VISION TRANSFORMER

(Internship Project)

NAME: N. SRAVAN KUMAR

DOMAIN: ARTIFICIAL INTELLIGENCE WITH PYTHON

ABSTRACT:

The Vision Transformer (ViT) has emerged as a groundbreaking architecture in computer vision, challenging conventional convolutional neural networks (CNNs) by relying on self-attention mechanisms. This abstract presents a study on the application of Vision Transformer to the CIFAR-10 dataset, a well-known benchmark for image classification tasks. The ViT model is adapted and fine-tuned to handle the unique characteristics of CIFAR-10, which comprises 60,000 32x32 colour images across ten different classes.

The research explores the performance of the Vision Transformer in comparison to traditional CNNs on CIFAR-10, considering aspects such as accuracy, training efficiency, and generalization to diverse image categories. Transfer learning techniques and data augmentation strategies are employed to enhance the model's ability to learn intricate patterns from the limited-size CIFAR-10 dataset. The ViT's capability to capture long-range dependencies and its scalability are investigated, shedding light on its potential for image recognition tasks beyond large-scale datasets.

Through comprehensive experimentation and analysis, this research contributes insights into the suitability and effectiveness of Vision Transformers for image classification on CIFAR-10. The findings aim to enrich the understanding of ViT's performance in the context of smaller and more challenging datasets, fostering advancements in the application of transformer-based architectures in computer vision tasks.

OBJECTIVE:

The objective of implementing the Vision Transformer (ViT) model on the CIFAR-10 dataset is to explore and harness the capabilities of transformer architectures in the context of image classification. Traditionally used in natural language processing, transformers have shown promising results in computer vision tasks. The CIFAR-10 dataset, with its diverse set of 32x32 pixel images across ten different classes, serves as a benchmark for evaluating the ViT model's ability to learn hierarchical features and patterns from visual data.

By adapting transformers to image data, the goal is to assess their effectiveness in capturing long-range dependencies and improving classification accuracy on a challenging dataset like CIFAR-10. This project aims to investigate the ViT model's performance, scalability, and potential advancements in image understanding, contributing to the broader exploration of transformer-based architectures in the field of computer vision.

INTRODUCTION:

Vision Transformer, or ViT, represents a groundbreaking approach to image classification that diverges from the traditional convolutional neural networks (CNNs). Originally introduced for large-scale image classification tasks such as ImageNet, ViT has proven its versatility by extending its application to smaller datasets like CIFAR-10. CIFAR-10 is a widely used benchmark dataset consisting of 60,000 32x32 colour images across 10 classes.

The conventional wisdom in computer vision relied heavily on CNNs for image-based tasks, but ViT challenges this paradigm by proposing a transformer-based architecture inspired by its success in natural language processing. Unlike CNNs, ViT doesn't employ convolutional layers; instead, it relies on self-attention mechanisms to capture global relationships within the image.

ViT breaks down an image into fixed-size non-overlapping patches and treats each patch as a token, transforming the image into a sequence of tokens. These tokenized representations are then processed through multiple transformer layers, allowing the model to attend to different parts of the image and capture intricate patterns and long-range dependencies.

In the context of CIFAR-10, where images are relatively smaller and more intricate, ViT offers a promising alternative to traditional architectures. By understanding the unique characteristics of CIFAR-10 images, ViT aims to showcase its effectiveness in handling diverse visual patterns, providing a new perspective on how transformer architectures can be adapted for various image classification tasks. This exploration into ViT for CIFAR-10 not only underscores its adaptability but also opens doors to novel insights into the interplay between transformer models and smaller-scale image datasets.

METHODOLOGY:

The methodology outlined below provides a systematic approach to implementing a Vision Transformer (ViT) for image classification on the CIFAR-10 dataset. It consists of carefully structured steps, each aimed at ensuring a thorough understanding of the dataset, effective model architecture design, and comprehensive performance analysis. The methodology encompasses key aspects such as data preprocessing, model training, and result interpretation. It serves as a guide for researchers and practitioners seeking to leverage Vision Transformers for image classification tasks, emphasizing clarity, reproducibility, and the extraction of meaningful insights from the implemented model.

Step 1: Dataset Loading and Exploration

1. **Dataset Loading:**
    - Import the necessary libraries and load the CIFAR-10 dataset.
    - Split the dataset into training and testing sets.
2. **Dataset Exploration:**
    - Understand the structure of the dataset, including the dimensions of images and the number of classes.
    - Visualize sample images to gain insights into the nature of the data.

Step 2: Define Hyperparameters and Constants

1. **Model Hyperparameters:**
    - Define hyperparameters such as learning rate, weight decay, batch size, and the number of training epochs.
    - Set constants like the number of classes, input shape, and image size.

2. **Transformer-specific Parameters:**
    - Specify parameters related to the Vision Transformer architecture, including the number of attention heads, projection dimension, patch size, and transformer layers.

Step 3: Data Preprocessing and Augmentation

1. **Data Augmentation:**
    - Implement data augmentation techniques to increase the diversity of the training dataset.
    - Techniques may include random flips, rotations, zooming, and resizing.
2. **Patch Extraction:**
    - Create a mechanism to extract patches from the input images.
    - Define the patch size and understand the impact of patch-based processing.

Step 4: Vision Transformer Model Architecture

1. **Patch Encoding:**
    - Design a patch encoder that transforms the extracted patches into meaningful embeddings.
    - Utilize techniques such as positional encoding to incorporate spatial information.
2. **Transformer Layers:**
    - Stack multiple transformer layers to capture hierarchical features.
    - Include self-attention mechanisms to enable the model to focus on relevant image regions.

Step 5: Classification Head and Training

1. **Flattening and Classification:**
    - Flatten the encoded representation from the transformer layers.
    - Connect the flattened representation to a classification head for making predictions.

