Task 2: Comparative Analysis of CNN and Vision Transformer on CIFAR-10 Image Classification

Introduction

This notebook implements and evaluates both a Convolutional Neural Network (CNN) and Vision Transformer (ViT) for image classification on the CIFAR-10 dataset. Both models will be trained on the same dataset under comparable conditions. The resulting accuracy, computational efficiency, and generalisation ability of each architecture will be analysed and compared, in order to draw insights into their respective strengths, weaknesses and differences in learning behaviours.

Setup

Settings for Reproducibility

To ensure reproducibility, the versions of python and PyTorch are reported below:

1. Python Version: 3.12.5
2. PyTorch Version: 2.5.1+cpu121

Disclaimer: While the seed is explicitly defined to ensure reproducibility, slight variations in the training results and various plots may still occur. These small differences are likely due to the specific hardware characteristics of the computer used, such as CPU and memory performance. We have minimized these discrepancies, but they may still be present. For reference on the exact output graphs used for our interpretations, please refer to the accompanying PDF document.

Import Libraries

```
# Import necessary libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import time
from PIL import Image
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader
```

```python
from sklearn.metrics import classification_report, confusion_matrix
import pytorch_lightning as pl
from pytorch_lightning import Trainer, LightningModule

# Device configuration (GPU support)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Set pytorch lightning seed for reproducibility
pl.seed_everything(42, workers=True)
```

```
Seed set to 42
```

```
42
```

Data Import and Preparation

The CIFAR-10 dataset consists of 60,000 RGB images, 50,000 in the training set and 10,000 in the test set. Each image is of size 32x32 pixels, with the pixel values normalised to [0, 1]. Each image has a label of 1 of 10 different classes, namely:

1. 'airplane'
2. 'automobile'
3. 'bird'
4. 'cat'
5. 'deer'
6. 'dog'
7. 'frog'
8. 'horse'
9. 'ship'
10. 'truck'

As part of our data preprocessing, we performed the following data augmentations on the CIFAR-10 dataset.

1. Flipping the images horizontally
2. Rotate images within +/- 15 degrees
3. Translate the images

These augmentations are useful for enabling computer vision models to generalise better to unseen data, as they learn to recognise the object regardless of the transformations made to them.

Additionally, since the pixel values of the images are in a [0, 1] scale, we further normalised the dataset using the channel-wise mean and standard deviation values, obtained by averaging across the entire CIFAR-10 dataset training images. This brings each RGB channel to mean = 0 and std dev = 1, thereby standardising the input in order to train the neural networks more effectively.

```
# Define transformations for the training set
# Includes data augmentation (flipping, rotation, translation) and
normalization
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),  # Randomly flip images
horizontally
    transforms.RandomRotation(15),           # Rotate images randomly
within ±15 degrees
    transforms.RandomAffine(0, translate=(0.1, 0.1)),  # Apply random
translations
    transforms.ToTensor(),                   # Convert images to
PyTorch tensors
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435,
0.2616))  # Normalize using CIFAR-10 stats
])
# Define transformations for the test set
# Only normalization is applied (no augmentation)
test_transform = transforms.Compose([
    transforms.ToTensor(),                   # Convert images to
PyTorch tensors
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435,
0.2616))  # Normalize using CIFAR-10 stats
])

batch_size = 1024
train_dataset = CIFAR10(root='./data', train=True, download=True,
transform=train_transform)
test_dataset = CIFAR10(root='./data', train=False, download=True,
transform=test_transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
classes = ('airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

Visualise sample images from the test dataset

```
def visualize_sample_images(dataloader, class_names=None):
    found = {}
    for images, labels in dataloader:
        for img, label in zip(images, labels):
            label = label.item()
            if label not in found:
                found[label] = img
            if len(found) == 10:
                break
```

```
        if len(found) == 10:
            break

    plt.figure(figsize=(15, 3))
    for idx, class_id in enumerate(sorted(found.keys())):
        img = found[class_id].cpu()
        img_np = img.permute(1, 2, 0).numpy()

        # Denormalize (CIFAR-10)
        img_np = img_np * np.array([0.2023, 0.1994, 0.2010]) +
np.array([0.4914, 0.4822, 0.4465])
        img_np = np.clip(img_np, 0, 1)

        plt.subplot(1, 10, idx + 1)
        plt.imshow(img_np)
        plt.axis('off')
        if class_names:
            plt.title(class_names[class_id])
        else:
            plt.title(f"Class {class_id}")

    plt.tight_layout()
    plt.show()

print("Sample transformed images from the training set:")
visualize_sample_images(train_loader, class_names=classes)
print("Sample images from the test set:")
visualize_sample_images(test_loader, class_names=classes)
```

```
Sample transformed images from the training set:
```


```
Sample images from the test set:
```

Defining Model Architectures

Convolutional Neural Network (CNN) Architecture

Our CNN architecture is implemented as follows:

- **Feature Extraction - 3x ResBlocks**
  - Each ResBlock contains 2 convolutional filters, BatchNorm, ReLU, and a residual connection added to the output.
  - Each filter has dimension of 3x3, and padding of 1 pixel is used to maintain the spatial dimensions of the input (i.e., the height and width of the image), since 32x32 is already very small.
  - The 1st filter performs a convolution on the input and splits it multiple feature maps. The 2nd filter performs another convolution without increasing the number of output feature maps.
  - We have noted that low resolution of the images makes it difficult to make out what the object in the images are. Hence, we made sure that the number of feature maps increases substantially with each ResBlock, from 3 RGB input channels -> 32 -> 64 -> 128.
  - Having a larger number of feature maps with each block of the CNN enables the model to capture a larger variety of abstract features, such as edges around the object or patterns of an animal, thus allowing the CNN to use more visual characteristics to differentiate between different classes more easily.
  - Batch normalisation is used to ensure numeric stability of the learned weights, especially with ReLU activation.
  - ReLU activation is used to introduce non-linearity to the model.
  - Residual connection adds the original input image to the output of the residual block. This is important for a dataset with low resolution images, as it preserves the original details of the image, which would have otherwise been lost due to the convolution. This also helps to address any vanishing gradient problems.
  - MaxPooling
    - Applied after each ResBlock to progressively downsample the image.
    - Although the spatial resolution of the images are lost from the act of downsampling, the most important 'meta' features of the image are preserved and carried forward to the next block of the CNN, rather than ensuring every pixel of the input is retained. This way, the CNN learns to recognise high level visual characteristics rather than individual pixels of the image.
    - Crucially, downsampling helps improve on computational efficiency of the CNN. In this case, the feature maps are shrunk from 32x32 → 16x16 → 8x8 → 4x4, which is effectively a substantial 64x reduction in computation needed in later layers.
  - Dropout
    - A low dropout rate of 0.1 was empirically tested and used to regularise the learning and prevent overfitting. A balance was achieved to prevent the model from underfitting as well.

- **MLP Layer**
  - Following the feature extraction, the output is flattened into a 128x4x4 = 2048 embedding and passed as input to a MLP layer of size 512.
  - This is followed by another MLP layer of size 10, corresponding to the 10 classes for final classification.
- **Input Batch Normalisation**
  - The model also includes batch normalization for the input image, labeled as self.whiten. This normalizes the input image's channels (RGB), potentially leading to better model performance by ensuring that the inputs have a mean of zero and a standard deviation of one.
- **He Kaiming Initialization**
  - Since the model utilises ReLU activations, He initialisation is best optimised for this.

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        # First convolutional layer in the residual block
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, 
padding=1, stride=stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        # Second convolutional layer in the residual block
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 
padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        # Shortcut connection to match dimensions if needed
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            # If input and output dimensions differ, use a 1x1 
convolution to match dimensions
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, 
stride=stride),  # 1x1 convolution
                nn.BatchNorm2d(out_channels)  # Batch normalization 
for the shortcut
            )

    def forward(self, x):
        residual = x  # Store the input for the shortcut connection
        x = self.relu(self.bn1(self.conv1(x)))  # Apply the first 
convolution, batch normalization, and ReLU activation
        x = self.bn2(self.conv2(x))  # Apply the second convolution 
and batch normalization
        x += self.shortcut(residual)  # Add the shortcut connection to 
the output
        return self.relu(x)  # Apply ReLU activation to the final 
output
```

```
class CNN(nn.Module):
    def __init__(self, dropout_prob=0.1):
        super(CNN, self).__init__()
        self.whiten = nn.BatchNorm2d(3, affine=True,
track_running_stats=True) # Batch normalization for input channels
(RGB)
        # Define the feature extraction layers
        # Modified feature extractor with residual blocks
        self.features = nn.Sequential(
            ResBlock(3, 32),          # 32x32x3 → 32x32x32
            nn.MaxPool2d(2),          # 32x32x32 → 16x16x32
            nn.Dropout(dropout_prob),

            ResBlock(32, 64),         # 16x16x32 → 16x16x64
            nn.MaxPool2d(2),          # 16x16x64 → 8x8x64
            nn.Dropout(dropout_prob),

            ResBlock(64, 128),        # 8x8x64 → 8x8x128
            nn.MaxPool2d(2),          # 8x8x128 → 4x4x128
            nn.Dropout(dropout_prob)
        )
        # Define the classifier (fully connected layers)
        self.classifier = nn.Sequential(
            nn.Linear(128 * 4 * 4, 512),     # Flattened input size:
128 * 4 * 4, Output size: 512
            nn.ReLU(),                       # Activation function
            nn.Dropout(dropout_prob),        # Dropout for
regularization
            nn.Linear(512, 10)               # Output size: 10 (number
of classes in CIFAR-10)
        )
        # Initialize weights for the layers
        self._initialize_weights()

    def forward(self, x):
        # Apply batch normalization (whitening) to the input
        x = self.whiten(x)
        # Pass input through the feature extraction layers
        x = self.features(x)
        # Flatten the output for the fully connected layers
        x = x.view(x.size(0), -1)
        # Pass through the classifier
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        # Initialize weights for Conv2D and Linear layers
        for m in self.modules():
```

```
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')  # He Kaiming initialization
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)  # Initialize biases
to 0
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)  # Initialize
weights with normal distribution
                nn.init.constant_(m.bias, 0)       # Initialize biases
to 0

    def visualise_feature_maps(self, img, layer_idx=0,
num_channels=4):
        # Function to visualize feature maps from a specific layer
        input_tensor = img.unsqueeze(0)
        assert input_tensor.dim() == 4 and input_tensor.size(0) == 1,
"Input must be a single image with shape [1, 3, H, W]"
        self.eval()

        # Pass the image through the network up to the specified layer
        with torch.no_grad():
            input_tensor = self.whiten(input_tensor)
            for i in range(layer_idx + 1):
                input_tensor = self.features[i](input_tensor)
        feature_map = input_tensor[0]  # shape: [C, H, W]

        # De-normalize original image for plotting
        img_np = img.permute(1, 2, 0).numpy()
        img_np = img_np * np.array([0.2023, 0.1994, 0.2010]) +
np.array([0.4914, 0.4822, 0.4465])  # De-normalize
        img_np = np.clip(img_np, 0, 1)

        # Plot individual channels
        fig, axes = plt.subplots(1, 5, figsize=(12, 3))
        axes = axes.flatten()

        axes[0].imshow(img_np)  # Plot original image
        axes[0].set_title("Original Image")
        axes[0].axis('off')

        for i in range(1, num_channels+1):  # Plot first 11 feature
maps
            fmap = feature_map[i - 1].cpu().numpy()
            axes[i].imshow(fmap, cmap='inferno')
            axes[i].set_title(f'Feature {i}')
            axes[i].axis('off')

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

Vision Transformer (ViT) Architecture

Our ViT architecture is implemented as follows:

- **Patch Embedding**
  - A Conv2D is used to perform both the patchification and embedding steps efficiently in a single step, significantly reducing the complexity of the patching process compared to a manual step.
  - The patching step involves splitting the 32x32 image into 64 non-overlapping patches of size 4x4. This is achieved using a filter size and stride of 4. By setting the kernel size and stride both to 4, the patches are non-overlapping, meaning that the entire image is divided into equal-sized patches without any loss or overlap between patches. This is crucial to preserve the spatial resolution of the image when learning embeddings.
  - The embedding step involves transforming each of the 64 patches into an embedding vector of length 128. This is achieved by applying 128 filters over the input, effectively projecting each patch into a 1D vector.
  - Effectively, the transformer model learns to embed the multiple patches of the image into a semantic space, where each patch carries semantic meaning understood by the model.
  - A Classify token (CLS) is initialised at the front of the sequence of patch embeddings, acting as the final embedding of all the information of the image for classification.
  - A learnable positional embedding is also randomly initialised.
- **Positional Embedding**
  - Positional embeddings are added to the patch embeddings, as they hold the spatial information of each patch relative to the others. This is an important step following the patch splitting, which had previously discarded the positional information of the patches. This way, the attention model is able to learn the relative positions of each patch, in addition to the semantic information of each patch.
  - We opted to use learnable positional embeddings, rather than fixed sinusoidal embeddings, as this would be more flexible for the model to semantically learn the relative positions of each patch in the image, rather than defining it for the model.
- **Transformer Encoder**
  - Multi-Head Attention
    - Multi-head attention (MHA) is used instead of single-head attention as it allows the model to split its focus into multiple perspectives over each patch embedding. Intuitively, it allows the model to analyse different parts or characteristics of the image.
    - 4 attention heads were selected for this model. Although increasing the number of attention heads allows for more diverse perspectives that lead to a richer contextual output, this significantly increased the computational cost of the training, and hence 4 attention heads achieved the desired balance of performance and computational efficiency.
  - LayerNorm

        - Used to ensure numeric stability in the learned weights, similar to the BatchNorm used in the CNN architecture.
    - MLP block
        - A 2-layer MLP block is used to process the embeddings, with a GELU activation function used to introduce non-linearity. GELU is used as it has been shown to outperform ReLU in transformer models.
    - Dropout
        - A low dropout rate of 0.1 was empirically tested and used to regularise the learning and reduce overfitting. A balance was achieved to prevent the model from underfitting as well.
    - Residual Connections
        - As per standard transformer architecture, 2 residual connections are used, 1 after the MHA followed by 1 after the MLP block. This helps to avoid the vanishing gradients problem.
    - Depth
        - 6 transformer layers are used, with each layer further refining the patch and CLS embeddings by learning more abstract information in the image, but not too deep to prevent overfitting.
- **CLS Token**
    - Extracted from the output and returned for final classification.

