Vision Transformer from Scratch Report

Azaad Katiyar (240249)

February 7, 2026

1 Introduction

In this assignment, I implemented a Vision Transformer (ViT) model completely from scratch using PyTorch and trained it on the CIFAR-10 dataset.
Unlike traditional convolutional neural networks (CNNs), which rely on convolution and pooling operations, Vision Transformers treat an image as a sequence of patches and process them using the Transformer architecture.

2 Data Preprocessing and Regularization

2.1 Dataset

The CIFAR-10 dataset contains:

1. 60,000 color images
2. Image size: $32 \times 32 \times 3$
3. 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)

2.2 Preprocessing Steps

The following preprocessing operations were applied:

```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),              # converts the images to a tensor
    transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.247, 0.243, 0.261])  # normalizes the image tensor
])
```

Figure 1: Preprocessing operations

1. Random Horizontal Flip:
   Randomly flips images horizontally and helps the model learn orientation invariant features.
   Effect → Improves generalization and reduces overfitting.
2. Random Crop with Padding:
   Adds padding around image and then crops back to 32×32 pixels.
   Effect → Makes the model invariant to small shifts and translations.

3. ToTensor:
   Converts images to PyTorch tensors and scales pixel values to [0,1].
   Effect → Allows the image to be processed by neural networks.

4. Normalization:
   Sets mean and standard deviation.
   Centers pixel values around zero.
   Effect → Faster convergence and more stable training.

2.3 Regularization Techniques Used

1. Dropout:
   Randomly disables neurons during training.
   Effect → Forces the network to learn redundant representations and improves generalization.

```
self.dropout = nn.Dropout(config.dropout)
```

2. Weight Decay (L2 Regularization):
   Penalizes large weights.
   Effect → Prevents extremely large parameter values and stabilizes training.

```
optimizer = optim.AdamW(model.parameters(), lr=3e-4, weight_decay=0.05)
```

3. Gradient Clipping:
   Maintains training stability against the "exploding gradients" which are common in deep transformer architectures.

```
# gradient clipping
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

3 Hyperparameter Tuning

3.1 Important Hyperparameters

Parameter	Tested Values	Final Value
Patch Size	2,4,8	4
Embedding Dim	256,384,768	364
Heads	4,8	8
Layers	4,6,8	6
Dropout	0.1,0.2	0.1
Batch Size	64,128	128
Learning Rate	1e-3, 3e-4, 1e-4	3e-4

3.2 Observations

1. Smaller patch size improves accuracy but increases computation.
2. More layers improved performance up to a point.
3. Too large learning rate caused unstable loss.

4 Vision Transformer Architecture

The core of the ViT is the ”tokenization” process, where a static image is transformed into a sequence of vectors that the Transformer can process as if it were text.

4.1 Patch Creation

The image is divided into small non-overlapping patches.
Each patch is equivalent to a “token” in NLP. Patches are important because Transformers work on sequences, and patches act as sequence elements.

```
self.patch_size = config.patch_size # each patch has a size of 4x4 pixels
# calculating the number of patches were going to get-
self.num_patches = (self.image_size // self.patch_size) ** 2  # 32/4 = 8x8 = 64 patches
```

4.2 Patch Embedding

Each flattened patch is projected into a fixed-size embedding vector.
Effect → Transforms raw pixels into a feature representation.

```
self.out_proj = nn.Linear(self.embed_dim, self.embed_dim, bias=True)  # this gives the output projection
```

4.3 Positional Embedding

Transformers have no inherent notion of order. Purpose of this is to add spatial information about where each patch is located.

```
# creating the positional embedding layer -> creates a lookup table of size num_patches x embed_dim
self.position_embedding = nn.Embedding(self.num_positions, self.embed_dim)
```

4.4 Class Token

It is a learnable token that gathers information from all patches. It is used for final classification.

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

4.5 Multi-Head Self Attention

In Multi-Head Self Attention, each patch attends to every other patch. The model learns global relationships instead of local ones.

```
# scaled dot product attention -
attn_weights = (q_states @ k_states.transpose(-2, -1)) / math.sqrt(self.head_dim)
attn_weights = F.softmax(attn_weights, dim=-1)

attn_weights = F.dropout(attn_weights, p=self.dropout, training=self.training)
```

4.6 Feed Forward Network (MLP Block)

This block introduces non-linear transformations that enhance representation capacity, enabling the model to refine token features and learn complex patterns beyond linear attention outputs.

```
def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
    hidden_states = self.fc1(hidden_states)
    # applying the non linearity activation function -
    hidden_states = F.gelu(hidden_states)
    hidden_states = self.dropout(hidden_states)
    hidden_states = self.fc2(hidden_states)
    return hidden_states
```

4.7 Classification Head

Maps class token representation to class probabilities.

```
# classification head
self.classifier = nn.Linear(config.hidden_size, config.num_classes)
```

5 Results

Training Results:
The 50 epoch training run yielded the following final results:

$$\textbf{Training Accuracy} = \textbf{95.03\%}$$
$$\textbf{Best Test Accuracy} = \textbf{79.30\%}$$

The model successfully learnt meaningful image representations. Accuracy is low on CIFAR-10 dataset due to small dataset size. With larger datasets, ViT performance would improve significantly.