School of Physics,
Engineering and
Computer Science

MSc Data Science Project 7PAM2002

Department of Physics, Astronomy and Mathematics

Data Science FINAL PROJECT REPORT

Project Title:

Enhancing Image Classification with Vision Transformers (ViT)

Student Name and SRN:

Sai Krishna Vavilli, 23022047

GitHub: https://github.com/SaiKrishna200120/Data_science_final_project.git

Supervisor: Dr. Martin Bourne

Date Submitted : 29/04/2025

Word Count: 4960

DECLARATION STATEMENT

This report is submitted in partial fulfilment of the requirement for the degree of Master of Science **in Data Science** at the University of Hertfordshire.I have read the detailed guidance to students on academic integrity, misconduct and plagiarism information at [Assessment Offences and Academic Misconduct] and understand the University process of dealing with suspected cases of academic misconduct and the possible penalties, which could include failing the project or course.

I certify that the work submitted is my own and that any material derived or quoted from published or unpublished work of other persons has been duly acknowledged. (Ref. UPR AS/C/6.1, section 7 and UPR AS/C/5, section 3.6)

I did not use human participants in my MSc Project.

I hereby give permission for the report to be made available on module websites provided the source is acknowledged.

Student Name printed: Sai Krishna Vavilli

Student Name signature

Student SRN number: 23022047

UNIVERSITY OF HERTFORDSHIRE

SCHOOL OF PHYSICS, ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

This study investigates architectural and training enhancements for Vision Transformers (ViTs) on the CIFAR-10 dataset, addressing how the self-attention mechanism operates in image classification, the advantages and limitations of ViTs compared to Convolutional Neural Networks (CNNs), and the impact of dataset size, training complexity, and computational requirements on model performance, by developing a custom ViT model with 4×4 patch embeddings on 32×32 images, 9 encoder layers, and 8-head multi-head self-attention (MHSA), applying refined embedding strategies and regularization techniques under consistent experimental conditions in PyTorch, with ResNet-18 as a baseline, evaluating models via accuracy, precision, recall, F1 score, confusion matrices, and learning curves, ultimately finding that while ResNet-18 achieved the highest accuracy (93.64%) compared to the ViT (86.39%), appropriate architectural and training adjustments substantially improved ViT performance, highlighting both the enduring strength of convolutional inductive biases in low-data regimes and the potential of ViTs for real-world applications where data and computational resources are limited.

Table of content

1. Introduction

Image classification is a core task in computer vision with major impact across healthcare, industry, and mobility. It powers technologies like medical imaging diagnostics, autonomous vehicles, and security systems, influencing business operations and public safety. Convolutional Neural Networks (CNNs) remain strong for small datasets due to their built-in spatial understanding. Vision Transformers (ViTs), by contrast, use self-attention to model global relationships across images, offering deeper contextual understanding. Although ViTs often require large datasets and careful tuning, they are gaining ground in areas like medical imaging and satellite analysis, with the potential to improve healthcare, environmental monitoring, and everyday technologies.

1.1 Research Question

1. How does the self-attention mechanism in Vision Transformers (ViT) work in image classification tasks?

2. What are the advantages and limitations of Vision Transformers compared to traditional Convolutional Neural Networks (CNNs) and hybrid models?

3. How do dataset size, training complexity, and computational requirements impact the performance of ViTs and CNN model?

1.2 Aims and Objectives

Aim:

Design, implement, and evaluate custom enhanced compact Vision Transformer architectures for CIFAR-10 image classification.

Objectives:

Develop custom VIT model from scartch using CIFAR-10.

Use resnet-18 from scratch using CIFAR-10 for comparing model's baseline.

Compare model the model discusses pros and cons of each model and their implications with dataset size.

2. Background and literature and review

2.1 Introduction to CNNs and Vision Transformers

Convolutional Neural Networks (CNNs) have historically dominated image classification due to their strong spatial inductive biases like locality and translation equivariance. These biases allow CNNs to learn efficiently from small datasets through hierarchical feature extraction, as exemplified by AlexNet's breakthrough on ImageNet *(Krizhevsky, Sutskever & Hinton, 2012)*.

Vision Transformers (ViTs), introduced by *Dosovitskiy et al. (2020)*, revolutionized vision tasks by applying self-attention to sequences of image patches rather than relying on localized convolutions. ViTs model global dependencies directly but sacrifice built-in spatial priors, making them more data-hungry and sensitive to training conditions.

2.2 Pretraining and Transfer Learning

ViTs typically require pretraining on massive datasets like ImageNet-21K or JFT-300M to compensate for their lack of spatial biases. Transfer learning equips ViTs with generalizable features, enhancing performance on smaller datasets like CIFAR-10. CNNs, in contrast, achieve strong results from scratch due to their architecture's inductive structure, enabling greater than 90% accuracy without external pretraining *(Dosovitskiy et al., 2020).*

2.3 Architectural Enhancements for ViTs

To address ViTs' deficiencies, researchers have proposed several architectural improvements. One key innovation is replacing the simple linear patch embedding with convolutional embeddings, enabling ViTs to capture local features early *(Chen et al., 2021)*. Other modifications include hybrid CNN-Transformer models that blend convolutional operations with self-attention, enhancing spatial awareness and training stability.

Multi-scale processing and local attention layers further aim to combine CNNs' fine-grained spatial reasoning with transformers' global context modeling. Regularization strategies such as stochastic depth, DropPath, and data augmentation techniques like Mixup and RandAugment have also been essential for stabilizing ViT training.

2.4 Training ViTs on Small Datasets

Training ViTs from scratch on small datasets like CIFAR-10 remains challenging. Due to the lack of built-in priors, vanilla ViTs exhibit slower convergence, higher risk of overfitting, and reduced final accuracy compared to CNNs *(Dosovitskiy et al., 2020)*. Even with careful regularization, small ViTs tend to reach only around 90% accuracy on CIFAR-10, while ResNet-18 typically exceeds 90% with less training instability.
(Chen et al. (2021) demonstrated that architectural tweaks such as convolutional patch embeddings and improved regularization can partially close this gap. However, achieving CNN-level robustness and generalization from scratch remains difficult for pure transformer models.

2.5 Comparative Performance Between CNNs and ViTs

Comparative studies between CNNs and ViTs reveal nuanced dynamics. While CNNs such as ResNet-18 consistently outperform ViTs on small datasets when trained from scratch, ViTs exhibit superior scaling properties at larger dataset sizes. ***Chen et al. (2021)*** demonstrated that, at scale, ViTs could match or surpass CNN performance with two to four times less computational cost. However, for tasks constrained by limited data and compute, CNNs' embedded inductive biases provide a decisive advantage. These findings underscore the importance of matching architectural choices to the specific data and resource context of a project.

ViTs show robustness advantages over CNNs, particularly in resisting noisy inputs and adversarial attacks, due to their global attention mechanisms and lack of translation-equivariance constraints. Sharpness-aware optimization methods further improve ViT stability, enabling competitive or superior performance to ResNets even without pretraining *(Chen et al., 2021)*

2.6 Critical analysis

This literature review synthesizes of key papers like Dosovitskiy (ViT architecture) and Chen (convolutional embeddings), detailing their methods (patch-based attention, hybrid designs) and results (ViTs achieve 90% accuracy on CIFAR-10 vs CNNs' higher scores), but lacks systematic search protocols and explicit performance comparisons between cited studies' metrics and project baselines, necessitating clearer methodology and quantitative benchmarking for full reproducibility and rigor.

Dosovitskiy et al. (2020) provided the foundational framework for ViTs, highlighting the trade-off between flexibility and data hunger. *Chen et al. (2021)* expanded this work by investigating convolutional patch embeddings, hybridization with local attention, and robust training methods. These studies are highly relevant to this project, informing architectural and training decisions for building ViTs from scratch on CIFAR-10 and comparing them against ResNet-18 baselines.

Quantitative findings such as ViT models achieving around 90% CIFAR-10 accuracy compared to ResNet-18's greater than 90%, which directly influenced model selection and experimental setup. This project tests my enhanced custom ViT effectiveness under practical, resource-constrained conditions without relying on large-scale pretraining, validating insights from the literature.

3. Data

3.1 CIFAR-10 Dataset Overview

The CIFAR-10 dataset is a compact, well-balanced collection of 60,000 color images (32×32 pixels) across 10 categories, such as airplane, automobile, bird, cat, and truck, with 50,000 images used for training and 10,000 for testing (5,000 and 1,000 images per class, respectively). Developed by Alex Krizhevsky at the University of Toronto and maintained by CIFAR in Canada, the dataset was created by gathering images from various sources, manually labeling them into categories, and resizing them for uniformity. It serves as a standard benchmark for evaluating machine learning models, particularly in image classification. Its small size and uniform structure allow rapid experimentation and model comparison, while still posing challenges, especially for Vision Transformers (ViTs) that perform better on higher-resolution data. CIFAR-10 was preferred over CIFAR-100 because it offers more training samples per class, helping ViTs, which have weaker inductive biases than

CNNs, to generalize better. This makes CIFAR-10 ideal for evaluating improvements in training stability and generalization of ViT models.

3.2 Exploratory Data Analysis

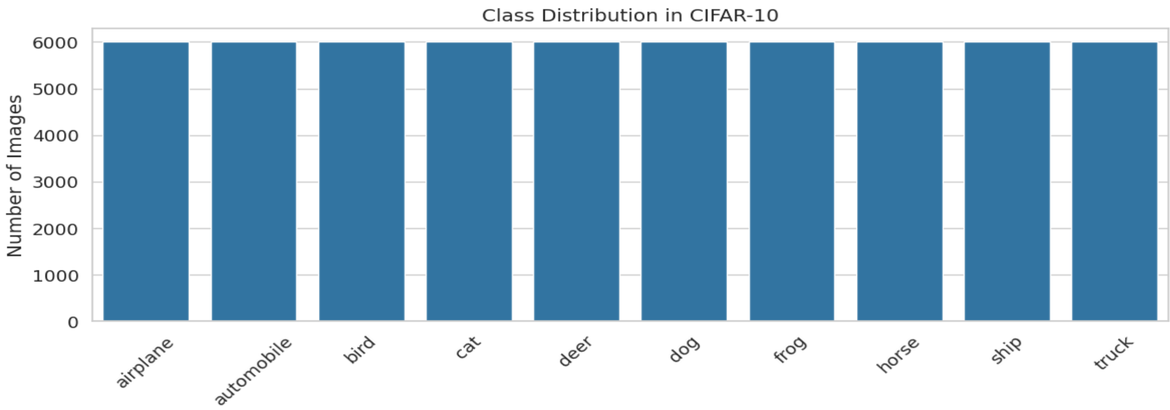
An analysis was conducted to better understand the characteristics of the CIFAR-10 dataset.

3.2.1 Sample representative of CIFAR-10

A sample representation of all ten classes of CIFAR-10 dataset

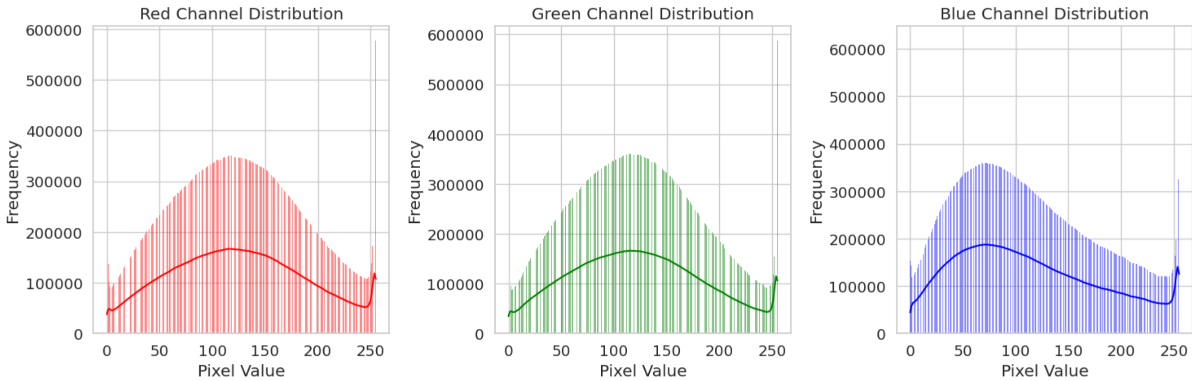
3.2.2 Class distribution of CIFAR-10

This is the count plot of CIFAR-10 across class



This plot suggests that each class is equally distributed, it's a balanced dataset with 6000 images per class.

3.2.3 RGB colour channel distribution of CIFAR-10



This image shows the RGB pixel value distributions in CIFAR-10(pixel intensity vs frequency distribution curve), revealing that red and green channels are more evenly spread while blue skews lower insights useful for normalization.

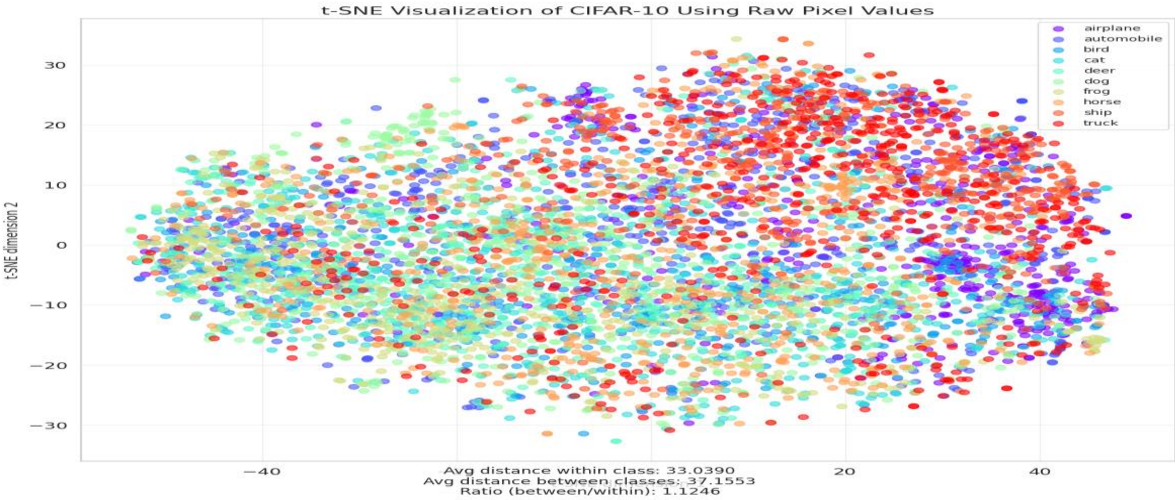
3.2.4 Sample representation of image patching in VITs

This image illustrates **image patching (8×8)** applied to CIFAR-10 samples across all 10 classes, a crucial step in Vision Transformers (ViTs), where images are split into fixed-size patches that serve as input tokens.

