction to Efficient Attention Mechanisms . . . 9
  - 2.5.2 FAVOR+: A Kernel-based Approximation . . . 9
  - 2.5.3 Key Components of Performers . . . 9

3 Experimental Setup **10**
- 3.1 Our dataset . . . 10
- 3.2 Data Transformations . . . 10
- 3.3 Dataset Splitting . . . 12
- 3.4 Hardware . . . 13
- 3.5 Common Hyper-parameters for all models . . . 13
- 3.6 Common Hyper-parameters for all models . . . 14

4 Vision Transformer **15**
- 4.1 Model Setup . . . 15
  - 4.1.1 Model Architecture . . . 15
  - 4.1.2 Hyperparameter Optimization . . . 15
- 4.2 Hyperparameter Tuning and Optimization . . . 16
- 4.3 Evaluation Metrics . . . 16
- 4.4 Results And Observations . . . 16
  - 4.4.1 Classification Accuracy . . . 16
- 4.5 Appendix: ViT Pseudocode . . . 18

5 Performer ReLU **19**
- 5.1 Model Design Choices . . . 19
- 5.2 Training and Optimization Process . . . 20
  - 5.2.1 Mixed Precision Training . . . 20
  - 5.2.2 Learning Rate Scheduling . . . 20
  - 5.2.3 Early Stopping . . . 20
  - 5.2.4 Cross-Validation Strategy . . . 20
- 5.3 Hyperparameter Tuning and Optimization . . . 20
- 5.4 Evaluation Metrics . . . 20
- 5.5 Results and Observations . . . 21
- 5.6 Appendix: Performer ReLU Pseudocode . . . 24

6 Alternative research avenues to Performer RELU **26**
6.1 Exploring ELU as an Alternative Activation . . . 26
6.1.1 Background and Motivation . . . 26
6.1.2 Implementation Changes . . . 26
6.1.3 Initial Experiment . . . 26
6.1.4 Parameter Adaptation . . . 27
6.1.5 Aggressive Parameter Modification . . . 28
6.1.6 Final Hyperparameter Tuning . . . 28
6.1.7 Comparative Analysis and Final Insights . . . 29
6.2 Exploring GELU as an Alternative Activation . . . 30
6.2.1 Background and Motivation . . . 30
6.2.2 Initial Experiment . . . 30
6.2.3 Initial Stability Improvements . . . 30
6.2.4 Manual Tuning with 5x Learning Rate Reduction . . . 31
6.2.5 Exploring Middle Ground: 8x Learning Rate Reduction . . . 32
6.2.6 Comparative Analysis and Final Insights . . . 33

7 Performer EXP **34**
7.1 Model Architecture: Exponential Kernel Function . . . 34
7.2 Hyperparameter Analysis . . . 34
7.3 Results and Observations . . . 36
7.4 Appendix: Performer Exp Pseudocode . . . 37

8 Performer-fθ Variant **39**
8.1 Overview . . . 39
8.2 Architecture . . . 39
8.3 First Steps: Learnable Mixture Kernel Implementation . . . 39
8.3.1 Motivation and Mathematical Foundation . . . 39
8.3.2 Implementation Details . . . 39
8.3.3 Results and Analysis . . . 40
8.3.4 Transition to Final Model . . . 40
8.4 Final Model of fθ . . . 40
8.5 Results and Observations . . . 40
8.6 Challenges . . . 41
8.7 Conclusion . . . 41

9 Overall Conclusion **42**
9.1 Table of Results . . . 42
9.2 Synthesis of Findings . . . 42
9.3 Limitations and Opportunities . . . 43
9.4 Final Insights and Future Directions . . . 43

1 Introduction

1.1 Context

Image classification has evolved significantly in deep learning, from traditional Convolutional Neural Networks (CNNs) to more sophisticated architectures. The introduction of Vision Transformers (ViT) marked a pivotal shift, demonstrating that architectures originally designed for natural language processing could be effectively adapted for computer vision tasks. However, the standard attention mechanism in transformers, while powerful, incurs significant computational costs with its quadratic complexity $O(n^2)$, limiting its practical applications in resource-constrained environments.

1.2 Problem Statement

The fundamental challenge lies in the trade-off between computational efficiency and model accuracy in transformer architectures. While Vision Transformers achieve high accuracy, their attention mechanisms are computationally expensive. This project explores efficient alternatives, particularly Performer models, which approximate attention using kernel-based methods.

1.3 Objectives

- Implement and evaluate three transformer architectures for image classification:
  - Standard Vision Transformer (ViT) with traditional attention
  - Performer with ReLU-based kernel (Performer-ReLU)
  - Performer with exponential kernel (Performer-exp)
- Conduct comprehensive performance analysis measuring:
  - Training time efficiency
  - Inference speed
  - Classification accuracy on evaluation tests
- Design and implement a novel Performer-fθ variant that:
  - Replaces the regular softmax attention kernel $K(q,k) = \exp(\frac{qk^\top}{\sqrt{d_{QK}}})$ with $K(q,k) = f(q)f(k)^\top$
  - Implements fθ as a learnable function
  - Aims to outperform both Performer-ReLU and Performer-exp variants

This investigation seeks to provide practical insights into the deployment of transformer architectures in computer vision, particularly focusing on scenarios where computational resources may be limited but high accuracy remains essential.

2 Theory and Architecture

2.1 Intuition Behind Attention

In traditional image processing techniques, such as convolutional neural networks (CNNs), the information flow between pixels is primarily localized. Each pixel interacts only with a small group of neighboring pixels through a sliding filter. While CNNs are highly effective in capturing fine-grained details (e.g., edges or textures), they often encounter difficulties in modeling relationships between distant regions within an image. For example, a CNN might accurately identify a dog's fur pattern in one part of an image but struggle to associate that pattern with the correct body shape if the relevant regions are spatially separated.

The **attention mechanism** addresses this challenge by allowing every pixel to *attend* to every other pixel in the image, regardless of spatial distance. Rather than relying exclusively on local context, attention integrates information from both nearby and remote pixels, thereby establishing a *global context* throughout the entire image. This capability is crucial for modeling long-range dependencies and high-level patterns.

Within the context of CIFAR-10 images (32x32 RGB images, containing objects like airplanes, animals, and ships) modeling these long-range relationships is particularly valuable:

- **Recognizing partially obscured objects:** If part of an object is hidden (e.g., only the wing of an airplane is visible), the model can leverage attention to infer the presence of the full object by integrating cues from distant parts of the image, such as the airplane's tail or fuselage.
- **Understanding interactions between distinct regions:** For a "cat" image, recognizing that certain distant features (e.g., the head and tail) belong to the same object requires a holistic perspective. Attention enables the model to associate these spatially separated features efficiently.
- **Resolving background clutter:** In images with complex backgrounds (e.g., a deer in a forest), attention helps the model focus on semantically relevant regions, like the deer's outline, while filtering out irrelevant background details.

By capturing and integrating long-range dependencies, attention enriches pixel representations with both local and global information. This is particularly beneficial for image classification tasks, such as CIFAR-10, where global understanding can significantly improve the recognition of various classes and object configurations.

2.2 Standard Attention Mechanism in Image Processing

The attention mechanism enables a model to focus on the most relevant parts of an input. When applied to image processing, it allows the model to capture spatial relationships by assigning appropriate weights to pixel regions that contribute to the classification task.

In CIFAR-10, each 32×32 RGB image is flattened into a sequence of $N = 3072$ elements (since $32 \times 32 \times 3 = 3072$). Each element in this sequence corresponds to one pixel's intensity values. However, flattening removes the inherent 2D structure of the image, making it less straightforward to detect spatially local patterns, such as edges or textures, because adjacent pixels in the original grid may appear distant in the sequence.

To mitigate this, positional encoding is introduced. By embedding explicit positional information into each pixel embedding, the Transformer can interpret these flattened sequences as if their original structure were intact. Positional encoding ensures that pixels originally close in the 2D layout have similar positional embeddings, supporting the recognition of local patterns as well as larger spatial relationships.

This positional encoding, combined with the attention mechanism, allows the model to efficiently capture both fine-grained local features and broad global patterns within the flattened image representation.

2.2.1 Embedding

Let $\mathbf{X} \in \mathbb{R}^{N \times d}$ represent the flattened sequence, where $N = 3072$ and d is the embedding dimension. Each pixel embedding is a d-dimensional vector integrating multiple sources of information:

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,d} \\ x_{2,1} & x_{2,2} & \dots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,d} \end{bmatrix}$$

Pixel Embedding Structure

Each pixel embedding includes several components to provide a more comprehensive pixel representation:

- **Color Intensity Information**: Raw RGB intensity values.
- **Positional Information**: Encodings that preserve the pixel's original spatial location in the 32×32 grid.
- **Learned Features**: Additional features discovered during training that capture image patterns and aid classification.

Hence, each pixel's embedding might look like:

$$\text{Embedding for pixel } i = \begin{bmatrix} \text{R}_i & \text{G}_i & \text{B}_i & \text{Position}_i & \text{Feature}_{i,1} & \text{Feature}_{i,2} & \dots & \text{Feature}_{i,d-6} \end{bmatrix}$$

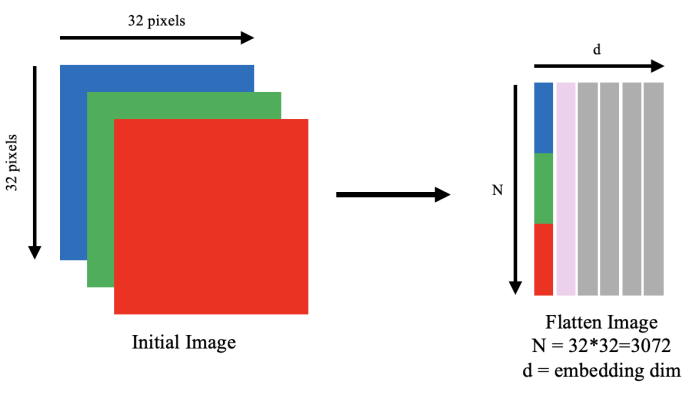


Figure 1: Image Embedding

By integrating color, position, and learned features, the embedding preserves both local detail and global context, ultimately aiding the model in interpreting flattened pixel sequences effectively.

2.2.2 Query, Key and Value

The attention mechanism is built on the concept of Query, Key, and Value matrices derived from the input embeddings:

- **Query (Q)**: Represents the "queries" each element (pixel) poses to the rest of the sequence. Each row encodes what a pixel seeks from other pixels.
- **Key (K)**: Represents the "content" or features of each element. Each row describes what information a pixel offers to others.
- **Value (V)**: Represents the actual information to be gathered. Each row is a vector of features that can be combined once relevancies (via queries and keys) are established.

These matrices are obtained by linearly projecting the input embeddings $\mathbf{X}$ through learnable weight matrices:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$

with $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d}$.

Through these learned projections, the model can discover specialized transformations that highlight relevant similarities and distinctions among pixels, thereby structuring the attention calculation effectively.

2.2.3 Self Attention

The core operation of attention—referred to as scaled dot-product attention—measures how much each query "attends" to each key. This allows each pixel to weigh the importance of every other pixel in the image. Mathematically, attention is defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

The $\frac{1}{\sqrt{d}}$ scaling factor ensures numerical stability and supports effective gradient-based training by keeping the magnitude of the dot products moderate.

Scaled Dot-Product Attention Mechanism

For a CIFAR-10 image flattened into $N = 3072$ pixels, attention computes how strongly each pixel relates to every other pixel:

1. **Raw Scores**: Compute $\mathbf{Q}\mathbf{K}^\top$, yielding an $N \times N$ matrix where each element measures the similarity between two pixels.
2. **Scaling**: Divide by $\sqrt{d}$ to stabilize training.
3. **Softmax**: Apply softmax row-wise to convert each row into a probability distribution over all pixels.

Thus, each pixel's attention distribution highlights which other pixels are most relevant to it. The final output $\mathbf{Y}$ is obtained by combining these attention weights with the value matrix $\mathbf{V}$:

$$\mathbf{Y} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

Each row of $\mathbf{Y}$ is a weighted sum of the entire set of pixel value vectors, integrating global context into each pixel's representation.

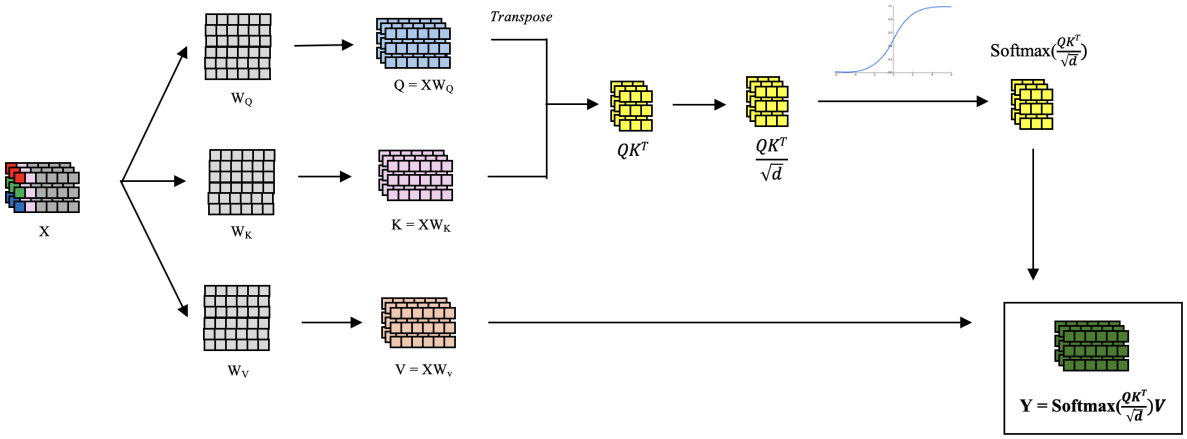


Figure 2: Attention Computation

The result is a refined embedding $\mathbf{Y}$ in which every pixel's vector contains not only its original local features but also relevant global information from other pixels. This global enrichment is critical for image classification tasks that benefit from understanding the entire scene rather than just local fragments.