```
# ViT Architecture
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=32, patch_size=4, in_channels=3,
embed_dim=128):
        super().__init__()
        self.proj = nn.Conv2d(in_channels, embed_dim,
kernel_size=patch_size, stride=patch_size)
        num_patches = (img_size // patch_size) ** 2  # Calculate the
number of patches
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.randn(1, num_patches + 1,
embed_dim))

    def forward(self, x):
        B = x.shape[0]
        x = self.proj(x).flatten(2).transpose(1, 2)   # Perform
patchification and embedding in one step using Conv2d
        cls_tokens = self.cls_token.expand(B, 1, -1)  # Initialise CLS
token
        x = torch.cat((cls_tokens, x), dim=1)         # Prepending the
CLS token
        x = x + self.pos_embed                        # Adding
positional embedding to patch embedding
        return x

class TransformerEncoder(nn.Module):
    def __init__(self, embed_dim, num_heads, hidden_dim, dropout=0.1):
```

```
        super().__init__()
        self.norm1 = nn.LayerNorm(embed_dim)  # 1st LayerNorm
        self.attn = nn.MultiheadAttention(embed_dim, num_heads,
batch_first=True)  # Multihead Attention
        self.norm2 = nn.LayerNorm(embed_dim)  # 2nd LayerNorm
        self.mlp = nn.Sequential(  # 2-layer MLP using GELU activation
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Linear(hidden_dim, embed_dim)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, return_attention=False):
        normed = self.norm1(x)
        attn_out, attn_weights = self.attn(normed, normed, normed, #
passing in Query, Key, Value tensors

need_weights=return_attention, # return weights for visualisation
                                           average_attn_weights=False)
# to inspect each head's weights
        x = x + self.dropout(attn_out)  # 1st residual connection
        x = x + self.dropout(self.mlp(self.norm2(x)))  # 2nd residual
connection

        if return_attention:
            return x, attn_weights
        return x

class ViT(nn.Module):
    def __init__(self, num_classes=10, img_size=32, patch_size=4,
embed_dim=128, depth=6, num_heads=4, mlp_dim=512):
        super().__init__()
        self.patch_embed = PatchEmbedding(img_size, patch_size, 3,
embed_dim)
        self.encoder = nn.Sequential(*[
            TransformerEncoder(embed_dim, num_heads, mlp_dim) for _ in
range(depth)
        ])
        self.norm = nn.LayerNorm(embed_dim)
        self.head = nn.Linear(embed_dim, num_classes)

    def forward(self, x):
        x = self.patch_embed(x) # Perform patch embedding
        x = self.encoder(x)     # Perform Transformer encoding
        x = self.norm(x[:, 0])  # Return only the CLS token
        return self.head(x)

    def visualise_attention_overlay(self, img, head=None):  #
Visualise attention heads from last encoder layer
        self.eval()
```

```
        device = next(self.parameters()).device

        input_tensor = img.unsqueeze(0)

        with torch.no_grad():
            embedding = self.patch_embed(input_tensor)  # [1, N+1, E]
            for layer in self.encoder[:-1]:
                embedding = layer(embedding)

            embedding, attn_weights = self.encoder[-1](embedding,
return_attention=True)  # [1, heads, N+1, N+1]

        # Get CLS token attention weights to all patches from all
heads
        if head is not None:
            cls_attn = attn_weights[0, head, 0, 1:]
        else:
            cls_attn = attn_weights[0, :, 0, 1:].mean(dim=0)  # shape:
[num_patches]

        patch_dim = int((embedding.shape[1] - 1) ** 0.5)
        attn_map = cls_attn.reshape(patch_dim,
patch_dim).unsqueeze(0).unsqueeze(0)
        attn_map = F.interpolate(attn_map, size=(32, 32),
mode='bilinear', align_corners=False)[0, 0]

        # De-normalize original image for plotting
        img_np = img.permute(1, 2, 0).numpy()
        img_np = img_np * np.array([0.2023, 0.1994, 0.2010]) +
np.array([0.4914, 0.4822, 0.4465])  # De-normalize
        img_np = np.clip(img_np, 0, 1)

        # Prepare overlay with attention map
        fig, axes = plt.subplots(1, 2, figsize=(6, 3))

        # Plot original image
        axes[0].imshow(img_np)
        axes[0].set_title("Original Image")
        axes[0].axis('off')

        # Image + attention overlay
        axes[1].imshow(img_np)
        axes[1].imshow(attn_map.cpu(), cmap='jet', alpha=0.5,
extent=(0, 32, 32, 0))
        axes[1].set_title("Attention Overlay (Head {})".format(head+1
if head is not None else "All Heads"))
        axes[1].axis('off')

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

Training Workflow

We have opted to use Pytorch Lightning to streamline our training process:

1. Performing forward pass
2. Defining the loss function
   - Since this is a multi-class classification task, the crossentropy loss function should be used as it penalises incorrect predictions more harshly when the model is confident.
3. Tracking the average training and validation losses and accuracies per epoch for loss and accuracy curve visualisations.
4. Hyperparameter tuning
   - Optimizer
     - AdamW Optimizer was used with a weight decay of 0.05, penalising overly large weights to prevent overfitting and help generalise better.
   - Learning rate
     - Empirically tested different learning rates, and eventually found a value of 1e-3 to be optimal for both models.
     - Cosine learning rate decay was used to gradually reduce the learning rate over time and achieve smoother convergence.
   - Max epochs
     - Empirically tested different training durations, and eventually found 80 epochs gave both models sufficient time to reach convergence.

```python
# Defining Pytorch Lightning Classifier that will handle training and
validation
class Classifier(pl.LightningModule):
    def __init__(self, learning_rate=1e-3, model=None,
criterion=None):
        super().__init__()
        self.model = model
        self.lr = learning_rate
        self.criterion = criterion

        # Store loss and accuracy metrics for training and validation
        self.train_loss = []
        self.train_acc = []
        self.val_loss = []
        self.val_acc = []

        self._epoch_train_loss = []
        self._epoch_train_acc = []
        self._epoch_val_loss = []
        self._epoch_val_acc = []

    def forward(self, x):
        return self.model(x)
```

```
    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        loss = self.criterion(preds, y)
        acc = (preds.argmax(dim=1) == y).float().mean()

        # Store per-step metrics
        self._epoch_train_loss.append(loss.item())
        self._epoch_train_acc.append(acc.item())

        return loss

    def on_train_epoch_end(self):
        # Log average metrics of the epoch
        avg_loss = sum(self._epoch_train_loss) /
len(self._epoch_train_loss)
        avg_acc = sum(self._epoch_train_acc) /
len(self._epoch_train_acc)

        self.train_loss.append(avg_loss)
        self.train_acc.append(avg_acc)

        print(f"Epoch {self.current_epoch + 1} - Train Loss:
{avg_loss:.4f}, "
              f"Train Acc: {avg_acc:.4f}")

        self._epoch_train_loss = []
        self._epoch_train_acc = []

    def validation_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        loss = self.criterion(preds, y)
        acc = (preds.argmax(dim=1) == y).float().mean()

        # Store metrics per batch
        self._epoch_val_loss.append(loss.item())
        self._epoch_val_acc.append(acc.item())

        return loss

    def on_validation_epoch_end(self):
        # Log average metrics of the epoch
        avg_loss = sum(self._epoch_val_loss) /
len(self._epoch_val_loss)
        avg_acc = sum(self._epoch_val_acc) / len(self._epoch_val_acc)

        self.val_loss.append(avg_loss)
        self.val_acc.append(avg_acc)
```

```
        self._epoch_val_loss = []
        self._epoch_val_acc = []

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.model.parameters(),
lr=self.lr, weight_decay=0.05, fused=True)
        scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=80)
        return {
            'optimizer': optimizer,
            'lr_scheduler': scheduler
        }
```

Testing Workflow

```
# Generate predictions on test dataset
def predict(model, test_loader, device):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            outputs = model(images)  # [batch_size, num_classes]
            _, preds = torch.max(outputs, 1)  # get predicted class
index

            all_preds.append(preds.cpu())
            all_labels.append(labels.cpu())

    # Concatenate all batches into single tensors
    all_preds = torch.cat(all_preds)
    all_labels = torch.cat(all_labels)
    return all_preds, all_labels

# Print metrics
def print_class_metrics(true_labels, pred_labels, class_names):
    # Plot confusion matrix
    cm = confusion_matrix(true_labels, pred_labels)

    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='rocket',
                xticklabels=class_names, yticklabels=class_names)

    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Confusion Matrix')
```

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

    # Convert to NumPy arrays
    true_labels = true_labels.numpy() if hasattr(true_labels, 'numpy')
else np.array(true_labels)
    pred_labels = pred_labels.numpy() if hasattr(pred_labels, 'numpy')
else np.array(pred_labels)

    # Print classification report
    report = classification_report(true_labels, pred_labels,
target_names=class_names, output_dict=True)

    print(f"\n{'Class':<15}{'Precision':>10}{'Recall':>10}{'F1-
Score':>12}{'Accuracy':>12}")
    print("-" * 60)

    for i, class_name in enumerate(class_names):
        precision = report[class_name]['precision']
        recall = report[class_name]['recall']
        f1 = report[class_name]['f1-score']
        support = report[class_name]['support']

        # Per-class accuracy = correct predictions / total for that
class
        class_accuracy = cm[i, i] / cm[i].sum()

        print(f"{class_name:<15}{precision:10.4f}{recall:10.4f}
{f1:12.4f}{class_accuracy:12.4f}")

    overall_accuracy = (pred_labels == true_labels).mean()
    print(f"\nOverall accuracy: {overall_accuracy:41.4f}")

# Plot loss and accuracy curves
def plot_curves(classifier, name=None):
    plt.figure(figsize=(12, 5))

    # Loss curve
    plt.subplot(1, 2, 1)
    plt.plot(classifier.train_loss, label='Train Loss')
    plt.plot(classifier.val_loss, label='Validation Loss')
    plt.title(f"{name} Loss Curve")
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.legend()

    # Accuracy curve
    plt.subplot(1, 2, 2)
    plt.plot(classifier.train_acc, label='Train Accuracy')
```

```
    plt.plot(classifier.val_acc, label='Validation Accuracy')
    plt.title(f"{name} Accuracy Curve")
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.grid(True)
    plt.legend()