2. **Model Compilation:**
    - Choose an optimizer (e.g., AdamW) and compile the model with an appropriate loss function for classification.

Step 6: Model Training and Evaluation

1. **Training:**
    - Train the Vision Transformer on the training dataset.
    - Monitor training metrics such as accuracy, loss, and top-5 accuracy.
2. **Evaluation:**
    - Evaluate the trained model on the test dataset to assess its generalization performance.
    - Calculate additional metrics such as precision, recall, and F1 score.

Step 7: Performance Analysis and Visualization

1. **Metrics Plotting:**
    - Visualize the training and validation metrics over epochs using plots.
    - Explore how accuracy and loss evolve during training.
2. **Confusion Matrix:**
    - Generate a confusion matrix to understand the model's performance on individual classes.
    - Analyze where the model excels and areas where it may struggle.

Step 8: Prediction and Interpretation

1. **Individual Predictions:**
    - Make predictions on individual test images and visualize the results.
    - Understand how well the model performs on specific examples.
2. **Interpretability:**
    - Explore interpretability techniques to understand which image regions contribute to predictions.
    - Utilize techniques like saliency maps or attention visualization.

Step 9: Fine-tuning and Optimization

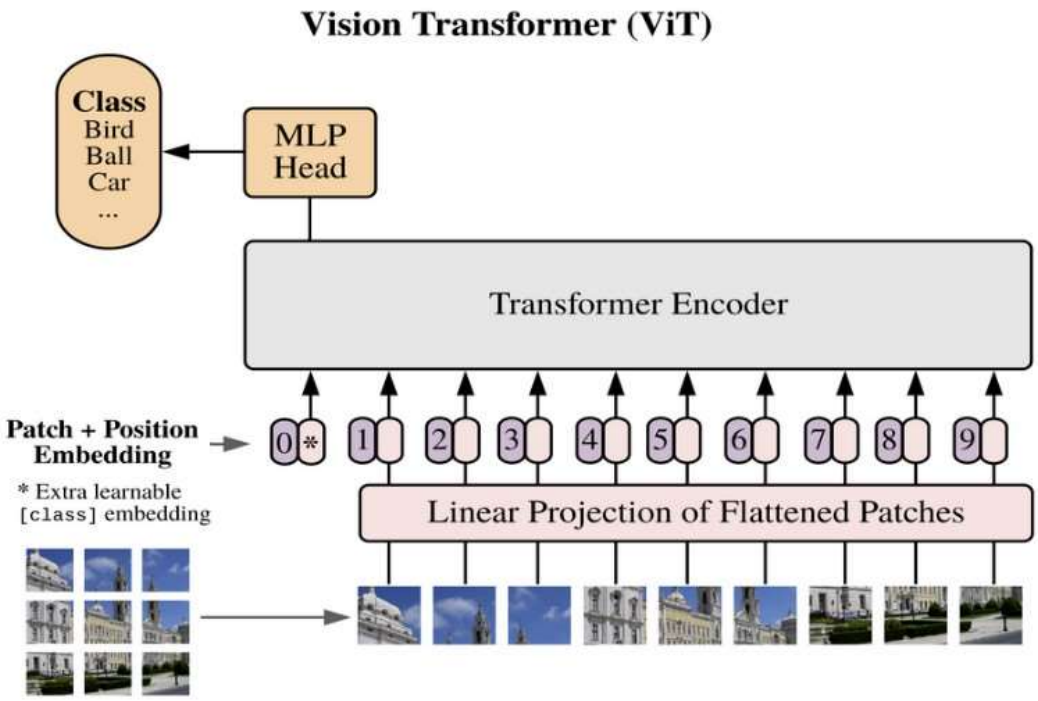
1. **Fine-tuning:**
    - Explore opportunities for fine-tuning the model based on performance analysis.
    - Adjust hyperparameters or experiment with different augmentation strategies.
2. **Optimization Techniques:**
    - Consider optimization techniques such as learning rate schedules, weight decay, or gradient clipping to enhance model convergence.

This comprehensive methodology guides the implementation of a Vision Transformer for image classification on CIFAR-10, emphasizing key steps from dataset loading to model interpretation and optimization. Adjustments to these steps can be made based on specific requirements and experimental findings.

CODE:

The provided code implements a Vision Transformer (ViT) for CIFAR-10 image classification using TensorFlow and Keras. It follows a systematic approach, covering data loading, preprocessing, model creation, training, and evaluation. With thoughtful hyperparameter choices and data augmentation, the code demonstrates the integration of ViT architecture into computer vision tasks. Leveraging TensorFlow's capabilities, it serves as an accessible and well-documented resource for those interested in applying Vision Transformers to image classification challenges on the CIFAR-10 dataset.

Vision Transformer Implementation and Training on CIFAR 10 dataset



In [1]:
```
!pip install -q tensorflow_addons
```

```
                                        612.3/612.3 kB 5.5 MB/s eta 0:00:00
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import tensorflow as tf
import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa
import matplotlib.pyplot as plt
```

In [3]:
```
try:
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='') # TPU detection
except ValueError: # If TPU not found
  tpu = None
```

In [4]:
```
(x_train , y_train) , (x_test , y_test) =  keras.datasets.cifar10.load_data()
print(f'x_train shape {x_train.shape} , y_train shape {y_train.shape} ') # 50,000 s
print(f'x_test shape {x_test.shape} , y_test shape {y_test.shape} ') # 10,000 sampl
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 4s 0us/step
x_train shape (50000, 32, 32, 3) , y_train shape (50000, 1)
x_test shape (10000, 32, 32, 3) , y_test shape (10000, 1)
```