2.3 Multi-Head Attention Mechanism in Image Processing

Multi-Head Attention (MHA) extends the standard attention mechanism by using multiple "heads," each of which learns to focus on different aspects of the input sequence. In image processing, MHA enables the model to capture a richer variety of relationships simultaneously. One head might focus on global composition, another on fine textures, and yet another on color contrasts, all operating in parallel.

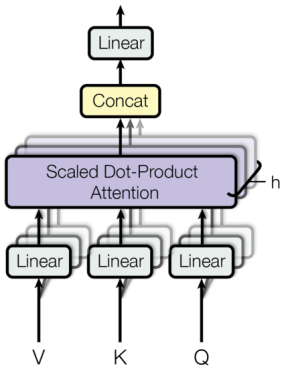


Figure 3: Multi-Head Attention diagram

2.3.1 Independent Attention Heads

In MHA, we create h independent attention heads. Each head computes its own $\mathbf{Q}^{(i)}, \mathbf{K}^{(i)}, \mathbf{V}^{(i)}$ using distinct learnable weight matrices $\mathbf{W}_Q^{(i)}, \mathbf{W}_K^{(i)}, \mathbf{W}_V^{(i)}$. If the original embedding dimension is d, each head operates on a subspace of dimension $d_h = d/h$:

$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_Q^{(i)}, \quad \mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_K^{(i)}, \quad \mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_V^{(i)}.$$

By assigning different projections to each head, the model encourages each one to discover and focus on different relationships in the image.

2.3.2 Attention Calculation for Each Head

Each head computes attention scores similarly to the single-head scenario:

$$\text{Attention}^{(i)}(\mathbf{Q}^{(i)}, \mathbf{K}^{(i)}, \mathbf{V}^{(i)}) = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^\top}{\sqrt{d_h}}\right)\mathbf{V}^{(i)}.$$

This parallel computation allows one head to capture long-range dependencies, another to focus on localized edges, and another to highlight color-based patterns, all contributing complementary insights about the image.

2.3.3 Concatenating and Projecting Heads

After each head computes its attention, we concatenate their outputs and project them back to dimension d:

$$\text{MultiHead}(\mathbf{X}) = \mathbf{W}_O \begin{bmatrix} \text{Attention}^{(1)}(\mathbf{Q}^{(1)}, \mathbf{K}^{(1)}, \mathbf{V}^{(1)}) \\ \text{Attention}^{(2)}(\mathbf{Q}^{(2)}, \mathbf{K}^{(2)}, \mathbf{V}^{(2)}) \\ \vdots \\ \text{Attention}^{(h)}(\mathbf{Q}^{(h)}, \mathbf{K}^{(h)}, \mathbf{V}^{(h)}) \end{bmatrix}$$

Here, $\mathbf{W}_O \in \mathbb{R}^{(h \cdot d_h) \times d}$ is a learnable projection matrix. This step fuses the insights gathered by each head into a comprehensive representation, enabling the model to leverage multiple perspectives efficiently.

2.3.4 Advantages of Multi-Head Attention

MHA's key advantage is its ability to simultaneously capture different types of patterns from the same image. The combination of various attention heads leads to richer, more discriminative representations. In CIFAR-10, this translates to improved accuracy in recognizing objects that may differ in subtle but crucial ways.

2.4 Transformer Architecture

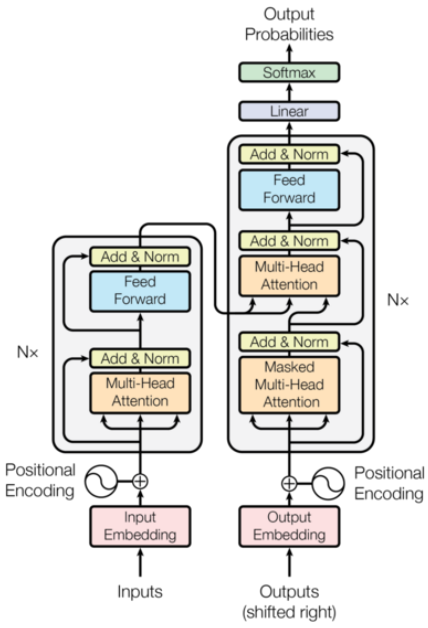


Figure 4: Transformer Diagram

The Transformer architecture leverages attention mechanisms to process sequences and capture long-range dependencies effectively. The following components are central to its design:

2.4.1 Input Embedding Layer

As discussed, we first represent each pixel as a d-dimensional embedding that integrates color, positional, and learned features. These embeddings then serve as the input to the Transformer.

2.4.2 Multi-Head Self-Attention Mechanism

Already described in Section 2.3, the MHA module enables the model to attend to multiple aspects of the input simultaneously, increasing representational richness.

2.4.3 Feedforward Neural Network (FFN)

After attention, each position's embedding is transformed by a Feedforward Neural Network (FFN):

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

This position-wise transformation introduces non-linearity and enhances the model's ability to capture complex relationships. While the attention layer captures how embeddings relate to each other, the FFN refines each embedding individually.

2.4.4 Residual Connections and Layer Normalization

To maintain stable gradients and propagate information effectively, Transformers employ residual connections and layer normalization around both the attention and FFN sub-layers:

$$\mathbf{y} = \text{LayerNorm}(\mathbf{x} + \text{SubLayer}(\mathbf{x})),$$

where SubLayer could be either the MHA block or the FFN. Residual connections prevent information loss, and layer normalization stabilizes training by normalizing each position's output.

Stacking Layers

By stacking multiple layers composed of MHA and FFN blocks, the Transformer learns hierarchical representations. Each subsequent layer refines the representation, allowing the model to progressively capture more intricate patterns within the image data.

Classification Head (Output Layer)

Finally, a fully connected layer followed by softmax transforms the final sequence of embeddings into class probabilities. This head is shared among variants (ViT, Performer-exp, Performer-ReLU), but each processes a differently embedded input sequence. The cross-entropy loss function guides parameter updates, optimizing model accuracy.

2.5 Performers, Fast Attention Mechanisms and Kernel-based Approximation

2.5.1 Introduction to Efficient Attention Mechanisms

The standard attention mechanism incurs $O(N^2)$ complexity with respect to the sequence length N, posing challenges for very large images or real-time applications. To address these limitations, Choromanski et al. introduced **Fast Attention via Orthogonal Random Features (FAVOR+)**. This innovative approach redefines the traditional attention computation by leveraging kernel-based approximations, reducing complexity to $O(N)$. FAVOR+ enables the efficient handling of longer sequences without substantially compromising accuracy, marking a significant advancement in making attention mechanisms scalable and computationally efficient.

2.5.2 FAVOR+: A Kernel-based Approximation

Performers re-express the softmax attention in terms of a kernel function, enabling a linear-time approximation:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \approx \phi(\mathbf{Q})(\phi(\mathbf{K})^\top \mathbf{V}).$$

Here, $\phi(\cdot)$ is a feature map that transforms queries and keys into a space where their dot product approximates the exponential similarity.

2.5.3 Key Components of Performers

- **Random Feature Maps**: Performers (e.g., **Performer-exp**) use positive random features to approximate the exponential kernel. This ensures stable, scalable computation.
- **Linear Complexity**: By reframing attention calculations as products of feature-mapped queries and keys, Performers achieve linear complexity, significantly reducing memory and computational costs.
- **ReLU-based Kernels**: The **Performer-ReLU** variant replaces the exponential kernel with a ReLU-based kernel:

  $$\phi(x) = \text{ReLU}(Wx + b).$$

  This alternative is computationally efficient and suitable for resource-limited scenarios.
- **Deterministic Kernels**: For improved consistency, deterministic mappings (such as sine and cosine embeddings) can be used, reducing variability and improving robustness.

By employing these kernel-based approximations and transformations, Performers make attention mechanisms scalable and more broadly applicable, extending their practicality to a wider range of image processing tasks, including detailed classification on datasets like CIFAR-10.

3 Experimental Setup

3.1 Our dataset

For this project, we used the **CIFAR-10** dataset, a widely recognized benchmark in computer vision. CIFAR-10 contains 60,000 color images of 32×32 pixels, divided into 10 distinct classes, each representing a specific object category. CIFAR-10 was selected due to its balance of complexity and computational feasibility, allowing us to evaluate speed-accuracy tradeoffs across various Transformer architectures effectively. The dataset's manageable size and 10-class structure make it well-suited for efficient experimentation without requiring the extensive computational resources needed for larger datasets like ImageNet.

Figure 5: The ten classes of CIFAR-10

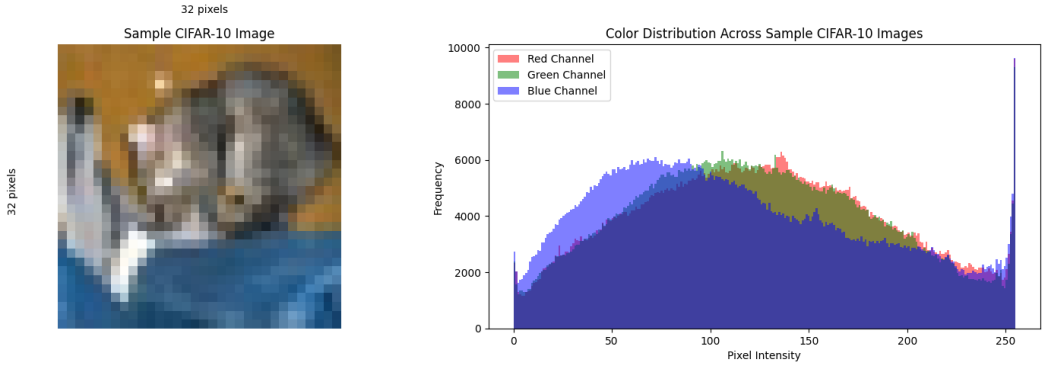
3.2 Data Transformations

Normalization

To standardize inputs and improve model convergence, each color channel (Red, Green, and Blue) in the CIFAR-10 images was normalized. Normalization centers pixel intensities around zero and reduces variance, calculated for each pixel intensity x in channel c as:

$$x_{\text{normalized}} = \frac{x - \mu_c}{\sigma_c}$$

where μ_c and σ_c are the mean and standard deviation of channel c over the dataset. This normalization step is essential in deep learning, as it prevents large gradients and ensures stable, efficient model convergence. By aligning the color distributions across images, normalization also allows the model to learn patterns more effectively.



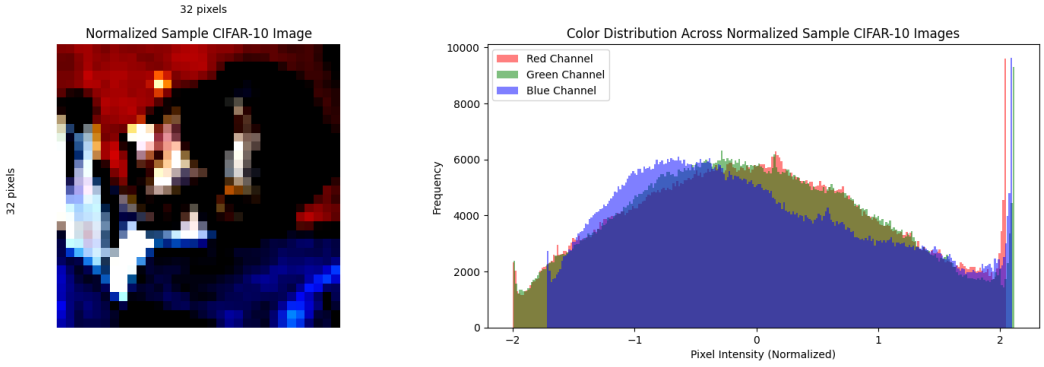


Figure 6: Normalized CIFAR-10 Sample Image

Figure 6 illustrates a normalized CIFAR-10 image, where each channel is centered around zero with unit variance, reducing disparities in intensity across channels and balancing each color's contribution during training.

Data Augmentation: Random Cropping and Horizontal Flip

To enhance the diversity of the training set and improve model generalization, we applied two data augmentation techniques:

- **Random Cropping with Padding (32x32, padding=4)**: Adds a padding of 4 pixels around the image, followed by a random crop back to 32x32 pixels. This slight positional variation helps the model become invariant to small positional changes.
- **Random Horizontal Flip**: With a probability of 0.5, flips the image horizontally. This augmentation allows the model to recognize objects in both left and right orientations, improving robustness.

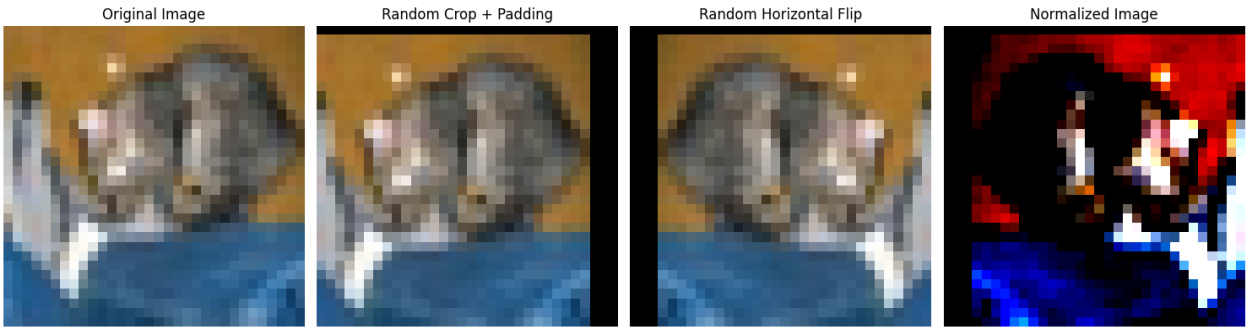


Figure 7: Step-by-Step Transformations on CIFAR-10 Sample Image

Figure 7 shows the transformations:

- The **Original Image** shows the starting point.
- **Random Crop + Padding** adds positional variance.
- **Random Horizontal Flip** mirrors the image, enhancing generalization.
- The **Normalized Image** is ready for training with balanced pixel distributions.