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

CNN implementation

Training Phase

```
# Creating PyTorch Lightning Classifier instances for CNN
cnn_classifier = Classifier(learning_rate=1e-3,
                               model=CNN().to(device),
                               criterion=nn.CrossEntropyLoss()
                               )

# Creating a PyTorch Lightning Trainer instance for training
cnn_trainer = Trainer(
    max_epochs=80,
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None
)

# Train the CNN model
start_time = time.time()  # Record time taken for training
cnn_trainer.fit(cnn_classifier, train_loader, test_loader)
end_time = time.time()
cnn_train_time = end_time - start_time
```

```
You are using the plain ModelCheckpoint callback. Consider using
LitModelCheckpoint which with seamless uploading to Model registry.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
You are using a CUDA device ('NVIDIA GeForce RTX 4060 Laptop GPU')
that has Tensor Cores. To properly utilize them, you should set
`torch.set_float32_matmul_precision('medium' | 'high')` which will
trade-off precision for performance. For more details, read
https://pytorch.org/docs/stable/generated/torch.set_float32_matmul_pre
cision.html#torch.set_float32_matmul_precision
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name      | Type              | Params | Mode
-------------------------------------------------------
```

```
0 | model     | CNN              | 1.4 M  | train
1 | criterion | CrossEntropyLoss | 0      | train
-------------------------------------------------------
1.4 M     Trainable params
0         Non-trainable params
1.4 M     Total params
5.413     Total estimated model params size (MB)
42        Modules in train mode
0         Modules in eval mode
```

{"model_id":"e75378700fc84a8e950d52e8dedc96ab","version_major":2,"version_minor":0}

```
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytorch_lightning\trainer\connectors\data_connector.py:425: The 'val_dataloader' does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` to `num_workers=15` in the `DataLoader` to improve performance.
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytorch_lightning\trainer\connectors\data_connector.py:425: The 'train_dataloader' does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` to `num_workers=15` in the `DataLoader` to improve performance.
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytorch_lightning\loops\fit_loop.py:310: The number of training batches (49) is smaller than the logging interval Trainer(log_every_n_steps=50). Set a lower value for log_every_n_steps if you want to see logs for the training epoch.
```

{"model_id":"b97ec51e824e407c91e5896b340c22ea","version_major":2,"version_minor":0}

{"model_id":"b2f7e71b694d47f0954c60d10767339c","version_major":2,"version_minor":0}

```
Epoch 1 - Train Loss: 1.7407, Train Acc: 0.3588
```

{"model_id":"8eb37f1d50df466191e2443663eef92a","version_major":2,"version_minor":0}

```
Epoch 2 - Train Loss: 1.3449, Train Acc: 0.5054
```

{"model_id":"b90c3c2b22b44496a46d587ee82d79e4","version_major":2,"version_minor":0}

```
Epoch 3 - Train Loss: 1.1735, Train Acc: 0.5748
```

{"model_id":"51706d0f5add4d92b993adba281aee8f","version_major":2,"version_minor":0}

```
Epoch 4 - Train Loss: 1.0652, Train Acc: 0.6161
```

{"model_id":"41d680c383b94055a8cd03b72a54ca66","version_major":2,"version_minor":0}

```
Epoch 5 - Train Loss: 0.9751, Train Acc: 0.6500
```

{"model_id":"5f53542c41a948c4a89694af2fd39bf8","version_major":2,"version_minor":0}

```
Epoch 6 - Train Loss: 0.9203, Train Acc: 0.6723
```

{"model_id":"d57ef2f020984a11971a37007281ba04","version_major":2,"version_minor":0}

```
Epoch 7 - Train Loss: 0.8599, Train Acc: 0.6935
```

{"model_id":"b0fd2c8ec8814495949fe1d1a6fa31f3","version_major":2,"version_minor":0}

```
Epoch 8 - Train Loss: 0.8194, Train Acc: 0.7075
```

{"model_id":"d288233b112a4853838573369337d2a8","version_major":2,"version_minor":0}

```
Epoch 9 - Train Loss: 0.7930, Train Acc: 0.7175
```

{"model_id":"5f583c2cd90f434787473a9f655dd79f","version_major":2,"version_minor":0}

```
Epoch 10 - Train Loss: 0.7547, Train Acc: 0.7321
```

{"model_id":"472063e210f94e2f8e1a3c8b39b37e09","version_major":2,"version_minor":0}

```
Epoch 11 - Train Loss: 0.7259, Train Acc: 0.7436
```

{"model_id":"48c6833b5ac1425184616bc6117cf449","version_major":2,"version_minor":0}

```
Epoch 12 - Train Loss: 0.7070, Train Acc: 0.7484
```

{"model_id":"3d1e34d8d8af49499c1b1c5f4f4e3c6c","version_major":2,"version_minor":0}

```
Epoch 13 - Train Loss: 0.6899, Train Acc: 0.7540
```

{"model_id":"e3e9be57192d40f8be32c0957c9fa5b2","version_major":2,"version_minor":0}

```
Epoch 14 - Train Loss: 0.6668, Train Acc: 0.7654
```

{"model_id":"50abd5426b024b8d915118b9221361bb","version_major":2,"version_minor":0}

```
Epoch 15 - Train Loss: 0.6539, Train Acc: 0.7700
```
{"model_id":"c53acd354bf04831870d97ada4fd1743","version_major":2,"version_minor":0}
```
Epoch 16 - Train Loss: 0.6318, Train Acc: 0.7773
```
{"model_id":"0285bb7884be4d3f9a7af47dd31ad744","version_major":2,"version_minor":0}
```
Epoch 17 - Train Loss: 0.6169, Train Acc: 0.7829
```
{"model_id":"031968fe73064df59a74baa441ffb151","version_major":2,"version_minor":0}
```
Epoch 18 - Train Loss: 0.6026, Train Acc: 0.7867
```
{"model_id":"7de7cbe847574e2c88b5a6f4a545fc2b","version_major":2,"version_minor":0}
```
Epoch 19 - Train Loss: 0.5879, Train Acc: 0.7926
```
{"model_id":"3fde654c2e0e4bd4a2f9219da60afe3c","version_major":2,"version_minor":0}
```
Epoch 20 - Train Loss: 0.5762, Train Acc: 0.7975
```
{"model_id":"f57ca164c0b04183a89b525e3234818d","version_major":2,"version_minor":0}
```
Epoch 21 - Train Loss: 0.5656, Train Acc: 0.7998
```
{"model_id":"7c44812e12d54a17aea196726f2d5c01","version_major":2,"version_minor":0}
```
Epoch 22 - Train Loss: 0.5529, Train Acc: 0.8049
```
{"model_id":"ab9c62c8ffa445cf9e5050ebb7324820","version_major":2,"version_minor":0}
```
Epoch 23 - Train Loss: 0.5401, Train Acc: 0.8120
```
{"model_id":"2485acf527e145ad9f328dbd81bc1609","version_major":2,"version_minor":0}
```
Epoch 24 - Train Loss: 0.5283, Train Acc: 0.8149
```
{"model_id":"3a6c3c3ab5b74386ad85c445285ce273","version_major":2,"version_minor":0}
```
Epoch 25 - Train Loss: 0.5170, Train Acc: 0.8148
```
{"model_id":"4ebcfda54faf4410b6c49e9beba1d7e9","version_major":2,"version_minor":0}

```
Epoch 26 - Train Loss: 0.5096, Train Acc: 0.8208
```

{"model_id":"3d6aba9b51354d9fbd061ce2af560206","version_major":2,"version_minor":0}

```
Epoch 27 - Train Loss: 0.5076, Train Acc: 0.8201
```

{"model_id":"9f0f461d7bc54b47879bf9672692f781","version_major":2,"version_minor":0}

```
Epoch 28 - Train Loss: 0.4927, Train Acc: 0.8246
```

{"model_id":"eee1ecbf89964b6d959a16f219a4b90d","version_major":2,"version_minor":0}

```
Epoch 29 - Train Loss: 0.4849, Train Acc: 0.8307
```

{"model_id":"4653ecd40a864258b2105773259d5805","version_major":2,"version_minor":0}

```
Epoch 30 - Train Loss: 0.4763, Train Acc: 0.8306
```

{"model_id":"c32607be359949cfa86cc449e2b79359","version_major":2,"version_minor":0}

```
Epoch 31 - Train Loss: 0.4698, Train Acc: 0.8324
```

{"model_id":"0f717d24b6da45c9887912ba89b33323","version_major":2,"version_minor":0}

```
Epoch 32 - Train Loss: 0.4686, Train Acc: 0.8348
```

{"model_id":"96ad8dfa0e3546b59571f5901705d478","version_major":2,"version_minor":0}

```
Epoch 33 - Train Loss: 0.4565, Train Acc: 0.8388
```

{"model_id":"7b3d301cd44a4e1aa7ed567d4adac910","version_major":2,"version_minor":0}

```
Epoch 34 - Train Loss: 0.4510, Train Acc: 0.8413
```

{"model_id":"88a5896efbbe454c998e598c75100377","version_major":2,"version_minor":0}

```
Epoch 35 - Train Loss: 0.4373, Train Acc: 0.8468
```

{"model_id":"e476f411e56b40e0b75876b8bdf0c35d","version_major":2,"version_minor":0}

```
Epoch 36 - Train Loss: 0.4365, Train Acc: 0.8440
```

{"model_id":"a04afa78471d4c59875351c7026ca924","version_major":2,"version_minor":0}

```
Epoch 37 - Train Loss: 0.4276, Train Acc: 0.8491
```

{"model_id":"41c7587da2444ef6b9c60994b714506b","version_major":2,"version_minor":0}

```
Epoch 38 - Train Loss: 0.4259, Train Acc: 0.8490
```

{"model_id":"08cc44d972f9456481d1a602476049c6","version_major":2,"version_minor":0}

```
Epoch 39 - Train Loss: 0.4195, Train Acc: 0.8519
```

{"model_id":"5c18b93812ed40eeb952575726248c84","version_major":2,"version_minor":0}

```
Epoch 40 - Train Loss: 0.4148, Train Acc: 0.8539
```

{"model_id":"d80ae9b3790b48ffa64d17bcbe1d1303","version_major":2,"version_minor":0}

```
Epoch 41 - Train Loss: 0.4099, Train Acc: 0.8542
```

{"model_id":"9439eda9f6c34aa28a66154f30360d12","version_major":2,"version_minor":0}

```
Epoch 42 - Train Loss: 0.4029, Train Acc: 0.8574
```

{"model_id":"70952fd147484d0f97443e5dd90f50a0","version_major":2,"version_minor":0}

```
Epoch 43 - Train Loss: 0.3989, Train Acc: 0.8601
```

{"model_id":"4bc92fb34e5e467db3c70392590b8774","version_major":2,"version_minor":0}

```
Epoch 44 - Train Loss: 0.3955, Train Acc: 0.8598
```

{"model_id":"111c6c7df0fb48578a617b2de56992a0","version_major":2,"version_minor":0}

```
Epoch 45 - Train Loss: 0.3843, Train Acc: 0.8635
```

{"model_id":"b6a2404f36a747e2a429beeb0afcb16b","version_major":2,"version_minor":0}

```
Epoch 46 - Train Loss: 0.3830, Train Acc: 0.8675
```

{"model_id":"24162aa00f5b4b41a0904ef15bfcf627","version_major":2,"version_minor":0}

```
Epoch 47 - Train Loss: 0.3786, Train Acc: 0.8654
```

{"model_id":"3e771b5dd74e4fbd8da3b53aa1a1b7e9","version_major":2,"version_minor":0}

```
Epoch 48 - Train Loss: 0.3720, Train Acc: 0.8685
```

{"model_id":"fcf0cb9696e84c11ac06257eda220f78","version_major":2,"version_minor":0}

```
Epoch 49 - Train Loss: 0.3716, Train Acc: 0.8687
```

{"model_id":"bb73c427218b4e6eb77b0a0d5fb8f4b7","version_major":2,"version_minor":0}

```
Epoch 50 - Train Loss: 0.3679, Train Acc: 0.8697
```

{"model_id":"b58e684415af48b7b302ecfdfba05ce7","version_major":2,"version_minor":0}

```
Epoch 51 - Train Loss: 0.3607, Train Acc: 0.8715
```