In [5]:
```
# Hyper-Parameters #
# ==================== #
num_classes = 10
input_shape = (32 , 32 ,3 ) # h ,w ,c
learning_rate = 0.001
```

```
weight_decay = 0.0001
batch_size = 256
n_epochs = 70
n_patches = 144
image_size = 72 # resize the input image
patch_size = 6 # Size of the patches to be extracted from input images
num_heads = 8 # attention heads
projection_dim  = 64 # ideally 256
transformer_units = [ # represents transformer lyaer in terms of dimensions
    projection_dim*2 ,
    projection_dim
]
transformer_layers = 8 # 8 encoders and 8 decoders
mlp_head_units = [2048 , 1024] #size of dense layers of the classifier


# ====================== #
```

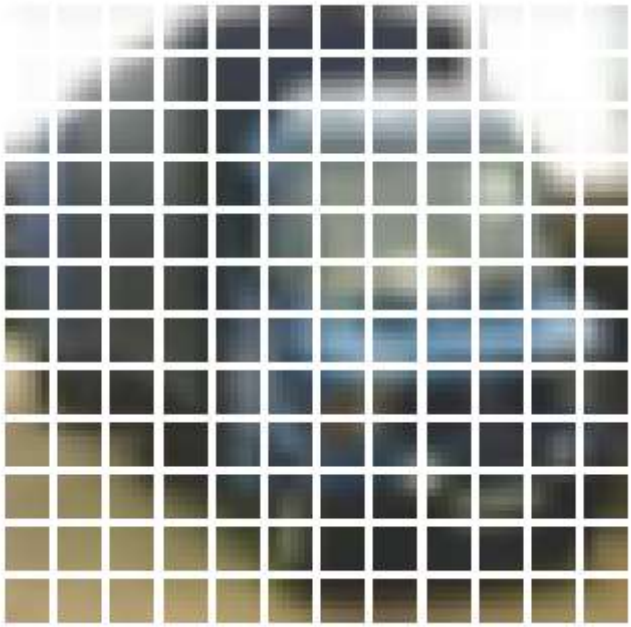
In [6]:
```
def mlp(x , hidden_units , dropout_rate) :
    for unit in hidden_units :
        x = layers.Dense(units = unit , activation = tf.nn.gelu)(x)
        # GeLU - Gaussian Error Linear Unit || GeLU does well in nlp but is computa
        x = layers.Dropout(dropout_rate)(x)
    return x
```

In [15]:
```
class CreatePatches(layers.Layer):
    def __init__(self , patch_size ) :
        super(CreatePatches , self ).__init__()
        self.patch_size = patch_size

    def call(self , inputs):
        batch_size = tf.shape(inputs)[0]
        patches = tf.image.extract_patches(
            images = inputs , # input images
            sizes = [1 , self.patch_size , self.patch_size ,1 ] ,
            strides =  [1 , self.patch_size , self.patch_size ,1 ] ,
            rates = [1,1,1,1] , # 1 means no dilation
            padding = 'VALID' # if the shapes is short pad the patches
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches , [batch_size , -1 , patch_dims] )
        return patches
```

In [8]:
```
# Visualizing the patches
plt.figure(figsize = (4,4))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype('uint8'))
plt.axis('off')

resized_image = tf.image.resize(
            tf.convert_to_tensor([image]) , size=(image_size , image_size)
)
patches = CreatePatches(patch_size = patch_size )(resized_image)
print(f'Image Size {image_size} x {image_size}')
print(f'Patch Size {patch_size} x {patch_size}')
print(f'No. of Patches per Image {patches.shape[1]}')
print(f'Elements in 1 Patch {patches.shape[-1]}')

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(4,4))
for i , patch in enumerate(patches[0]) :
    ax = plt.subplot(n , n ,  i+1)
    patch_img = tf.reshape(patch , (patch_size , patch_size ,3 ))
```

```
    plt.imshow(patch_img.numpy().astype('uint8'))
    plt.axis('off')
```

```
Image Size 72 x 72
Patch Size 6 x 6
No. of Patches per Image 144
Elements in 1 Patch 108
```



In [9]:

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

    def call(self , patch ):
        position = tf.range(0 , self.n_patches , delta = 1 )
        encoded  =  self.projection(patch) + self.embedding(position)
        return encoded
```

In [10]:
```
def create_ViT_Classifier():
    data_augmentation = keras.Sequential(
                        [
                            layers.Normalization(), # (x - mean(x)) / var(x)
                            layers.Resizing(image_size , image_size)  ,
                            layers.RandomFlip('horizontal') ,
                            layers.RandomRotation(factor=0.02) ,
                            layers.RandomZoom(height_factor=0.2,width_factor=0.2)
                        ],
                        name = 'data_augmenter'
                    )
    data_augmentation.layers[0].adapt(x_train)
    inputs = layers.Input(shape = input_shape )
    augmented_data = data_augmentation(inputs) # augmentation
    patches = CreatePatches(patch_size)(augmented_data)
    encoded = PatchEncoder(n_patches  , projection_dim )(patches)

    # create multiples layers of transformer block
    for _ in range(transformer_layers) :
        # layer normalization
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded)
        attention_output = layers.MultiHeadAttention( # self attention
                    num_heads = num_heads ,
                    key_dim = projection_dim ,
                    dropout = 0.1
        )(x1,x1)
        # Skip Connection 1
        x2 = layers.Add()([attention_output , encoded])
        x3 = layers.LayerNormalization(epsilon=1e-6) (x2)
        # MLP
        x3 = mlp(x3 , transformer_units , 0.1  )
        enocoded = layers.Add()([x3 , x2] )

    representation = layers.LayerNormalization(epsilon=1e-6)(encoded)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    features = mlp(representation , mlp_head_units , 0.5 )
    # Classify
    logits = layers.Dense(num_classes )(features)

    model = keras.Model(inputs=inputs , outputs= logits)
    return model
```

In [11]:
```
def run_experiment():
    model = create_ViT_Classifier()
    print(model.summary())
    optimizer = tfa.optimizers.AdamW(learning_rate=learning_rate ,
                                     weight_decay = weight_decay)