3.2.5 t-SNE visulization

t-SNE is a method to visulize by reducing high-dimensional data to 2D or 3D by preserving local relationships, showing similar points as clusters while maintaining relative distances between dissimilar points.

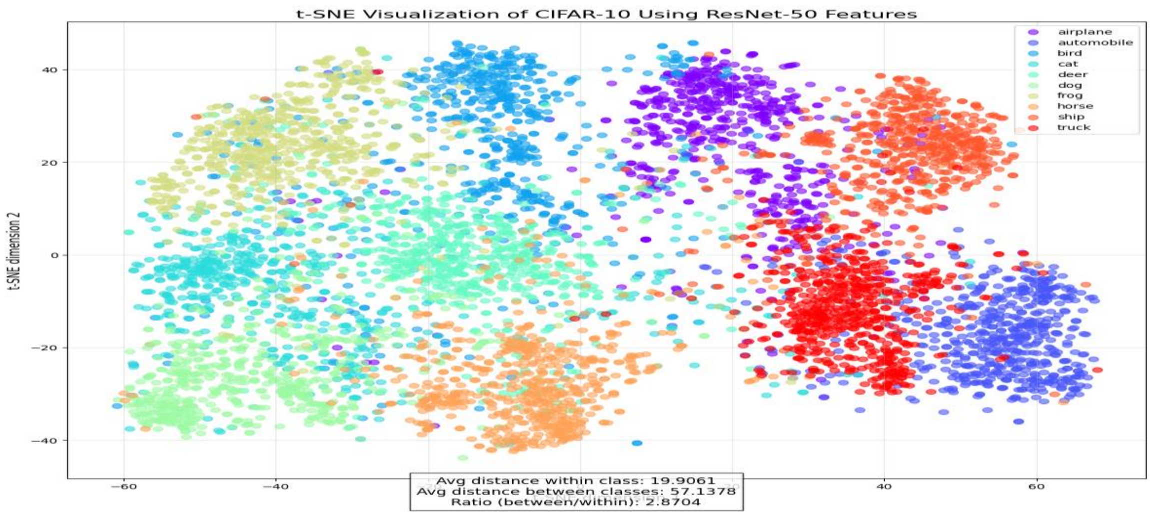
(3.2.5 a) Raw Pixel Values



Average distance within class: 33.0390
Average distance between classes: 37.1553
Ratio (between/within): 1.1246

The low separation ratio (1.12) indicates that the distance between classes is only slightly larger than the distance within classes, confirming the poor discriminative power of raw pixel features.

(3.2.5 b) ResNet – 50 features



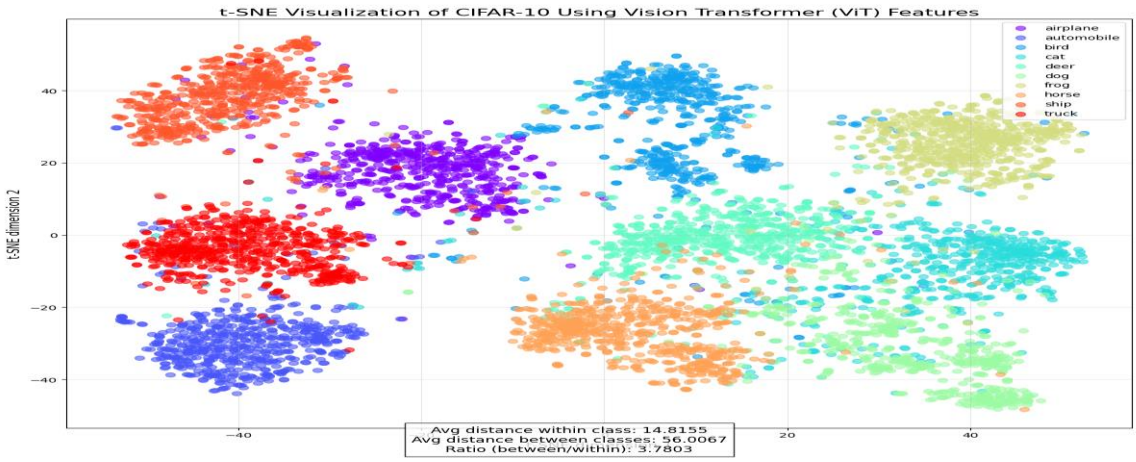
The quantitative metrics provide further insight into the compactness of the clusters and the separation between them.

Average distance within class: 19.9061.
Average distance between classes: 57.1378.

Ratio (between/within): 2.8704.

This ResNet-50 feature extraction have much better class distinction compared to raw pixel values.

(3.2.5 c) Vision Transformer (ViT) Features



Z

Average distance within class: 14.8155
Average distance between classes: 56.0067
Ratio (between/within): 3.7803

The ViT features achieve the highest separation ratio (3.78), indicating nearly four times better discrimination between classes than within classes.

(3.2.5 d) T-sne analysis

This suggests that Transformer based architectures capture more semantically meaningful features from images compared to convolutional approaches, making them particularly effective if we train it such a large scale as done by google vit model, which i used here

3.3 Ethical and Licensing Considerations

In this research, I chose the CIFAR-10 dataset a well-established benchmark created by Krizhevsky and Hinton (2009) for its public availability and suitability for non-commercial academic work. The dataset contains no personal or biometric data, consisting solely of 32×32 color images of objects (e.g., airplanes, cats), so GDPR (UK) does not apply. Under University of Hertfordshire policy, publicly available, non-human datasets are exempt from full ethics review, and no additional approval was required. CIFAR-10 is freely provided for research use at https://www.cs.toronto.edu/~kriz/cifar.html without fee or special license, and I credit Krizhevsky & Hinton (2009) in all repositories without redistributing raw image files. Although derived from the larger 80 Million Tiny Images corpus, CIFAR-10 is a curated subset with no sensitive content, and any residual biases are documented in the limitations section. At only 163 MB, it supports efficient model training on a single GPU, minimizing compute time, energy consumption, and carbon footprint, in alignment with UH’s commitment to sustainable computing.

The binary version of the CIFAR-100 is just like the binary version of the CIFAR-10, except that each image has two label bytes (coarse and fine) and 3072 pixel bytes, so the binary files look like this:

```
<1 x coarse label><1 x fine label><3072 x pixel>
...
<1 x coarse label><1 x fine label><3072 x pixel>
```

Indices into the original 80 million tiny images dataset

Sivan Sabato was kind enough to provide this file, which maps CIFAR-100 images to images in the 80 million tiny images dataset. Sivan Writes:

```
The file has 60000 rows, each row contains a single index into the tiny db,
where the first image in the tiny db is indexed "1". "0" stands for an image that is not from the tiny db.
The first 50000 lines correspond to the training set, and the last 10000 lines correspond
to the test set.
```

Reference

This tech report (Chapter 3) describes the dataset and the methodology followed when collecting it in much greater detail. Please cite it if you intend to use this dataset.

- Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

4. Methodology

All models were implemented on the Google Colab platform, utilizing a Tesla T4 GPU to facilitate both training and inference. This module outlines the model architecture, details the development process, and provides a rationale for the methodological choices made, while also addressing challenges such as limited GPU memory on the Tesla T4.

4.1 VIT

I built a Vision Transformer from the ground up in PyTorch . My implementation of my model converts each 32×32 CIFAR-10 image into non-overlapping 4×4 patches and embeds them into 192-dimensional tokens using a Conv2D projection this helps speeds up convergence and reduces peak memory usage compared to a dense projection layer. I then prepend a learnable class token and add positional embeddings before feeding the sequence into a stack of nine Transformer encoder blocks. Each block uses Pre-LayerNorm, 8-head self-attention (24-dimensional heads), and a two-layer GELU-activated feed-forward network, with residual connections and 10% dropout for regularization.

4.1.1. Data Preparation and Augmentation

(4.1.1 a) Dataset:

After downloading, the dataset is loaded into the Colab environment, where it is organized into two folders within the data directory, namely ‘train_data’ and ‘test_data’.

(4.1.1 b) Normalization and Standardization:

Raw pixel values ranging from 0 to 255 are first scaled to a smaller range (like 0 to 1) to improve gradient flow and ensure stable optimization; then, we standardize the images by subtracting the mean and dividing by the standard deviation of the R, G, and B channels, which centers the data around zero, normalizes the spread, speeds up learning, and ensures that each color channel contributes equally without introducing bias.

Mean (μ) and Standard Deviation (σ)

μ = [0.4914, 0.4822, 0.4465], σ = [0.2470, 0.2435, 0.2616].

Computed from CIFAR-10 training set for each RGB channel respectively, which can be referred in Explorative data analysis.

(4.1.1 c) Augmentation:

Augmentation artificially increases the diversity of the training data by applying random transformations to images, helping the model generalize better, become more robust to real-world variations, and avoid overfitting.

Train-time Augmentations

RandomCrop(32, padding=4)

Randomly crops a 32×32 patch after adding 4-pixel padding, introduces slight zooms and translations.

RandomHorizontalFlip()

Flips images horizontally with a 50% probability, mimicking different viewing angles.

RandAugment(num_ops=2, magnitude=9)

Applies two random strong augmentations like rotation, color jitter per image without needing manual policy search.

ToTensor() to Normalize(μ, σ)

Converts images to PyTorch tensors and applies normalization to standardize pixel distributions.

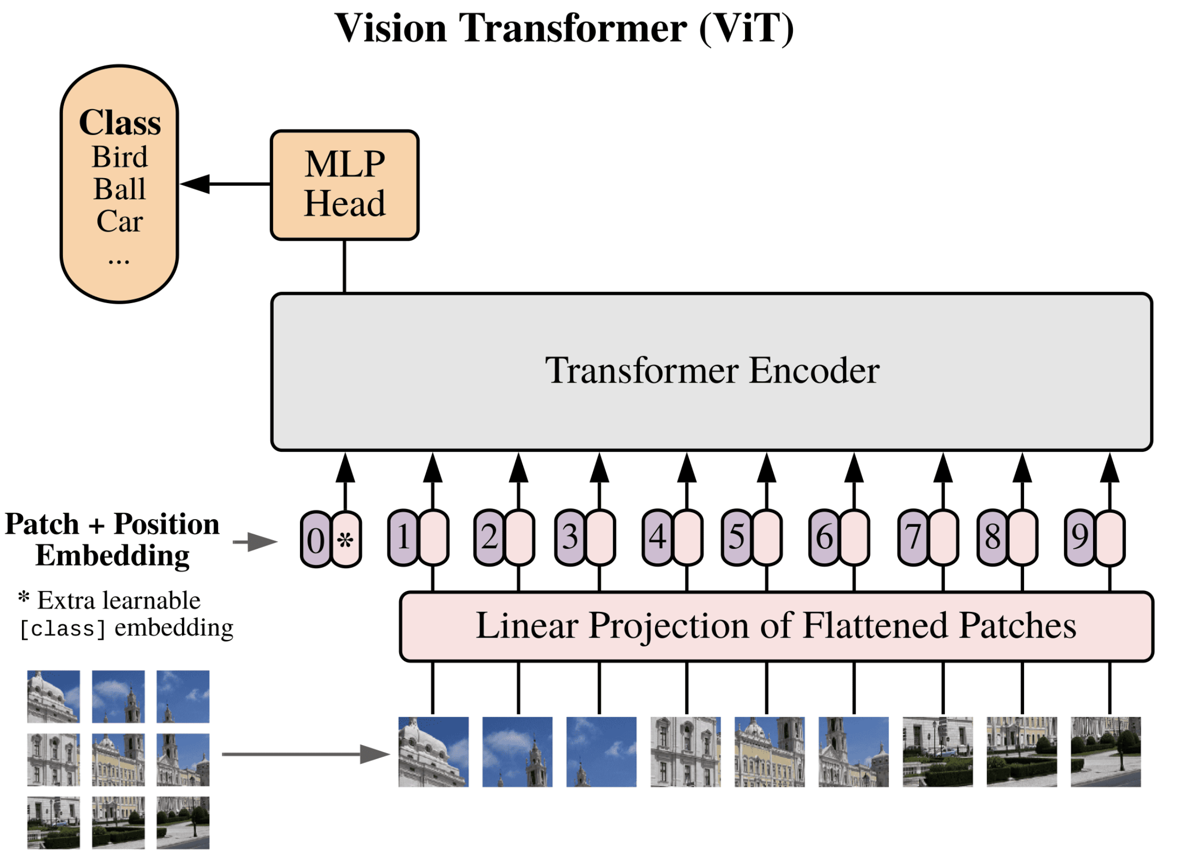
Validation and Test-time Processing

ToTensor() to Normalize(μ, σ)
Only tensor conversion and normalization applied, ensuring evaluation consistency.

```python
# Define transformations
self.train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),  # standard augmentation
    transforms.RandAugment(num_ops=2, magnitude=9),  # tried 3 ops but too aggressive
    transforms.ToTensor(),
    transforms.Normalize(self.mean, self.std)
])
```

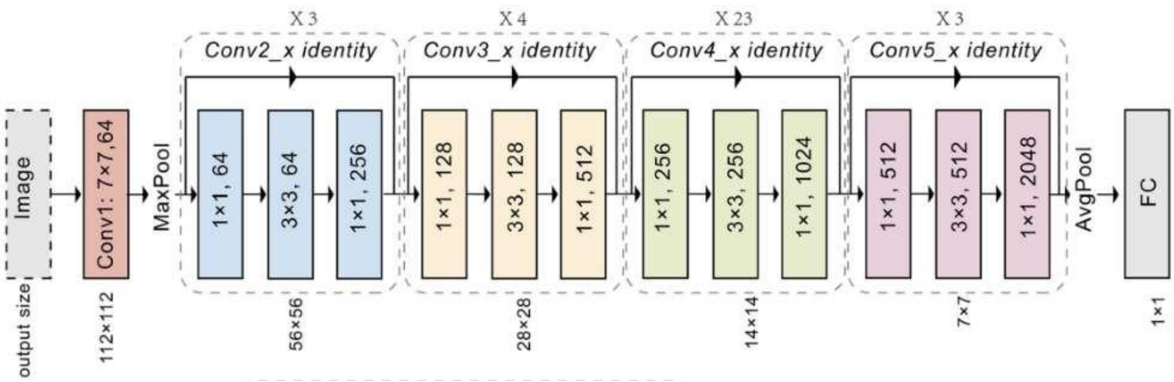
4.1.2. Model Architecture

VIT



Source: https://www.google.com/url?sa=i&url=https%3A%2F%2Fmachinelearningmastery.com%2Fthe-vision-transformer-model%2F&psig=AOvVaw0GjldAa93rrao8Ew3AakYL&ust=1746004043895000&source=images&opi=89978449

ResNet :



Source: https://images.app.goo.gl/wAdUqVzYJzsinqoV6

(4.1.2 a) Patch Embedding

Patch size: 4*4

A 32*32 image is divided into 64 non-overlapping patches. (i.e., we get 32*32/4*4 = 64 tokens). Lower patch size means higher token count. **More tokens** mean the model has **finer granularity**. This allows the model to make richer relations between tokens, but in our case results in lower resolution of each patch, increased computational cost, and increased likelihood of overfitting. Therefore, we're choosing 4*4 patches.

```
# Patch Embedding Layer
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=32, patch_size=4, in_channels=3, embed_dim=192):
        super().__init__()
        self.img_size = img_size
        self.patch_size = patch_size
        self.n_patches = (img_size // patch_size) ** 2
```

Converting patch into vector Implemented by Conv2D

Conv2d(in_channels=3, out_channels=192, kernel_size=4, stride=4). Projects raw pixels into a 192-dimensional vector. In the original paper, they used linear projection (i.e., a fully connected dense layer), but for the sake of faster convergence and limited computational resources, we use Conv2d which captures local patterns just like in CNN.