{"model_id":"1ce69088bca847da824d7bba2d501eb9","version_major":2,"version_minor":0}

```
Epoch 52 - Train Loss: 0.3552, Train Acc: 0.8737
```

{"model_id":"58ce018b52cd499ebd94b47414af9a8c","version_major":2,"version_minor":0}

```
Epoch 53 - Train Loss: 0.3556, Train Acc: 0.8740
```

{"model_id":"3a61e1db1298401d860d8abe6d56ce77","version_major":2,"version_minor":0}

```
Epoch 54 - Train Loss: 0.3476, Train Acc: 0.8769
```

{"model_id":"17d2c44d77ca436fa9ffa8f41cf07f88","version_major":2,"version_minor":0}

```
Epoch 55 - Train Loss: 0.3449, Train Acc: 0.8789
```

{"model_id":"ef905cf1f0cf4895bbe31e795d38fc23","version_major":2,"version_minor":0}

```
Epoch 56 - Train Loss: 0.3448, Train Acc: 0.8768
```

{"model_id":"b01d5882603c4bd7a664edeae0513c1e","version_major":2,"version_minor":0}

```
Epoch 57 - Train Loss: 0.3417, Train Acc: 0.8781
```

{"model_id":"7e6748d27425458abd6fb5a153dfd9bd","version_major":2,"version_minor":0}

```
Epoch 58 - Train Loss: 0.3358, Train Acc: 0.8813
```

{"model_id":"63d8ac6591144801af398fdf0afad44c","version_major":2,"version_minor":0}

```
Epoch 59 - Train Loss: 0.3371, Train Acc: 0.8794
```

{"model_id":"e6abb50e0f7641d7a5e99a1ba8348864","version_major":2,"version_minor":0}

```
Epoch 60 - Train Loss: 0.3303, Train Acc: 0.8823
```

{"model_id":"b017d00beabd44b4bfba63a3b458c465","version_major":2,"version_minor":0}

```
Epoch 61 - Train Loss: 0.3335, Train Acc: 0.8813
```

{"model_id":"f3b22b9254ee40ecb98aee31a2809440","version_major":2,"version_minor":0}

```
Epoch 62 - Train Loss: 0.3292, Train Acc: 0.8811
```

{"model_id":"fa818a77fa764d47beb7370f969e7e83","version_major":2,"version_minor":0}

```
Epoch 63 - Train Loss: 0.3279, Train Acc: 0.8829
```

{"model_id":"ab25eb1d97e14b5497c85342611a0d7d","version_major":2,"version_minor":0}

```
Epoch 64 - Train Loss: 0.3264, Train Acc: 0.8834
```

{"model_id":"2e4f57e4b7b24c38a3db00e67574a224","version_major":2,"version_minor":0}

```
Epoch 65 - Train Loss: 0.3190, Train Acc: 0.8874
```

{"model_id":"fffc397d275d4bdca0a5b79a121123e7","version_major":2,"version_minor":0}

```
Epoch 66 - Train Loss: 0.3213, Train Acc: 0.8848
```

{"model_id":"830217dcb4154899bc112b58b9c4dd92","version_major":2,"version_minor":0}

```
Epoch 67 - Train Loss: 0.3152, Train Acc: 0.8888
```

{"model_id":"a4d5b933c84a49d58cc226b47a5c69ea","version_major":2,"version_minor":0}

```
Epoch 68 - Train Loss: 0.3150, Train Acc: 0.8894
```

{"model_id":"3c9bc6b9b0a146e68b2d72fc5c707507","version_major":2,"version_minor":0}

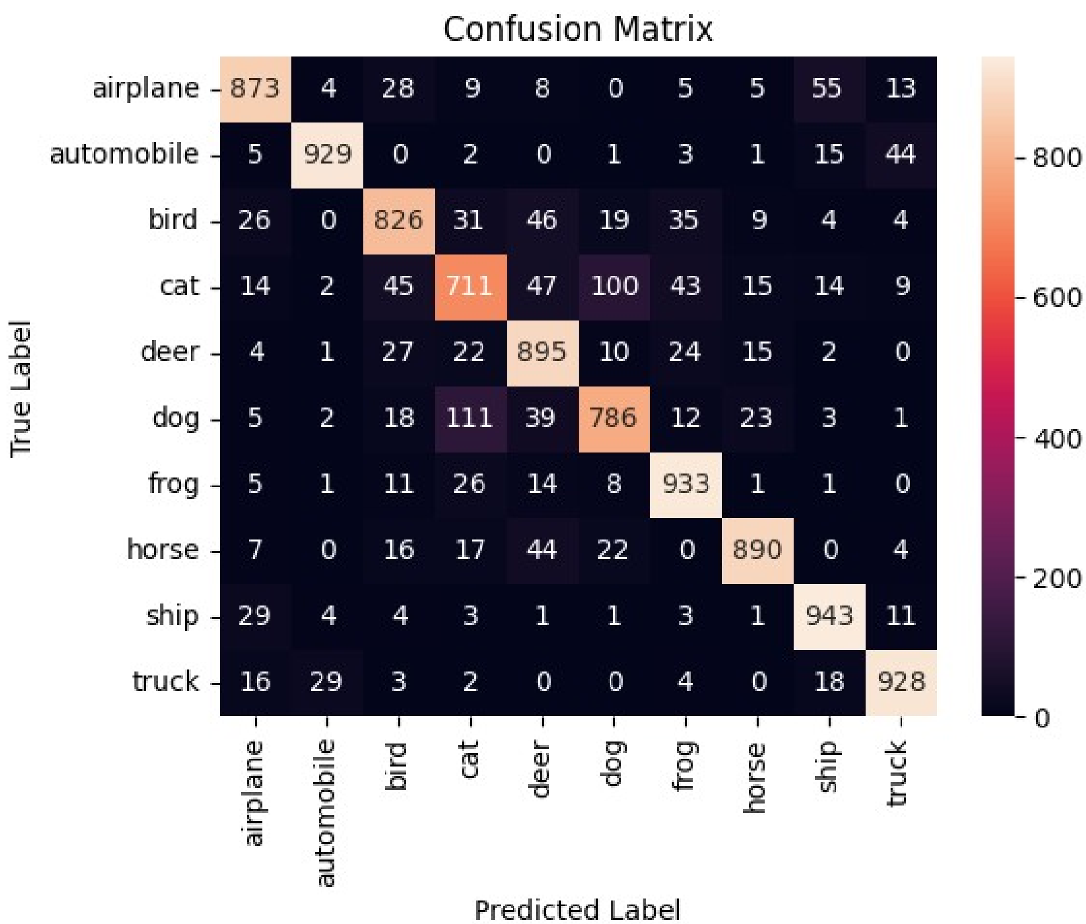
```
Epoch 69 - Train Loss: 0.3147, Train Acc: 0.8888
```

{"model_id":"e58cfa8b1cf846e98c0d7d8d90557447","version_major":2,"version_minor":0}

```
Epoch 70 - Train Loss: 0.3130, Train Acc: 0.8908
```
{"model_id":"ab434b58304c491ab071018af48d41ae","version_major":2,"version_minor":0}
```
Epoch 71 - Train Loss: 0.3117, Train Acc: 0.8891
```
{"model_id":"999ab663713b415099865326b4882ad5","version_major":2,"version_minor":0}
```
Epoch 72 - Train Loss: 0.3084, Train Acc: 0.8907
```
{"model_id":"5f089e6364c24597b6394d757cfddde7","version_major":2,"version_minor":0}
```
Epoch 73 - Train Loss: 0.3119, Train Acc: 0.8891
```
{"model_id":"8aa4b9379aeb48078f88e449f120d4b2","version_major":2,"version_minor":0}
```
Epoch 74 - Train Loss: 0.3075, Train Acc: 0.8920
```
{"model_id":"468b4fcf118944338b6762eecd5b1198","version_major":2,"version_minor":0}
```
Epoch 75 - Train Loss: 0.3113, Train Acc: 0.8895
```
{"model_id":"124e338012114cdb9f09b08eb585c147","version_major":2,"version_minor":0}
```
Epoch 76 - Train Loss: 0.3115, Train Acc: 0.8898
```
{"model_id":"b5fa5eff9f7d40369add9d6ca29e2791","version_major":2,"version_minor":0}
```
Epoch 77 - Train Loss: 0.3089, Train Acc: 0.8906
```
{"model_id":"061e784b0caf43bfb6b2763358b19f11","version_major":2,"version_minor":0}
```
Epoch 78 - Train Loss: 0.3082, Train Acc: 0.8914
```
{"model_id":"9d1f30e95f7a4bbb82afb93fbf1df428","version_major":2,"version_minor":0}
```
Epoch 79 - Train Loss: 0.3063, Train Acc: 0.8926
```
{"model_id":"a35a4766d18b4ecbbb1571b1ee84ea62","version_major":2,"version_minor":0}
```
`Trainer.fit` stopped: `max_epochs=80` reached.

Epoch 80 - Train Loss: 0.3108, Train Acc: 0.8890
```

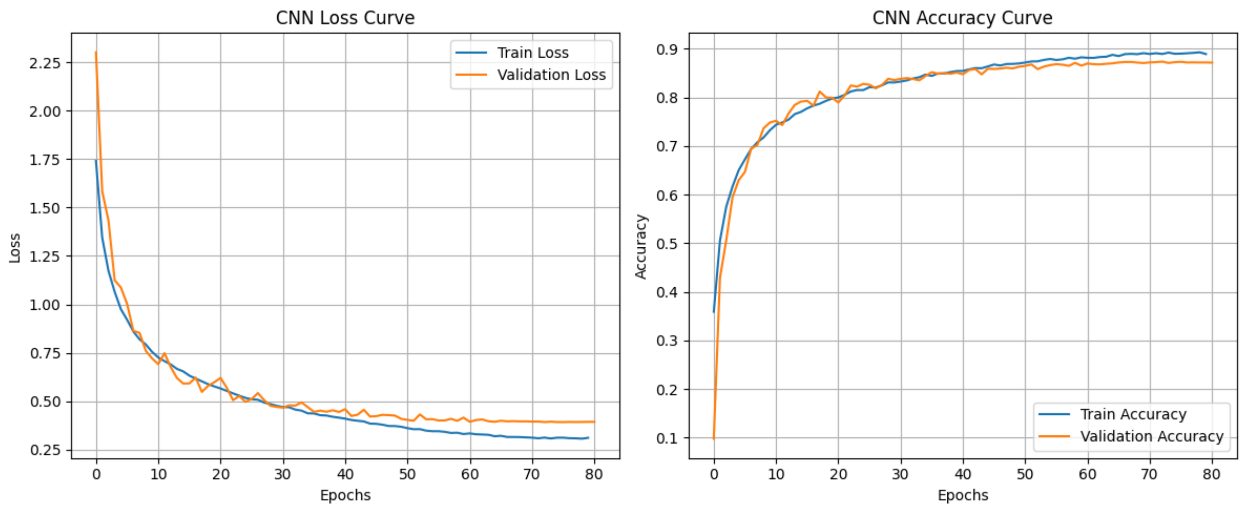
Testing, Metrics & Visualisations

```
# Generate predictions for CNN model
predictions_cnn, labels_cnn = predict(cnn_classifier.model,
test_loader, cnn_classifier.device)

# Print metrics, loss and accuracy curves for CNN model
print_class_metrics(labels_cnn, predictions_cnn, classes)
plot_curves(cnn_classifier, name='CNN')
print(f"Time taken to train CNN: {cnn_train_time:.2f} seconds")
```




```
Class           Precision    Recall    F1-Score    Accuracy
-----------------------------------------------------------
airplane           0.8872    0.8730      0.8800      0.8730
automobile         0.9558    0.9290      0.9422      0.9290
bird               0.8446    0.8260      0.8352      0.8260
cat                0.7612    0.7110      0.7353      0.7110
deer               0.8181    0.8950      0.8548      0.8950
dog                0.8300    0.7860      0.8074      0.7860
frog               0.8785    0.9330      0.9049      0.9330
```