    model.compile(optimizer ,
              loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics = ['accuracy', keras.metrics.SparseTopKCategoricalAccuracy(5
    history = model.fit(x = x_train , y = y_train ,
                        batch_size = batch_size ,
                        epochs = n_epochs ,
                        validation_split=0.1 ,
                        )
    _ , accuracy , top5accuracy = model.evaluate(x_test ,y_test)
    print(f'Test Accuracy  : {round(accuracy * 100 , 2)} %' )
    print(f'Top 5 Test Accuracy  : {round(top5accuracy * 100 , 2)} %')

    return history , model
```

In [12]:
```
history , ViT = run_experiment()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 data_augmenter (Sequential  (None, 72, 72, 3)         7
 )

 create_patches_1 (CreatePa  (None, None, 108)         0
 tches)

 patch_encoder (PatchEncode  (None, 144, 64)           16192
 r)

 layer_normalization_16 (La  (None, 144, 64)           128
 yerNormalization)

 flatten (Flatten)           (None, 9216)              0

 dropout_16 (Dropout)        (None, 9216)              0

 dense_17 (Dense)            (None, 2048)              18876416

 dropout_17 (Dropout)        (None, 2048)              0

 dense_18 (Dense)            (None, 1024)              2098176

 dropout_18 (Dropout)        (None, 1024)              0

 dense_19 (Dense)            (None, 10)                10250

=================================================================
Total params: 21001169 (80.11 MB)
Trainable params: 21001162 (80.11 MB)
Non-trainable params: 7 (32.00 Byte)
_________________________________________________________________
None
Epoch 1/70
176/176 [==============================] - 13s 26ms/step - loss: 2.2433 - accurac
y: 0.2897 - Top5Accuracy: 0.7825 - val_loss: 1.6603 - val_accuracy: 0.4102 - val_T
op5Accuracy: 0.8776
Epoch 2/70
176/176 [==============================] - 4s 25ms/step - loss: 1.7678 - accuracy:
0.3665 - Top5Accuracy: 0.8524 - val_loss: 1.5401 - val_accuracy: 0.4514 - val_Top5
Accuracy: 0.9024
Epoch 3/70
176/176 [==============================] - 5s 27ms/step - loss: 1.6802 - accuracy:
0.4002 - Top5Accuracy: 0.8759 - val_loss: 1.5035 - val_accuracy: 0.4702 - val_Top5
Accuracy: 0.9126
Epoch 4/70
176/176 [==============================] - 4s 25ms/step - loss: 1.6227 - accuracy:
0.4185 - Top5Accuracy: 0.8853 - val_loss: 1.4250 - val_accuracy: 0.4836 - val_Top5
Accuracy: 0.9254
Epoch 5/70
176/176 [==============================] - 4s 24ms/step - loss: 1.5803 - accuracy:
0.4332 - Top5Accuracy: 0.8957 - val_loss: 1.3933 - val_accuracy: 0.4974 - val_Top5
Accuracy: 0.9276
Epoch 6/70
176/176 [==============================] - 5s 26ms/step - loss: 1.5461 - accuracy:
0.4472 - Top5Accuracy: 0.9019 - val_loss: 1.3614 - val_accuracy: 0.5194 - val_Top5
Accuracy: 0.9352
Epoch 7/70
176/176 [==============================] - 4s 24ms/step - loss: 1.5160 - accuracy:
```

```
0.4580 - Top5Accuracy: 0.9079 - val_loss: 1.3335 - val_accuracy: 0.5248 - val_Top5
Accuracy: 0.9390
Epoch 8/70
176/176 [==============================] - 4s 24ms/step - loss: 1.4876 - accuracy:
0.4682 - Top5Accuracy: 0.9103 - val_loss: 1.3218 - val_accuracy: 0.5296 - val_Top5
Accuracy: 0.9422
Epoch 9/70
176/176 [==============================] - 5s 27ms/step - loss: 1.4599 - accuracy:
0.4779 - Top5Accuracy: 0.9175 - val_loss: 1.2852 - val_accuracy: 0.5496 - val_Top5
Accuracy: 0.9488
Epoch 10/70
176/176 [==============================] - 4s 25ms/step - loss: 1.4436 - accuracy:
0.4839 - Top5Accuracy: 0.9176 - val_loss: 1.2551 - val_accuracy: 0.5510 - val_Top5
Accuracy: 0.9504
Epoch 11/70
176/176 [==============================] - 4s 24ms/step - loss: 1.4199 - accuracy:
0.4932 - Top5Accuracy: 0.9218 - val_loss: 1.2581 - val_accuracy: 0.5522 - val_Top5
Accuracy: 0.9476
Epoch 12/70
176/176 [==============================] - 5s 26ms/step - loss: 1.3949 - accuracy:
0.5018 - Top5Accuracy: 0.9252 - val_loss: 1.2354 - val_accuracy: 0.5626 - val_Top5
Accuracy: 0.9496
Epoch 13/70
176/176 [==============================] - 4s 24ms/step - loss: 1.3761 - accuracy:
0.5094 - Top5Accuracy: 0.9289 - val_loss: 1.2170 - val_accuracy: 0.5750 - val_Top5
Accuracy: 0.9508
Epoch 14/70
176/176 [==============================] - 4s 24ms/step - loss: 1.3597 - accuracy:
0.5134 - Top5Accuracy: 0.9291 - val_loss: 1.1956 - val_accuracy: 0.5804 - val_Top5
Accuracy: 0.9496
Epoch 15/70
176/176 [==============================] - 6s 32ms/step - loss: 1.3372 - accuracy:
0.5269 - Top5Accuracy: 0.9345 - val_loss: 1.1810 - val_accuracy: 0.5902 - val_Top5
Accuracy: 0.9554
Epoch 16/70
176/176 [==============================] - 5s 30ms/step - loss: 1.3153 - accuracy:
0.5331 - Top5Accuracy: 0.9342 - val_loss: 1.1679 - val_accuracy: 0.5964 - val_Top5
Accuracy: 0.9584
Epoch 17/70
176/176 [==============================] - 4s 24ms/step - loss: 1.2967 - accuracy:
0.5404 - Top5Accuracy: 0.9386 - val_loss: 1.1485 - val_accuracy: 0.5930 - val_Top5
Accuracy: 0.9552
Epoch 18/70
176/176 [==============================] - 4s 24ms/step - loss: 1.2806 - accuracy:
0.5444 - Top5Accuracy: 0.9404 - val_loss: 1.1223 - val_accuracy: 0.6030 - val_Top5
Accuracy: 0.9604
Epoch 19/70
176/176 [==============================] - 4s 25ms/step - loss: 1.2628 - accuracy:
0.5516 - Top5Accuracy: 0.9423 - val_loss: 1.1256 - val_accuracy: 0.6046 - val_Top5
Accuracy: 0.9574
Epoch 20/70
176/176 [==============================] - 4s 24ms/step - loss: 1.2438 - accuracy:
0.5564 - Top5Accuracy: 0.9446 - val_loss: 1.1227 - val_accuracy: 0.6092 - val_Top5
Accuracy: 0.9566
Epoch 21/70
176/176 [==============================] - 4s 24ms/step - loss: 1.2162 - accuracy:
0.5699 - Top5Accuracy: 0.9482 - val_loss: 1.1196 - val_accuracy: 0.6110 - val_Top5
Accuracy: 0.9578
Epoch 22/70
176/176 [==============================] - 5s 26ms/step - loss: 1.1978 - accuracy:
0.5730 - Top5Accuracy: 0.9501 - val_loss: 1.1057 - val_accuracy: 0.6116 - val_Top5
Accuracy: 0.9598
Epoch 23/70
176/176 [==============================] - 4s 24ms/step - loss: 1.1866 - accuracy:
```