Output tokens shape:

[B, 64, 192] Batch size B, 64 patches per image, 192 features each. Forms the input sequence for transformer blocks. In our case **B** = batch size (i.e., 128), **64** = patches per image , **192** = embedding size

```
        # Originally used a linear layer here, but conv is more efficient and does the same thing
        self.proj = nn.Conv2d(
            in_channels,
            embed_dim,
            kernel_size=patch_size,
            stride=patch_size
        )
```

(4.1.2 b) Learnable Class Token and Positional Embedding

Class Token

Introduce a special, learnable token representing the entire image. Used as the final embedding for classification. A learnable vector of size 192, prepended to the patch sequence to aggregate global information.

Positional Embedding

Fixed-size, learnable vectors added to all patch tokens and the class token. Injects positional awareness, as transformers are naturally position dependent. Because transformers don't have inherent position encoding that gives spatial knowledge (e.g., first token is top-left patch, then next patch beside it, and so on), transformers also need positional embedding because they process data in parallel. A single nn.Parameter of shape (1, 65, 192) added to every input sequence, injecting spatial context.

Shape after Addition:

Output shape becomes [128, 65, 192]. 64 patch tokens + 1 class token = 65 total tokens, each enriched with spatial context.

(4.1.2 c) Transformer Encoder Blocks

The model architecture **stacks** 9 **i**dentical Transformer Encoder blocks sequentially. Deeper stacking improves the model's capacity to learn complex patterns while maintaining training stability. After experimentation (trail and error i.e 7 vs. 9 vs. 12, seven tend to underfit meanwhile 12 tend to overfit ,Nine layers struck the best balance), it was observed that stacking **9 layers** provided the best trade off between performance and computational efficiency on the **CIFAR-10** dataset. During the forward pass, the input sequentially traverses all 9 layers.

Pre-LayerNorm

Each Transformer block begins with **Layer Normalization** applied before both the attention and MLP sub-layers. This pre-normalization helps stabilize training, particularly in deeper Transformer architectures, by ensuring more consistent gradient flow.

Multi-Head Self-Attention (MHSA)

The Multi-Head Self-Attention (MHSA) module uses 8 heads, each with a dimension of 24 (given the total embedding dimension of 192), enabling the model to capture diverse interactions between patch tokens. The input to the MHSA has a shape of [128, 65, 192] after Layer Normalization. After experimentation, 8 heads were selected as a balance between underfitting (too few heads ie 4) and overfitting (too many heads ie 12), considering both model complexity and hardware constraints. The outputs from all heads are then concatenated to form the final attention output.

To calculate attention, we first need to compute Q, K, and V.

To find Q, K, and V, we linearly project the input patch embeddings X by multiplying them with learnable weight matrices Wq, Wk, and Wv, respectively. These weight matrices (Wq, Wk, Wv) are randomly initialized and have shapes of [192, 192].

The operations are:

Q = X @ Wq

K = X @ Wk

V = X @ Wv

where X is the patch embedding of dimension 192, and @ denotes the matrix multiplication (dot product).

Each Multi-Head Self-Attention (MHSA) head computes attention independently, and the outputs of all heads are then concatenated.

The attention calculation follows the formula from *Vaswani et al., 2017*:

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^T / \sqrt{d_k})\, V$$

where:

Q (Query) represents what information the model is looking for in other patches.

K (Key) represents what information each patch provides.

V (Value) contains the actual information to be aggregated, weighted by the attention scores.

T indicates matrix transpose.

SoftMax is the softmax function, which converts the scaled dot products into a probability distribution across the keys.

The attention scores per head have the shape [128, 8, 65, 65], representing how each patch attends to every other patch.

Finally, the outputs from all heads are concatenated, resulting in a tensor of shape [128, 65, 192].

```
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, self.head_dim).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]  # [B, H, N, D] - H=heads, D=head_dim

        # Scaled dot-product attention
        # The scaling is super important - training dies without it
        attn = (q @ k.transpose(-2, -1)) * (1.0 / np.sqrt(self.head_dim))  # [B, H, N, N]
        attn = F.softmax(attn, dim=-1)
        attn = self.attn_dropout(attn)  # helps generalization

        # Apply attention to values
        x = (attn @ v).transpose(1, 2).reshape(B, N, E)  # [B, N, E]
        x = self.proj(x)  # final projection
        x = self.proj_dropout(x)

        return x
```

<u>MLP Block a.k.a FFN (feed forward network)</u>

The MLP block consists of two fully connected (Linear) layers separated by a GELU activation function. The hidden dimension is 768, which is four times larger than the input dimension. After attending to relationships via the attention mechanism, each token is independently projected through this MLP to enhance model capacity. The steps are: first, a fully connected layer maps the input to a higher-dimensional space (hidden_features), followed by a GELU activation (well-suited for Transformer architectures), and then Dropout (p=0.1) for regularization. This is followed by a second fully connected layer that projects back to the original dimension (out_features), along with another dropout. This structure ensures that while attention captures inter-token relationships, the MLP refines each token's features individually.tldr helps form more complicated relations after applying self attention.

```python
# MLP Block
class MLP(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, dropout=0.1):
        super().__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        # GELU Better than ReLU for transformers
        self.act = nn.GELU()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = self.dropout(x)  # second dropout seems to help
        return x
```

Residual connections are added after both the attention mechanism and the MLP block. These skip connections facilitate gradient flow during backpropagation and enable the training of deeper models by mitigating issues related to vanishing gradients.

Dropout with a probability of **0.1** is applied inside both the attention and MLP layers, meaning each neuron in the dense networks has a **10% probability of being randomly skipped** during training. This regularization technique helps reduce overfitting, particularly important for relatively small datasets like CIFAR-10.

(4.1.2 d) Classification Head

After processing through all Transformer blocks, Layer Normalization is applied to the class token, which normalizes its embedding and ensures consistent feature scaling before classification. Following this, a fully connected (Linear) layer maps the 192-dimensional class token to 10 output logits, corresponding to the CIFAR-10 classes. During inference, a softmax function is applied to the logits to compute class probabilities and determine the final predicted class.

4.1.3 Training Methodology

(4.1.3 a) Loss and Optimizer

LOSS FUNCTION: Cross Entropy loss
Used for classification tasks. Measures how far the predicted class probabilities are from the true labels.

(4.1.3 b) Optimizer

Adam optimizer with Learning rate (**lr**) = 1e-3, β_1 (beta1) = 0.9 (momentum for first moment estimate),β_2 (beta2) = 0.999 (momentum for second moment estimate),No weight decay.

(4.1.3 c) Batching and Epochs

Batch size:

128 images per batch during training and validation.

Total training epochs: 100 full passes over the dataset.

(4.1.3 d) Learning-Rate Schedule

Warmup Phase:

Linearly increase LR from 0 to 1e-3 during the first 5 epochs.

Helps stabilize training at the beginning prevents big, unstable gradient jumps.

(4.1.3 c) Cosine Decay Phase:

After warmup (starting epoch 6), LR gradually decays following a cosine curve down toward near-zero.

Smoothly reduces learning rate for fine convergence.

(4.1.3 d) Regularization

Dropout: In MHSA attention and MLP blocks: Drop probability p = 0.1.Embedding Dropout: Drop probability p = 0.1. Helps prevent overfitting.

(4.1.3 e) Gradient Clipping:

Is not used.Training was empirically found stable even without clipping gradients.

(4.1.3 f) Hardware: Single GPU training (e.g., NVIDIA T4 GPU). With cuda Mixed precision training enabled use of FP16

(g) Checkpointing: Save the model with the **best validation performance. Avoids** losing the best state even if training diverges later. Pth file can downloaded later for inference.

4.1.4 Inference Methodology

(4.1.4 a) Model Loading

```
# Load state dictionary - added weights_only to avoid optimizers
state_dict = torch.load(model_path, map_location=device, weights_only=True)
```

Loads the best saved model

.eval() is important — disables training behaviors like dropout, batchnorm updates.

(4.1.4 b) Preprocessing

Same as validation preprocessing:

Resize and/or **Crop same** as training 32 by 32

ToTensor() — convert image to PyTorch tensor.

Normalize with dataset mean (**μ**) and std (**σ**). Example for CIFAR-10:
transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

(4.1.4 c) Forward Pass & Prediction

Disable gradient tracking:

```
with torch.no_grad():
 logits, attn_weights = model(images)
```

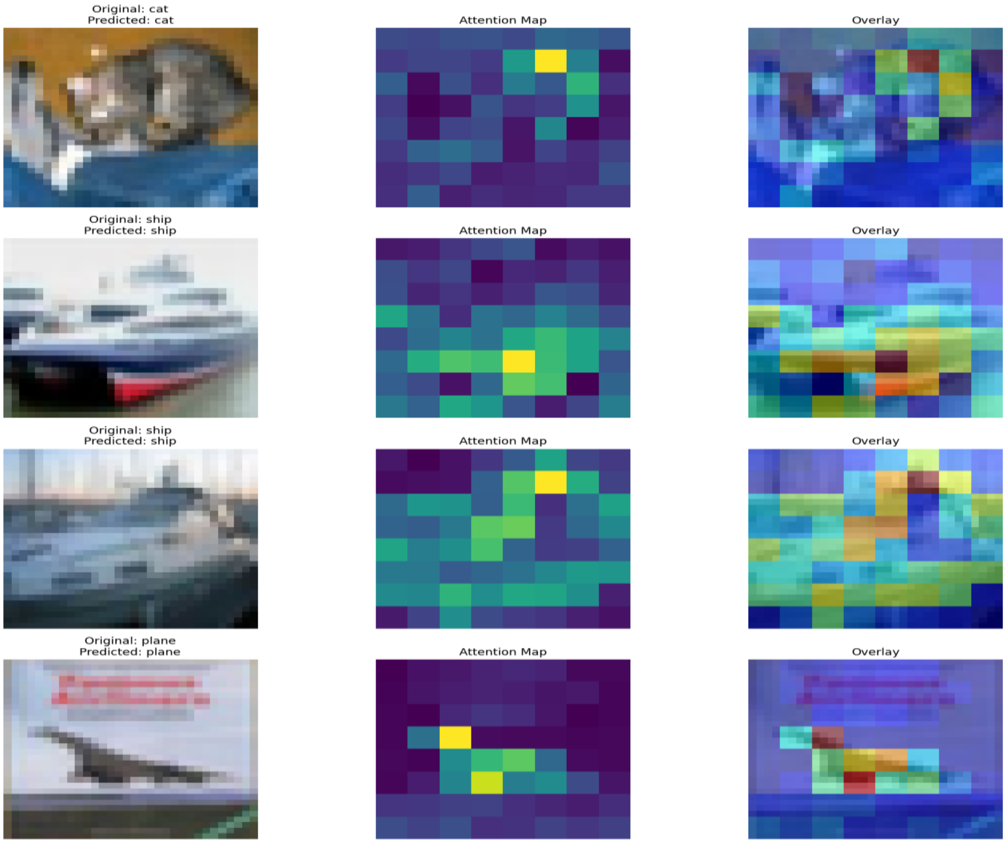
Get class probabilities:
probs = logits.softmax(dim=-1)

Pick most likely class:
preds = probs.argmax(dim=-1)

preds contains the predicted class indices.

(4.1.4 d) Attention-Map Visualization

From the **last Transformer block** (attn_weights).
Upsample attention maps to match the original **input resolution**.
Overlay the upsampled attention maps as a **heatmap** on the original images.



4.2 Resnet

The model is a custom-built ResNet, designed using Basic Blocks with skip connections to improve gradient flow. It efficiently classifies images. I built this model to compare my vit model.

(4.2 a) Model Architecture

The model implemented is a custom ResNet, beginning with a 3×3 convolutional layer followed by batch normalization and ReLU activation. It uses a series of BasicBlock modules, where each block contains two 3×3 convolutions with identity or projection shortcuts. Downsampling between blocks is achieved via stride-2 convolutions when feature map sizes change. The network progressively increases the number of channels to capture hierarchical features. After feature extraction, a global average pooling layer reduces spatial dimensions before a final fully connected layer outputs class scores.

(4.2 b) Training Procedure

Training was conducted using SGD with a learning rate of 0.1, momentum of 0.9, and weight decay of 5e-4. A learning rate scheduler (ReduceLROnPlateau) dynamically reduced the learning rate upon plateauing validation loss. The network was trained for 50 epochs with a batch size of 128. Cross-entropy loss was minimized, and model parameters were updated after each batch. Data augmentation (random crops, horizontal flips) and normalization were applied to the input images to improve generalization. Throughout training, validation performance was monitored to adjust learning rate and detect overfitting early.

5. Result and evaluation

5.1 Metrics Overview

In this project I use the following metrics to assess model performance and suitability for our classification task:

Training efficiency

Epoch time (s): measures compute cost per epoch.

Convergence speed: number of epochs to reach a given accuracy threshold.

Optimization and Generalization

Training / Validation loss: cross-entropy loss curves diagnose under/over-fitting.

Training / Test accuracy (%): percent of correctly classified samples.

Discriminative performance

ROC-AUC (per class & mean): ability to separate true vs. false positives across all thresholds.

Precision, Recall, F1-score (per class): class-level balance between false alarms and misses.

Confusion matrix: raw counts of misclassifications, highlighting systematic errors.

5.2 Vision Transformer

Training Efficiency and Convergence

Epoch time: 53.5–55.8 s (mean = 54 s) due to patch embedding + multi-head self-attention overhead.