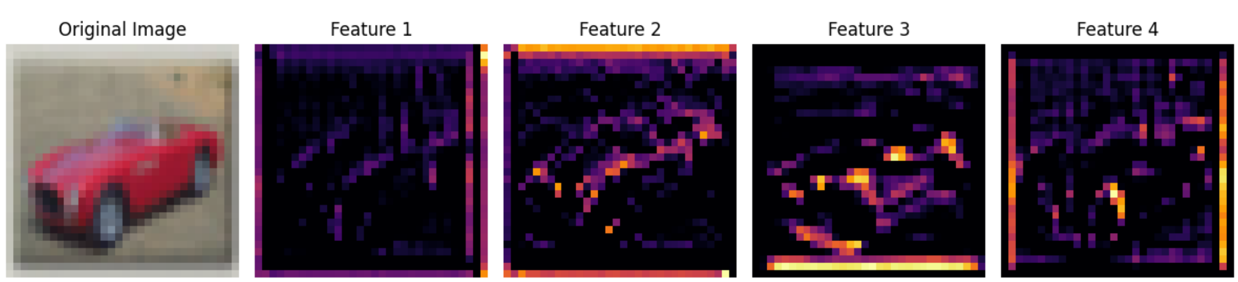
```
horse                  0.9271    0.8900      0.9082      0.8900
ship                   0.8938    0.9430      0.9178      0.9430
truck                  0.9152    0.9280      0.9215      0.9280

Overall accuracy:                                        0.8714
```

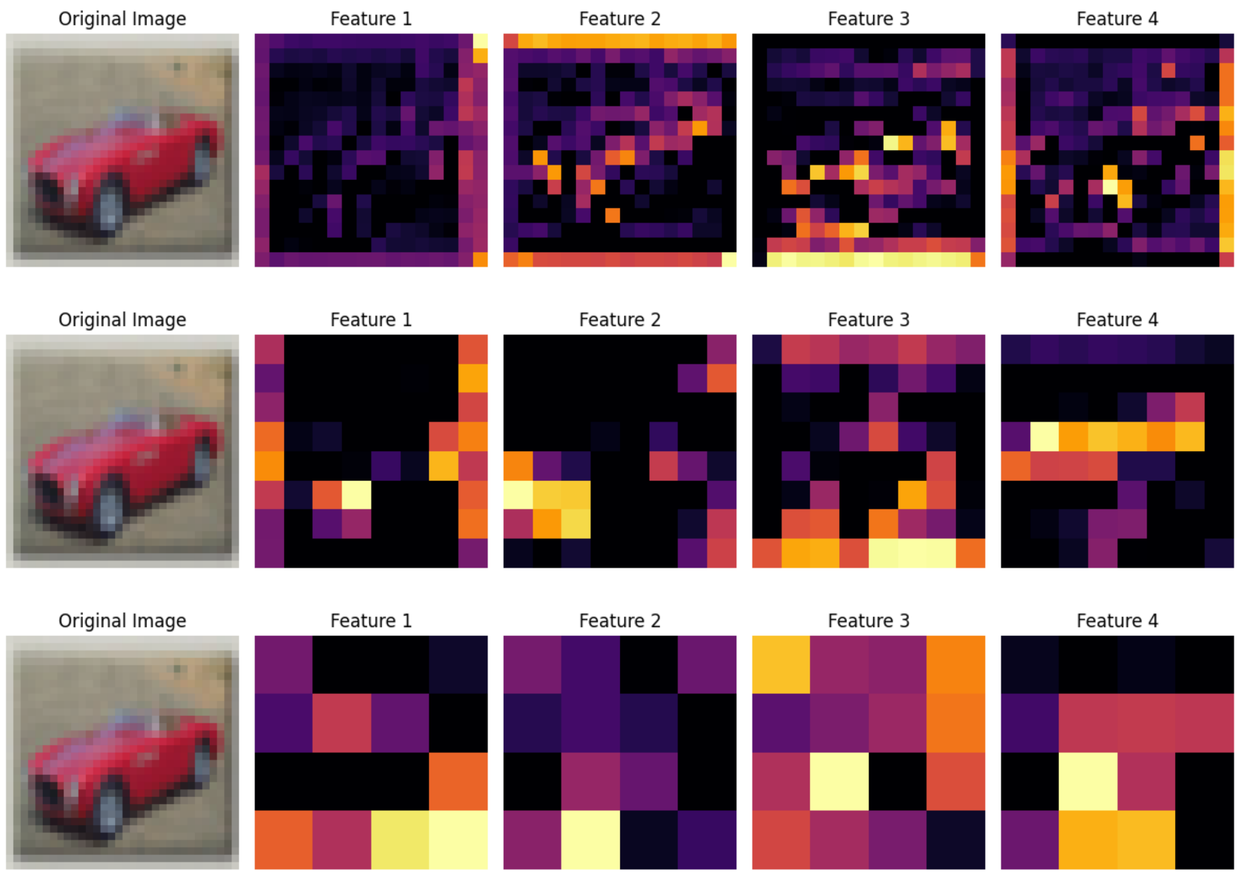



```
Time taken to train CNN: 1127.00 seconds
```

Visualise Sample Feature Maps

```
# Get one image from the training set
img = test_dataset[1234][0]

# Call the visualisation function to see feature maps
cnn_classifier.model.visualise_feature_maps(img, layer_idx=0,
num_channels=4)
cnn_classifier.model.visualise_feature_maps(img, layer_idx=1,
num_channels=4)
cnn_classifier.model.visualise_feature_maps(img, layer_idx=4,
num_channels=4)
cnn_classifier.model.visualise_feature_maps(img, layer_idx=7,
num_channels=4)
```





ViT implementation

Training Phase

```
# Creating PyTorch Lightning Classifier instances for ViT
vit_classifier = Classifier(learning_rate=1e-3,
                               model=ViT().to(device),
                               criterion=nn.CrossEntropyLoss()
                               )

# Creating a PyTorch Lightning Trainer instance for training
vit_trainer = Trainer(
    max_epochs=80,
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None
)

# Train the ViT model
start_time = time.time()  # Record time taken for training
vit_trainer.fit(vit_classifier, train_loader, test_loader)
end_time = time.time()
vit_train_time = end_time - start_time
```

```
You are using the plain ModelCheckpoint callback. Consider using
LitModelCheckpoint which with seamless uploading to Model registry.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name      | Type             | Params | Mode
-------------------------------------------------------
0 | model     | ViT              | 1.2 M  | train
1 | criterion | CrossEntropyLoss | 0      | train
-------------------------------------------------------
1.2 M     Trainable params
0         Non-trainable params
1.2 M     Total params
4.824     Total estimated model params size (MB)
67        Modules in train mode
0         Modules in eval mode
```

{"model_id":"ed9a64b486074648a05b9d581c39a162","version_major":2,"version_minor":0}

```
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:425:
The 'val_dataloader' does not have many workers which may be a
bottleneck. Consider increasing the value of the `num_workers`
argument` to `num_workers=15` in the `DataLoader` to improve
performance.
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:425:
The 'train_dataloader' does not have many workers which may be a
bottleneck. Consider increasing the value of the `num_workers`
argument` to `num_workers=15` in the `DataLoader` to improve
performance.
c:\Users\andre\AppData\Local\Programs\Python\Python312\Lib\site-
packages\pytorch_lightning\loops\fit_loop.py:310: The number of
training batches (49) is smaller than the logging interval
Trainer(log_every_n_steps=50). Set a lower value for log_every_n_steps
if you want to see logs for the training epoch.
```

{"model_id":"346333eaab0f430ca2b6aeb50f62ddd3","version_major":2,"version_minor":0}

{"model_id":"5ea9b91e66e540b9b701462456e6e5ae","version_major":2,"version_minor":0}

```
Epoch 1 - Train Loss: 2.0774, Train Acc: 0.2286
```

{"model_id":"84b5b88adb8949f4b0dac1be9101dac3","version_major":2,"version_minor":0}

```
Epoch 2 - Train Loss: 1.7724, Train Acc: 0.3473
```

{"model_id":"cf56001e3b8f4bcba9be9a1f5480a91c","version_major":2,"version_minor":0}

```
Epoch 3 - Train Loss: 1.5595, Train Acc: 0.4327
```

{"model_id":"cc13222bd7e0407ea09d922b57340b9c","version_major":2,"version_minor":0}

```
Epoch 4 - Train Loss: 1.4607, Train Acc: 0.4710
```

{"model_id":"6b5e499049324820a1645e9c5f33965c","version_major":2,"version_minor":0}

```
Epoch 5 - Train Loss: 1.3891, Train Acc: 0.4952
```

{"model_id":"79d50840ab684522b5527adf44a9fe8d","version_major":2,"version_minor":0}

```
Epoch 6 - Train Loss: 1.3285, Train Acc: 0.5201
```

{"model_id":"b78e0fe3f048456f928904241349085c","version_major":2,"version_minor":0}

```
Epoch 7 - Train Loss: 1.2870, Train Acc: 0.5376
```

{"model_id":"4cdef95a83e24d208769fb19f678a8e2","version_major":2,"version_minor":0}

```
Epoch 8 - Train Loss: 1.2378, Train Acc: 0.5528
```

{"model_id":"4e83a78c1e43440e9eaf7f8fda2a6d02","version_major":2,"version_minor":0}

```
Epoch 9 - Train Loss: 1.2156, Train Acc: 0.5626
```

{"model_id":"3d5135810d3347ceb543f90bb6aa67c4","version_major":2,"version_minor":0}

```
Epoch 10 - Train Loss: 1.1745, Train Acc: 0.5790
```

{"model_id":"cd8a943fb7824eb98ed499f417740df1","version_major":2,"version_minor":0}

```
Epoch 11 - Train Loss: 1.1529, Train Acc: 0.5868
```

{"model_id":"9ca9a3dc8a204620b26f7a86722195ed","version_major":2,"version_minor":0}

```
Epoch 12 - Train Loss: 1.1246, Train Acc: 0.5976
```

{"model_id":"592836cdba6b424e806c9fadcb28cfc7","version_major":2,"version_minor":0}

```
Epoch 13 - Train Loss: 1.0918, Train Acc: 0.6114
```

{"model_id":"1c26d539fca6453ea1e51bebc5ad0cea","version_major":2,"version_minor":0}

```
Epoch 14 - Train Loss: 1.0608, Train Acc: 0.6218
```

{"model_id":"546c1861061c478db6eca15267e9f21e","version_major":2,"version_minor":0}

```
Epoch 15 - Train Loss: 1.0350, Train Acc: 0.6294
```

{"model_id":"32c6d29e301b46e89962a80e20e785ca","version_major":2,"version_minor":0}

```
Epoch 16 - Train Loss: 1.0099, Train Acc: 0.6406
```

{"model_id":"a0e51e4f2a844bcc9c8a7d5f401852a1","version_major":2,"version_minor":0}

```
Epoch 17 - Train Loss: 0.9914, Train Acc: 0.6473
```

{"model_id":"1826e01461194300a60f18850e733cc8","version_major":2,"version_minor":0}

```
Epoch 18 - Train Loss: 0.9697, Train Acc: 0.6533
```

{"model_id":"a08dd6ed44ba428aaa773a2989f51a26","version_major":2,"version_minor":0}

```
Epoch 19 - Train Loss: 0.9454, Train Acc: 0.6631
```

{"model_id":"12c79bb0652e401186da204c5e0334c6","version_major":2,"version_minor":0}

```
Epoch 20 - Train Loss: 0.9264, Train Acc: 0.6720
```

{"model_id":"aed1c657cac44458a2ef2542060541f7","version_major":2,"version_minor":0}

```
Epoch 21 - Train Loss: 0.9014, Train Acc: 0.6799
```

{"model_id":"6cc25e25cfcf431696b869878b0d449e","version_major":2,"version_minor":0}

```
Epoch 22 - Train Loss: 0.9042, Train Acc: 0.6795
```

{"model_id":"0071c6684f8f4db2a6f01c5db95a924d","version_major":2,"version_minor":0}

```
Epoch 23 - Train Loss: 0.8701, Train Acc: 0.6908
```

{"model_id":"779c8f5a6655400dac94802728687899","version_major":2,"version_minor":0}

```
Epoch 24 - Train Loss: 0.8558, Train Acc: 0.6959
```

{"model_id":"cff33cde04a14ffbae9c34ec6b21e5b7","version_major":2,"version_minor":0}

```
Epoch 25 - Train Loss: 0.8387, Train Acc: 0.7020
```

{"model_id":"385c5df938224a41be24c85e0a4ceb32","version_major":2,"version_minor":0}

```
Epoch 26 - Train Loss: 0.8195, Train Acc: 0.7085
```