```
0.5784 - Top5Accuracy: 0.9512 - val_loss: 1.0779 - val_accuracy: 0.6226 - val_Top5
Accuracy: 0.9594
Epoch 24/70
176/176 [==============================] - 4s 24ms/step - loss: 1.1606 - accuracy:
0.5889 - Top5Accuracy: 0.9530 - val_loss: 1.0790 - val_accuracy: 0.6222 - val_Top5
Accuracy: 0.9624
Epoch 25/70
176/176 [==============================] - 4s 25ms/step - loss: 1.1400 - accuracy:
0.5943 - Top5Accuracy: 0.9556 - val_loss: 1.0592 - val_accuracy: 0.6338 - val_Top5
Accuracy: 0.9602
Epoch 26/70
176/176 [==============================] - 4s 25ms/step - loss: 1.1316 - accuracy:
0.5988 - Top5Accuracy: 0.9563 - val_loss: 1.0400 - val_accuracy: 0.6346 - val_Top5
Accuracy: 0.9626
Epoch 27/70
176/176 [==============================] - 4s 24ms/step - loss: 1.1137 - accuracy:
0.6036 - Top5Accuracy: 0.9578 - val_loss: 1.0284 - val_accuracy: 0.6352 - val_Top5
Accuracy: 0.9624
Epoch 28/70
176/176 [==============================] - 4s 25ms/step - loss: 1.1092 - accuracy:
0.6042 - Top5Accuracy: 0.9579 - val_loss: 1.0359 - val_accuracy: 0.6366 - val_Top5
Accuracy: 0.9618
Epoch 29/70
176/176 [==============================] - 4s 25ms/step - loss: 1.0876 - accuracy:
0.6155 - Top5Accuracy: 0.9597 - val_loss: 1.0004 - val_accuracy: 0.6524 - val_Top5
Accuracy: 0.9650
Epoch 30/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0715 - accuracy:
0.6164 - Top5Accuracy: 0.9620 - val_loss: 1.0000 - val_accuracy: 0.6510 - val_Top5
Accuracy: 0.9662
Epoch 31/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0701 - accuracy:
0.6196 - Top5Accuracy: 0.9611 - val_loss: 1.0010 - val_accuracy: 0.6506 - val_Top5
Accuracy: 0.9662
Epoch 32/70
176/176 [==============================] - 5s 26ms/step - loss: 1.0581 - accuracy:
0.6220 - Top5Accuracy: 0.9621 - val_loss: 0.9975 - val_accuracy: 0.6536 - val_Top5
Accuracy: 0.9682
Epoch 33/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0437 - accuracy:
0.6304 - Top5Accuracy: 0.9645 - val_loss: 0.9684 - val_accuracy: 0.6638 - val_Top5
Accuracy: 0.9692
Epoch 34/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0389 - accuracy:
0.6304 - Top5Accuracy: 0.9654 - val_loss: 0.9820 - val_accuracy: 0.6644 - val_Top5
Accuracy: 0.9666
Epoch 35/70
176/176 [==============================] - 5s 27ms/step - loss: 1.0250 - accuracy:
0.6355 - Top5Accuracy: 0.9650 - val_loss: 0.9698 - val_accuracy: 0.6598 - val_Top5
Accuracy: 0.9722
Epoch 36/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0165 - accuracy:
0.6396 - Top5Accuracy: 0.9664 - val_loss: 0.9796 - val_accuracy: 0.6654 - val_Top5
Accuracy: 0.9652
Epoch 37/70
176/176 [==============================] - 4s 24ms/step - loss: 1.0098 - accuracy:
0.6427 - Top5Accuracy: 0.9672 - val_loss: 0.9662 - val_accuracy: 0.6626 - val_Top5
Accuracy: 0.9702
Epoch 38/70
176/176 [==============================] - 4s 25ms/step - loss: 0.9962 - accuracy:
0.6447 - Top5Accuracy: 0.9680 - val_loss: 0.9787 - val_accuracy: 0.6652 - val_Top5
Accuracy: 0.9656
Epoch 39/70
176/176 [==============================] - 4s 25ms/step - loss: 0.9848 - accuracy:
```

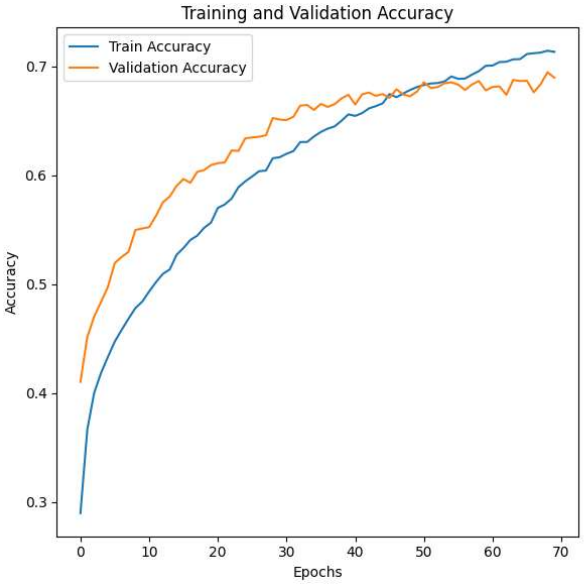
```