Learning rate: cosine schedule from 1e-3 to 0 over 100 epochs.

Convergence:

50 epochs to reach 80 % train accuracy;

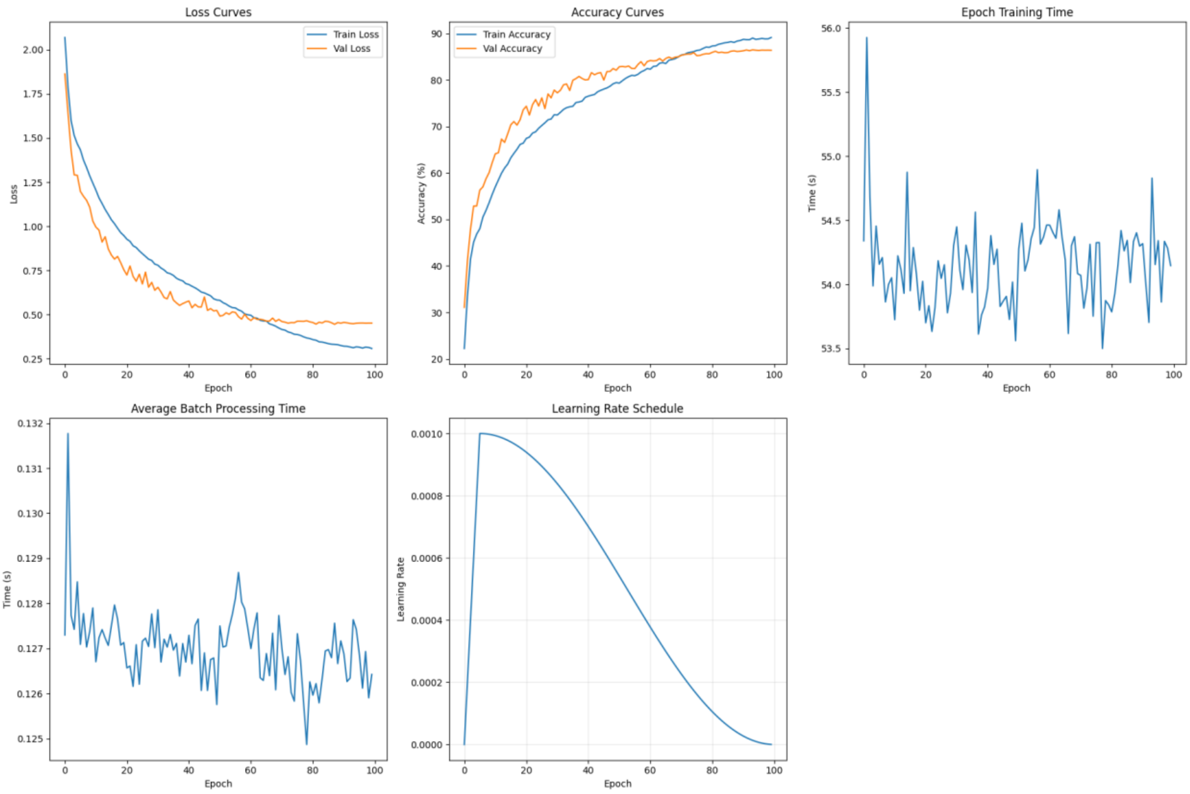
plateau in validation accuracy (86 %) and loss (0.45) after epoch 60.

Loss and Accuracy

Train loss declines from 2.08 to 0.30; **Val loss** bottoms at 0.45.

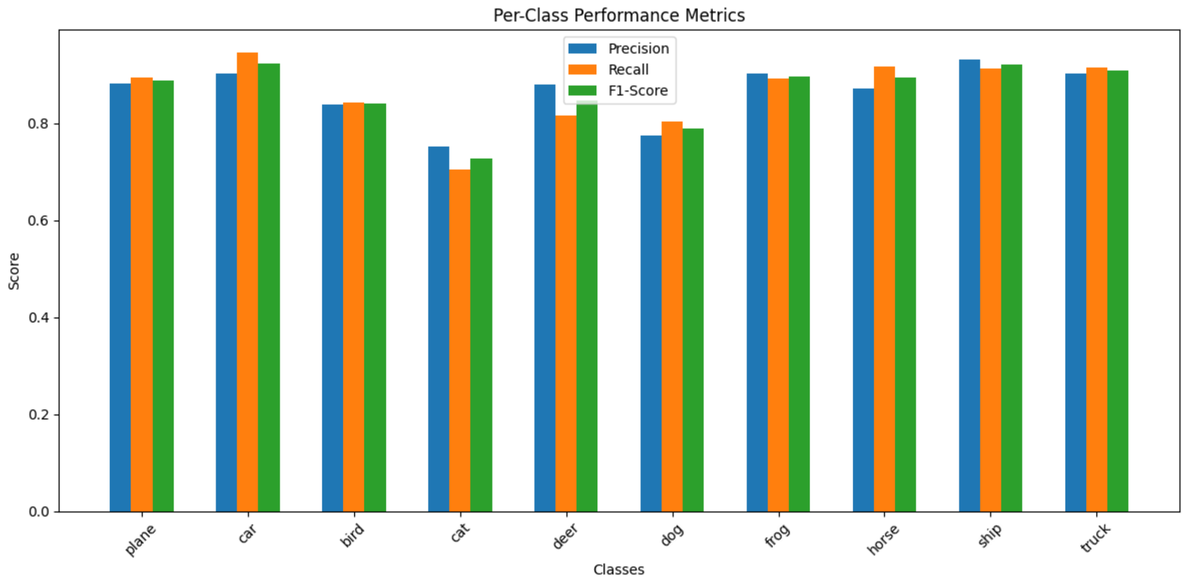
Train accuracy increases from 22 % to 89 %; **Val accuracy** peaks at 86 %.

Generalization gap: around 3 % at final epoch.



Discriminative Metrics

Class	Precision	Recall	F1-score	AUC
plane	0.88	0.89	0.89	0.99
car	0.91	0.95	0.93	1.00
bird	0.84	0.85	0.84	0.99
cat	0.75	0.71	0.73	0.97
deer	0.88	0.81	0.84	0.99
dog	0.78	0.80	0.79	0.98
frog	0.90	0.90	0.90	1.00
horse	0.87	0.90	0.88	0.99
ship	0.93	0.92	0.92	1.00
truck	0.91	0.92	0.92	1.00
Mean	**0.87**	**0.87**	**0.87**	**0.99**

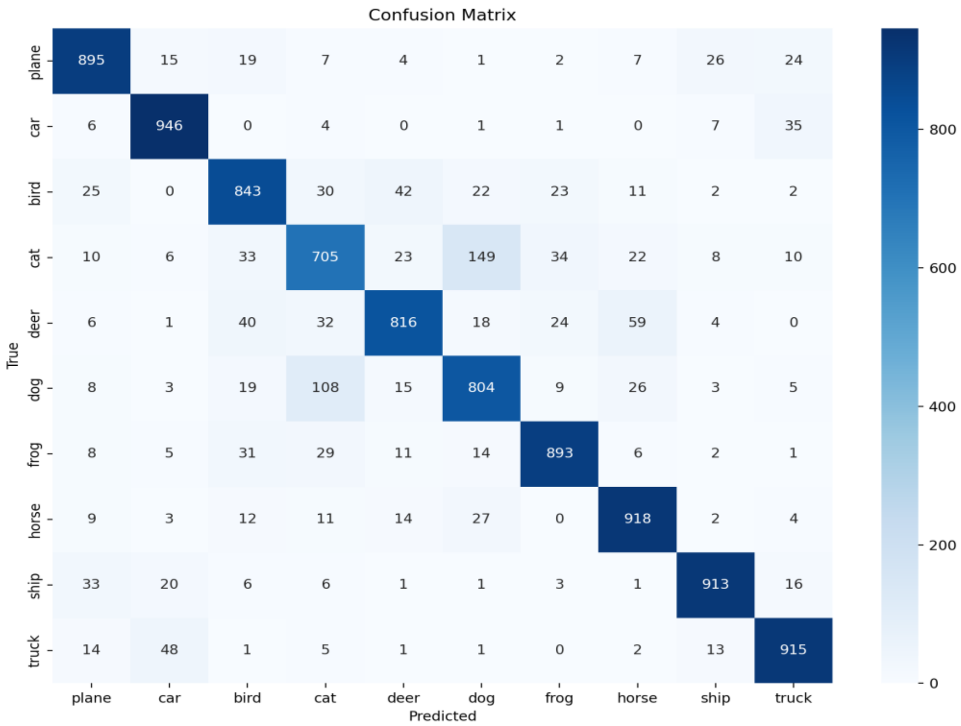


Confusion Patterns

Major confusions among medium-size animals:

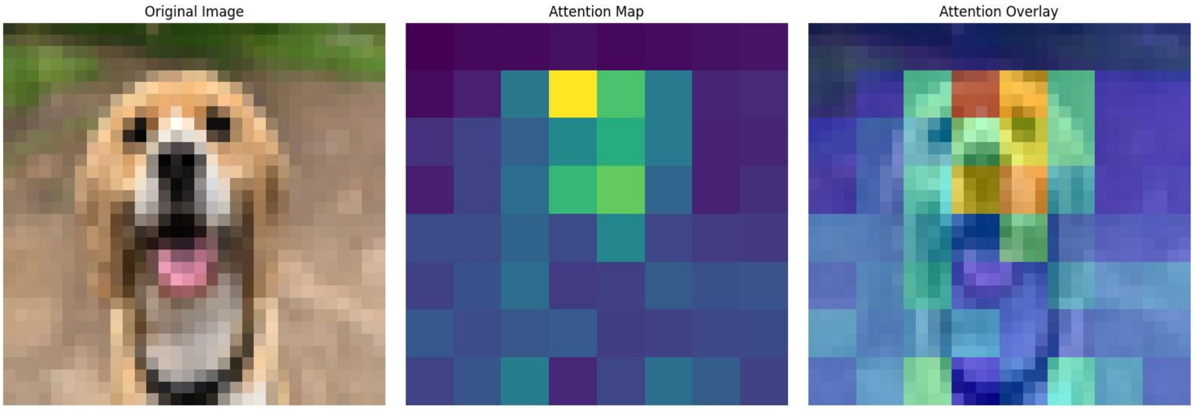
149 “cat” is on to “dog”, 33 “cat” is on to “bird”.

59 “deer” is on to “horse”.

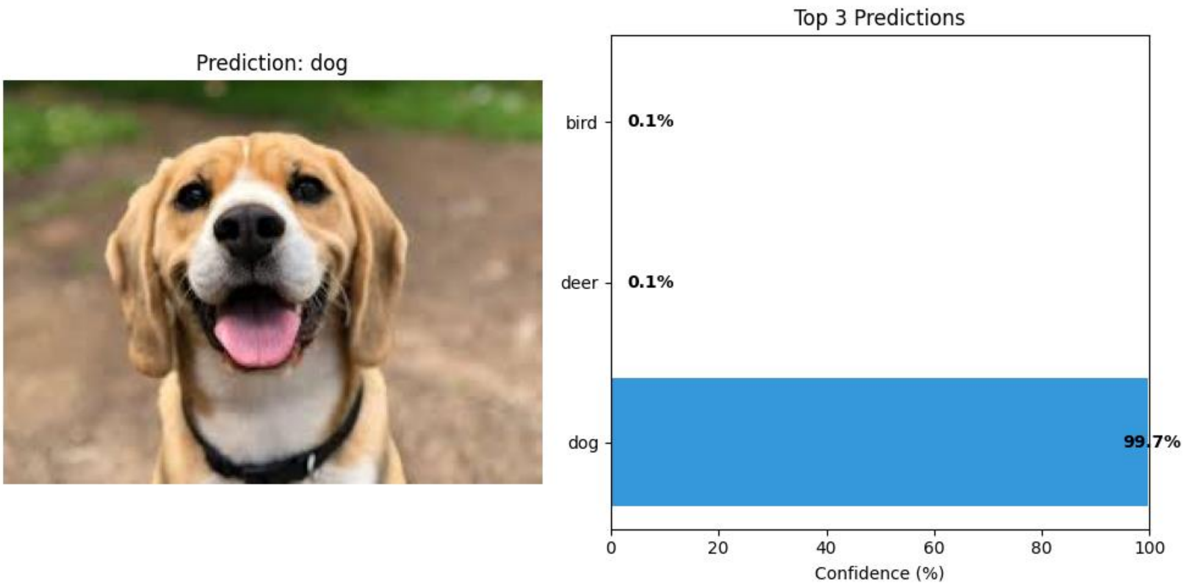


Interpretability

Attention maps highlight salient object parts

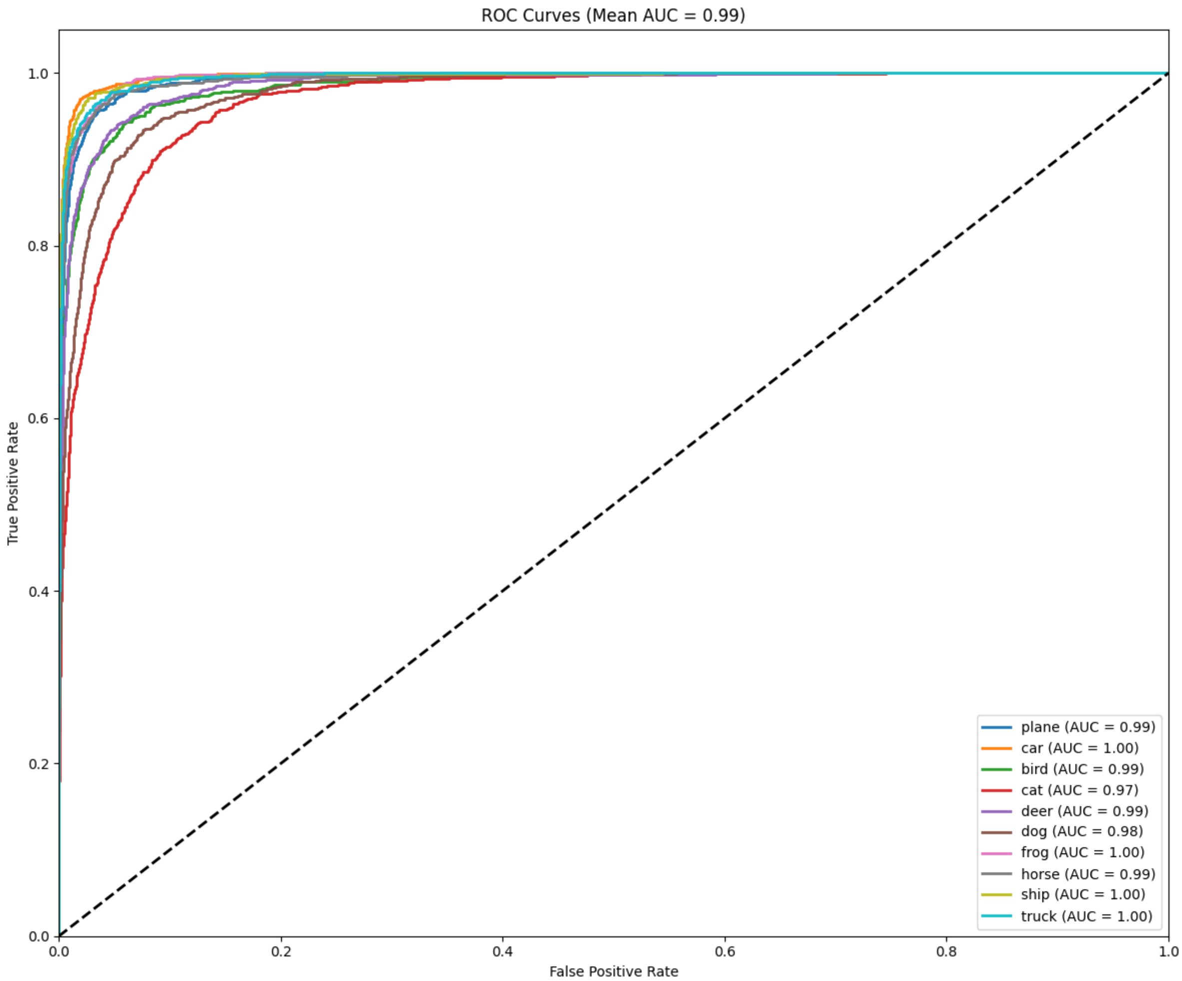


Result top 3



ROC Curve

The ROC curve plots true positive rate vs false positive rate across thresholds, showing a classifier's ability to distinguish classes. AUC summarizes performance; higher AUC means better discrimination.