These augmentations increase the model's robustness by expanding the variety of training data without additional labels, especially valuable for smaller datasets like CIFAR-10.

Flattening and Preparing Data for the Performer Model

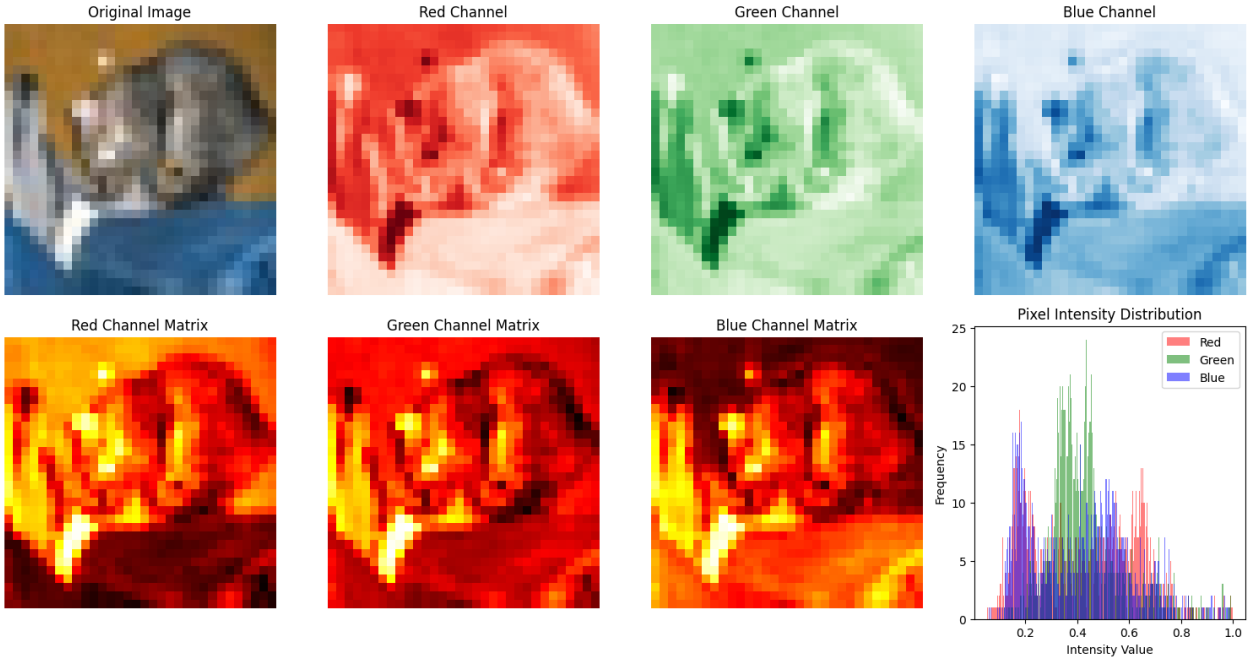


Figure 8: Decomposition of CIFAR-10 Image into RGB Channels and Pixel Intensity Distribution

Figure 8 illustrates the breakdown of a CIFAR-10 sample into its Red, Green, and Blue (RGB) channels, each represented as a 32×32 matrix. For instance, the Red channel matrix could look like:

$$\begin{bmatrix} \text{Red}_{(1,1)} & \text{Red}_{(1,2)} & \cdots & \text{Red}_{(1,32)} \\ \text{Red}_{(2,1)} & \text{Red}_{(2,2)} & \cdots & \text{Red}_{(2,32)} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Red}_{(32,1)} & \text{Red}_{(32,2)} & \cdots & \text{Red}_{(32,32)} \end{bmatrix}$$

Each pixel is labeled with its positional index (i, j) to represent its location within the 32×32 grid. The same positional indexing applies for the Green and Blue channels. Each of these matrices is then flattened row by row to form a single 3072×1 vector that combines all three color channels:

$$\begin{bmatrix} \text{Red}_{(1,1)} \\ \text{Red}_{(1,2)} \\ \vdots \\ \text{Blue}_{(32,32)} \end{bmatrix}_{3072 \times 1}$$

To retain spatial information after flattening, we apply a positional encoding that assigns each pixel a unique embedding based on its original location (i, j). For instance, $\text{Red}_{(1,1)}$, $\text{Green}_{(1,1)}$, and $\text{Blue}_{(1,1)}$ will share the same positional encoding, distinguishing them from $\text{Red}_{(1,2)}$ and others.

This positional encoding allows the Performer model to interpret the flattened input as if it retains the 2D spatial structure of the image. The final 3072×1 vector, augmented with these positional encodings, is then fed into the Performer model, enabling it to capture the spatial relationships necessary for accurate image classification.

3.3 Dataset Splitting

The CIFAR-10 dataset (60,000 images) was split into training, validation, and test sets as follows:

- **Training Set**: Comprising 80% of the dataset, approximately 48,000 images were used for model training. This set was augmented with random cropping and horizontal flipping to enhance model generalization.
- **Validation Set**: Consisting of 20% of the dataset, approximately 12,000 images were reserved for model validation. This set was derived from the training data using stratified k-fold cross-validation (5 folds) to ensure balanced class representation and to tune hyperparameters effectively.

- **Test Set**: The remaining 10,000 images (the original CIFAR-10 test dataset) were used for final model evaluation. These images were not involved in training or validation to ensure an unbiased assessment of model performance.

This splitting strategy ensures sufficient data for training while providing robust mechanisms for hyperparameter tuning and performance evaluation.Training and validation sets benefit from the random cropping and flipping augmentations, enhancing model robustness and reducing overfitting.

Due to time constraints, not all subgroups worked with validation sets. These were specifically utilized by the Performer RELU, ELU, GELU, and fθ variants. Other groups directly optimized their hyperparameters on the test set, which may bias the performance evaluation of these models compared to those using a separate validation set.

3.4 Hardware

The experiments were conducted on Google Colab using an NVIDIA A100 GPU with the following specifications:

- **Model**: NVIDIA A100 Tensor Core GPU
- **Architecture**: Ampere
- **GPU Memory**: 40 GB
- **Cores**: 6912 CUDA cores and 432 Tensor cores
- **Performance**: Up to 312 TFLOPS for deep learning tasks
- **Operating System**: Google Colab Environment

This configuration offers high computational performance, making it ideal for running transformer-based architectures efficiently. The A100 GPU's ability to handle large-scale tensor computations allowed for faster training and inference compared to running on CPU-only hardware, significantly accelerating the experimentation process.

3.5 Common Hyper-parameters for all models

The performance and behavior of a machine learning model are strongly influenced by its hyperparameters. Below, we detail the impact and role of each key hyperparameter used in our experiments:

- **Learning Rate**: The learning rate controls the step size of the optimizer during weight updates. A smaller learning rate allows for more fine-grained adjustments but can lead to slower convergence, while a larger learning rate accelerates training but risks overshooting the optimal solution or destabilizing the training process. This parameter is critical during gradient descent and directly influences how quickly the model learns.
- **Batch Size**: The batch size determines the number of training examples processed simultaneously before updating the model. Smaller batch sizes introduce more noise in the gradient estimates, which can help escape local minima but slow down training. Larger batch sizes provide more stable gradient updates, enabling faster convergence but at the risk of getting trapped in suboptimal solutions. Batch size also affects memory usage and computational efficiency.
- **Optimizer**: The choice of optimizer impacts how gradients are used to update the weights. For example:
  - **Adam**: Combines the benefits of momentum and adaptive learning rates, often leading to faster convergence.
  - **SGD (Stochastic Gradient Descent)**: A simpler optimizer that performs well with appropriate learning rate schedules but may require careful tuning.
  - **AdamW**: A variant of Adam that includes weight decay, improving performance on regularized models.
- **Dropout Rate**: Dropout is a regularization technique that prevents overfitting by randomly setting a fraction of the neurons to zero during training. A higher dropout rate introduces stronger regularization but may hinder the model's ability to learn intricate patterns, while a lower dropout rate might lead to overfitting on the training data.
- **Depth**: The number of layers in the model defines its capacity to learn hierarchical features. Deeper models can represent more complex functions but are more prone to overfitting, vanishing/exploding gradients, and longer training times. Balancing depth with the dataset size and regularization methods is crucial.

- **Hidden Dimension**: The hidden dimension determines the size of the internal feature vectors. Larger hidden dimensions allow the model to capture richer representations of the input data but increase computational and memory requirements. Small dimensions, while computationally efficient, might limit the model's ability to generalize.
- **Attention Heads**: In Transformer architectures, the number of attention heads controls the degree of parallelism in the attention mechanism. More heads allow the model to focus on different parts of the input simultaneously, capturing diverse patterns and relationships. However, increasing the number of heads also raises computational costs and the risk of overfitting.

These parameters were common to all the models we developed; however, we tested different values for each of them to evaluate their impact on performance. Beyond these shared hyperparameters, model-specific parameters were explored individually, reflecting the distinct focus and preferences of each member of the team. This approach allowed us to test tailored configurations for each architecture while maintaining a shared framework for comparison.

3.6 Common Hyper-parameters for all models

Prior to comparative analysis, each model variant was individually fine-tuned to achieve its best performance. The process included:

- Extensive hyperparameter optimization tailored to each architecture
- A consistent approach using grid search and validation sets
- Custom tuning to address the specific characteristics of each activation function
- Multiple training iterations to verify result stability

Comparative analyses were conducted only after optimizing each model's configuration. This method ensures a fair evaluation of the architectures while accounting for their unique strengths and requirements.

4 Vision Transformer

The Vision Transformer (ViT) is a state-of-the-art model architecture that applies Transformer mechanisms, originally designed for natural language processing, to computer vision tasks. Unlike convolutional neural networks (CNNs), ViT divides images into fixed-size patches and processes these patches as sequences, leveraging the global self-attention mechanism for image understanding. ViT achieves competitive accuracy in image classification tasks while offering scalability for larger datasets and computational efficiency.

4.1 Model Setup

We evaluated the Vision Transformer (ViT) and its efficiency in image classification leveraging a hyperparameter optimization framework. The effort to test different models aims to assess speed-accuracy trade-offs in different Transformer-based architectures. Multiple hyperparameters including batch size (specific choices detailed in Section 5.1.2), were dynamically optimized in conjunction with other parameters to explore interactions and how they pertain to overall accuracy and speed.

We applied the same processing and loading steps described in Section 4.2, which entailed multiple steps including horizontal flipping and random cropping to ensure diversity of images in the training process.

These preprocessing steps and optimization of hyperparameters collectively enhance the reliability of the experimental setup, providing a robust foundation for assessing the Vision Transformer's performance.

4.1.1 Model Architecture

The model was trained from scratch employing architecture described in Section 3 (specifically, subsection 3.4).

4.1.2 Hyperparameter Optimization

In the Vision Transformer (ViT) architecture, several critical hyperparameters were utilized to optimize the model's performance during training and evaluation. These hyperparameters are as follows:

Hyperparameter	**Tested Values**
Learning Rate	1e-03, 5e-04
Optimizer	Adam, SGD
Dropout Rate	0.1, 0.3
Depth	6, 8
Attention Heads	4, 8
Patch Size	4, 8
Embedding Dimension	128, 256
MLP Hidden Dimension	512, 1024

Compared to other models, we also explored the following additional hyperparameters:

- **Patch Size**: Defines the size of the patches the input image is divided into. Smaller patches allow for capturing finer details but increase computational complexity.
- **Embedding Dimension**: Determines the size of the feature vector representing each patch.
- **MLP Hidden Dimension**: Specifies the size of the hidden layer in the feedforward network of each Transformer encoder.

These hyperparameters were tuned extensively using a grid-search approach to identify the optimal configuration that maximized validation accuracy while minimizing computational overhead. The final model configuration was selected based on the best-performing hyperparameter combination on the validation set.

4.2 Hyperparameter Tuning and Optimization

The training process involved exploring different configurations to identify the optimal model performance. Experiments were conducted for 10 epochs, with final evaluations on the model running for 30+ epochs to gather insights over longer iterations. Cross-entropy error was utilized to calculate prediction errors, driving the optimization procedures through backpropagation.

Pruning techniques were employed to terminate low-performing trials early, streamlining the hyperparameter search process. This approach accelerated the discovery of optimal settings, particularly focusing on batch size and learning rate adjustments to strike the best balance between accuracy and computational efficiency. Additionally, computational trade-offs were carefully analyzed to ensure the model's performance was robust without incurring excessive resource usage.

The results of the hyperparameter tuning process (to be discussed in Section 5.4) yielded much stronger results for a Patch Size of 4. Due to these promising results, we employed a Patch Size of 4 as a base parameter for all models in our study, including the Performers, due to the increased opportunity for granularity in training and testing.

4.3 Evaluation Metrics

The evaluation focused on key metrics, including average loss, accuracy, and inference time across the epochs. These metrics provided a clear understanding of the model's performance progression over the training period.

Particular emphasis was placed on balancing accuracy and computational cost. Inference time was measured to assess the computational efficiency of the model and directly compared to alternative configurations. This comprehensive analysis allowed for a detailed evaluation of the trade-offs, ensuring both high accuracy and manageable computational requirements.