{"model_id":"52fd243257cc4c5fb8046f8779ea07e4","version_major":2,"version_minor":0}

```
Epoch 27 - Train Loss: 0.8036, Train Acc: 0.7150
```

{"model_id":"7cf5491838434c518b8e0eeacada20ae","version_major":2,"version_minor":0}

```
Epoch 28 - Train Loss: 0.7840, Train Acc: 0.7235
```

{"model_id":"beeaacedbd584fa5b14c108f2cca6a2c","version_major":2,"version_minor":0}

```
Epoch 29 - Train Loss: 0.7794, Train Acc: 0.7212
```

{"model_id":"d3c2188a4a9f4924892511098880e1cd","version_major":2,"version_minor":0}

```
Epoch 30 - Train Loss: 0.7643, Train Acc: 0.7281
```

{"model_id":"1dadd7b8e23d44069395e833d00ee40c","version_major":2,"version_minor":0}

```
Epoch 31 - Train Loss: 0.7482, Train Acc: 0.7346
```

{"model_id":"f2b9e0499db24439877e9944172dbd59","version_major":2,"version_minor":0}

```
Epoch 32 - Train Loss: 0.7415, Train Acc: 0.7363
```

{"model_id":"77c63332d3144c5d92878f1a2059af41","version_major":2,"version_minor":0}

```
Epoch 33 - Train Loss: 0.7223, Train Acc: 0.7441
```

{"model_id":"d60f269b1ef242c489c97402cfdb96d7","version_major":2,"version_minor":0}

```
Epoch 34 - Train Loss: 0.7035, Train Acc: 0.7505
```

{"model_id":"c07c4e6f4cb34a3e8f6847db834cd335","version_major":2,"version_minor":0}

```
Epoch 35 - Train Loss: 0.6813, Train Acc: 0.7572
```

{"model_id":"3aea353cb0c24381af83828e190d849b","version_major":2,"version_minor":0}

```
Epoch 36 - Train Loss: 0.6750, Train Acc: 0.7590
```

{"model_id":"8ebf8e07c5a34ebfb06d6101647bb708","version_major":2,"version_minor":0}

```
Epoch 37 - Train Loss: 0.6611, Train Acc: 0.7644
```

{"model_id":"cdec4f5db214418899fe5a03b9fd2a8d","version_major":2,"version_minor":0}

```
Epoch 38 - Train Loss: 0.6494, Train Acc: 0.7667
```

{"model_id":"b9f5f1af8f7c427da50c02f74bbccd46","version_major":2,"version_minor":0}

```
Epoch 39 - Train Loss: 0.6317, Train Acc: 0.7764
```

{"model_id":"ed0d0a09b3f746cf90a6e3cc3898a91a","version_major":2,"version_minor":0}

```
Epoch 40 - Train Loss: 0.6255, Train Acc: 0.7771
```

{"model_id":"3947183f9a214d6f9d95e93e0dcec752","version_major":2,"version_minor":0}

```
Epoch 41 - Train Loss: 0.6111, Train Acc: 0.7836
```

{"model_id":"37abba4a23d54a52b81c4bd01476bf71","version_major":2,"version_minor":0}

```
Epoch 42 - Train Loss: 0.5949, Train Acc: 0.7876
```

{"model_id":"04009bd70f6a4c5f966053523c478774","version_major":2,"version_minor":0}

```
Epoch 43 - Train Loss: 0.5850, Train Acc: 0.7922
```

{"model_id":"18656254e43f4040b7952d7af17b2c03","version_major":2,"version_minor":0}

```
Epoch 44 - Train Loss: 0.5755, Train Acc: 0.7953
```

{"model_id":"392044142edb48cdb3d78e55e33747dc","version_major":2,"version_minor":0}

```
Epoch 45 - Train Loss: 0.5650, Train Acc: 0.7989
```

{"model_id":"c16e003c51644c5b97e84736f6aa09b1","version_major":2,"version_minor":0}

```
Epoch 46 - Train Loss: 0.5503, Train Acc: 0.8020
```

{"model_id":"374e649a12e04d57aad74914358fed09","version_major":2,"version_minor":0}

```
Epoch 47 - Train Loss: 0.5351, Train Acc: 0.8086
```

{"model_id":"7f4b527652144a7abfdc7893ec37e7c8","version_major":2,"version_minor":0}

```
Epoch 48 - Train Loss: 0.5240, Train Acc: 0.8144
```

{"model_id":"49d02c2d31b246258e8350637655ece0","version_major":2,"version_minor":0}

```
Epoch 49 - Train Loss: 0.5103, Train Acc: 0.8182
```

{"model_id":"08eaa5ae8f8e4f898da3ae7c6c3169b0","version_major":2,"version_minor":0}

```
Epoch 50 - Train Loss: 0.5062, Train Acc: 0.8186
```

{"model_id":"32c2007a9708430591cc406118555251","version_major":2,"version_minor":0}

```
Epoch 51 - Train Loss: 0.4926, Train Acc: 0.8227
```

{"model_id":"42f5592420e54bb5a7c1fd9d1ec281e4","version_major":2,"version_minor":0}

```
Epoch 52 - Train Loss: 0.4798, Train Acc: 0.8299
```

{"model_id":"615aef8dcffd454388c594c0019173f5","version_major":2,"version_minor":0}

```
Epoch 53 - Train Loss: 0.4758, Train Acc: 0.8293
```

{"model_id":"ec74d18c2d5a4af99f0913bfe3717f14","version_major":2,"version_minor":0}

```
Epoch 54 - Train Loss: 0.4631, Train Acc: 0.8332
```

{"model_id":"5ef31dc31b7549b083f52cecbecc16da","version_major":2,"version_minor":0}

```
Epoch 55 - Train Loss: 0.4516, Train Acc: 0.8383
```

{"model_id":"21c3e184cc3c422a9806b36c5bb0419d","version_major":2,"version_minor":0}

```
Epoch 56 - Train Loss: 0.4411, Train Acc: 0.8427
```

{"model_id":"26482d946be44f8ea668d386e00e8e26","version_major":2,"version_minor":0}

```
Epoch 57 - Train Loss: 0.4316, Train Acc: 0.8469
```

{"model_id":"81e98dac257247b586a5506373de9ec4","version_major":2,"version_minor":0}

```
Epoch 58 - Train Loss: 0.4285, Train Acc: 0.8468
```

{"model_id":"500254895da14a2eaa4a8d348ecd84bf","version_major":2,"version_minor":0}

```
Epoch 59 - Train Loss: 0.4235, Train Acc: 0.8499
```

{"model_id":"8f7cb48e1c034a308c13aa13bbd90167","version_major":2,"version_minor":0}

```
Epoch 60 - Train Loss: 0.4132, Train Acc: 0.8510
```

{"model_id":"4298a83da0014e368f07eb738a1667bb","version_major":2,"version_minor":0}

```
Epoch 61 - Train Loss: 0.3993, Train Acc: 0.8560
```

{"model_id":"0eb42999ea874ae2ac37f32df3a0b427","version_major":2,"version_minor":0}

```
Epoch 62 - Train Loss: 0.3996, Train Acc: 0.8569
```

{"model_id":"2c0b0253f554443b9854f8fbad9a15bb","version_major":2,"version_minor":0}

```
Epoch 63 - Train Loss: 0.3922, Train Acc: 0.8605
```

{"model_id":"5ca5c69382a4445893337b3e60adff08","version_major":2,"version_minor":0}

```
Epoch 64 - Train Loss: 0.3819, Train Acc: 0.8646
```

{"model_id":"d9b2f36ea0b54373b76d0b2a6d08b76e","version_major":2,"version_minor":0}

```
Epoch 65 - Train Loss: 0.3751, Train Acc: 0.8655
```

{"model_id":"8d689718738e4448a2ef669d0043a35c","version_major":2,"version_minor":0}

```
Epoch 66 - Train Loss: 0.3732, Train Acc: 0.8667
```

{"model_id":"3b3d43abe5aa446f863d941ab4d87b69","version_major":2,"version_minor":0}

```
Epoch 67 - Train Loss: 0.3723, Train Acc: 0.8657
```

{"model_id":"b0061bcbf3404d8abc9dd614b9a6fe39","version_major":2,"version_minor":0}

Epoch 68 - Train Loss: 0.3652, Train Acc: 0.8709

{"model_id":"339cdb852ee445a9b769eda02593608f","version_major":2,"version_minor":0}

Epoch 69 - Train Loss: 0.3631, Train Acc: 0.8698

{"model_id":"e037b805392642fe8342a9e3153cc6ae","version_major":2,"version_minor":0}

Epoch 70 - Train Loss: 0.3564, Train Acc: 0.8737

{"model_id":"b884501c0f6d4e1ba7bee2c7c534ede3","version_major":2,"version_minor":0}

Epoch 71 - Train Loss: 0.3558, Train Acc: 0.8741

{"model_id":"8f5cb3bcf1cb4a9fa089156636fe24e4","version_major":2,"version_minor":0}

Epoch 72 - Train Loss: 0.3489, Train Acc: 0.8739

{"model_id":"136cefd646314478b2b49138ebc61a40","version_major":2,"version_minor":0}

Epoch 73 - Train Loss: 0.3505, Train Acc: 0.8745

{"model_id":"01b54641450a4375b7ebc12b7851f5da","version_major":2,"version_minor":0}

Epoch 74 - Train Loss: 0.3481, Train Acc: 0.8766

{"model_id":"acd2c83380494737aabee03261c84877","version_major":2,"version_minor":0}

Epoch 75 - Train Loss: 0.3460, Train Acc: 0.8764

{"model_id":"df813e4af3f54e5c951f05674d21e2e4","version_major":2,"version_minor":0}

Epoch 76 - Train Loss: 0.3441, Train Acc: 0.8767

{"model_id":"c4c70f049eee41d19421c01b5fa4e0ca","version_major":2,"version_minor":0}

Epoch 77 - Train Loss: 0.3436, Train Acc: 0.8762

{"model_id":"e4ce73a397da482298f33a6fd3d5f317","version_major":2,"version_minor":0}

Epoch 78 - Train Loss: 0.3436, Train Acc: 0.8790

{"model_id":"86a8df225ea24130ba7f91eb34e53601","version_major":2,"version_minor":0}

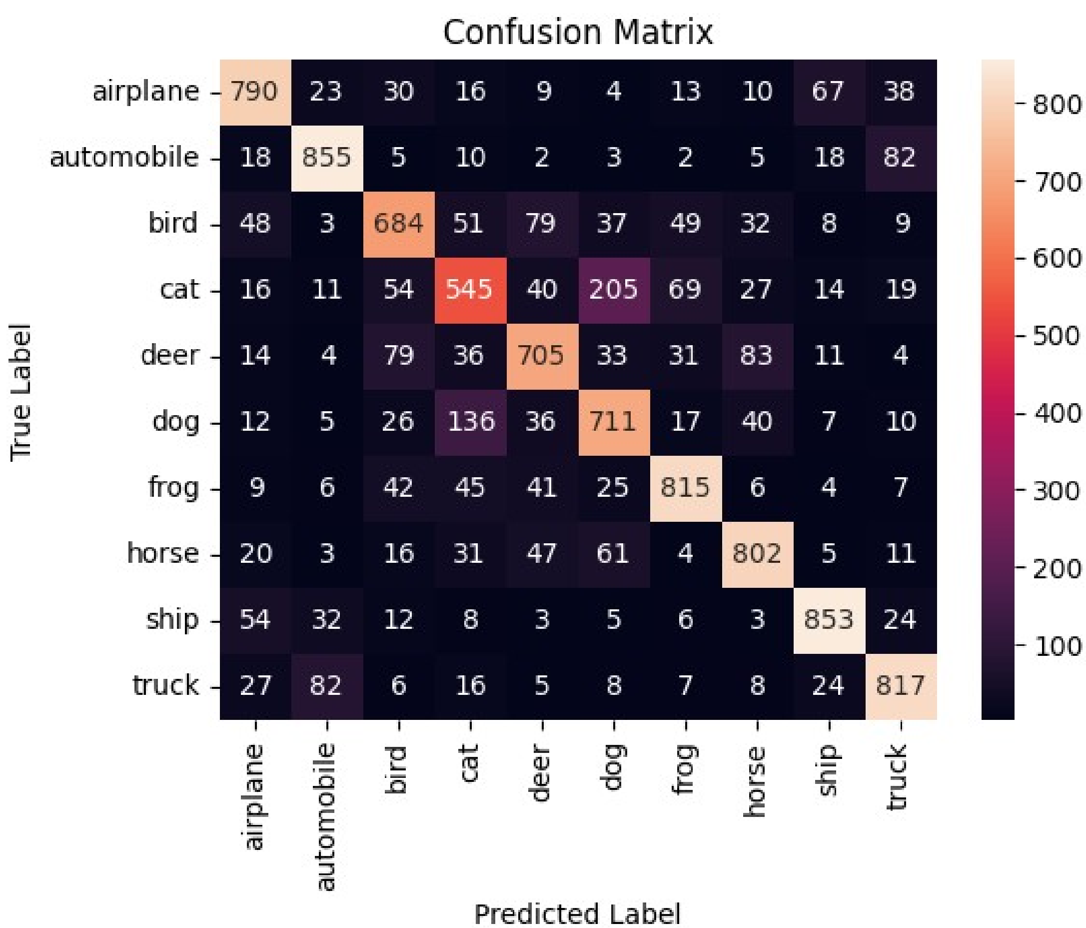
```
Epoch 79 - Train Loss: 0.3409, Train Acc: 0.8783
```