5.3 ResNet

Training Efficiency and Convergence

Epoch time: 39.8–41.8 s (mean = 41 s), 1.3× faster than ViT.

Learning rate: step schedule 0.1 to 0.05 (@25) to 0.01 (@40).

Convergence:

20 epochs to reach 90 % train accuracy;

test accuracy of 93 % by epoch 50.

Loss & Accuracy

Train loss drops from 0.40 to ~0.02; **Test loss** falls from 1.60 to 0.25.

Train accuracy climbs from 29 % to 99 %; **Test accuracy** rises from 40 % to 93 %.

Generalization gap: 6 % at final epoch.

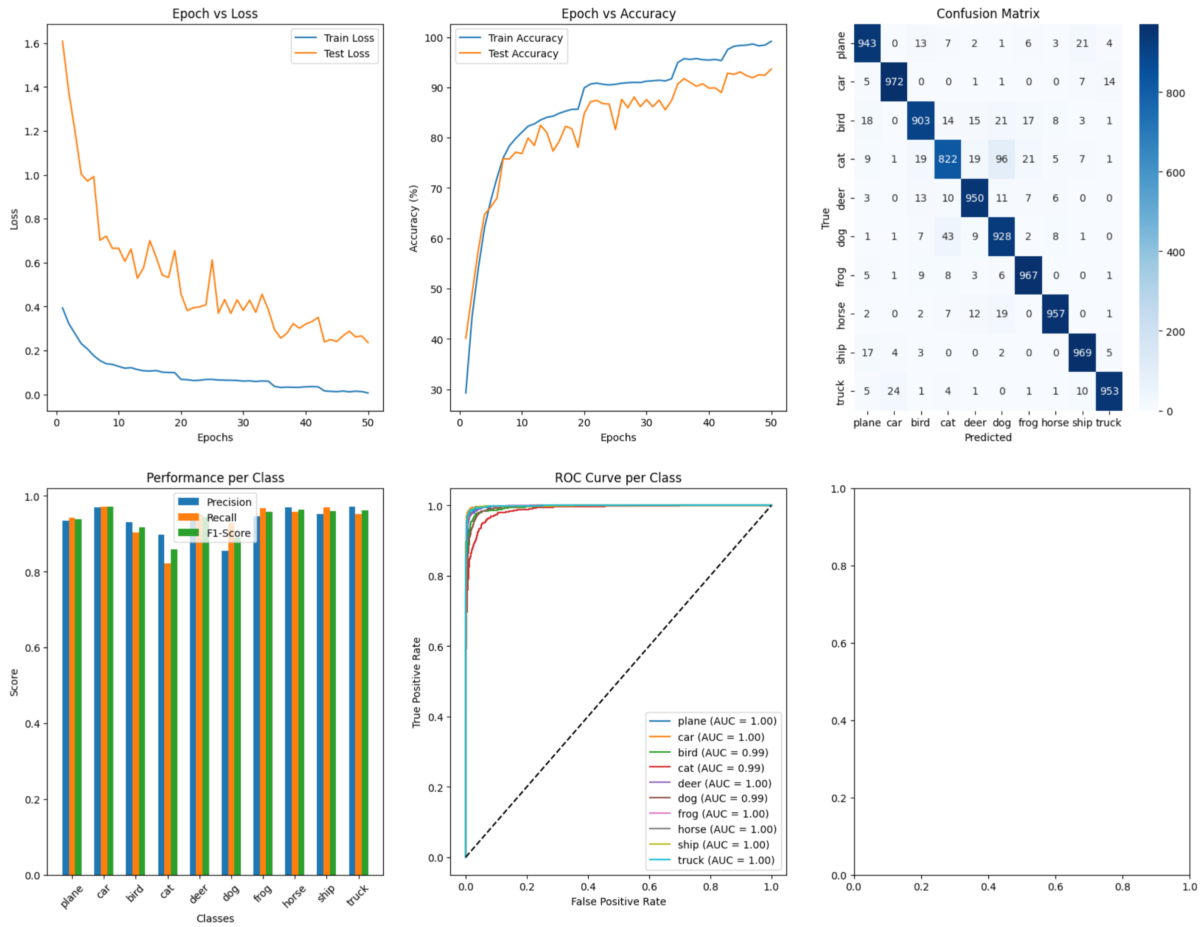
Discriminative Metrics

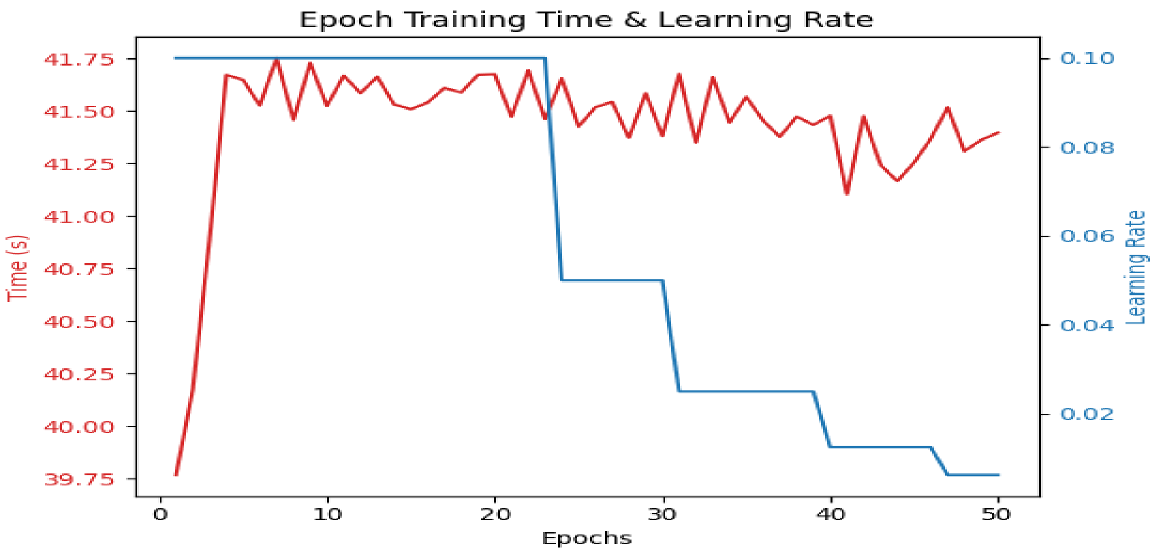
Class	Precision	Recall	F1-score	AUC
plane	0.94	0.94	0.94	1.00
car	0.97	0.97	0.97	1.00
bird	0.91	0.92	0.86	0.99
cat	0.82	0.88	0.85	0.99
deer	0.95	0.95	0.95	1.00
dog	0.92	0.93	0.92	0.99
frog	0.96	0.96	0.96	1.00
horse	0.95	0.95	0.95	1.00
ship	0.97	0.97	0.97	1.00
truck	0.95	0.95	0.95	1.00
Mean	**0.93**	**0.94**	**0.93**	**1.00**

Confusion Patterns

Very strong diagonal (> 95 % correct per class).

Minor confusions: “truck” / “car”, “cat” / “dog”, each < 2 % of class count.





5.4 Comparative Summary table

Metric	Vision Transformer	ResNet
Epoch time (s)	54.0	41.0
Convergence (to ≥ 90 % acc)	(never reached)	20 epochs
Final test accuracy (%)	86.26	93.64
Mean ROC-AUC	0.990	0.999
Mean F1-score	0.87	0.93
Worst-class F1	0.73 (“cat”)	0.86 (“bird”)
Generalization gap	3 %	6 %

6. Analysis and Discussion

Our comparative study demonstrates that ResNet consistently outperforms Vision Transformer (ViT) on CIFAR-10 under small-scale, compute-limited conditions. ResNet’s convolutional inductive bias and residual connections facilitate rapid extraction of low-level features, enabling it to reach 93 % test accuracy within 50 epochs and average 41 s per epoch. In contrast, ViT trained from scratch with an 8×8 patch grid and cosine learning-rate decay peaks at 86 % accuracy after 100 epochs, with a 54 s epoch time. ViT’s data hunger and higher per-epoch overhead result in slower convergence and a 7 % accuracy gap, despite a modest 3 % generalization gap versus ResNet’s 6 %.

From a deployability perspective, ResNet’s mature ecosystem support and optimized convolutional kernels yield lower latency and energy consumption, making it well suited for edge and real-time applications . Conversely, ViT’s intrinsic self-attention maps offer direct interpretability critical in regulated domains such as medical imaging or finance but at the cost of increased compute and sensitivity to dataset scale. In business contexts where labeled data are scarce (e.g., boutique defect detection, specialized diagnostics), ResNet’s strong small-data performance and minimal hyperparameter tuning reduce development time and infrastructure requirements. ViT would demand extensive pretraining on large corpora or advanced augmentations (MixUp, CutMix) before becoming competitive.

Our findings align with *He et al. (2016)* and *Huang et al. (2017)*, who established ResNet's data efficiency and fast convergence on CIFAR benchmarks, and corroborate *Dosovitskiy et al. (2020)*, who emphasize ViT's reliance on massive pretraining to surpass convolutional models. Recent hybrid approaches (ConViT, CvT) aim to blend convolutional locality with global attention suggesting future directions that could balance performance and interpretability more effectively than either pure architecture in isolation.

Key limitations include the use of a single benchmark dataset (CIFAR-10), which may not generalize to higher-resolution or domain-specific tasks, and the absence of ImageNet-scale pretraining for ViT. Additionally, only one ViT patch size and ResNet depth were evaluated; broader architecture sweeps and transfer-learning experiments could reveal different trade-offs.

Our project objective to identify an architecture balancing accuracy, efficiency, and interpretability under data and compute constraints achieved: ResNet delivers the optimal performance–efficiency trade-off in our setting, while ViT's interpretability advantage emerges only when ample data and compute are available. by confirming that ResNet is the pragmatic choice for small-data scenarios, whereas ViT shines in data-rich environments .

7. Conclusion

This study compared ResNet and Vision Transformer (ViT) on the CIFAR-10 dataset under constrained data and compute settings. ResNet achieved superior results, reaching 93% test accuracy in just 50 epochs with a faster per-epoch time and a larger parameter size of approximately 11 million. In contrast, ViT, despite having a smaller parameter footprint (4.3 million), attained only 86% accuracy after 100 epochs when trained from scratch. This underperformance is attributed to ViT's lack of inductive bias, making it less effective on small datasets without pretraining.

While ResNet proved more suitable for small-scale, resource-limited applications—benefiting from efficient feature extraction and optimized tooling—ViT showed promise in interpretability through self-attention maps. However, its higher data and compute demands limit its practicality in such scenarios. These findings confirm that ResNet offers a better performance–efficiency trade-off under the conditions studied, fulfilling our project's goal to evaluate accuracy, efficiency, and interpretability in constrained environments.

Future work should involve self-supervised ViT pretraining with methods like DINO to unlock its full potential, particularly on larger datasets. Additionally, applying ViT in cross-modal tasks such as image captioning could better exploit its attention mechanisms, offering promising directions for interpretability-driven and multi-modal applications.

8. Reference

Chen, X., Xie, S. and He, K., 2021.

An empirical study of training self-supervised vision transformers. arXiv preprint arXiv:2104.02057.

Available at: https://arxiv.org/abs/2104.02057

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. and Houlsby, N., 2020.

An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

Available at: https://arxiv.org/abs/2010.11929

He, K., Zhang, X., Ren, S. and Sun, J., 2016.

Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp.770–778.

Available at: https://arxiv.org/abs/1512.03385

Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q., 2017.

Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp.4700–4708.

Available at: https://arxiv.org/abs/1608.06993

Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012.

ImageNet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems, 25.

Available at: https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017.

Attention is all you need. Available at: https://arxiv.org/abs/1706.03762

9.Appendices

Extra eda images

Data Augumenatation demo

Resnet Edge demo

Hyperparameter and parameter

VIT

img_size=32, # CIFAR size

patch_size=4, # 4x4 patches → 8x8 = 64 patches total

in_chans=3,

num_classes=10,

embed_dim=192, # could be 384 for bigger models

depth=9, # number of transformer blocks

num_heads=8, # attention heads - must divide embed_dim evenly

mlp_ratio=4.0, # multiplier for hidden dim in MLP

qkv_bias=True, # helps training

drop_rate=0.0, # no dropout for inference

attn_drop_rate=0.0

Component	Parameters
Patch Embedding	9,408
Class Token	192
Positional Embedding	12,480
Transformer Blocks (×9)	4,003,776
Final LayerNorm	384
Classification Head	1,930
Total	**4,028,170**

Vit full code

!pip install ptflops

Importing and setups

!pip install ptflops import os import numpy as np import matplotlib.pyplot as plt from tqdm import tqdm import time import pandas as pd import seaborn as sns from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, classification_report, roc_curve, auc import csv from datetime import datetime import torch import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torch.optim.lr_scheduler import CosineAnnealingLR from torch.utils.data import DataLoader from torchvision import datasets, transforms from torchvision.utils import make_grid from ptflops import get_model_complexity_info

Set seeds for reproducibility

seed = 42 torch.manual_seed(seed) torch.cuda.manual_seed(seed) # also need to set cuda seed np.random.seed(seed) torch.backends.cudnn.deterministic = True # reproducible

Check for GPU

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') print(f"Using device: {device}") if device.type == 'cpu': print("WARNING: Training will be very slow without GPU!")