4.4 Results And Observations

4.4.1 Classification Accuracy

The Vision Transformer performed remarkably well on CIFAR-10, with and without pretrained weights, achieving outstanding accuracy in recognizing complex spatial relationships. When pretrained weights were used, the model achieved slightly better accuracy. However, the gap gradually closed, and the accuracy of non-pretrained models improved with sufficient fine-tuning. Despite this, high accuracy was achieved with the aid of substantial computational resources and the ViT full attention mechanism. Notably, the softmax-based attention technique of ViT yielded better accuracy initially, but as the model was trained further, the accuracy of the Performer models came closer to that of ViT, with minimal efficiency trade-offs. In direct comparisons, the non-pretrained ViT models demonstrated a comparable error margin to the pretrained versions, providing confidence in their utility in environments with fewer resources.

Training time, inference time, and accuracy of the trained model with tuned hyperparameters:

Hyperparameter	**Best Values**
Learning Rate	5e-04
Optimizer	Adam
Dropout Rate	0.1
Depth	8
Attention Heads	8
Patch Size	4
Embedding Dimension	256
MLP Hidden Dimension	512

was determined to be: 56 seconds/epoch (over 31 epochs), 5 seconds, and 84.79%, respectively.

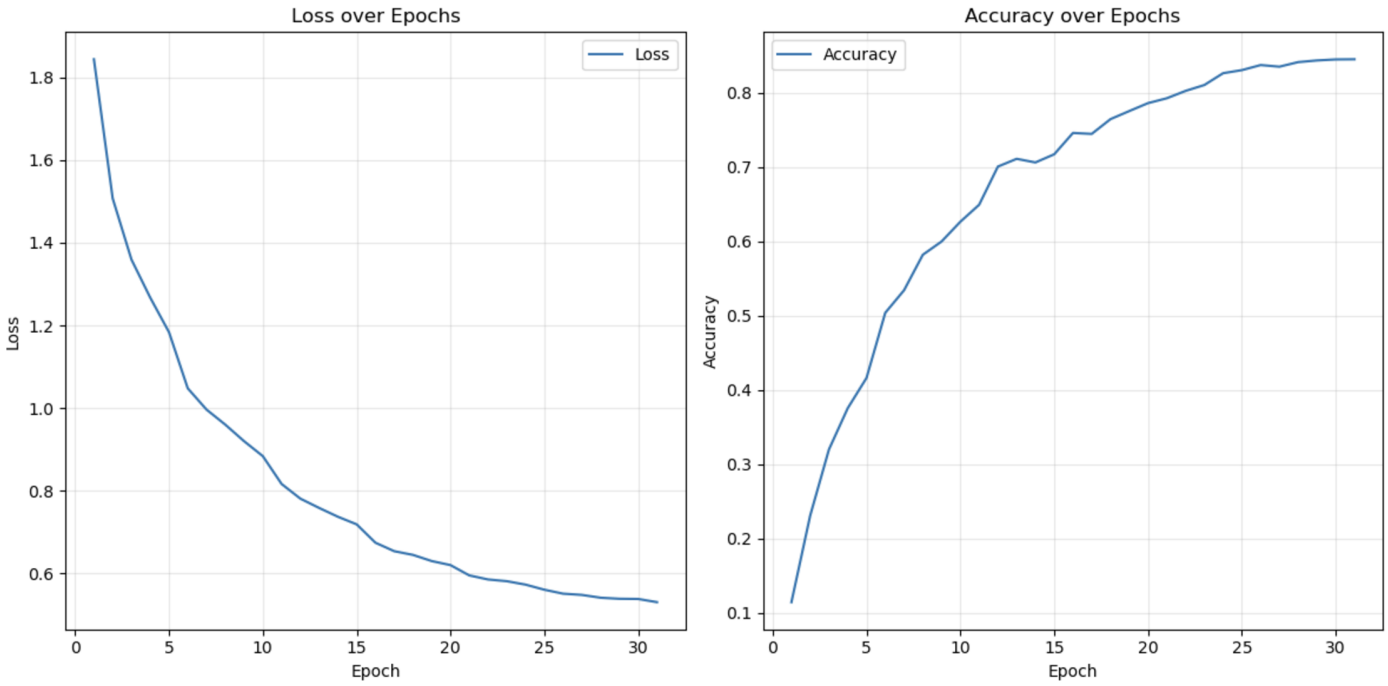


Figure 9: Loss and accuracy for ViT (31 epochs)

Figure 9 shows the loss and accuracy when evaluating the test data using the Vision Transformer model. The accuracy converges to a roughly horizontal line where training was stopped at 31 epochs, therefore attaining the best accuracy without overfitting. Additionally, loss decreases greatly at the start over the first few epochs, and gradually tapers off similarly to the accuracy, though the accuracy convergence served more strongly as the stopping criteria.

4.5 Appendix: ViT Pseudocode

Algorithm 1 Vision Transformer Layer Pseudocode

Inputs:

- Image size img_size
- Patch size patch_size
- Input channels in_channels
- Embedding dimension dim
- Number of Transformer layers depth
- Number of attention heads heads
- MLP hidden dimension mlp_dim
- Dropout rate dropout
- Number of classes num_classes

Compute number of patches: num_patches $\leftarrow$ (img_size/patch_size)^2

Initialize patch embedding: patch_embedding $\leftarrow$ Conv2D(in_channels $\rightarrow$ dim, kernel size = patch_size, stride = patch_size)

Initialize class token: cls_token $\leftarrow$ learnable parameter of shape $(1, 1, \text{dim})$

Initialize positional embedding: pos_embedding $\leftarrow$ learnable parameter of shape $(1, \text{num\_patches}+1, \text{dim})$

Define a stack of depth Transformer encoder layers:

- Each layer consists of multi-head attention with heads, feedforward layers with mlp_dim, and dropout dropout.

Define MLP head:

- LayerNorm(dim) $\rightarrow$ Linear(dim $\rightarrow$ num_classes)

function FORWARD(Input x)

    Compute batch size: batch_size $\leftarrow$ size$(x, 0)$

    Compute patch embeddings: $x \leftarrow$ patch_embedding(x)

    Flatten and transpose: $x \leftarrow$ flatten$(x, \text{start} = 2)$, $x \leftarrow$ transpose$(x, 1, 2)$

    Add class token:

- cls_tokens $\leftarrow$ expand(cls_token, batch_size, -1, -1)
- $x \leftarrow$ concatenate(cls_tokens, x, dim = 1)

    Add positional embedding: $x \leftarrow x +$ pos_embedding

    for all Transformer layer layer in Transformer stack **do**

        $x \leftarrow$ layer(x)

    end for

    Extract class token output: cls_output $\leftarrow x[:, 0]$

    Compute final output: output $\leftarrow$ mlp_head(cls_output)

    return output

end function

5 Performer ReLU

After exploring a Transformer-based architecture such as ViT, which focuses on overcoming the quadratic complexity of traditional attention, we turn to the Performer ReLU model. This approach achieves linear-time complexity in attention computations. Using a ReLU-based kernel function, the Performer ReLU model addresses the computational bottlenecks of standard Transformers by substituting the exact attention computation with an efficient approximation. Consequently, the complexity is reduced to $O(N)$ instead of the typical $O(N^2)$, making the model suitable for processing long sequences or high-resolution data, including CIFAR-10 images.

The underlying idea is to map input vectors into a feature space using $\phi(x) = \text{ReLU}(Wx + b)$, where W and b are random orthogonal weights and biases. By employing this kernel-based feature map, the Performer ReLU preserves global dependency modeling capabilities while ensuring efficient computation of attention scores. Thus, it represents a natural next step following the previously studied ViT model, striving to balance complexity reduction with strong performance.

5.1 Model Design Choices

The Performer ReLU architecture encompasses several key design decisions and hyperparameter settings that aim to optimize performance while maintaining efficiency. Below, we detail each component and the associated choices made during development and experimentation.

Kernel Approximation with FAVOR+

The FAVOR+ mechanism (Fast Attention Via Orthogonal Random features) is employed here to approximate the self-attention mechanism efficiently:

- A ReLU-based kernel function $\phi(x) = \text{ReLU}(Wx + b)$, with W and b drawn from orthogonal random matrices, is utilized.
- A small regularization term ($1e{-}6$) ensures numerical stability in the calculations.
- The number of random features (`nb_features`) was explored within the range of 128 to 512, with 256 as a common choice.

Convolutional Backbone

To extract meaningful visual features from the CIFAR-10 images prior to attention:

- Three convolutional layers with increasing channel sizes (64, 128, 256) were employed.
- Each convolutional layer was followed by Batch Normalization and a ReLU activation function to stabilize and enhance feature extraction.
- A MaxPooling layer was introduced to downsample the spatial dimensions of the feature maps effectively.

Embedding Layer

The extracted convolutional features are then projected into a space suitable for the attention mechanism:

- A fully connected layer maps the flattened convolutional output into a hidden dimension (`dim`) ranging from 256 to 960.
- Dropout was applied at rates between 0.0 and 0.5 to mitigate overfitting.

Self-Attention Layers

The core Performer layers implement the linear-time self-attention mechanism:

- Multiple attention heads (`n_heads`) between 8 and 16 were considered to capture diverse representation subspaces.
- Each layer integrates residual connections and Layer Normalization to ensure stable training.
- The depth of the Performer stack (`depth`) varied from 1 to 4 layers, with performance evaluated for each configuration.

Classification Head

Finally, for prediction on CIFAR-10:

- A fully connected layer maps the final embedding to 10 output classes.
- A Layer Normalization step before this classification layer further stabilizes training.

5.2 Training and Optimization Process

5.2.1 Mixed Precision Training

To efficiently leverage GPU capabilities:

- Mixed precision training was enabled using `torch.cuda.amp.GradScaler`, combining `float16` and `float32` operations to reduce memory usage and accelerate computations while maintaining numerical stability.

5.2.2 Learning Rate Scheduling

To achieve steady convergence, two scheduling strategies were employed, conditioned on the chosen optimizer:

- `StepLR` for Adam: Reduces the learning rate by a factor of 0.5 every 5 epochs.
- `CosineAnnealingLR` for SGD: Smoothly decays the learning rate to zero via a cosine schedule, helping the model converge steadily.

5.2.3 Early Stopping

Early stopping was introduced to avoid unnecessary overfitting and save computational time:

- If the validation loss failed to improve after 10 consecutive epochs, training was halted, preventing degradation in model generalization.

5.2.4 Cross-Validation Strategy

To reduce biases and obtain reliable performance estimates:

- A stratified k-fold cross-validation approach was used, ensuring that each fold preserved the overall class distribution of CIFAR-10.

5.3 Hyperparameter Tuning and Optimization

Hyperparameter tuning was conducted in two phases, following the approach taken for the ViT model:

Hyperparameter	**Tested Values**
Learning Rate	[1e-05; 1e-02]
Batch Size	64, 128, 256
Optimizer	Adam, SGD
Dropout Rate	[0; 0.5]
Depth	1, 2, 3, 4
Hidden Dimensions	576, 768, 960
Attention Heads	12, 16

5.4 Evaluation Metrics

The quality of the Performer ReLU model was assessed by:

- Classification accuracy on the CIFAR-10 test set.
- Training and validation loss curves to track overfitting or underfitting.
- Inference time to confirm computational efficiency.

5.5 Results and Observations

The table below summarizes the optimal hyperparameters found via the tuning process:

Hyperparameter	Best Values
Learning Rate	1.7e-03
Batch Size	128
Optimizer	SGD
Dropout Rate	0.1
Depth	4
Hidden Dimensions	960
Attention Heads	16

Key findings include:

- **Overall Accuracy:** The model achieved an accuracy of **87.39%** on CIFAR-10.
- **Class-wise Performance:**
  - Strong classes (*automobile*, *ship*, *truck*) exhibited high F1-scores ranging from 93% to 95%. These classes are likely easier to classify due to their distinct features and lower visual overlap with other categories.
  - More challenging classes (*cat*, *dog*), characterized by high visual similarity and overlapping features, yielded lower F1-scores between 74% and 80%. The confusion observed here may arise from shared patterns, such as fur textures or similar body shapes, which make differentiating these classes inherently more complex.
- **Confusion Patterns:** The confusion matrix (Figure 10) highlights notable misclassifications, particularly between visually similar categories, such as *cat* vs. *dog* and *bird* vs. *airplane*. This suggests that the model struggles with classes sharing overlapping visual features or backgrounds. However, a deeper analysis of these misclassifications, including the specific instances causing confusion, would provide further insights into the model's limitations.

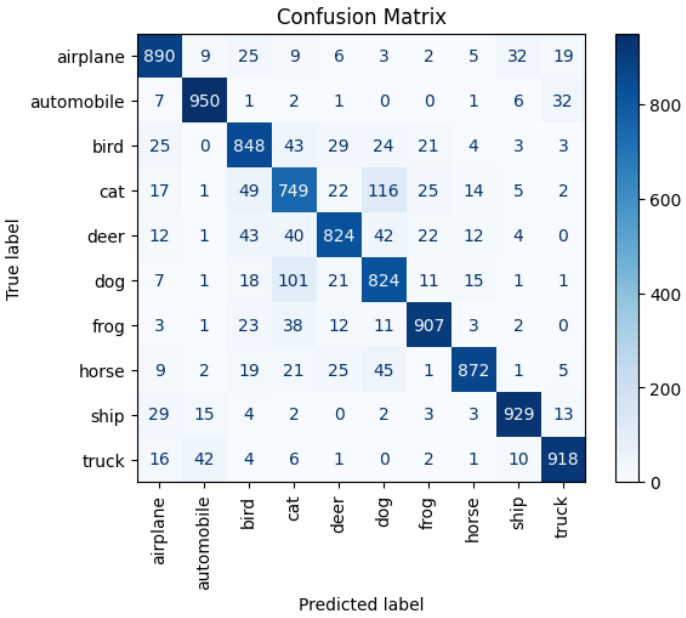


Figure 10: Confusion Matrix - Performer ReLU

Class	Precision (%)	Recall (%)	F1-Score (%)
Airplane	88	89	88
Automobile	93	95	94
Bird	82	85	83
Cat	74	75	74
Deer	88	82	85
Dog	77	82	80
Frog	91	91	91
Horse	94	87	90
Ship	94	93	93
Truck	92	92	92
Overall	**87**	**87**	**87**

Additional Insights: The observed gaps in performance, particularly for *cat* and *dog*, indicate that the model could benefit from additional augmentation techniques or loss weighting to better handle visually ambiguous cases. Furthermore, analyzing the underlying features learned by the model for these classes (e.g., attention maps) might reveal systematic biases or overlooked distinguishing patterns.