{"model_id":"4bf2e7a3b1ce4e0a84ecf2e0b163a001","version_major":2,"version_minor":0}

```
`Trainer.fit` stopped: `max_epochs=80` reached.

Epoch 80 - Train Loss: 0.3386, Train Acc: 0.8791
```

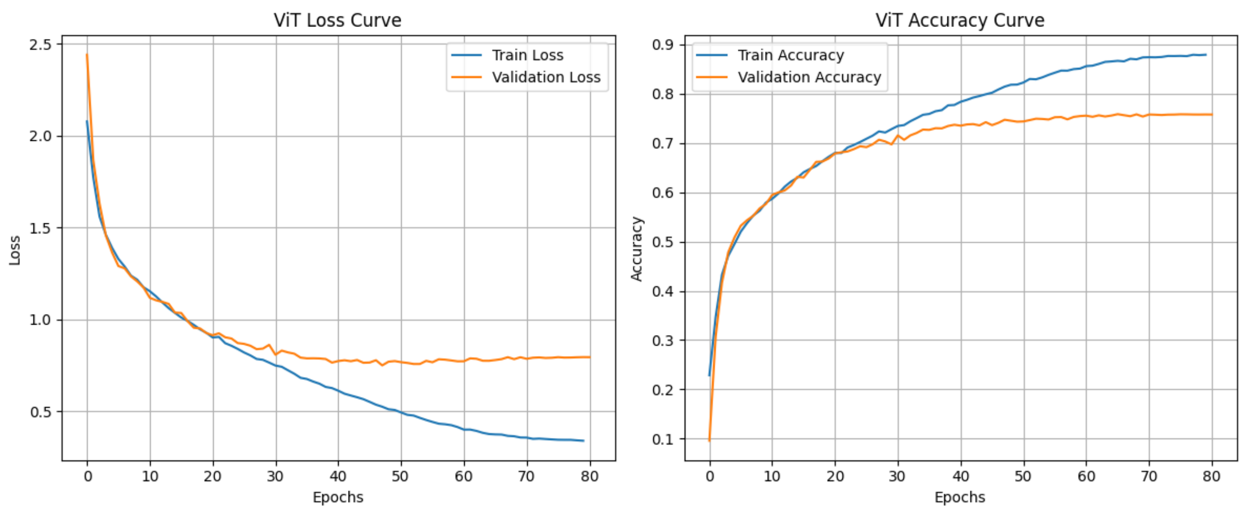
Testing, Metrics & Visualisations

```
# Generate predictions for ViT model
predictions_vit, labels_vit = predict(vit_classifier.model,
test_loader, vit_classifier.device)

# Print metrics, loss and accuracy curves for ViT model
print_class_metrics(labels_vit, predictions_vit, classes)
plot_curves(vit_classifier, name='ViT')
print(f"Time taken to train ViT: {vit_train_time:.2f} seconds")
```



True \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	790	23	30	16	9	4	13	10	67	38
automobile	18	855	5	10	2	3	2	5	18	82
bird	48	3	684	51	79	37	49	32	8	9
cat	16	11	54	545	40	205	69	27	14	19
deer	14	4	79	36	705	33	31	83	11	4
dog	12	5	26	136	36	711	17	40	7	10
frog	9	6	42	45	41	25	815	6	4	7
horse	20	3	16	31	47	61	4	802	5	11
ship	54	32	12	8	3	5	6	3	853	24
truck	27	82	6	16	5	8	7	8	24	817

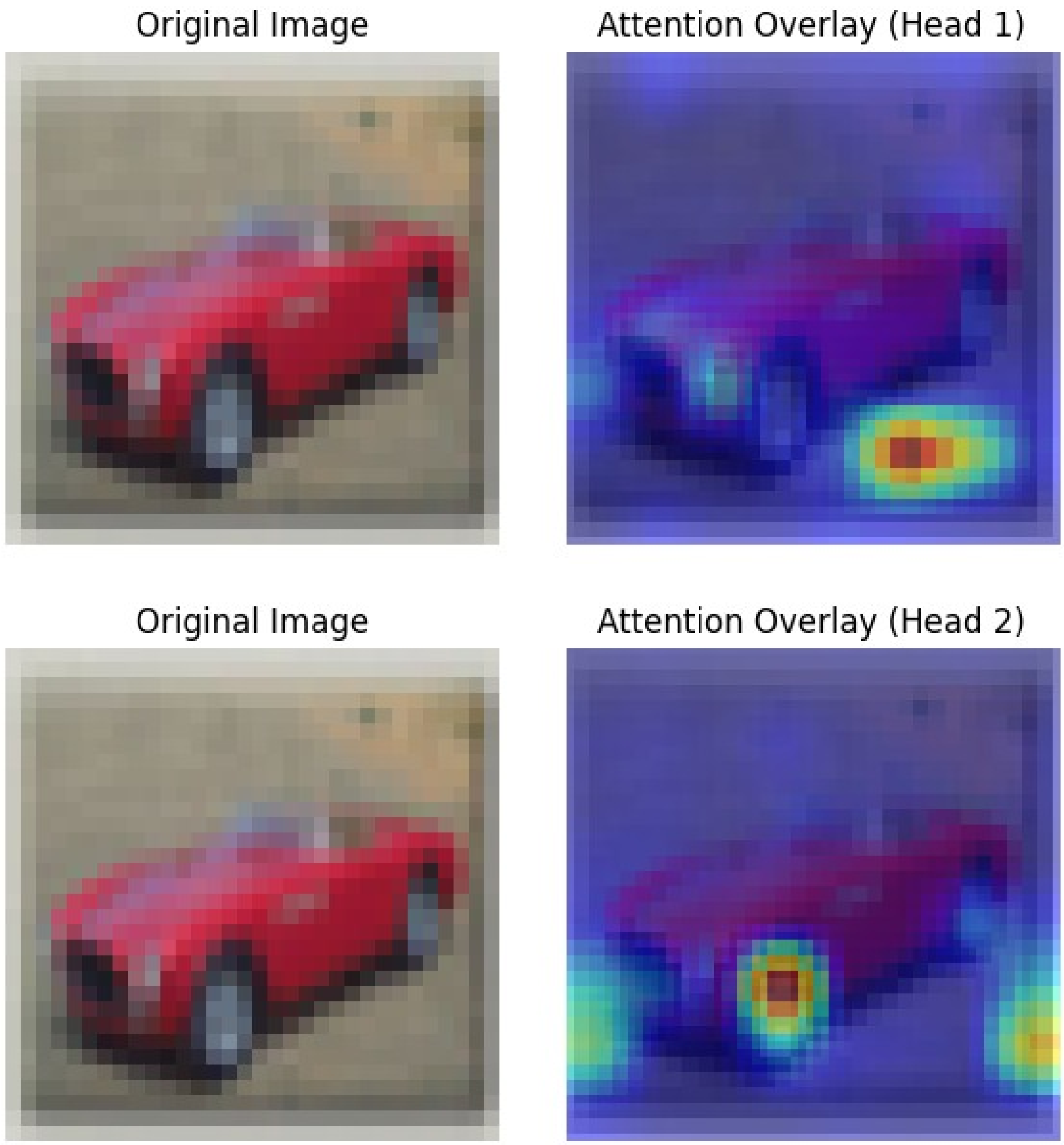
```
Class              Precision    Recall     F1-Score     Accuracy
------------------------------------------------------------------
airplane              0.7837    0.7900       0.7869       0.7900
automobile            0.8350    0.8550       0.8449       0.8550
bird                  0.7170    0.6840       0.7001       0.6840
cat                   0.6096    0.5450       0.5755       0.5450
deer                  0.7291    0.7050       0.7168       0.7050
dog                   0.6511    0.7110       0.6797       0.7110
frog                  0.8045    0.8150       0.8097       0.8150
horse                 0.7894    0.8020       0.7956       0.8020
ship                  0.8437    0.8530       0.8483       0.8530
truck                 0.8002    0.8170       0.8085       0.8170

Overall accuracy:                                         0.7577
```