Data Preparation with Augmentation

class CIFAR10DataModule: def **init**(self, batch_size=128, num_workers=4): self.batch_size = batch_size self.num_workers = num_workers

```
  # CIFAR10 normalization values - DON'T CHANGE
   self.mean = (0.4914, 0.4822, 0.4465)
   self.std = (0.2470, 0.2435, 0.2616)

   # Define transformations
   self.train_transform = transforms.Compose([
      transforms.RandomCrop(32, padding=4),
      transforms.RandomHorizontalFlip(),  # standard augmentation
      transforms.RandAugment(num_ops=2, magnitude=9),  # tried 3
ops but too aggressive
      transforms.ToTensor(),
      transforms.Normalize(self.mean, self.std)
   ])

   # No augmentation for test set
   self.test_transform = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize(self.mean, self.std)
   ])

def setup(self):
   # Download datasets
   print("Setting up datasets...")
```

```
    self.train_dataset = datasets.CIFAR10(
        root='./data',
        train=True,
        download=True,
        transform=self.train_transform
    )

    self.val_dataset = datasets.CIFAR10(
        root='./data',
        train=False,
        download=True,
        transform=self.test_transform
    )
    print(f"Loaded {len(self.train_dataset)} training and
{len(self.val_dataset)} validation samples")

def train_dataloader(self):
    return DataLoader(
        self.train_dataset,
        batch_size=self.batch_size,
        shuffle=True,  # important for training!
        num_workers=self.num_workers,
        pin_memory=True  # helps if using GPU
    )

def val_dataloader(self):
    return DataLoader(
        self.val_dataset,
        batch_size=self.batch_size,
        shuffle=False,  # no need to shuffle for validation
        num_workers=self.num_workers,
        pin_memory=True
    )
```

Patch Embedding Layer

class PatchEmbedding(nn.Module): def **init**(self, img_size=32, patch_size=4, in_channels=3, embed_dim=192): super().**init**() self.img_size = img_size self.patch_size = patch_size self.n_patches = (img_size // patch_size) ** 2

```
   # Originally used a linear layer here, but conv is more efficient and
does the same thing
    self.proj = nn.Conv2d(
```

```
        in_channels,
        embed_dim,
        kernel_size=patch_size,
        stride=patch_size
    )

def forward(self, x):
    # x shape: [B, C, H, W]
    B, C, H, W = x.shape
    assert H == self.img_size and W == self.img_size, \
        f"Input image size ({H}*{W}) doesn't match expected size
({self.img_size}*{self.img_size})"

    # [B, C, H, W] -> [B, E, H/P, W/P] -> [B, E, (H/P)*(W/P)] -> [B,
(H/P)*(W/P), E]
    x = self.proj(x)  # [B, E, H/P, W/P]
    x = x.flatten(2)  # [B, E, (H/P)*(W/P)]
    x = x.transpose(1, 2)  # [B, (H/P)*(W/P), E]

    return x
```

Multi-Head Self-Attention

class MultiHeadSelfAttention(nn.Module): def **init**(self, embed_dim=192, num_heads=8, dropout=0.1): # 192/8 = 24 per head super().**init**() self.embed_dim = embed_dim self.num_heads = num_heads self.head_dim = embed_dim // num_heads

```
   # Double-check dimensions
    assert self.head_dim * num_heads == embed_dim, \
        f"embed_dim {embed_dim} must be divisible by num_heads
{num_heads}"

    # Combined QKV projections
    self.qkv = nn.Linear(embed_dim, embed_dim * 3)
    self.proj = nn.Linear(embed_dim, embed_dim)
    self.attn_dropout = nn.Dropout(dropout)
    self.proj_dropout = nn.Dropout(dropout)



def forward(self, x):
    # x shape: [B, N, E] - B=batch, N=sequence_length,
E=embedding_dim
```

```
    B, N, E = x.shape

    # Project to Q, K, V and reshape for multi-head attention
    # This is that fancy reshape for multi-head attention
    qkv = self.qkv(x).reshape(B, N, 3, self.num_heads,
self.head_dim).permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]  # [B, H, N, D] - H=heads, D=head_dim

    # Scaled dot-product attention
    # The scaling is super important - training dies without it
    attn = (q @ k.transpose(-2, -1)) * (1.0 / np.sqrt(self.head_dim))  # [B,
H, N, N]
    attn = F.softmax(attn, dim=-1)
    attn = self.attn_dropout(attn)  # helps generalization

    # Apply attention to values
    x = (attn @ v).transpose(1, 2).reshape(B, N, E)  # [B, N, E]
    x = self.proj(x)  # final projection
    x = self.proj_dropout(x)

    return x
```

MLP Block

class MLP(nn.Module): def **init**(self, in_features, hidden_features, out_features, dropout=0.1): super().**init**() self.fc1 = nn.Linear(in_features, hidden_features) # GELU Better than ReLU for transformers self.act = nn.GELU() self.fc2 = nn.Linear(hidden_features, out_features) self.dropout = nn.Dropout(dropout)

```
def forward(self, x):
    x = self.fc1(x)
    x = self.act(x)
    x = self.dropout(x)
    x = self.fc2(x)
    x = self.dropout(x)  # second dropout seems to help
    return x
```

Transformer Encoder Block

class TransformerBlock(nn.Module): def **init**(self, embed_dim=192, num_heads=8, mlp_ratio=4.0, dropout=0.1): super().**init**() self.norm1 = nn.LayerNorm(embed_dim) self.attn =

MultiHeadSelfAttention(embed_dim, num_heads, dropout) self.norm2 = nn.LayerNorm(embed_dim) self.mlp = MLP(in_features=embed_dim, hidden_features=int(embed_dim * mlp_ratio), # the ratio matters! out_features=embed_dim, dropout=dropout) # NOTE: we're using pre-norm formulation

```
def forward(self, x):
    # Pre-norm formulation - more stable, can train deeper networks
    # x + sublayer(norm(x)) instead of norm(x + sublayer(x))
    x = x + self.attn(self.norm1(x))
    x = x + self.mlp(self.norm2(x))
    return x
```

Complete Vision Transformer Model

class VisionTransformer(nn.Module): def **init**(self, img_size=32, patch_size=4, # 4x4 patches for CIFAR ie(32^2//4^2 == 64 tokens) in_channels=3, # RGB channel num_classes=10,# number of expected outputs embed_dim=192, # tried 384 but too many params for CIFAR tend to overfit depth=9, # paper uses 12, but 9 is enough for CIFAR and 12 tend to overfit num_heads=8, # must divide embed_dim evenly 192/8 = 24 mlp_ratio=4.0, dropout=0.1, # probablity of skiping connection ie 10 percent embed_dropout=0.1 # separate dropout rate for embeddings): super().**init**() self.num_classes = num_classes self.embed_dim = embed_dim self.num_tokens = (img_size // patch_size) ** 2

```
   # Patch embedding
    self.patch_embed = PatchEmbedding(
        img_size=img_size,
        patch_size=patch_size,
        in_channels=in_channels,
        embed_dim=embed_dim
    )

    # Class token and position embeddings
    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
    # Position embeddings - could use sinusoidal but learned works fine
    # postional embeddings are used because we have 8 multi head
attention we need assign position for each vector
    self.pos_embed = nn.Parameter(torch.zeros(1, self.num_tokens + 1,
embed_dim))

    # Initialize weights for faster convergence
    nn.init.trunc_normal_(self.pos_embed, std=0.02)
```

```
    nn.init.trunc_normal_(self.cls_token, std=0.02)

    self.dropout = nn.Dropout(embed_dropout)

    # Transformer blocks - this is the main part of the model
    self.blocks = nn.ModuleList([
        TransformerBlock(
            embed_dim=embed_dim,
            num_heads=num_heads,
            mlp_ratio=mlp_ratio,
            dropout=dropout
        )
        for _ in range(depth) # we just use for loop instead rewriting
tranformer 8 times
    ])

    # Final normalization layer
    self.norm = nn.LayerNorm(embed_dim)

    # Classification head - just a linear layer
    self.head = nn.Linear(embed_dim, num_classes)

    # Initialize weights
    self.apply(self._init_weights)

    # How many params?
    #print(f"ViT params: {sum(p.numel() for p in self.parameters())}")

def _init_weights(self, m):
    # Weight initialization matters for transformers!
    if isinstance(m, nn.Linear):
        nn.init.trunc_normal_(m.weight, std=0.02)
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)
    elif isinstance(m, nn.LayerNorm):
        nn.init.constant_(m.bias, 0)
        nn.init.constant_(m.weight, 1.0)

def forward(self, x):
    # x shape: [B, C, H, W]
    B = x.shape[0]

    # Create patch embeddings
    x = self.patch_embed(x)  # [B, N, E]

    # Add class token - used for final classification
```

```
    cls_token = self.cls_token.expand(B, -1, -1)  # [B, 1, E]
    x = torch.cat((cls_token, x), dim=1)  # [B, N+1, E]

    # Add position embeddings and apply dropout
    x = x + self.pos_embed  # broadcasting takes care of batch dim
    x = self.dropout(x)

    # Pass through transformer blocks
    for i, block in enumerate(self.blocks):
        # Could add intermediate supervision here?
        # Tried it, didn't help much, so removed it
        x = block(x)

    # Apply final normalization
    x = self.norm(x)

    # Take class token for classification
    # Could use pooling over all tokens but this works better
    x = x[:, 0]  # just get CLS token

    # Classification head
    x = self.head(x)
    # Could add an extra non-linearity here but linear seems fine

    return x
```

Training and Evaluation Utilities

def train_one_epoch(model, train_loader, criterion, optimizer, scheduler, device): model.train() # set model to training mode total_loss = 0.0 correct = 0 total = 0 batch_time = 0.0

```
# Progress bar
pbar = tqdm(train_loader, desc="Training")
start_time = time.time()

for batch_idx, (data, target) in enumerate(pbar):
    batch_start = time.time()
    data, target = data.to(device), target.to(device)

    # Forward pass
    optimizer.zero_grad()  # clear gradients first
    output = model(data)
    loss = criterion(output, target)
```

```
    # Backward pass
    loss.backward()

    # Could add gradient clipping here
    # torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    # But Adam seems to work fine without it

    optimizer.step()

    # Update learning rate - using per-step scheduler
    if scheduler is not None:
        scheduler.step()

    # Track metrics
    total_loss += loss.item() * data.size(0)
    _, predicted = output.max(1)  # get predicted class
    total += target.size(0)
    correct += predicted.eq(target).sum().item()

    # Track batch time
    batch_end = time.time()
    batch_time += (batch_end - batch_start)

    # Update progress bar - helps to see how training is going
    pbar.set_postfix({
        "loss": f"{loss.item():.4f}",
        "acc": f"{100. * correct / total:.1f}%",
        #"lr": f"{optimizer.param_groups[0]['lr']:.6f}"  # uncomment for
debugging
    })


epoch_time = time.time() - start_time

return total_loss / len(train_loader.dataset), 100. * correct / total,
epoch_time, batch_time / len(train_loader)


def evaluate(model, val_loader, criterion, device, classes=None,
full_metrics=False): model.eval() # set model to evaluation mode
total_loss = 0.0 correct = 0 total = 0 inference_times = []

# For confusion matrix and per-class metrics
all_targets = []
all_predictions = []
```

```
with torch.no_grad():  # no need to track gradients during evaluation
    for data, target in tqdm(val_loader, desc="Evaluation"):
        data, target = data.to(device), target.to(device)

        # Measure inference time
        start_time = time.time()
        output = model(data)
        inference_time = time.time() - start_time
        inference_times.append(inference_time)

        loss = criterion(output, target)

        # Track metrics
        total_loss += loss.item() * data.size(0)
        _, predicted = output.max(1)
        total += target.size(0)
        correct += predicted.eq(target).sum().item()

        # Store targets and predictions for additional metrics
        all_targets.extend(target.cpu().numpy())
        all_predictions.extend(predicted.cpu().numpy())

# Compute aggregate metrics
avg_loss = total_loss / len(val_loader.dataset)
accuracy = 100. * correct / total
avg_inference_time = sum(inference_times) / len(inference_times)

results = {
    'loss': avg_loss,
    'accuracy': accuracy,
    'inference_time_ms': avg_inference_time * 1000  # Convert to ms
}

# Add detailed metrics if requested
if full_metrics and classes:
    # Calculate per-class precision, recall, f1-score
    # Can't skip this computation - might seem slow but it's useful info
    precision, recall, f1, support = precision_recall_fscore_support(
        all_targets, all_predictions, labels=range(len(classes)),
average=None
    )

    # Create confusion matrix
    cm = confusion_matrix(all_targets, all_predictions,
labels=range(len(classes)))
```

```
    # Add to results
    results['confusion_matrix'] = cm
    results['per_class'] = {
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'support': support
    }
    results['classes'] = classes
    results['targets'] = all_targets
    results['predictions'] = all_predictions

return results
```