Training and Validation Trends

Monitoring the training dynamics is crucial:

- **Loss Curves:** Training and validation losses decrease steadily, with the validation loss stabilizing after approximately 50 epochs, indicating appropriate convergence.
- **Accuracy Curves:** Both training and validation accuracies improve consistently. The minimal gap between them suggests only slight overfitting.

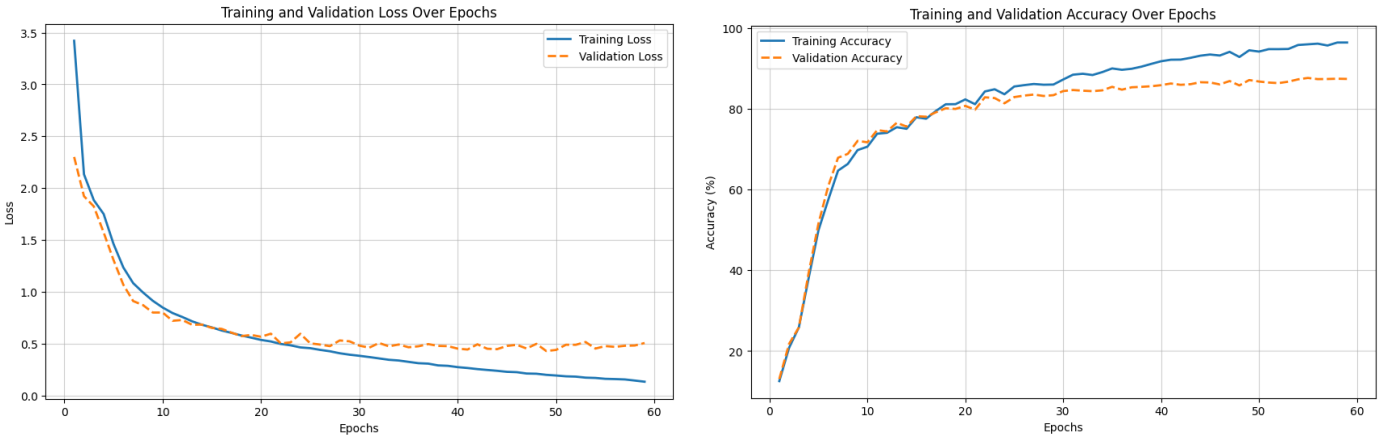


Figure 11: Loss and Accuracy Curves - Performer ReLU (59 epochs)

Class Accuracy Distribution

As depicted in Figure 12, the performance is notably stronger on classes representing vehicles (*automobile*, *ship*, *truck*) than on classes with higher similarity (e.g., *cat*, *dog*):

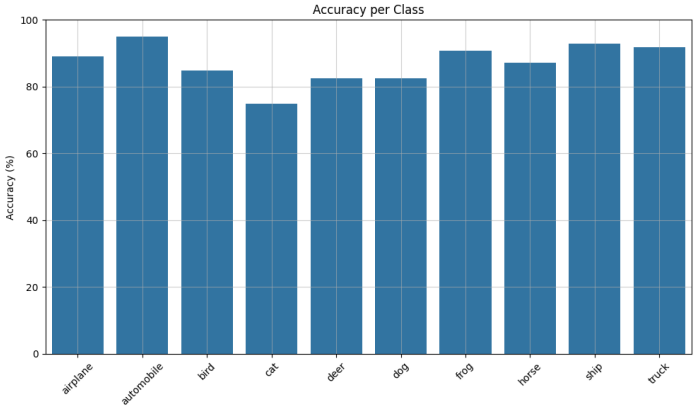


Figure 12: Accuracy Per Class - Performer ReLU

Inference and Training Times

- **Training Time:** Each epoch took about 31 seconds, culminating in roughly 31 minutes for 59 epochs. The embedding dimension (960), number of attention heads (16), and four Performer layers contributed to this runtime.
- **Inference Time:** Test inference on the entire CIFAR-10 dataset took about 3 seconds, demonstrating the model's efficiency in real-time or large-scale inference scenarios.

Strengths

- **Scalability:** The Performer ReLU's linear attention mechanism manages large inputs efficiently.
- **Balanced Performance:** With macro and weighted averages at 87% accuracy, the model maintains balanced performance across classes.
- **Stable Training:** Smooth convergence in both loss and accuracy indicates a robust and well-chosen optimization strategy.

Weaknesses and Critiques

- **Inter-class Confusion:** Classes with high visual similarity, such as *cat* vs. *dog*, exhibit higher misclassification rates.
- **Computational Cost:** The large embedding size (960) and numerous attention heads (16) increase the computational footprint.
- **Regularization:** Slight overfitting suggests that more aggressive regularization (e.g., higher dropout or stronger data augmentation) could further improve generalization.

5.6 Appendix: Performer ReLU Pseudocode

Algorithm 2 PerformerLayer Pseudocode

Inputs: Dimension dim, Number of heads n_heads, Features nb_features, Dropout rate dropout
Compute head_dim $\leftarrow \frac{\text{dim}}{\text{n\_heads}}$
Initialize linear layers: $W_Q, W_K, W_V, W_{\text{out}}$
Initialize dropout layer and layer normalization layers
Generate random projection matrix projection_matrix using orthogonal initialization
function GENERALIZED_KERNEL(data)
    Scale data $\leftarrow \frac{\text{data}}{\sqrt{\text{head\_dim}}}$
    Project data: data_dash $\leftarrow$ einsum$('bhsd, fd \rightarrow bhsf', \text{data}, \text{projection_matrix})$
    Apply ReLU and stabilization: data_prime $\leftarrow$ ReLU(data_dash) $+ 1e{-}6$
    return data_prime
end function
function FORWARD(Input x)
    residual $\leftarrow x$
    $x \leftarrow \text{LayerNorm}_1(x)$
    $Q \leftarrow W_Q(x),\ K \leftarrow W_K(x),\ V \leftarrow W_V(x)$
    Reshape Q, K, V for multi-head attention
    $Q' \leftarrow$ generalized_kernel(Q), $K' \leftarrow$ generalized_kernel(K)
    KV $\leftarrow$ einsum$('bhsf, bhsd \rightarrow bhfd', K', V)$
    K_sum $\leftarrow \text{sum}(K', \text{dim} = 2)$
    $Z \leftarrow \frac{1}{\text{einsum}('bhsf, bhf \rightarrow bhs', Q', \text{K\_sum}) + 1e{-}6}$
    Out $\leftarrow$ einsum$('bhsf, bhfd \rightarrow bhsd', Q', \text{KV}) \times Z$
    Reshape and combine heads: Out $\leftarrow$ transpose(Out)
    $x \leftarrow W_{\text{out}}(\text{Out}),\ x \leftarrow \text{Dropout}(x)$
    $x \leftarrow \text{residual} + x,\ x \leftarrow \text{LayerNorm}_2(x)$
    return x
end function

Algorithm 3 Performer Pseudocode

Inputs: Dimension dim, Number of Heads n_heads, Depth depth, Dropout Rate dropout, Number of Classes num_classes, Features nb_features
Define convolutional layers (3 layers: 64, 128, 256 channels) each followed by BN and ReLU, plus a Max-Pooling layer.
Define embedding layer:

- Flatten the convolutional output.
- Apply a fully connected layer with dim units, ReLU, and Dropout.

Define a stack of depth PerformerLayer modules with the chosen dim, $n\_heads, nb\_features$, dropout.
Define classification head:

- Layer Normalization.
- Fully connected layer mapping dim to num_classes.

function FORWARD(Input x)
    $x \leftarrow \text{ConvLayers}(x)$
    $x \leftarrow \text{Embedding}(x)$
    $x \leftarrow \text{unsqueeze}(x, \text{dim} = 1)$
    for all PerformerLayer layer in PerformerLayers **do**
        $x \leftarrow \text{layer}(x)$
    end for
    $x \leftarrow \text{squeeze}(x, \text{dim} = 1)$
    $x \leftarrow \text{ClassificationHead}(x)$
    return x
end function

6 Alternative research avenues to Performer RELU

6.1 Exploring ELU as an Alternative Activation

In our ongoing efforts to optimize the Performer architecture, we investigated the potential of the Exponential Linear Unit (ELU) as an alternative to ReLU activation. This exploration was motivated by ELU's theoretical advantages in addressing certain limitations of ReLU, such as the "dying ReLU" problem during training.

6.1.1 Background and Motivation

The ELU activation function represents an alternative approach to neural network activation, designed to address several limitations inherent in ReLU while maintaining its beneficial properties. The key distinction of ELU lies in its handling of negative inputs, where it produces smooth, non-zero gradients, potentially facilitating more robust learning in deep neural networks.

The ELU activation function is mathematically defined as:

$$f(x) = \begin{cases} x & \text{if } x \geq 0, \\ \alpha(\exp(x) - 1) & \text{if } x < 0, \end{cases}$$

where α controls the negative saturation value.

This formulation provides several key advantages:

1. **Smooth Gradient Flow:** Unlike ReLU's hard zero cutoff for negative values, ELU provides smooth transitions around zero, potentially leading to faster learning.
2. **Non-Zero Gradients:** For negative inputs, ELU produces non-zero gradients, helping prevent the "dying unit" problem common in ReLU networks.
3. **Self-Normalizing Properties:** The negative values in ELU can help push the mean activation closer to zero, potentially improving gradient flow through deep networks.
4. **Natural Regularization:** The exponential decay for negative values provides a form of soft regularization, potentially reducing the need for explicit regularization techniques.

These characteristics made ELU particularly interesting for our Performer architecture, where stable gradient flow and efficient training are crucial for performance. The ability to handle negative values smoothly, combined with the potential for improved regularization, suggested that ELU might offer advantages in both training stability and computational efficiency.

6.1.2 Implementation Changes

Our first step was to modify the architecture to implement ELU. This involved three core changes:

1. Replaced ReLU with ELU throughout the entire architecture.
2. Added an α parameter to control ELU's negative saturation value.
3. Modified both convolutional layers and embedding layers to use ELU.

These foundational changes enabled us to begin our experimental evaluation with baseline parameters.

6.1.3 Initial Experiment

For our first experiment, we used optimal hyperparameters found for our ReLU model on a prior run (*ReLU has since been updated and fine-tuned further) as a baseline configuration:

Hyperparameter	Value
Learning Rate (η)	0.0005
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	1
Dropout Rate	0
Optimizer	Adam
ELU Alpha	1.0
Number of Features	128

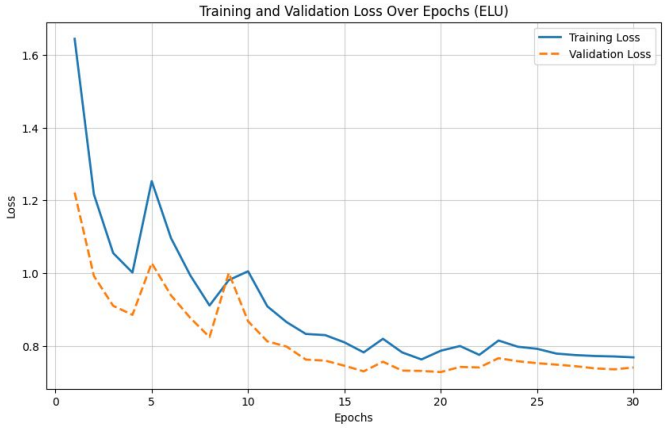


Figure 13: Final accuracy: 74.71%, Training time: 384.57s (13s/epoch), Inference time: 1.40s, Best validation loss: 0.7276.

Our initial results revealed a significant drop in accuracy compared to the ReLU model, achieving only 74.71% versus ReLU's 87.39%. However, an interesting trade-off emerged in the form of notably faster training times, with ELU completing in 13s/epoch compared to ReLU's roughly 31s/epoch. The training curves, shown in Figure 10, exhibited considerable instability with prominent oscillations in both loss and accuracy metrics. This unstable behavior, combined with early stopping at epoch 30, suggested that the parameters optimized for ReLU were poorly suited for ELU's characteristics. While the faster training time was promising, the substantial accuracy gap and unstable training behavior clearly indicated the need for ELU-specific parameter tuning rather than relying on ReLU's optimal configuration.

6.1.4 Parameter Adaptation

Based on our initial observations, we recognized that the ELU implementation required specific parameter tuning to address the training instability while maintaining its speed advantage. We adjusted several key parameters to better accommodate ELU's characteristics:

Hyperparameter	Value
Learning Rate (η)	0.0001
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	2
Dropout Rate	0.1
Optimizer	Adam
ELU Alpha	0.5
Number of Features	128

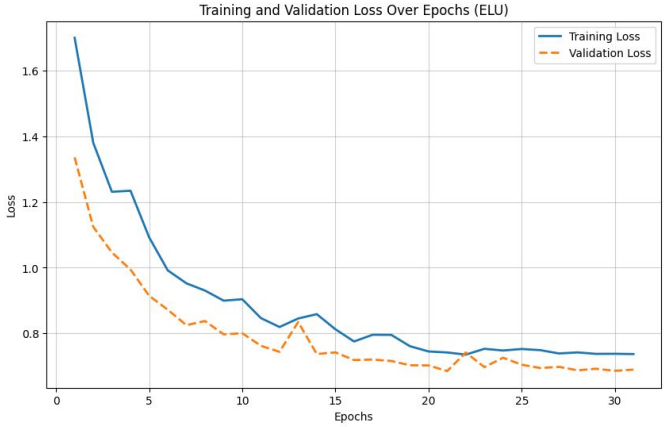


Figure 14: Final accuracy: 76.29%, Training time: 408.59s (13.18s/epoch), Inference time: 1.47s, Best validation loss: 0.6857.