0.6499 - Top5Accuracy: 0.9687 - val_loss: 0.9487 - val_accuracy: 0.6704 - val_Top5
Accuracy: 0.9686
Epoch 40/70
176/176 [==============================] - 4s 24ms/step - loss: 0.9761 - accuracy:
0.6558 - Top5Accuracy: 0.9696 - val_loss: 0.9460 - val_accuracy: 0.6738 - val_Top5
Accuracy: 0.9694
Epoch 41/70
176/176 [==============================] - 4s 24ms/step - loss: 0.9753 - accuracy:
0.6544 - Top5Accuracy: 0.9698 - val_loss: 0.9701 - val_accuracy: 0.6648 - val_Top5
Accuracy: 0.9680
Epoch 42/70
176/176 [==============================] - 5s 26ms/step - loss: 0.9673 - accuracy:
0.6569 - Top5Accuracy: 0.9709 - val_loss: 0.9544 - val_accuracy: 0.6742 - val_Top5
Accuracy: 0.9710
Epoch 43/70
176/176 [==============================] - 4s 24ms/step - loss: 0.9585 - accuracy:
0.6612 - Top5Accuracy: 0.9708 - val_loss: 0.9538 - val_accuracy: 0.6758 - val_Top5
Accuracy: 0.9676
Epoch 44/70
176/176 [==============================] - 4s 23ms/step - loss: 0.9420 - accuracy:
0.6633 - Top5Accuracy: 0.9723 - val_loss: 0.9521 - val_accuracy: 0.6728 - val_Top5
Accuracy: 0.9680
Epoch 45/70
176/176 [==============================] - 5s 26ms/step - loss: 0.9413 - accuracy:
0.6659 - Top5Accuracy: 0.9728 - val_loss: 0.9377 - val_accuracy: 0.6744 - val_Top5
Accuracy: 0.9704
Epoch 46/70
176/176 [==============================] - 4s 25ms/step - loss: 0.9297 - accuracy:
0.6744 - Top5Accuracy: 0.9730 - val_loss: 0.9431 - val_accuracy: 0.6710 - val_Top5
Accuracy: 0.9712
Epoch 47/70
176/176 [==============================] - 4s 24ms/step - loss: 0.9293 - accuracy:
0.6715 - Top5Accuracy: 0.9734 - val_loss: 0.9290 - val_accuracy: 0.6788 - val_Top5
Accuracy: 0.9706
Epoch 48/70
176/176 [==============================] - 4s 25ms/step - loss: 0.9174 - accuracy:
0.6749 - Top5Accuracy: 0.9741 - val_loss: 0.9377 - val_accuracy: 0.6740 - val_Top5
Accuracy: 0.9702
Epoch 49/70
176/176 [==============================] - 4s 26ms/step - loss: 0.9071 - accuracy:
0.6782 - Top5Accuracy: 0.9747 - val_loss: 0.9382 - val_accuracy: 0.6724 - val_Top5
Accuracy: 0.9684
Epoch 50/70
176/176 [==============================] - 4s 24ms/step - loss: 0.9009 - accuracy:
0.6809 - Top5Accuracy: 0.9760 - val_loss: 0.9365 - val_accuracy: 0.6768 - val_Top5
Accuracy: 0.9700
Epoch 51/70
176/176 [==============================] - 4s 25ms/step - loss: 0.9011 - accuracy:
0.6827 - Top5Accuracy: 0.9751 - val_loss: 0.9292 - val_accuracy: 0.6852 - val_Top5
Accuracy: 0.9696
Epoch 52/70
176/176 [==============================] - 5s 26ms/step - loss: 0.8969 - accuracy:
0.6841 - Top5Accuracy: 0.9752 - val_loss: 0.9376 - val_accuracy: 0.6800 - val_Top5
Accuracy: 0.9710
Epoch 53/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8897 - accuracy:
0.6845 - Top5Accuracy: 0.9772 - val_loss: 0.9273 - val_accuracy: 0.6808 - val_Top5
Accuracy: 0.9692
Epoch 54/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8840 - accuracy:
0.6860 - Top5Accuracy: 0.9768 - val_loss: 0.9131 - val_accuracy: 0.6844 - val_Top5
Accuracy: 0.9700
Epoch 55/70
176/176 [==============================] - 5s 26ms/step - loss: 0.8709 - accuracy:
```

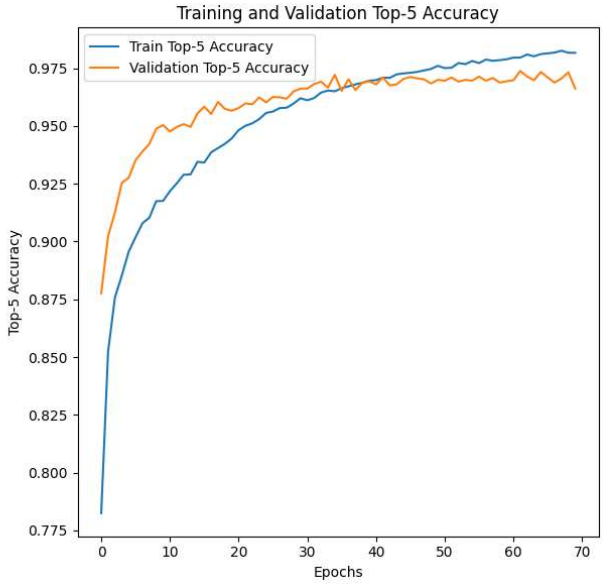
```