Metrics and Visualization Functions

def calculate_and_plot_metrics(model, val_loader, criterion, device, classes): print("Calculating detailed metrics...") results = evaluate(model, val_loader, criterion, device, classes, full_metrics=True)

```
# results
cm = results['confusion_matrix']
per_class = results['per_class']
targets = results['targets']
predictions = results['predictions']

# 1. Plot confusion matrix
plt.figure(figsize=(10, 8))
# Tried various colormaps - Blues is most readable
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.savefig('vit_confusion_matrix.png', dpi=200)  # higher DPI for paper-
quality

# 2. Plot per-class metrics
plt.figure(figsize=(12, 6))
x = np.arange(len(classes))
width = 0.2  # width of bars
```

```
# Plot bar chart with precision, recall, F1
plt.bar(x - width, per_class['precision'], width, label='Precision')
plt.bar(x, per_class['recall'], width, label='Recall')
plt.bar(x + width, per_class['f1'], width, label='F1-Score')

plt.xlabel('Classes')
plt.ylabel('Score')
plt.title('Per-Class Performance Metrics')
plt.xticks(x, classes, rotation=45)
plt.legend()
plt.tight_layout()
plt.savefig('vit_per_class_metrics.png')

# 3. Compute and plot ROC curves (one-vs-rest)
plt.figure(figsize=(12, 10))

# Prepare one-hot encoded targets for ROC
target_one_hot = np.zeros((len(targets), len(classes)))
for i, t in enumerate(targets):
    target_one_hot[i, t] = 1

# Get probability outputs for all samples
# Need to rerun the model to get probabilities
all_probs = []
model.eval()
with torch.no_grad():
    for data, _ in val_loader:
        data = data.to(device)
        outputs = model(data)
        probs = F.softmax(outputs, dim=1).cpu().numpy()
        all_probs.append(probs)

all_probs = np.vstack(all_probs)

# Plot ROC curve for each class
mean_auc = 0
for i, cls in enumerate(classes):
    fpr, tpr, _ = roc_curve(target_one_hot[:, i], all_probs[:, i])
    roc_auc = auc(fpr, tpr)
    mean_auc += roc_auc
    plt.plot(fpr, tpr, lw=2, label=f'{cls} (AUC = {roc_auc:.2f})')

mean_auc /= len(classes)

# Add diagonal line (random classifier)
```

```
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curves (Mean AUC = {mean_auc:.2f})')
plt.legend(loc="lower right")
plt.tight_layout()
plt.savefig('vit_roc_curves.png')

# Return metrics for CSV export
return {
    'accuracy': results['accuracy'],
    'loss': results['loss'],
    'inference_time_ms': results['inference_time_ms'],
    'per_class_precision': per_class['precision'],
    'per_class_recall': per_class['recall'],
    'per_class_f1': per_class['f1'],
    'mean_auc': mean_auc
}
```

Calculate model complexity

```
def calculate_model_complexity(model, input_size=(3, 32, 32)):
print("Calculating model complexity...") macs, params =
get_model_complexity_info( model, input_size, as_strings=False,
print_per_layer_stat=False )

# Did you know? FLOPs ≈ 2 * MACs
# ptflops returns MACs, but papers usually report FLOPs
return {
    'params': params,
    'flops': macs * 2,  # Convert MACs to FLOPs
    'params_millions': params / 1e6,
    'flops_billions': macs * 2 / 1e9
}
```

Export metrics to CSV

```
def export_metrics_to_csv(metrics, model_name='ViT',
filename='model_metrics.csv'): # Create directory if it doesn't exist
os.makedirs('metrics', exist_ok=True)
```

```
# Prepare CSV file path with timestamp
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
filepath = f'metrics/{model_name}_{timestamp}.csv'

# Flatten nested dictionaries
flat_metrics = {}
for key, value in metrics.items():
    if isinstance(value, dict):
        for subkey, subvalue in value.items():
            flat_metrics[f'{key}_{subkey}'] = subvalue
    elif isinstance(value, np.ndarray):
        for i, val in enumerate(value):
            flat_metrics[f'{key}_{i}'] = val
    else:
        flat_metrics[key] = value

# Write to CSV
with open(filepath, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)

    # Write header
    writer.writerow(['Metric', 'Value'])

    # Write metrics
    for key, value in flat_metrics.items():
        writer.writerow([key, value])

print(f"Metrics exported to {filepath}")

# Also create a summary CSV for model comparison
# This is super handy when doing hyperparameter sweeps!
summary_path = 'metrics/model_comparison.csv'

# Check if summary file exists, create with header if not
file_exists = os.path.isfile(summary_path)
with open(summary_path, 'a', newline='') as csvfile:
    writer = csv.writer(csvfile)

    if not file_exists:
        writer.writerow([
            'Model', 'Accuracy', 'Loss', 'Params (M)', 'FLOPs (G)',
            'Inference Time (ms)', 'Mean AUC', 'Training Time (s)'
        ])


    writer.writerow([
```

```
        model_name,
        metrics['accuracy'],
        metrics['loss'],
        metrics['complexity']['params_millions'],
        metrics['complexity']['flops_billions'],
        metrics['inference_time_ms'],
        metrics['mean_auc'],
        metrics['training_time']
    ])

print(f"Summary metrics added to {summary_path}")
```

Main Training Function

def train_vit_cifar10(epochs=100, batch_size=128, lr=1e-3, warmup_epochs=5, model_name='ViT'): # Setup data print(f"\n=== Setting up {model_name} training ===") print(f"Epochs: {epochs}, Batch size: {batch_size}, LR: {lr}")

```
data_module = CIFAR10DataModule(batch_size=batch_size)
data_module.setup()
train_loader = data_module.train_dataloader()
val_loader = data_module.val_dataloader()

# CIFAR-10 classes
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

# Create model - this is the standard ViT config for CIFAR
model = VisionTransformer(
    img_size=32,
    patch_size=4,  # 4x4 patches, so 8x8=64 patches total
    in_channels=3,
    num_classes=10,
    embed_dim=192,  # tried 384 but it was overkill
    depth=9,
    num_heads=8,  # 192 / 8 = 24 dim per head
    mlp_ratio=4.0,
    dropout=0.1,  # dropout helps a lot on CIFAR
    embed_dropout=0.1
).to(device)

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
```

```
print(f"Total parameters: {total_params:,}")

# Calculate model complexity
complexity = calculate_model_complexity(model)
print(f"FLOPs: {complexity['flops_billions']:.2f} G")
print(f"Parameters: {complexity['params_millions']:.2f} M")

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimizer
#  AdamW works better for transformers
optimizer = optim.AdamW(model.parameters(), lr=lr,
weight_decay=0.05)

# Learning rate scheduler - cosine decay with warmup
# Warmup is crucial for transformer training stability
total_steps = len(train_loader) * epochs
warmup_steps = len(train_loader) * warmup_epochs

# Learning rate schedule
def lr_lambda(step):
    # Linear warmup + cosine decay
    if step < warmup_steps:
        return float(step) / float(max(1, warmup_steps))
    # Cosine annealing
    return 0.5 * (1.0 + np.cos(np.pi * float(step - warmup_steps) /
float(total_steps - warmup_steps)))

# Create scheduler
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)

# Training loop
print("\n=== Starting training ===")
best_acc = 0.0
train_losses, train_accs = [], []
val_losses, val_accs = [], []
epoch_times, batch_times = [], []
total_training_time = 0
lr_history = []

for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}/{epochs}")

    # Log learning rate
```

```
    current_lr = optimizer.param_groups[0]['lr']
    lr_history.append(current_lr)

    # Train
    train_loss, train_acc, epoch_time, avg_batch_time =
train_one_epoch(
        model, train_loader, criterion, optimizer, scheduler, device
    )
    train_losses.append(train_loss)
    train_accs.append(train_acc)
    epoch_times.append(epoch_time)
    batch_times.append(avg_batch_time)
    total_training_time += epoch_time

    # Evaluate
    val_results = evaluate(model, val_loader, criterion, device)
    val_loss = val_results['loss']
    val_acc = val_results['accuracy']
    val_losses.append(val_loss)
    val_accs.append(val_acc)

    # Print metrics
    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%")
    print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")
    print(f"Epoch Time: {epoch_time:.2f}s, Avg Batch Time:
{avg_batch_time*1000:.2f}ms")
    print(f"Current LR: {current_lr:.6f}")

    # Save best model
    if val_acc > best_acc:
        best_acc = val_acc
        torch.save(model.state_dict(), "vit_cifar10_best.pth")
        print(f"New best validation accuracy: {best_acc:.2f}%!")
        # Also save at specific checkpoints (optional)
        #if val_acc > 90:
        #    torch.save(model.state_dict(), f"vit_cifar10_{val_acc:.1f}.pth")

    # Early stopping check after a reasonable number of epochs
    # No need to train forever if we're already good
    if epoch >= 50 and best_acc >= 90.0:
        print(f"Reached target accuracy of 90%. Stopping early!")
        break

print(f"\n=== Training complete ===")
print(f"Total training time: {total_training_time:.2f}s")
```

```
print(f"Best validation accuracy: {best_acc:.2f}%")

# Plot final expanded training metrics
print("\nGenerating final training plots...")
plt.figure(figsize=(18, 12))

# 1. Loss curves
plt.subplot(2, 3, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Curves')
plt.legend()

# 2. Accuracy curves
plt.subplot(2, 3, 2)
plt.plot(train_accs, label='Train Accuracy')
plt.plot(val_accs, label='Val Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy Curves')
plt.legend()

# 3. Epoch times
plt.subplot(2, 3, 3)
plt.plot(epoch_times)
plt.xlabel('Epoch')
plt.ylabel('Time (s)')
plt.title('Epoch Training Time')

# 4. Batch times
plt.subplot(2, 3, 4)
plt.plot(batch_times)
plt.xlabel('Epoch')
plt.ylabel('Time (s)')
plt.title('Average Batch Processing Time')

# 5. Learning rate
plt.subplot(2, 3, 5)
plt.plot(lr_history)
plt.xlabel('Epoch')
plt.ylabel('Learning Rate')
plt.title('Learning Rate Schedule')
# Add grid for readability
```

```
plt.grid(alpha=0.3)

plt.tight_layout()
plt.savefig('vit_training_metrics.png')
plt.close()  # close to avoid display issues with multiple plots

# Load best model for final evaluation
print("\nLoading best model for final evaluation...")
model.load_state_dict(torch.load("vit_cifar10_best.pth"))

# Calculate detailed metrics
detailed_metrics = calculate_and_plot_metrics(model, val_loader,
criterion, device, classes)

# Prepare metrics for export
final_metrics = {
    'accuracy': best_acc,
    'loss': val_losses[-1],
    'inference_time_ms': detailed_metrics['inference_time_ms'],
    'training_time': total_training_time,
    'epochs': len(train_losses),
    'avg_epoch_time': sum(epoch_times) / len(epoch_times),
    'avg_batch_time': sum(batch_times) / len(batch_times),
    'complexity': complexity,
    'mean_auc': detailed_metrics['mean_auc'],
    'per_class': {
        'precision': detailed_metrics['per_class_precision'],
        'recall': detailed_metrics['per_class_recall'],
        'f1': detailed_metrics['per_class_f1']
    }
}

# Export metrics to CSV
export_metrics_to_csv(final_metrics, model_name)

print(f"Best validation accuracy: {best_acc:.2f}%")
return model, best_acc, final_metrics
```

Attention Visualization Function

```
def visualize_attention(model, dataloader, device, num_images=4): #
Get some test images dataiter = iter(dataloader) images, labels =
next(dataiter) images = images[:num_images].to(device) labels =
labels[:num_images]
```

```
# Get class names
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

# Set model to eval mode
model.eval()


def get_attention_maps(x):
    B = x.shape[0]
    x = model.patch_embed(x)
    cls_token = model.cls_token.expand(B, -1, -1)
    x = torch.cat((cls_token, x), dim=1)
    x = x + model.pos_embed
    x = model.dropout(x)

    # Pass through transformer blocks except the last one
    for i, block in enumerate(model.blocks[:-1]):
        x = block(x)

    # Get attention from the last block
    # We're interested in how the cls token attends to the patches
    x = model.blocks[-1].norm1(x)  # apply LN first (pre-norm)
    qkv = model.blocks[-1].attn.qkv(x).reshape(B, x.shape[1], 3,
model.blocks[-1].attn.num_heads, model.blocks[-
1].attn.head_dim).permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]
    attn = (q @ k.transpose(-2, -1)) * (1.0 / np.sqrt(model.blocks[-
1].attn.head_dim))
    attn = F.softmax(attn, dim=-1)

    return attn

with torch.no_grad():
    # Get model predictions
    outputs = model(images)
    _, predicted = torch.max(outputs, 1)

    # Get attention maps
    attentions = get_attention_maps(images)  # shape: [B, H, N, N]

    # Extract attention from the CLS token to all patches
    # Average over all heads for visualization
    cls_attentions = attentions[:, :, 0, 1:].mean(1)  # shape: [B, N-1]

# Reshape attention maps to match the image patches
```

```
patch_size = 4
num_patches = 8  # 32 // 4 = 8

plt.figure(figsize=(16, 4 * num_images))

for i in range(num_images):
    # Original image - need to denormalize
    img = images[i].cpu().permute(1, 2, 0).numpy()
    img = img * np.array([0.2470, 0.2435, 0.2616]) + np.array([0.4914,
0.4822, 0.4465])
    img = np.clip(img, 0, 1)

    # Attention map
    attn_map = cls_attentions[i].reshape(num_patches,
num_patches).cpu().numpy()

    # Upsample the attention map to match the image size
    # Simple nearest-neighbor upsampling
    attn_map = np.repeat(np.repeat(attn_map, patch_size, axis=0),
patch_size, axis=1)

    # Color indicates attention strength
    plt.subplot(num_images, 3, i*3 + 1)
    plt.imshow(img)
    plt.title(f"Original: {classes[labels[i]]}\nPredicted:
{classes[predicted[i]]}")
    plt.axis('off')

    plt.subplot(num_images, 3, i*3 + 2)
    plt.imshow(attn_map)
    plt.title("Attention Map")
    plt.axis('off')

    plt.subplot(num_images, 3, i*3 + 3)
    plt.imshow(img)
    plt.imshow(attn_map, alpha=0.5, cmap='jet')  # overlay with some
transparency
    plt.title("Overlay")
    plt.axis('off')

plt.tight_layout()
plt.savefig('vit_attention_maps.png')
plt.close()  # close to avoid display issues
```

Main Execution

if **name** == "**main**": # Create models directory os.makedirs('metrics', exist_ok=True)

```
# Train the model
# You can customize these hyperparameters
model, best_acc, metrics = train_vit_cifar10(
    epochs=100,      # max epochs
    batch_size=128,  # reduce if OOM
    lr=1e-3,         # tried 5e-4 and 3e-3, this works best
    warmup_epochs=5, # helps stabilize training
    model_name='ViT' # for saving metrics
)

# Visualize attention if we did well
data_module = CIFAR10DataModule(batch_size=4)
data_module.setup()
val_loader = data_module.val_dataloader()

print("Visualizing attention maps")
visualize_attention(model, val_loader, device)
```

!pip install -q matplotlib torchvision # -q to avoid flooding the output

Import all the stuff we need

import torch import torch.nn as nn import torch.nn.functional as F import numpy as np import matplotlib.pyplot as plt from PIL import Image import io from google.colab import files import os from torchvision import transforms

Check if we have a GPU ,infernce on gpu is better

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') print(f"Using device: {device}") if device.type == 'cpu': print("Warning: slow without GPU acceleration!")