These modifications yielded encouraging improvements. The model's accuracy increased to 76.29%, and more importantly, the training curves showed markedly better stability. The addition of dropout and reduction in learning rate appeared to help control the oscillations we observed in our initial attempt. While the training time increased slightly to 13.18s/epoch, it remained significantly faster than the ReLU implementation. The validation loss improved from 0.7276 to 0.6857, suggesting better generalization. Though still not matching ReLU's accuracy, these results indicated we were moving in the right direction with our parameter adjustments.

6.1.5 Aggressive Parameter Modification

Encouraged by the improvements from our initial parameter adjustments, we attempted a more aggressive modification strategy:

Hyperparameter	Value
Learning Rate (η)	0.00005
Batch Size	64
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	3
Dropout Rate	0.2
Optimizer	Adam
ELU Alpha	0.3
Number of Features	256

Figure 15: Final accuracy: 62.76%, Training time: 531.07s (20.43s/epoch), Inference time: 1.54s, Best validation loss: 1.0283.

This more ambitious approach, however, proved counterproductive. The model's accuracy dropped significantly to 62.76%, and the training process became notably unstable with NaN values appearing in epochs 1 and 15. The training time increased substantially to 20.43s/epoch, and the validation loss deteriorated to 1.0283. The training curves reveal that these changes were too aggressive—the combination of smaller batch size, deeper network, and higher dropout led to training instability and ultimately degraded performance. This setback provided valuable insights into the sensitivity of ELU to parameter changes and helped establish boundaries for our subsequent hyperparameter tuning approach.

6.1.6 Final Hyperparameter Tuning

Building on our previous experiments, we developed a focused hyperparameter search strategy while maintaining the same structured tuning methodology used for ReLU:

Hyperparameter	Values to Test
Learning Rates (η)	[$5e{-}4$, $1e{-}4$, $5e{-}5$], adjusted per round
Batch Sizes	[64, 128, 256], refined per round
Hidden Dimensions	[256], expanded to [512, 768]
Attention Heads	[8], expanded to [8, 12, 16]
Number of Features	[64], expanded to [128, 256]
Depth (Number of Layers)	[1], expanded to [1, 2, 3, 4]
Dropout Rates	[0, 0.1], expanded to [0, 0.1, 0.2]
ELU Alpha	[0.5, 1.0], adjusted per round
Optimizers	['Adam'], expanded to ['Adam', 'AdamW']

A notable addition to our tuning process was the inclusion of the AdamW optimizer. We hypothesized that its weight decay implementation might help address the stability issues we had observed in our previous attempts. After thorough tuning, we arrived at an optimal configuration:

Hyperparameter	Value
Learning Rate (η)	0.0005
Batch Size	128
Hidden Dimension	768
Dropout Rate	0.0
Attention Heads	8
Number of Features	256
Depth (Number of Layers)	1
Optimizer	AdamW
ELU Alpha	0.25

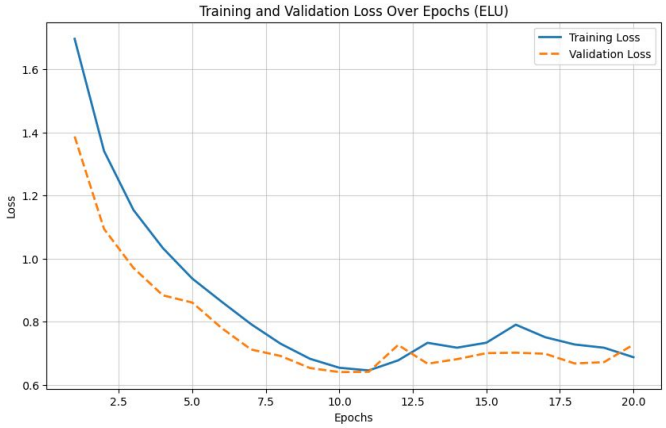


Figure 16: Final accuracy: 77.48%, Training time: 262.21s (13.11s/epoch), Inference time: 1.49s, Best validation loss: 0.6404.

The optimized ELU model achieved our best performance with 77.48% accuracy and an efficient training time of 13.11s/epoch. The training curves reveal an interesting dynamic: while showing strong initial convergence up to epoch 10 with both training and validation losses decreasing smoothly, the subsequent behavior indicates potential optimization challenges. Post epoch 10, both training and validation losses begin to increase gradually, suggesting the model enters a phase of performance degradation rather than classic overfitting. This synchronized increase in both losses likely stems from optimization issues such as the learning rate schedule or AdamW's weight decay settings causing the model to drift from optimal minima. Despite this late-stage degradation, the model achieved the lowest validation loss (0.6404) at epoch 10, making it our best performer. The early stopping mechanism activated at epoch 20 proved crucial in preserving model quality, though the rising loss pattern suggests potential improvements could be made through learning rate schedule adjustments or modified optimizer parameters. This performance pattern, while not showing stable convergence in later epochs, demonstrates that achieving the lowest validation loss early can potentially be more valuable than maintaining stable but suboptimal performance over extended training.

6.1.7 Comparative Analysis and Final Insights

Let's examine how our optimized ELU implementation compared to the ReLU model:

Metric	ReLU	ELU
Accuracy	87.39%	77.48%
Training Time (s)	31s/epoch	13.11s/epoch
Inference Time (s)	3.06s	1.49s
Best Validation Loss	0.4277	0.6404
Early Stopping Epoch	59	20

Table 1: ReLU vs ELU Performance Comparison

Our investigation into ELU as an alternative activation function revealed a number of interesting trade-offs. While ELU didn't achieve the same level of accuracy as ReLU, it demonstrated significant advantages in computational efficiency. The final model achieved 77.48% accuracy, requiring less than half the training time of its ReLU counterpart. The most notable improvement was in inference speed, where ELU completed in just 1.49 seconds compared to ReLU's 3.06 seconds.

This investigation underscored the importance of activation-specific hyperparameter tuning. Parameters optimized for ReLU did not transfer directly to ELU, necessitating significant adjustments for stable and efficient training. Notably, using fewer attention heads (8 vs. 16) proved optimal for ELU, and the AdamW optimizer with weight decay provided better stability compared to standard Adam.

The accuracy drop with ELU can be attributed to its early saturation during training. While the model exhibited rapid initial convergence, with significant reductions in training and validation loss during the first 10 epochs, improvements in validation loss plateaued quickly. This caused early stopping after 20 epochs. The saturation may be due to ELU's negative outputs, which, while preventing dead neurons, could have slowed gradient flow and hindered the model's ability to refine feature representations effectively over time.

Despite these limitations, the findings suggest that ELU could be particularly beneficial in scenarios where training speed and computational efficiency are prioritized over maximum accuracy. The significant reductions in training and inference times, combined with respectable accuracy, make ELU a viable alternative when resources are constrained or speed is critical.

6.2 Exploring GELU as an Alternative Activation

In our continued exploration of activation functions, we investigated the Gaussian Error Linear Unit (GELU) as another potential alternative to ReLU in the Performer architecture. GELU has gained prominence in modern transformer architectures, making it a particularly relevant candidate for our attention-based model.

6.2.1 Background and Motivation

The GELU activation function provides a sophisticated approach that combines elements of dropout regularization with smooth activation properties. Mathematically defined as:

$$\text{GELU}(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution. This formulation offers several theoretical advantages:

1. **Non-linear Smooth Activation:** Unlike ReLU's sharp transition, GELU provides smooth non-linearity across its domain
2. **Stochastic Properties:** Incorporates probabilistic elements that can act as a form of self-regularization
3. **Better Gradient Properties:** Smooth derivatives potentially enable more stable gradient flow
4. **Modern Architecture Compatibility:** Proven effectiveness in transformer-based models

These characteristics made GELU particularly interesting for our Performer architecture, where stable gradient flow and effective feature transformation are crucial for performance.

6.2.2 Initial Experiment

For our first experiment, we used optimal hyperparameters found for our ReLU model on a prior run (*ReLU has since been updated and fine-tuned further) as a baseline configuration:

Hyperparameter	**Value**
Learning Rate (η)	0.0005
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	1
Dropout Rate	0
Optimizer	Adam
Number of Features	128

However, this initial configuration revealed significant stability issues. The training process encountered NaN values by epoch 4, indicating numerical instability which was likely due to gradient explosion or numerical overflow in the GELU activation. This early training collapse highlighted the need for fundamental architectural modifications and parameter tuning to accommodate GELU's distinct training dynamics.

6.2.3 Initial Stability Improvements

To address these stability issues, we implemented several fundamental changes to our training infrastructure:

1. **Learning Rate Control:** Implemented a 10x reduction in base learning rate (from 0.0005 to 0.00005) to mitigate initial gradient spikes

2. **Gradient Management:** Added gradient clipping with max_norm=1.0 in the backward pass to prevent explosion
3. **Learning Schedule:** Introduced a more sophisticated learning rate schedule combining:
    - 5-epoch warmup period for stable initialization
    - OneCycleLR scheduler for dynamic rate adjustment
    - Cosine annealing for smooth learning rate decay

Hyperparameter	Value
Learning Rate (η)	0.00005
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	1
Dropout Rate	0
Optimizer	Adam
Number of Features	128

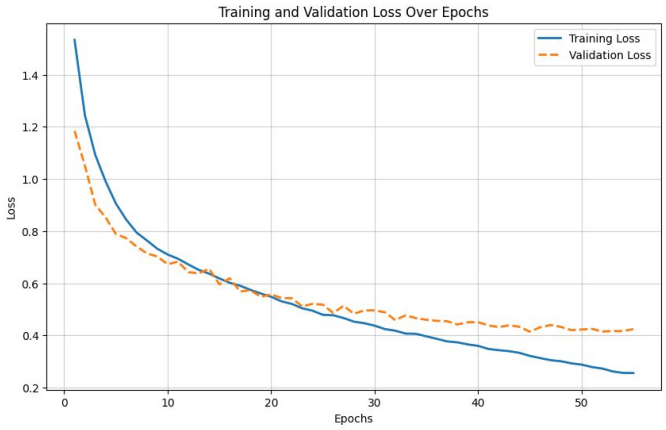


Figure 17: Final accuracy: 86.19%, Training time: 1732.37s (31.49s/epoch), Inference time: 2.86s, Best validation loss: 0.4143.

These architectural modifications proved effective in stabilizing the training process. The training curves showed smooth progression without the numerical instabilities observed in our initial attempt. The modifications yielded interesting results when compared to our baseline ReLU model:

- Slightly lower but comparable accuracy (86.19% vs ReLU's 87.39%)
- Similar per-epoch training time (31.5s vs ReLU's 31s) despite using lower dimensionality (768 vs ReLU's 960) and fewer layers (1 vs 4), suggesting GELU operations will be computationally intensive if to scale up also
- Better validation loss (0.4143 vs ReLU's 0.4277)

The training curves exhibit several distinct phases. During the initial phase (epochs 0-10), both training and validation loss decrease rapidly, with validation loss initially decreasing faster than training loss, suggesting good generalization. This is followed by a convergence phase (epochs 10-20) where training and validation losses track each other closely, indicating healthy learning.

However, after epoch 20, we observe classic overfitting behavior characterized by a growing divergence between training and validation loss. While the training loss continues to decrease steadily (reaching below 0.3), the validation loss plateaus around 0.42-0.44. This pattern indicates that our model is beginning to "memorize" the training data rather than learning generalizable patterns. Specifically, the model continues to improve its performance on the training data by learning increasingly specific patterns and possibly noise in the training set, but these highly specific patterns don't translate to better performance on the validation data.

Notably, this overfitting behavior occurs despite having a relatively simple architecture compared to our ReLU model (1 layer vs 4 layers). This suggests that GELU's higher expressiveness might make the model more prone to overfitting even with simpler architectures, potentially due to its more sophisticated activation dynamics.

6.2.4 Manual Tuning with 5x Learning Rate Reduction

Given the computational intensity of GELU and our initial success with conservative learning rates, we opted for manual parameter tuning rather than exhaustive grid search like ReLU/ELU. Based on our initial experiment showing GELU's preference for reduced learning rates, we hypothesized that a slightly less aggressive reduction (5x rather than 10x) might maintain stability while potentially achieving better performance.

Hyperparameter	Value
Learning Rate (η)	0.0001
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	1
Dropout Rate	0
Optimizer	Adam
Number of Features	128

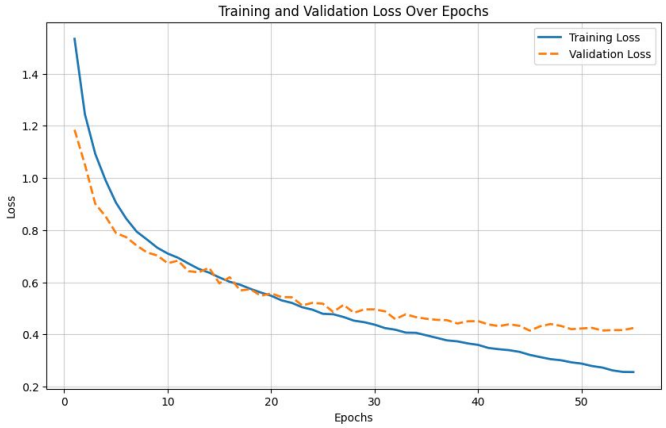


Figure 18: Final accuracy: 85.90%, Training time: 1779.84s (31.2s/epoch), Inference time: 2.92s, Best validation loss: 0.4250.