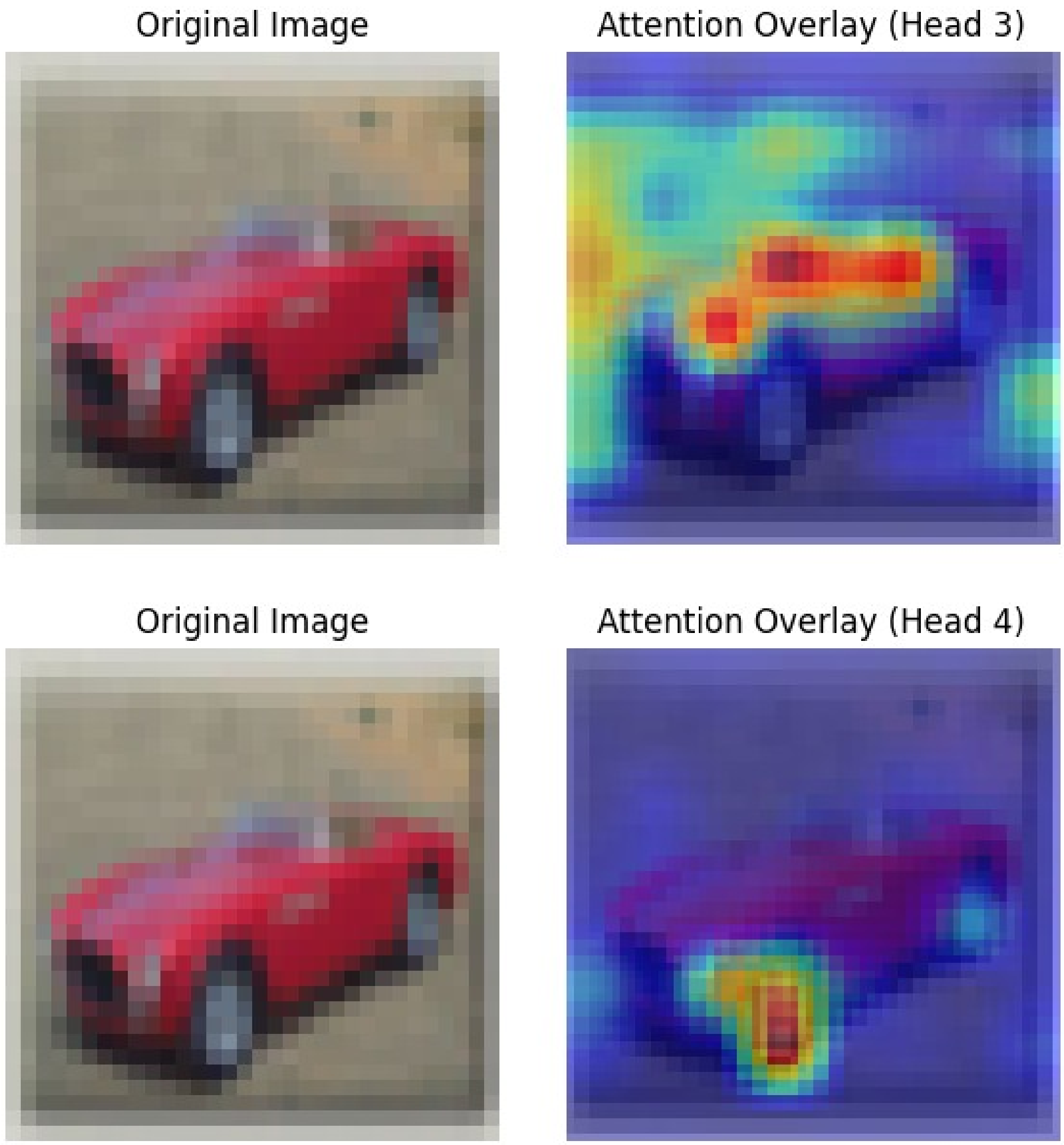


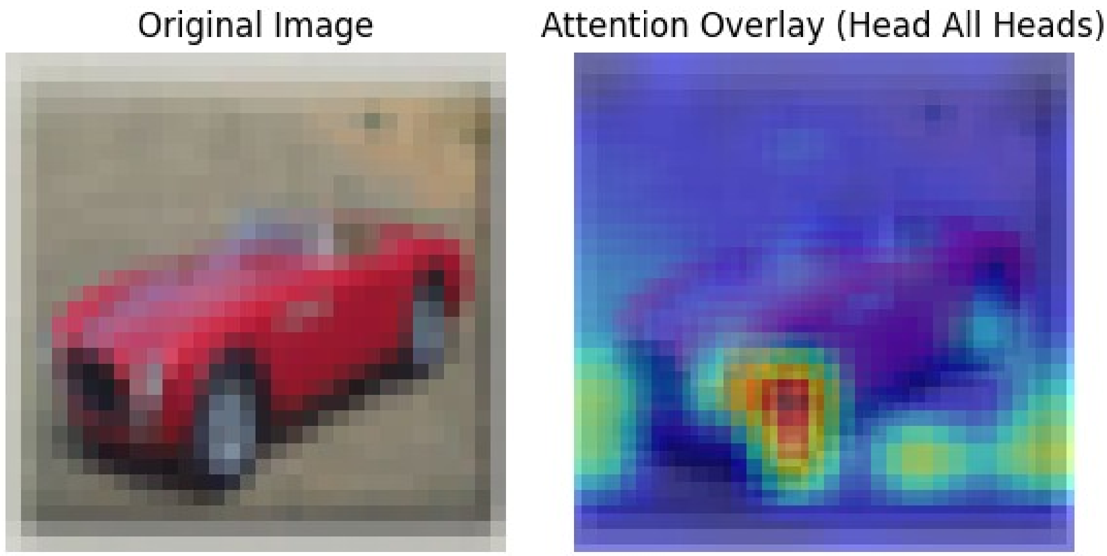

```
Time taken to train ViT: 1604.47 seconds
```

Visualise Attention Overlay

```python
# Get one image from the training set
img = test_dataset[1234][0]

# Call the visualisation function to see attention overlay
vit_classifier.model.visualise_attention_overlay(img, head=0)
vit_classifier.model.visualise_attention_overlay(img, head=1)
vit_classifier.model.visualise_attention_overlay(img, head=2)
vit_classifier.model.visualise_attention_overlay(img, head=3)
vit_classifier.model.visualise_attention_overlay(img, head=None)  # 
Average attention across all heads
```


Original Image
Attention Overlay (Head 1)
Original Image
Attention Overlay (Head 2)


Original Image
Attention Overlay (Head 3)
Original Image
Attention Overlay (Head 4)



Analysis

1. Performance Comparison

Accuracy

- The CNN architecture achieved a higher overall accuracy of 87.14% compared to the ViT architecture's overall accuracy of 75.77%. Moreover, the CNN had achieved a better score in virtually every metric (precision, recall, F1, accuracy) for every single class. This can be confirmed from the CNN confusion matrix, which shows a higher number of True Positives for every single class as compared to the ViT confusion matrix.
- As a sidenote, the common misclassifications by both models can be clustered into 2 broad categories: animals and vehicles. Concretely, the models tend to make misclassifications between 2 different classes of animals or 2 different classes of vehicles, but rarely between an animal and a vehicle, e.g. mistaking a bird for a car, although the bird tends to be mistaken for a plane at times. This is to be expected, since these 2 'meta' groups of classes are visually and conceptually very distinct. It can also be noted that the main confusion is between the 'cat' and 'dog' classes, which is reasonable since they both look visually very similar and the resolution of the dataset makes it harder to discern their features.

Training Speed & Computational Efficiency

- The CNN takes much less time (~ 19 mins) to train compared to the (~ 27 mins) ViT. This is surprising at first glance, considering that the CNN has slightly more parameters (1.4 mil) compared to the ViT (1.2 mil). This faster training speed and efficiency can be attributed to the respective architectures of the CNN and ViT.
- The CNN's inherent spatial hierarchy reduces computational complexity through progressive downsampling. Furthermore, CNNs demonstrate superior computational

efficiency due to parameter sharing in convolutional layers, by using the same filter over the entire image.

- On the other hand, the ViT utilises multi-head attention whereby every patch pays attention to every other patch in the image. This becomes highly memory-demanding, especially with larger batch sizes, making inherently slower per epoch.

Generalisation

- Overall, based on the loss curves observed, the CNN demonstrates very good generalisation, while the ViT demonstrates rather poor generalisation. This will be further elaborated in the loss curve analysis below.
- As a side observation, the ViT performed even worse at roughly 60% test accuracy when an unaugmented train loader was used for our initial trial runs. Meanwhile, train accuracy remained high, showing a case of severe overfitting. Upon performing the various augmentations to the train data loader, the ViT's accuracy made a dramatic improvement to 75%. This strongly suggests that ViTs have generalisation potential by introducing random noise to the image, which forces the ViT to learn that even if the object is rotated or flipped, it still remains the same object.

2. Learning Behaviour Comparison

Loss & Accuracy Curves

- CNN
  - The training and validation loss curves both drop steadily and converge well, with the validation loss being just slightly higher than the training loss.
  - The training and validation accuracy curves follow a similar trend, rising steadily before converging above 85%, with the validation accuracy being just slightly lower than the training accuracy.
  - The relatively small gap between the training and validation curves for both loss and accuracy shows that there is minimal overfitting, and the model is able to generalise well to unseen data.
- ViT
  - The training and validation loss curves appear to drop more slowly than those of the CNN. This can be noted from how the loss curves only dip below crossentropy loss of 1.0 after epoch 15, as compared to CNN after epoch 5. The training loss curve continues to decrease gradually to 0.3386 (still slightly higher than CNN training loss curve reaching 0.3105), and appears to be decreasing still. Meanwhile, the validation loss began to converge much earlier around epoch 40 and remains significantly higher than training loss.
  - Similarly, the training and validation accuracy curves also rose more slowly, crossing 70% accuracy after 20+ epochs while it only took 5+ epochs for the CNN. Validation accuracy also begins to plateau around 75% after 40 epochs, while training accuracy continues to rise steadily close to 90%.
  - This wide gap between the training and validation curves for both loss and accuracy shows that there is clear overfitting, and there is much room to improve the ViT's generalisation.

Feature Maps vs Attention Maps

- CNN feature maps
  - We plotted a total of 16 feature map samples of a red car image from the test set. Each row of feature maps corresponds to the output of a ResBlock layer of the CNN, the top row being the earliest layer.
  - Starting from the top, the feature maps appear more detailed, retaining the 32x32 dimensions of the original image. This is because the output of the ResBlock occurs before the first MaxPooling.
  - Different patterns can be observed in each feature map. For example, some feature maps have learned to identify only diagonal streaks across the surface of the car, or only vertical edges of the car, or just the outline of the car. This reveals how each filter has learned to identify a particular characteristic in the image through the training of the neural network.
  - Additionally, some of the feature maps appear to have brighter pixel highlights in them. This is probably because the original image consisting of 3 RGB channels is taken as input into the first ResBlock. As such, not only can the filters capture spatial patterns, but they are sensitive to colour as well.
  - Going further down the rows, the feature maps appear more low-resolution and abstract. For example, the outputs in the last row are completely unrecognisable to the human eye. This is the result of repeated maxpoolings performed that downsample and only preserve the most important characteristics of each feature map.
  - Given the CNN architecture that starts with 32 feature maps after the first layer and 128 feature maps in the final layer, this means that the model has learned to capture 128 high-level features within the image, instead of learning the weights for every single pixel like in a fully-connected MLP.
  - In conclusion, the CNN learns certain visual patterns associated with different classes of images, and utilises feature maps to capture these patterns in order to identify the image class.
- ViT attention maps
  - We plotted a total of 5 attention overlay maps, the first 4 corresponding to each of the 4 attention heads, and the last being the aggregated attention maps across all 4 heads.
  - It can be observed from the attention overlays that the more intense red spots correspond to regions where the head focused its attention on.
  - It is noticed that the attention heads tend to pay attention to the body of the car, rather than the background. This shows that the model has understood the focus subject of the image should be the car, and ignores less relevant aspects of the image like the background that may not contribute as much to the model's classification.
  - It is also noted that each individual attention head is able to focus on and identify unique features of the car (e.g. the wheels) even when not explicitly taught what that part is. This shows how each attention head hold a different perspective of the image, which get aggregated to give the transformer a holistic contextual understanding of the image. This is much like how humans are able to identify objects based on discrete parts of the object, such as recognising a car by identifying its wheels, headlights and car doors.

    - Furthermore, the ability to pinpoint the location of various car parts within the picture goes to show the importance of the positional embeddings learned in the model. This means that the model has also learned the relative positions of different parts of the object within an image, which would aid in its recognition ability.

Differences in learning behaviour

- Each visualisation type (CNN feature maps vs ViT attention maps) reveals how the different models adopt different approaches to gain semantic understanding of images
- Feature maps show how CNNs deconstruct an image into its visual characteristics and patterns, and pays little regard to the position of the feature in the image. This is because the same filter is applied across the entire image, so it is not concerned about where the feature is, but only whether it exists.
- In contrast, attention maps show how ViTs specifically learn the positions and relationships between different image patches using attention mechanism and positional embeddings.
- The trends observed in the loss and accuracy curves point to the better generalisation of the CNN compared to the ViT on the CIFAR-10 dataset. However, ViT training curves have yet to reach a plateau, which shows that it still has room to learn. On the whole, ViTs might benefit greatly from having a much larger dataset to train on, as well as more training epochs.

3. Conclusion

Overall Advantages/Disadvantages

- CNN
    - Advantages
        - CNNs process images efficiently by reusing learned patterns across locations and simplifying pictures consecutively, making them quicker to train and cheaper to run, even achieving relatively high performance for a medium sized dataset with low resolution images.
    - Disadvantages
        - The progressive downsampling can lead to a loss of information in the finer details.
        - Since they do not possess an attention mechanism like a ViT, they may fail to understand the relationship between distant parts of the image, such as the inability to draw a connection between a horse's tail to its head if they are far apart.
- ViT
    - Advantages
        - ViTs analyse the connections among all the patches. This allows them to highlight the regions in the image that matter most to its classification decision, hence the name 'attention'.
        - With enough data and compute, ViTs can be trained to become a robust classification model as they are able to better understand the relationships of different parts of an image than a CNN could.

- Disadvantages
    - Training a ViT is highly memory-intensive, especially when scaling up the number of attention heads and dataset size.
    - Reliant on effective data augmentation to train well.

Closing Remarks

- In this report, we evaluated and compared the performance of a custom CNN and a Vision Transformer (ViT) on the CIFAR-10 dataset.
- The CNN demonstrated strong generalisation and faster convergence, benefiting from its efficient use of weight-sharing and pooling layers that allow it to recognise patterns through the image.
- In contrast, the ViT initially underperformed but showed significant improvement with data augmentation, highlighting its reliance on data augmentation and positional embeddings to grasp the spatial structure of the image.

Suggestions for Future Refinements

- Future work could focus on enhancing the ViT's performance through stronger augmentation techniques, such as shearing and colour-shifting. These augmented versions can also be added to the dataset along with the original copies, as an efficient way of increasing the dataset size.
- More advanced regularisation techniques can also be used to reduce overfitting.
- Additionally, more types of attention visualisations can be implemented, so as to offer deeper insights into how the ViT model learns and attends to important image features.