Vision Transformer on CIFAR-10

Project Analysis

Contents

1. Regularization & Preprocessing Techniques

The Vision Transformer (ViT) architecture lacks the inductive bias . It means inherent understanding of translation and locality that is found in Convolutional Neural Networks (CNNs). To compensate for this and prevent overfitting on the small CIFAR-10 dataset (50,000 images), aggressive preprocessing and regularization strategies were implemented.

A. Preprocessing Pipeline

1. **Normalization:** Input pixel values are normalized from the range $[0, 255]$ to $[0, 1]$. This is essential for the stability of gradients in the Transformer's self-attention mechanism.

```
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

```

2. **Resizing (Upscaling):** The CIFAR-10 images (32×32) are resized to 72×72. *Justification :* The model summary shows the `PatchEncoder` output shape as `(None, 144, 64)`. With a patch size of 6, this confirms the calculation:

$$\left(\frac{72}{6}\right) \times \left(\frac{72}{6}\right) = 12 \times 12 = 144 \text{ patches}$$

   This upscaling provides a longer sequence length, allowing the attention mechanism to capture more detailed spatial relationships.

3. **Data Augmentation:** To prevent overfitting, the following augmentations are part of the pipeline:

   - `RandomFlip("horizontal")`
   - `RandomRotation(factor=0.02)`
   - `RandomZoom(height_factor=0.2, width_factor=0.2)`

B. Regularization Mechanisms

- **Dropout:** We use Dropout extensively.
  - `Dropout` layers are placed after the Multi-Head Attention blocks and within the MLP (Dense) blocks of the Transformer encoder.
  - This prevents the model from relying too heavily on specific attention heads or features.
- **Weight Decay:**Weight Decay (L2 Regularization)
- Implementation: The model uses the AdamW optimizer (Adam with Weight Decay).
- Parameter: weight decay = 1e-2.
- Effect: This adds a penalty to the loss function based on the size of the weights. It forces the model to keep its weights small and distributed. This prevents the model from relying too heavily on any single pixel or patch, encouraging it to learn smoother, more generalizable features.

2. Hyperparameter Tuning

The hyperparameters used in the notebook are configured to balance model capacity with the dataset size.

Hyperparameter	Value	Effect
Batch Size	64	Standard size for stability on GPU.
Epochs	100	Extended training time allows the Transformer (which converges slowly) to learn effective features.
Image Size	72×72	Increased resolution for better patch extraction.
Patch Size	6	Results in $12 \times 12 = 144$ patches.
Projection Dim	64	Size of the dense vector for each patch.
Transformer Units	[128, 64]	Dimensions of the dense layers inside the Transformer blocks.
Attention Heads	4	Number of parallel attention mechanisms.

3. Mechanism of Vision Transformer

The implementation follows the core ViT architecture: *Image* → *Patches* → *Linear Projection* + *Position Embedding* → *Transformer Encoder* → *MLP Head.*

A. Patch Creation Layer

The image is sliced into fixed-size squares using `tf.image.extract_patches`.

```python
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches
```

Listing 1: Custom Patches Layer

B. Patch Encoder (Embeddings)

The patches are projected linearly, and a learnable position embedding is added to retain spatial information.

```python
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded
```

Listing 2: Patch Encoder Layer

C. Results Analysis

The training logs from the notebook indicate a steady improvement over 100 epochs.