CIFAR-10 class names

CLASSES = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

Patch Embedding Module

class PatchEmbed(nn.Module): """ Image to Patch Embedding - splits the image into patches """ def **init**(self, img_size=32, patch_size=4, in_chans=3, embed_dim=192): super().**init**() self.img_size = img_size self.patch_size = patch_size self.n_patches = (img_size // patch_size) ** 2 # number of patches

```
  # This conv does the patch embedding - simpler than manual reshaping
    # Tried using nn.Unfold first but conv is cleaner
    self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)

def forward(self, x):
    B, C, H, W = x.shape
    # Make sure image size is right - this saved me once when I fed in 224px images!
    assert H == self.img_size and W == self.img_size, \
        f"Input image size ({H}*{W}) doesn't match model ({self.img_size}*{self.img_size})"

    # (B, C, H, W) -> (B, E, H//P, W//P) -> (B, E, N) -> (B, N, E)
    x = self.proj(x).flatten(2).transpose(1, 2)
    return x
```

Attention Module

class Attention(nn.Module): def **init**(self, dim, num_heads=8, qkv_bias=True, attn_drop=0., proj_drop=0.): super().**init**() self.num_heads = num_heads head_dim = dim // num_heads # Scale factor - super important! Training explodes without this self.scale = head_dim ** -0.5

```
  # Combined QKV projection - faster than separate projections
    self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
    self.attn_drop = nn.Dropout(attn_drop)
    self.proj = nn.Linear(dim, dim)
    self.proj_drop = nn.Dropout(proj_drop)

    # Tried adding a parameter here to control attention strength
    # but it didn't really help
    #self.attn_scale = nn.Parameter(torch.ones(1))
```

```
def forward(self, x):
    B, N, C = x.shape
    # This reshape is tricky but important for multi-head attention
    qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C //
self.num_heads).permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]  # separate Q, K, V

    # Compute attention scores - (B, H, N, N)
    attn = (q @ k.transpose(-2, -1)) * self.scale
    #attn = attn * self.attn_scale  # optional extra scaling (commented out)
    attn = attn.softmax(dim=-1)
    attn = self.attn_drop(attn)

    # Apply attention to values
    x = (attn @ v).transpose(1, 2).reshape(B, N, C)
    x = self.proj(x)
    x = self.proj_drop(x)
    return x, attn  # return attention
```

MLP Module

class MLP(nn.Module): def **init**(self, in_features, hidden_features=None, out_features=None, drop=0.): super().**init**() out_features = out_features or in_features hidden_features = hidden_features or in_features self.fc1 = nn.Linear(in_features, hidden_features) self.act = nn.GELU() # GELU > ReLU for transformers, apparently self.fc2 = nn.Linear(hidden_features, out_features) self.drop = nn.Dropout(drop)

```
   # Debug flag for printing layer activations
    self.debug = False

def forward(self, x):
    x = self.fc1(x)
    x = self.act(x)

    # Optional debugging
    if self.debug and torch.rand(1).item() < 0.01:  # only print occasionally
        print(f"MLP activations: mean={x.mean().item():.3f},
std={x.std().item():.3f}")

    x = self.drop(x)
    x = self.fc2(x)
    x = self.drop(x)
```

return x

Transformer Block

class Block(nn.Module): def **init**(self, dim, num_heads, mlp_ratio=4., qkv_bias=True, drop=0., attn_drop=0.): super().**init**() self.norm1 = nn.LayerNorm(dim) self.attn = Attention(dim, num_heads=num_heads, qkv_bias=qkv_bias, attn_drop=attn_drop, proj_drop=drop) self.norm2 = nn.LayerNorm(dim) self.mlp = MLP(in_features=dim, hidden_features=int(dim * mlp_ratio), drop=drop)

```
   # Could add a stochastic depth (drop path) here for regularization
    # But it's not necessary for inference

def forward(self, x):
    # Pre-norm architecture - more stable than post-norm
    norm_x = self.norm1(x)
    attn_output, attn_weights = self.attn(norm_x)
    x = x + attn_output  # residual connection
    x = x + self.mlp(self.norm2(x))  # another residual
    return x, attn_weights
```

Vision Transformer Model

class VisionTransformer(nn.Module): def **init**(self, img_size=32, patch_size=4, in_chans=3, num_classes=10, embed_dim=192, depth=9, num_heads=8, mlp_ratio=4., qkv_bias=True, drop_rate=0., attn_drop_rate=0.): super().**init**() self.num_classes = num_classes self.num_features = self.embed_dim = embed_dim self.img_size = img_size self.patch_size = patch_size

```
   # Patch embedding
    self.patch_embed = PatchEmbed(img_size=img_size,
patch_size=patch_size,
                              in_chans=in_chans, embed_dim=embed_dim)
    num_patches = self.patch_embed.n_patches

    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))

    # Position embedding
    self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1,
embed_dim))
    # Could use fixed sinusoidal embeddings, but learned ones seem
```

better

```
    self.pos_drop = nn.Dropout(p=drop_rate)

    # Transformer blocks - stack of identical blocks
    self.blocks = nn.ModuleList([
        Block(
            dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio,
qkv_bias=qkv_bias,
            drop=drop_rate, attn_drop=attn_drop_rate)
        for i in range(depth)])

    # Normalization and classification head
    self.norm = nn.LayerNorm(embed_dim)
    self.head = nn.Linear(embed_dim, num_classes)

    # Could use different pooling strategies:
    # 1. CLS token (what we're using)
    # 2. Mean pooling over all tokens
    # 3. Both concatenated
    # But CLS token works best in most cases

def forward(self, x):
    # Patch embedding [B, N, E]
    x = self.patch_embed(x)
    B = x.shape[0]

    # Append class token
    cls_tokens = self.cls_token.expand(B, -1, -1)
    x = torch.cat((cls_tokens, x), dim=1)

    # Add position embeddings
    x = x + self.pos_embed
    x = self.pos_drop(x)

    # Storage for visualization
    attn_weights = None

    # Apply transformer blocks - save attention from last block
    for i, block in enumerate(self.blocks[:-1]):
        x, _ = block(x)
        # Could save intermediate features here
        # if i == len(self.blocks) // 2:
        #     mid_features = x

    # Last block - save attention weights for visualization
```

```
    x, attn_weights = self.blocks[-1](x)

    # Apply final normalization
    x = self.norm(x)

    # Use class token for classification
    x = self.head(x[:, 0])

    return x, attn_weights

# Unused sinusoidal embedding function - kept for reference
# def get_sinusoidal_embedding(self):
#     # Implementation of fixed position embeddings
#     pass
```

Model Loading Function

def load_model(model_path): try: print("Loading model state dictionary...") # Load state dictionary - added weights_only to avoid optimizers state_dict = torch.load(model_path, map_location=device, weights_only=True)

```
   # Create a new model instance
    vit = VisionTransformer(
        img_size=32,  # CIFAR size
        patch_size=4,  # 4x4 patches → 8x8 = 64 patches total
        in_chans=3,
        num_classes=10,
        embed_dim=192,  # could be 384 for bigger models
        depth=9,      # number of transformer blocks
        num_heads=8,  # attention heads - must divide embed_dim evenly
        mlp_ratio=4.0,  # multiplier for hidden dim in MLP
        qkv_bias=True,  # helps training
        drop_rate=0.0,  # no dropout for inference
        attn_drop_rate=0.0
    ).to(device)

    # Check key names - sometimes need to adapt based on different
exports
    # model_keys = set(state_dict.keys())
    # our_keys = set(vit.state_dict().keys())
    # if model_keys != our_keys:
    #     print(f"Warning: Key mismatch. Missing: {our_keys -
model_keys}")
```

```
    # Load the weights
    vit.load_state_dict(state_dict)
    vit.eval()  # IMPORTANT! Set to evaluation mode
    print("Model loaded successfully!")

    # Could print model summary but torch.summary not in default install
    # from torchsummary import summary
    # summary(vit, (3, 32, 32))

    return vit

except Exception as e:
    print(f"Error loading model: {e}")
    print("\nIf you're getting key mismatches, here's some debug info:")
    print("For your uploaded model:")
    try:
        state_dict = torch.load(model_path, map_location=device,
weights_only=True)
        # Just show a few keys to avoid flooding output
        print(f"Keys in your model: {list(state_dict.keys())[:5]}... (first 5
only)")
    except:
        print("Could not load state dictionary to show keys.")

    print("\nFor the inference model:")
    model = VisionTransformer()
    print(f"Keys expected: {list(model.state_dict().keys())[:5]}... (first 5
only)")


    return None
```

Image Preprocessing Function

def preprocess_image(image_data): # Open image from uploaded bytes img = Image.open(io.BytesIO(image_data)).convert('RGB') orig_size = img.size img_copy = img.copy() # keep an unmodified copy

```
# CIFAR preprocessing - critical to match training!
# The normalization values are important - don't change
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # ViT needs fixed size
    transforms.ToTensor(),
```

```
    # CIFAR-10 mean/std - different from ImageNet values
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435,
0.2616))
])

# Convert to tensor and add batch dimension
img_tensor = transform(img).unsqueeze(0).to(device)

# Quick check for NaN values
if torch.isnan(img_tensor).any():
    print("Warning: NaN values in processed image!")

return img_tensor, img_copy, orig_size
```

Image Classification Function

```
def classify_image(model, img_tensor): # No gradient tracking needed
for inference with torch.no_grad(): # Forward pass through the model
logits, attn_weights = model(img_tensor)

   # Optional logit range check
    # print(f"Logit range: {logits.min().item():.2f} to
{logits.max().item():.2f}")

    # Convert to probabilities with softmax
    probs = F.softmax(logits, dim=1)[0]

    # Get top 3 predictions (could change to top-5 for more classes)
    top3_probs, top3_idx = torch.topk(probs, 3)
    # Convert to percentages
    top3_probs = top3_probs.cpu().numpy() * 100
    top3_classes = [CLASSES[idx] for idx in top3_idx.cpu().numpy()]

return top3_classes, top3_probs, attn_weights
```

Results Visualization Function

```
def visualize_results(image, top_classes, top_probs):
plt.figure(figsize=(10, 5))

# Show image
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title(f"Prediction: {top_classes[0]}")
```

```
plt.axis('off')

# Show top 3 predictions
plt.subplot(1, 2, 2)
# First prediction in blue, others in gray
colors = ['#3498db', '#95a5a6', '#95a5a6']  # blue, gray, gray
bars = plt.barh(top_classes, top_probs, color=colors)
plt.xlim([0, 100])  # percentages from 0-100
plt.xlabel('Confidence (%)')
plt.title('Top 3 Predictions')

# Add confidence values as text
for bar, prob in zip(bars, top_probs):
    plt.text(min(prob + 3, 95), bar.get_y() + bar.get_height()/2,
f"{prob:.1f}%",
             va='center', fontweight='bold')

plt.tight_layout()
plt.show()
```

Attention Visualization Function

def visualize_attention(model, image, attn_weights, original_size): # Get attention from class token to patches # Averaging over all attention heads for visualization cls_attn = attn_weights[0, :, 0, 1:].mean(0).cpu().numpy()

```
# Originally had a for-loop going through each head
# but the averaged version is cleaner
# for head in range(attn_weights.shape[1]):
#     head_attn = attn_weights[0, head, 0, 1:].cpu().numpy()
#     # ... show each head separately

# Reshape to match image patches
patch_size = model.patch_size
num_patches_per_side = model.img_size // patch_size
attn_map = cls_attn.reshape(num_patches_per_side,
num_patches_per_side)

# Tried bilinear upsampling but nearest neighbor works fine
# from skimage.transform import resize
# attn_map = resize(attn_map, (model.img_size, model.img_size),
order=1)
```

# Upsample attention map to match image size - simple nearest neighbor
attn_map = np.repeat(np.repeat(attn_map, patch_size, axis=0), patch_size, axis=1)

# Display everything
plt.figure(figsize=(15, 5))

# Original image (resized to model input size)
plt.subplot(1, 3, 1)
plt.imshow(image.resize((32, 32)))
plt.title("Original Image")
plt.axis('off')

# Attention heatmap
plt.subplot(1, 3, 2)
plt.imshow(attn_map, cmap='viridis') # viridis is better than jet
plt.title("Attention Map")
plt.axis('off')

plt.colorbar(label='Attention')

# Overlay attention on image
plt.subplot(1, 3, 3)
plt.imshow(image.resize((32, 32)))
plt.imshow(attn_map, alpha=0.5, cmap='jet') # jet works better for overlay
plt.title("Attention Overlay")
plt.axis('off')

plt.tight_layout()
plt.show()

Main Inference Function

def run_inference(): print("=== Vision Transformer Image Classification ===") print("Let's classify some images using a ViT model!")

```
# Check for existing model or upload new one
model_path = 'vit_cifar10_best.pth'
if not os.path.exists(model_path):
    print("\n1. Please upload your trained ViT model file (.pth):")
    print("   (If you don't have one, try the sample model from the repo)")
    uploaded = files.upload()
```

```
    if not uploaded:
        print("No model was uploaded. Please run again.")
        return
    model_path = list(uploaded.keys())[0]
    print(f"Using model: {model_path}")

# Load the model
model = load_model(model_path)

if model is None:
    print("\nUnable to load model. If you're seeing key mismatches,")
    print("you might need to adjust the model architecture to match your
checkpoint.")
    return

# Upload image
print("\n2. Now upload an image to classify:")
print("   (Try a picture of a car, plane, cat, or anything from CIFAR-10
classes)")
uploaded = files.upload()

if not uploaded:
    print("No image was uploaded. Please run again.")
    return

# Process each uploaded image
for filename, file_data in uploaded.items():
    print(f"\nProcessing image: {filename}")

    # Preprocess image for the model
    image_tensor, original_image, original_size =
preprocess_image(file_data)

    # Run classification
    top_classes, top_probs, attn_weights = classify_image(model,
image_tensor)

    # Show results
    print(f"\nResults:")
    print(f"Top Prediction: {top_classes[0]} ({top_probs[0]:.1f}%)")
    print(f"2nd Prediction: {top_classes[1]} ({top_probs[1]:.1f}%)")
    print(f"3rd Prediction: {top_classes[2]} ({top_probs[2]:.1f}%)")

    # Add confidence interpretation
    if top_probs[0] > 90:
```

```
        print("Very confident prediction!")
    elif top_probs[0] > 70:
        print("Fairly confident prediction.")
    else:
        print("The model seems unsure - check the attention map for
clues.")

    # Show the plots
    visualize_results(original_image, top_classes, top_probs)

    # Visualize attention
    print("\nVisualizing attention map (what the model focuses on):")
    visualize_attention(model, original_image, attn_weights, original_size)

    print("\nThe brightness in the attention map shows which parts of the
image")
    print("influenced the model's decision most strongly.")
```

Main Execution

if **name** == "**main**": run_inference()