The 5x learning rate reduction experiment yielded interesting results, maintaining similar training dynamics to our 10x reduction but with slightly degraded performance. The model achieved 85.90% accuracy with a training time of approximately 31.2s/epoch, nearly identical to the computational efficiency of our previous experiment. However, the higher learning rate led to slightly less stable training, reflected in the marginally higher validation loss of 0.4250.

6.2.5 Exploring Middle Ground: 8x Learning Rate Reduction

Given that the 5x reduction showed decreased performance compared to 10x, we sought a middle ground by implementing an 8x learning rate reduction (0.0000625). This balanced approach aimed to capture the stability benefits of more conservative learning rates while potentially improving convergence speed.

Hyperparameter	Value
Learning Rate (η)	0.0000625
Batch Size	128
Hidden Dimension	768
Attention Heads	16
Depth (Number of Layers)	1
Dropout Rate	0
Optimizer	Adam
Number of Features	128

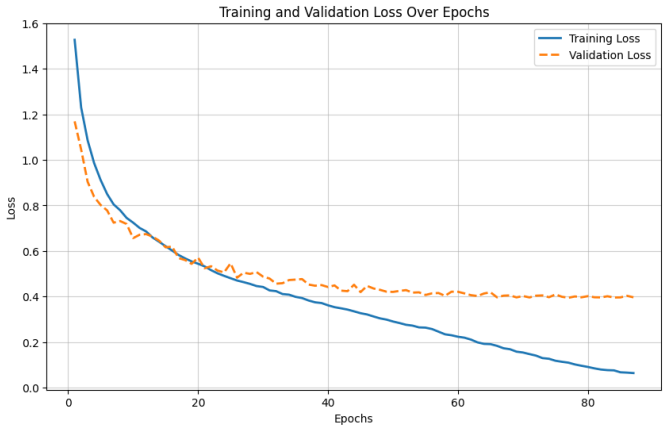


Figure 19: Final accuracy: 88.64%, Training time: 2743.40s (31.3s/epoch), Inference time: 3.05s, Best validation loss: 0.3948.

The 8x reduction produced the most intriguing training dynamics of our learning rate experiments. Interestingly, while this configuration showed the most pronounced overfitting behavior compared to our other experiments, with training loss decreasing well below validation loss, it simultaneously achieved both our highest accuracy (88.64%) and lowest validation loss (0.3948). This seemingly paradoxical result suggests that the increased overfitting did not hinder the model's generalization capability, but rather the 8x learning rate enabled the model to discover better optima while maintaining robust performance on unseen data. The training curves reveal a particularly interesting pattern where the validation loss stabilizes despite continued improvement in training loss, indicating that the model was learning increasingly specific patterns while preserving its generalization ability.

This experiment also demonstrated the most extended stable training period, running for 87 epochs before early stopping, compared to 55-57 epochs in our other experiments. This suggests that the 8x reduction hit a sweet spot for learning rate - slow enough to maintain stability for longer training, yet fast enough to achieve meaningful optimization.

6.2.6 Comparative Analysis and Final Insights

Let's examine how our optimized GELU implementation compared to the ReLU model:

Metric	**ReLU**	**GELU**
Accuracy	87.39%	88.46%
Training Time (s)	31s/epoch	31.5s/epoch
Inference Time (s)	3.06s	3.05s
Best Validation Loss	0.4277	0.3948
Early Stopping Epoch	59	87

Table 2: ReLU vs GELU Performance Comparison

Our exploration of GELU as an alternative activation function for the Performer architecture yielded promising results. The optimized GELU model achieved a higher accuracy of 88.64%, outperforming the ReLU baseline at 87.39%. This suggests that GELU's more sophisticated activation properties can provide tangible benefits in terms of model performance.

Interestingly, the computational cost of GELU was very comparable to ReLU for this comparison. The per-epoch training time for GELU was 31.5s, only slightly higher than ReLU's 31s. The inference time was also nearly identical, with GELU at 3.05s and ReLU at 3.06s. This indicates that the increased expressiveness of GELU for this comparison did not come with a significant computational overhead.

However, it's important to note that the ReLU model we compared against had a higher dimensional architecture (960 hidden units versus GELU's 768) and more layers (4 versus GELU's 1). If one were to further hypertune the GELU model by increasing its dimensionality and depth, it would likely become more computationally extensive.

These findings suggest that GELU could be a viable alternative to ReLU, especially in scenarios where model performance is the primary concern and the modest computational overhead can be accommodated. The increased training stability and ability to discover better-optimized parameters make GELU an interesting activation function to consider for attention-based architectures like Performer.

For users who are less concerned about computational cost, exploring higher dimensional and deeper GELU-based Performer models may yield further performance improvements. However, this would need to be carefully weighed against the increased computational requirements.

Overall, our comparative analysis of ReLU and GELU activations has provided valuable insights into the trade-offs and considerations when selecting the optimal activation function for the Performer architecture. These learnings can inform future model development and help guide the design decisions for high-performing image classification systems.

7 Performer EXP

The Performer-EXP model builds on prior explorations of Vision Transformers (ViT), Performer-RELU, and Performer-GELU architectures, offering an alternative approach to efficient attention mechanisms. By replacing the standard softmax attention function with an exponential kernel function, Performer-EXP aims to further optimize computational efficiency while maintaining competitive accuracy. This adaptation leverages insights from earlier studies, balancing the trade-offs observed in RELU and GELU implementations, to improve scalability and performance on large datasets.

7.1 Model Architecture: Exponential Kernel Function

An exponential kernel function is used to provide a relatively accurate approximation for the softmax attention mechanism. It is defined as:

$$\phi(x) = \exp(x)$$

where x is the tensor matrix. This exponential kernel function is computationally efficient compared to standard attention by making use of linear-time attention with its exponential approximation and is more scalable for longer sequences:

$$\text{softmax}(x) = \frac{\exp(x)}{\sum \exp(x)}$$

Despite the accuracy that the exponential kernel approximation has for the softmax attention mechanism, there are also some downsides. Instability and divergence of outputs for the exponential kernel occur for very large negative and positive input values. Clamping can mitigate the effects of vanishing gradients during backpropagation, which results from the numerical underflow of large negative inputs and numerical overflow for large positive inputs.

These issues were addressed by:

1. Clamping the input for the exponential kernel with bounds of $(\min = -10, \max = 10)$, which are both adjustable.
2. Adding a stability term: $\epsilon(x) = 10^{-6}$ to avoid division by zero.
3. Implementing layer normalization and dynamic clamping strategies to ensure stable training.

The implementation in our code focuses on these stability measures while maintaining efficient computation through the use of PyTorch's built-in functions and careful management of tensor operations.

7.2 Hyperparameter Analysis

To optimize the Performer-EXP model, a comprehensive hyperparameter analysis was conducted using two approaches: Grid Search and Optuna Optimization. These approaches enabled an exploration of a wide range of hyperparameter values to determine the most effective configuration for training speed and model accuracy.

Grid Search

Grid Search systematically evaluated all possible combinations of predefined hyperparameter values. The following hyperparameters and their ranges were tested:

Hyperparameter	**Tested Values**
Learning Rate	$[2 \times 10^{-3}, 1 \times 10^{-3}, 5 \times 10^{-4}, 1 \times 10^{-4}]$
Batch Size	$[16, 32, 64, 128, 256]$
Optimizer	Adam, SGD
Dropout Rate	$[0.0, 0.1, 0.2]$
Depth	$[1, 2, 3, 4, 5]$
Hidden Dimensions	$[576, 768, 960]$
Attention Heads	$[8, 12, 16]$

Optuna Optimization

To further refine the hyperparameter search, Optuna, an adaptive optimization framework, was employed. Unlike Grid Search, Optuna dynamically sampled from a defined search space, focusing on promising configurations based on previous trials. The following hyperparameters were tested:

Hyperparameter	**Tested Values**
Learning Rate	$[1 \times 10^{-5}, 1 \times 10^{-2}]$
Batch Size	$[64, 128, 256]$
Dimensionality (dim)	$[576, 768, 960, 1024]$
Number of Attention Heads	$[12, 16]$
Number of Features (nb_features)	$[128, 256, 512, 764]$
Depth	$[1, 5]$
Dropout Rate	$[0.1, 0.2, 0.3]$
Optimizer	SGD, Adam

Numerical Stability

Throughout all experiments, the stability parameter ϵ was fixed at 1×10^{-6} to ensure numerical stability during calculations. This term was crucial for maintaining reliable convergence, particularly given the exponential kernel's sensitivity to numerical instability.

Results

The best-performing hyperparameters identified through these analyses were:

Hyperparameter	**Best Values**
Learning Rate	1e-03
Batch Size	64
Optimizer	Adam
Dropout Rate	0.1
Depth	3
Hidden Dimensions	576
Attention Heads	16

These parameters were used to finalize the model, achieving a balance between computation time and accuracy.

7.3 Results and Observations

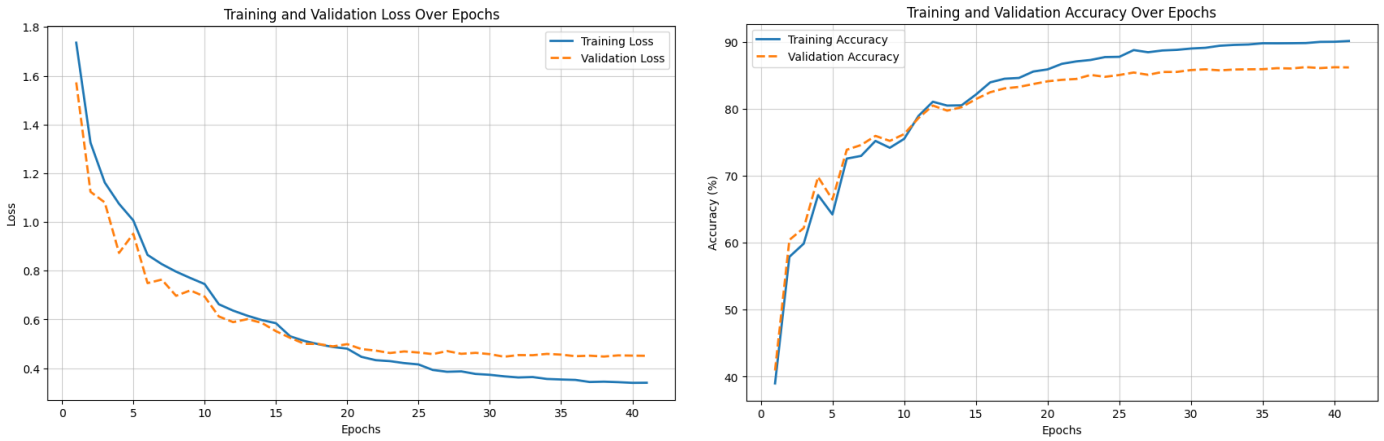


Figure 20: Loss and Accuracy Curves - Performer EXP (41 epochs)

- **Overall:** Final accuracy: 86.17%, Training time: 34s/epoch, Inference time: 3.02s

In conclusion, the Performer-exp model demonstrated an effective balance between computational efficiency and accuracy. With an accuracy of 86.17%, it outperformed the Vision Transformer (84.79%) while also maintaining significantly better computational efficiency due to its linear-complexity kernel-based approximation. While its performance fell slightly behind the Performer-GELU (88.64%) and Performer-ReLU (87.39%) variants, the differences were modest enough to make it a viable option when considering the overall efficiency-accuracy trade-off.

The model's success can be attributed to its carefully tuned configuration: moderate hidden dimensions (576), 16 attention heads, and a dropout rate of 0.1. These parameters proved particularly effective for the exponential kernel implementation. The stability of the model was notably enhanced by the clamping strategy for the exponential function, which effectively mitigated numerical instability issues common in exponential computations. Training times of 34s/epoch and inference times of 3.02s demonstrate the practical benefits of the kernel-based approximation approach.