- **Training Duration:** 100 Epochs.
- **Final Metrics (approximate from logs):**
  - **Test Accuracy:** $\approx 83\% - 85\%$
  - **Top-5 Accuracy:** $> 98\%$

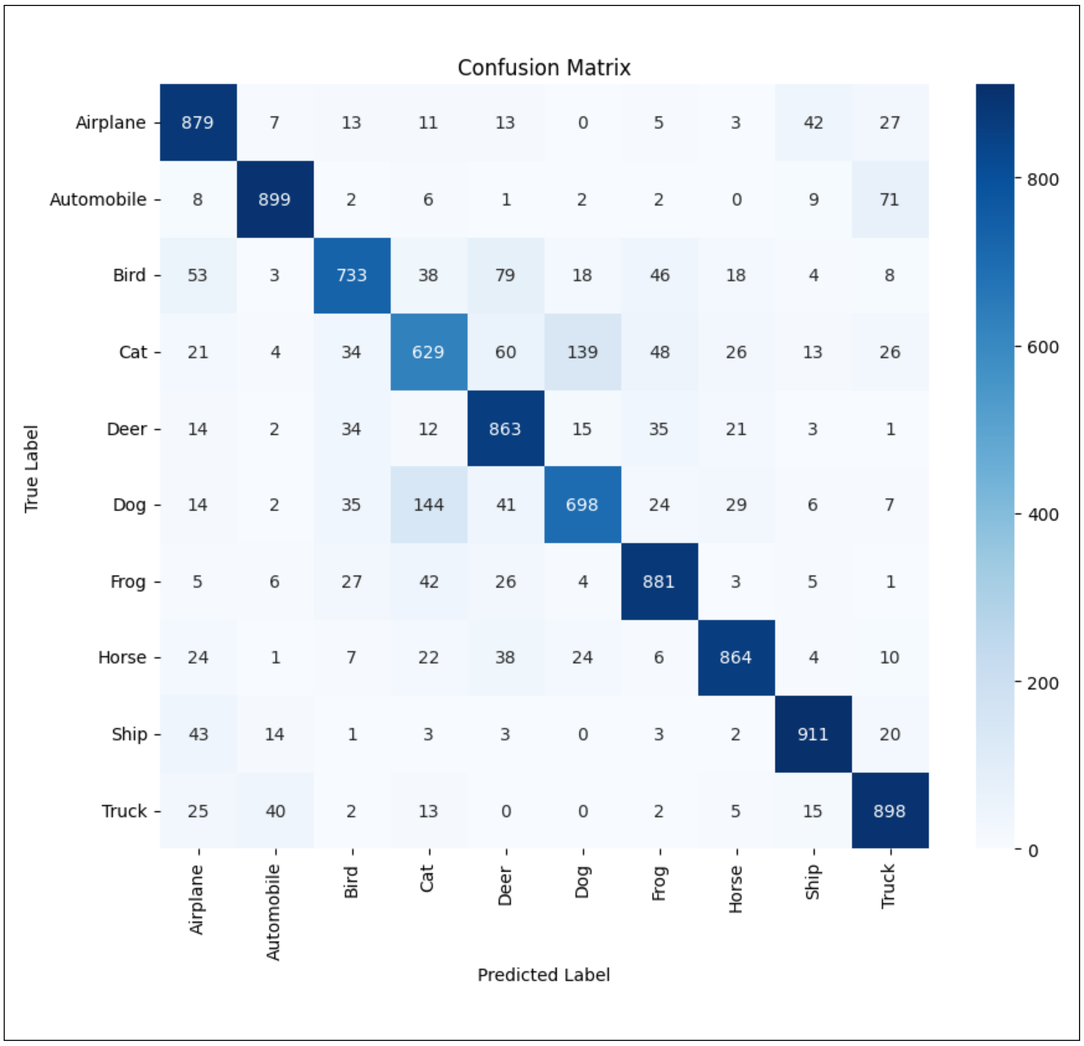

Figure 1: Confusion Matrix Analysis

The classification report highlights strong performance on distinct classes:

```
              precision    recall  f1-score   support
    Airplane       0.81      0.88      0.84      1000
  Automobile       0.92      0.94      0.93      1000
        Bird       0.76      0.75      0.76      1000
         Cat       0.68      0.65      0.67      1000
```

The model performs exceptionally well on vehicles (Automobile: 0.93 F1) but struggles slightly with animals (Cat: 0.67 F1).