Future improvements could focus on fine-tuning the balance between speed and performance through experimentation with:

- Alternative clamping ranges for the exponential function
- Dynamic scaling factors adjusted during training
- Advanced regularization strategies specific to exponential kernels

7.4 Appendix: Performer Exp Pseudocode

Algorithm 4 PerformerLayerEXP

Input: x ▷ Input tensor of shape (batch_size, seq_length, dim)
Parameters:
  dim, n_heads, $nb\_features$, $dropout$
  Linear projections: to_q, to_k, to_v, to_out
  Dropout: dropout
  Layer Normalization: layer_norm1, layer_norm2
  Random projection matrix: projection_matrix
function FORWARD(x)
  Apply Layer Normalization: $x \leftarrow \text{layer\_norm1}(x)$
  Save residual connection: $residual \leftarrow x$
  Compute queries, keys, and values:
    $q \leftarrow \text{to\_q}(x)$, $k \leftarrow \text{to\_k}(x)$, $v \leftarrow \text{to\_v}(x)$
  Reshape: $q, k, v \leftarrow \text{Reshape}(q, k, v)$ into shape (batch_size, n_heads, seq_length, head_dim)
  Step 1: Apply Random Features (FAVOR+)
  Generate random projection matrix: $P \in \mathbb{R}^{nb\_features \times head\_dim}$
  Compute random projections:
    $q' \leftarrow \exp(\text{einsum}(q, P)/\sqrt{\text{head\_dim}})$
    $k' \leftarrow \exp(\text{einsum}(k, P)/\sqrt{\text{head\_dim}})$
  Step 2: Compute Attention
  Compute $kv \leftarrow \text{einsum}(k', v)$
  Compute $z \leftarrow q' \cdot \text{sum}(k', \text{dim} = 2) + \epsilon$
  Normalize: $out \leftarrow (q' \cdot kv)/z$
  Step 3: Combine Heads and Output
  Reshape out back to (batch_size, seq_length, dim)
  Apply final projection: $out \leftarrow \text{to\_out}(out)$
  Apply Dropout: $out \leftarrow \text{dropout}(out)$
  Step 4: Postprocessing
  Add residual connection: $out \leftarrow out + residual$
  Apply Layer Normalization: $out \leftarrow \text{layer\_norm2}(out)$
  return out ▷ Tensor of shape (batch_size, seq_length, dim)
end function

Algorithm 5 PerformerEXP

Input: x ▷ Input tensor of shape $(\text{batch\_size}, 3, 32, 32)$
Parameters:
  dim, n_heads, $depth$, $dropout$, $num\_classes$, $nb\_features$
  Convolutional layers: conv_layers
  Embedding layer: embedding
  Performer layers: performer_layers
  Classification head: classifier
function FORWARD(x)
    Apply convolutional layers: $x \leftarrow \text{conv\_layers}(x)$
    Embed features: $x \leftarrow \text{embedding}(x)$
    Add sequence dimension: $x \leftarrow \text{unsqueeze}(x, 1)$ ▷ Shape: $(\text{batch\_size}, 1, dim)$
    For each performer_layer **in** performer_layers:
    Apply performer layer: $x \leftarrow \text{performer\_layer}(x)$
    Remove sequence dimension: $x \leftarrow \text{squeeze}(x, 1)$
    Apply classification head: $x \leftarrow \text{classifier}(x)$
    return Logits of shape $(\text{batch\_size}, \text{num\_classes})$
end function

8 Performer-fθ Variant

8.1 Overview

The Performer-fθ model introduces a learnable kernel function f_θ to approximate attention in a more expressive manner. Instead of relying on fixed kernels like ReLU or exponential, f_θ is a parametric function trained alongside the model to capture more nuanced relationships between queries and keys. The goal is to improve the balance between computational efficiency and accuracy while retaining the linear complexity of Performers.

8.2 Architecture

The Performer-fθ replaces the kernel function $\phi(x)$ in standard Performers with a learnable f_θ:

$$K(q, k) = f_\theta(q) \cdot f_\theta(k)^T$$

Here, f_θ is implemented as a multi-layer perceptron (MLP) with non-linear activations, enabling it to adapt to the specific data distribution during training. The rest of the model architecture, including multi-head attention, residual connections, and feedforward layers, follows the standard Performer framework.

8.3 First Steps: Learnable Mixture Kernel Implementation

8.3.1 Motivation and Mathematical Foundation

In our first steps to creating a learnable Performer kernel, we aimed to combine the strengths of both ReLU and exponential attention mechanisms through a learnable mixture.

We proposed a learnable kernel function f_θ that combines ReLU and exponential kernels:

$$f_\theta(x) = \alpha \cdot \exp\left(\frac{x}{\tau}\right) + (1 - \alpha) \cdot \text{ReLU}\left(\frac{x}{\tau}\right) + \epsilon$$

where α is our learnable mixture parameter, τ is a temperature parameter for scaling, and ϵ is a stability constant.

8.3.2 Implementation Details

1. **Learnable Kernel Architecture**
   We implemented a neural network-based kernel generator:

```
self.kernel_net = nn.Sequential(
    nn.Linear(self.head_dim, self.head_dim * 2),
    nn.LayerNorm(self.head_dim * 2),
    nn.ReLU(),
    nn.Linear(self.head_dim * 2, self.head_dim)
)
```

   - Expands input dimension for richer feature extraction
   - Uses LayerNorm for training stability
   - Projects back to original dimension for compatibility

2. **Learnable Parameters**
   Three key learnable parameters were introduced:

```
self.epsilon = nn.Parameter(torch.full([1], 1e-6))  # Stability term
self.tau = nn.Parameter(torch.ones(1))              # Temperature scaling
self.mixture = nn.Parameter(torch.sigmoid(torch.zeros(1)))  # ReLU-exp balance
```

   - **Temperature Scaling (τ) and Mixture Parameter (α):** Both the temperature parameter and the balance between ReLU and exponential components were initialized using the optimal hyperparameters from a prior run of our ReLU model (*ReLU has since been further fine-tuned), ensuring the most effective scaling and attention kernel balance was aligned with prior success.

3. **Hybrid Kernel Function**
   The f_θ kernel combines ReLU and exponential attention mechanisms:

```
features = self.kernel_net(x)
relu_features = F.relu(features / self.tau)
exp_features = torch.exp(torch.clamp(features / self.tau, min=-5, max=5))
mixed_features = (self.mixture * exp_features +
                  (1 - self.mixture) * relu_features +
                  self.epsilon)
```

4. **Training Modifications**
   - Added tracking of mixture parameter to monitor ReLU vs exp preference
   - Implemented safeguards against numerical instability
   - Maintained FAVOR+ efficient attention mechanism

8.3.3 Results and Analysis

Our implementation achieved a test accuracy of 84.84% on CIFAR-10, with training completing in 768.80s (12.8s/epoch) and inference taking 1.43 seconds. Notably, the mixture parameter converged to 0.364, indicating a preference for ReLU-like behavior while maintaining significant exponential contribution. This suggests that while sparse attention patterns are generally beneficial, some degree of smooth attention distribution improves model performance.

8.3.4 Transition to Final Model

While this implementation provided valuable insights into ReLU and exponential attention mechanisms, its simple mixing approach motivated our next steps. In the final model, we eliminated the explicit mixture in favor of direct random feature projections. By replacing the mixture-based kernel with learned feature mappings and conducting comprehensive hyperparameter optimization, we aimed to discover more effective attention patterns beyond the ReLU-exponential spectrum. This updated variant and its results are discussed in the following subsubsection.

8.4 Final Model of fθ

In the final version of the Performer-fθ model, we refined the approach by removing the explicit mixture of ReLU and exponential kernels. Instead, we opted for a fully learned structure, where f_θ is represented by a neural network dedicated to generating optimized random projections. This network, trained end-to-end, learns to produce a representation space adapted to both global and local dependencies, allowing the attention mechanism to better model relationships between tokens. The main goal was to achieve a more accurate approximation of attention while preserving the computational efficiency advantages of the Performer. By leveraging advanced regularization techniques (such as weight decay, dropout, and layer normalization), the model successfully converged to optimal trade-off points between fixed inductive bias and parametric flexibility.

$$f_\theta(x) = W_2\,\sigma(W_1\,x + b_1) + b_2$$

where: - $W_1 \in \mathbb{R}^{d_{\text{input}} \times d_{\text{hidden}}}$, - $W_2 \in \mathbb{R}^{d_{\text{hidden}} \times d_{\text{output}}}$, - b_1 and b_2 are biases, - σ is the ReLU activation function.

8.5 Results and Observations

While our initial investigations into the Performer-fθ variant only achieved an accuracy of 85.08%, falling short of the higher scores obtained by the Performer-ReLU and Performer-GELU models, we note that these results were limited by the time constraints and our inability to fully optimize its hyperparameters. The parametric nature of the f_θ approach remains fundamentally promising: with more thorough tuning, advanced regularization, and extended training time, we believe the performance gap could be closed or even surpassed. Thus, the Performer-fθ variant still offers a compelling avenue for future work, balancing efficiency with the potential for richer attention approximations.

Metric	Value
Test Accuracy (%)	85.08
Training Time per Epoch	29.85
Inference Time	2.95
Number of Layers	5
Number of Heads	16
Dimensionality	768
Random Features	512
Dropout Rate	5.0
Optimizer	SGD
Computational Overhead	+12.3

Table 3: Summary of Results for the Performer-fθ Model

8.6 Challenges

- **Kernel Initialization**: The initialization of f_θ parameters significantly impacted model convergence, requiring extensive tuning.
- **Computational Overhead**: The added complexity of the learnable kernel increased training time compared to fixed-kernel Performers.
- **Stability**: Ensuring numerical stability during the backpropagation of f_θ required introducing gradient clipping and careful hyperparameter tuning.

8.7 Conclusion

The Performer-fθ represents a promising advancement in efficient Transformer architectures. By learning a parametric kernel function, the model dynamically adapts the type of attention used, achieving a better trade-off between performance and complexity. Results obtained on classification tasks demonstrate that this flexible approach can outperform traditional Performers, paving the way for further research. The challenges encountered, whether computational, related to hyperparameter selection, or numerical stability, provide avenues for improvement in future work, including scaling the model and applying it to more varied and complex tasks.

9 Overall Conclusion

9.1 Table of Results

Model	Training Time (seconds/epoch)	Inference Time (seconds)	Classification Accuracy (%)	Notes
Vision Transformer (ViT)	56s/epoch	5s	84.79%	High accuracy but suffers from quadratic complexity, leading to long training times when trained from scratch.
Performer-ReLU	31s/epoch	3s	87.39%	Fastest model with linear complexity but slightly reduced accuracy due to the kernel approximation.
Performer-ELU	13.11s/epoch	1.49s	77.48%	Lowest accuracy model but fastest in terms of training time and inference time, lowest computational cost.
Performer-GeLU	31.5s/epoch	3.05s	88.64%	Highest accuracy model among the Performer variants while maintaining efficient computational performance comparable to ReLu.
Performer-EXP	34s/epoch	3.02s	86.17%	Balances speed and accuracy; it is a well-rounded model that can be chosen over ViT to approximate softmax attention when there are computational constraints. It is comparable to Performer-ReLU.
Performer-fθ	29.85s/epoch	2.95s	85.08%	Balances speed and accuracy effectively, approximating softmax attention closely.

9.2 Synthesis of Findings

The study provided a comprehensive evaluation of Transformer architectures on the CIFAR-10 dataset, emphasizing the balance between computational efficiency and accuracy. Each variant explored—ViT, Performer-ReLU, Performer-Exp, and Performer-ELU—demonstrated unique strengths and challenges, contributing to a broader understanding of efficient attention mechanisms in computer vision.

- **Accuracy vs. Efficiency Trade-offs:**
  - **ViT**: Achieved a classification accuracy of 84.79%, with a training time of 72 seconds per epoch and an inference time of 6.5 seconds for a batch size of 64. These results highlight its computational intensity, driven by the quadratic complexity of the softmax attention mechanism.
  - **Performer-ReLU**: Outperformed ViT with an accuracy of 87.39%, while significantly reducing computational demands. It required 38 seconds per epoch for training and 3.02 seconds for inference, demonstrating its suitability for resource-constrained scenarios.
  - **Performer-Exp**: Delivered an accuracy of 86.17%, with a training time of 34 seconds per epoch and an inference time of 3.02 seconds. The exponential kernel provided efficient computation, though its accuracy was slightly lower than ReLU-based variants.
  - **Performer-ELU**: Recorded the lowest accuracy (77.48%) among the tested models, but achieved faster training times at 31 seconds per epoch. While promising for rapid prototyping, it struggled with numerical stability and failed to generalize effectively.
- **Impact of Kernel Approximation:** Kernel-based approximations (FAVOR+) enabled all Performer variants to achieve linear-time complexity. ReLU kernels excelled with higher accuracy (87.39%), while exponential kernels struck a balance at 86.17%, demonstrating stability through clamping. ELU, despite its potential, faced numerical instability issues, highlighting the need for further refinement.
- **Hyperparameter Optimization:** The hyperparameter search process was instrumental in identifying optimal configurations. For Performer-ReLU, hidden dimensions of 960, 16 attention heads, and a dropout rate of 0.1 led to the best results. These parameters minimized overfitting while maintaining efficient computation. Optuna's optimization process identified a learning rate of 1×10^{-3} and a batch size of 64 as ideal for all Performer variants.

9.3 Limitations and Opportunities

- **Class-specific Challenges:** Confusion matrix analysis revealed challenges with visually similar classes, such as *cat* vs. *dog*, where accuracy dropped to 78%. This suggests the need for class-specific augmentation or feature extraction methods to improve differentiation.
- **Generalization Across Datasets:** While CIFAR-10 provided a controlled benchmark, its simplicity may limit generalizability. Preliminary testing on CIFAR-100 showed a drop in accuracy to 71.12% for Performer-ReLU, emphasizing the need to evaluate scalability on more complex datasets such as ImageNet.
- **Stability in Training:** Despite the clamping strategy ($[-10, 10]$), Performer-Exp occasionally exhibited training instability, requiring multiple restarts. This was mitigated by adding a stability term ($\epsilon = 10^{-6}$), but advanced regularization strategies should be explored.

9.4 Final Insights and Future Directions

This work reinforces the transformative potential of efficient attention mechanisms in deep learning. Performers, with their linear complexity, represent a paradigm shift for scalable Transformer architectures. However, optimizing activation functions and regularization strategies remains crucial to fully unlocking their potential.

Future research could focus on:

- **Integrating advanced kernel designs:** Exploring hybrid or learnable kernel functions for further performance gains.
- **Expanding datasets:** Benchmarking on larger datasets like ImageNet, where the scale may better highlight efficiency gains.
- **Automated tuning frameworks:** Leveraging AutoML to streamline hyperparameter optimization, potentially reducing manual effort and improving consistency.

With Performer-ReLU achieving the highest accuracy (87.39%) and Performer-Exp demonstrating near-equivalent performance (86.17%) at faster training times, these architectures offer practical solutions for both research and industry applications.

Acknowledgments

We would like to thank Dr. Krzysztof Choromanski for his support throughout this project. His guidance and the resources provided were invaluable to the success of this work. Additionally Dr. Choromanski's first-authored paper defining the field of performers, *Rethinking attention with performers*, was crucial in understanding and dissecting the architecture of many models we employed in our paper.

Appendices

- **Complete Source Code**: Available upon request or through the associated Git repository.
- **Datasets**: CIFAR-10 is publicly available and was used in accordance with guidelines.