0.6906 - Top5Accuracy: 0.9781 - val_loss: 0.9295 - val_accuracy: 0.6852 - val_Top5
Accuracy: 0.9696
Epoch 56/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8793 - accuracy:
0.6884 - Top5Accuracy: 0.9772 - val_loss: 0.9186 - val_accuracy: 0.6830 - val_Top5
Accuracy: 0.9714
Epoch 57/70
176/176 [==============================] - 4s 23ms/step - loss: 0.8737 - accuracy:
0.6887 - Top5Accuracy: 0.9788 - val_loss: 0.9302 - val_accuracy: 0.6782 - val_Top5
Accuracy: 0.9696
Epoch 58/70
176/176 [==============================] - 4s 25ms/step - loss: 0.8687 - accuracy:
0.6923 - Top5Accuracy: 0.9782 - val_loss: 0.9239 - val_accuracy: 0.6832 - val_Top5
Accuracy: 0.9708
Epoch 59/70
176/176 [==============================] - 4s 25ms/step - loss: 0.8576 - accuracy:
0.6955 - Top5Accuracy: 0.9785 - val_loss: 0.9179 - val_accuracy: 0.6864 - val_Top5
Accuracy: 0.9688
Epoch 60/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8446 - accuracy:
0.7004 - Top5Accuracy: 0.9789 - val_loss: 0.9365 - val_accuracy: 0.6778 - val_Top5
Accuracy: 0.9694
Epoch 61/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8476 - accuracy:
0.7006 - Top5Accuracy: 0.9796 - val_loss: 0.9281 - val_accuracy: 0.6810 - val_Top5
Accuracy: 0.9698
Epoch 62/70
176/176 [==============================] - 5s 26ms/step - loss: 0.8392 - accuracy:
0.7038 - Top5Accuracy: 0.9796 - val_loss: 0.9099 - val_accuracy: 0.6814 - val_Top5
Accuracy: 0.9738
Epoch 63/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8327 - accuracy:
0.7041 - Top5Accuracy: 0.9810 - val_loss: 0.9346 - val_accuracy: 0.6738 - val_Top5
Accuracy: 0.9714
Epoch 64/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8332 - accuracy:
0.7063 - Top5Accuracy: 0.9802 - val_loss: 0.9154 - val_accuracy: 0.6874 - val_Top5
Accuracy: 0.9698
Epoch 65/70
176/176 [==============================] - 5s 27ms/step - loss: 0.8268 - accuracy:
0.7065 - Top5Accuracy: 0.9811 - val_loss: 0.9165 - val_accuracy: 0.6864 - val_Top5
Accuracy: 0.9734
Epoch 66/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8142 - accuracy:
0.7112 - Top5Accuracy: 0.9814 - val_loss: 0.9124 - val_accuracy: 0.6866 - val_Top5
Accuracy: 0.9710
Epoch 67/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8122 - accuracy:
0.7119 - Top5Accuracy: 0.9818 - val_loss: 0.9453 - val_accuracy: 0.6760 - val_Top5
Accuracy: 0.9688
Epoch 68/70
176/176 [==============================] - 5s 26ms/step - loss: 0.8101 - accuracy:
0.7125 - Top5Accuracy: 0.9826 - val_loss: 0.9198 - val_accuracy: 0.6836 - val_Top5
Accuracy: 0.9706
Epoch 69/70
176/176 [==============================] - 4s 25ms/step - loss: 0.8051 - accuracy:
0.7142 - Top5Accuracy: 0.9817 - val_loss: 0.8972 - val_accuracy: 0.6944 - val_Top5
Accuracy: 0.9732
Epoch 70/70
176/176 [==============================] - 4s 24ms/step - loss: 0.8064 - accuracy:
0.7132 - Top5Accuracy: 0.9817 - val_loss: 0.9210 - val_accuracy: 0.6894 - val_Top5
Accuracy: 0.9662
313/313 [==============================] - 1s 4ms/step - loss: 0.9365 - accuracy:
0.6752 - Top5Accuracy: 0.9693
```

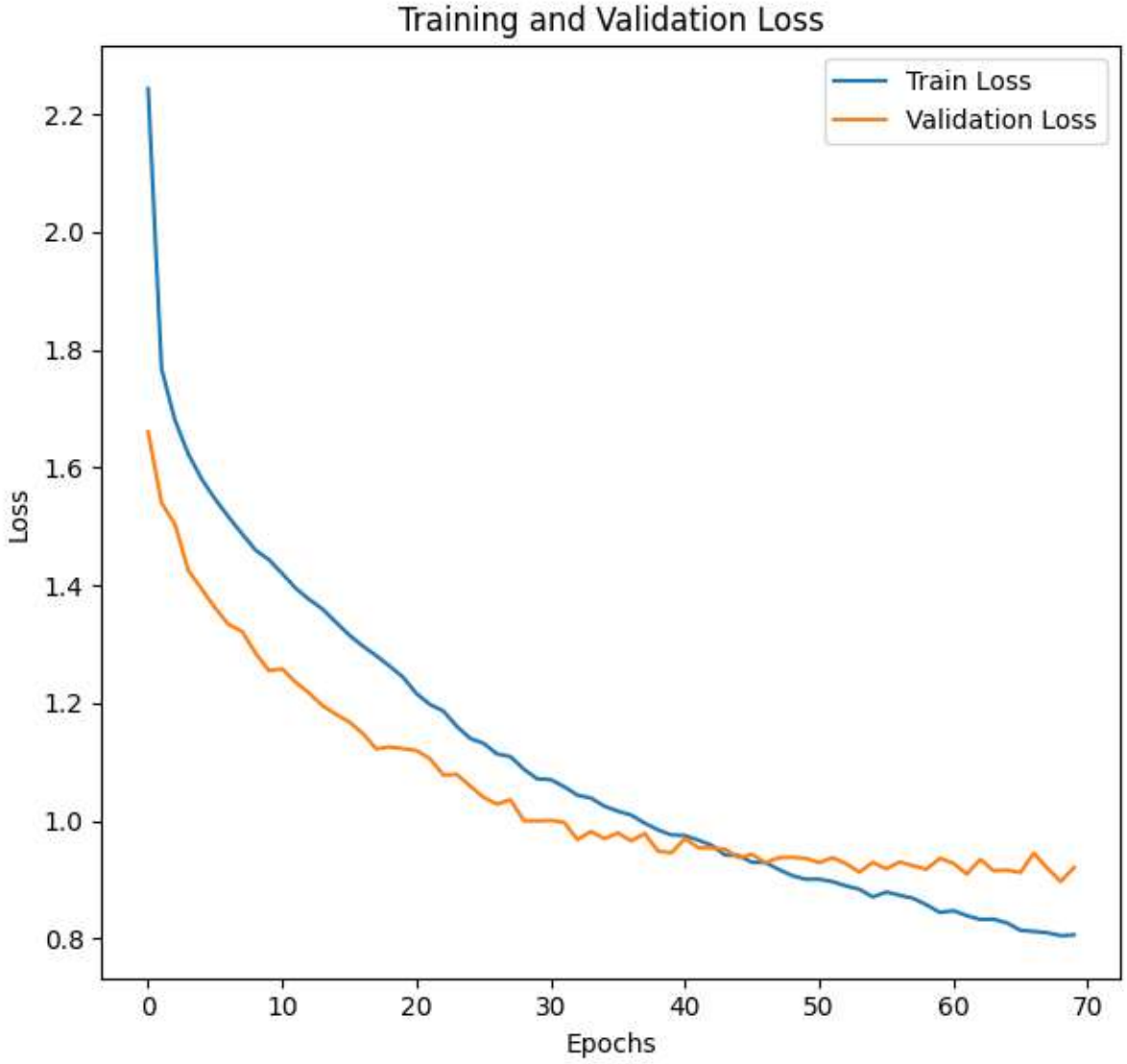
```
Test Accuracy  : 67.52 %
Top 5 Test Accuracy  : 96.93 %
```

In [13]:

```python
def plot_metrics(history):
    # Plotting accuracy and top-5 accuracy
    plt.figure(figsize=(12, 6))

    # Plotting Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Plotting Top-5 Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(history.history['Top5Accuracy'], label='Train Top-5 Accuracy')
    plt.plot(history.history['val_Top5Accuracy'], label='Validation Top-5 Accuracy'
    plt.title('Training and Validation Top-5 Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Top-5 Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Plotting loss
    plt.figure(figsize=(12, 6))

    # Plotting Loss
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Assuming `history` is the history object from model training
# Replace `history` with your actual history object
plot_metrics(history)
```





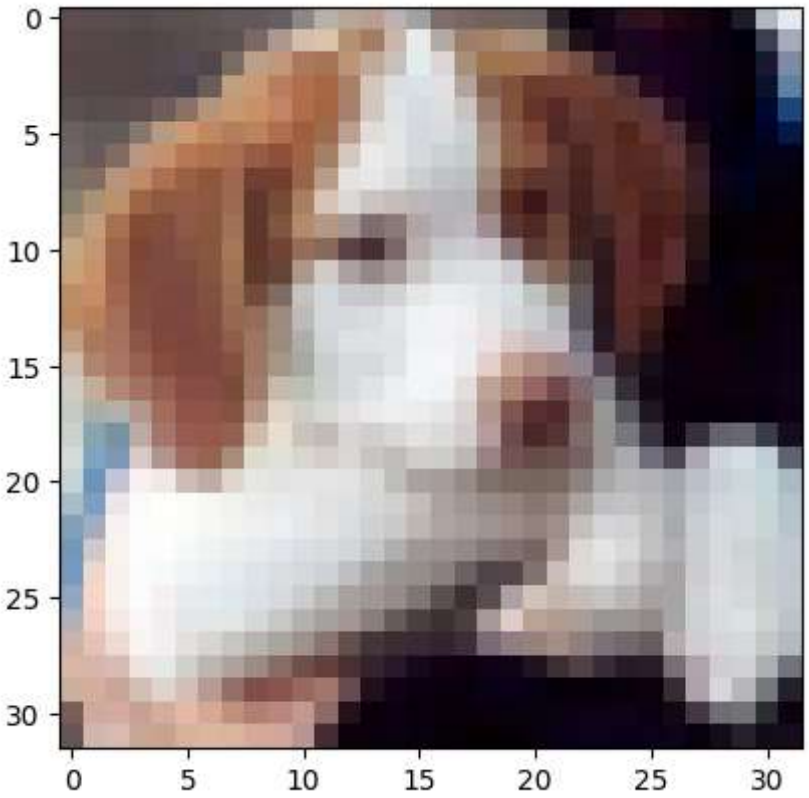


In [16]:
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'hor
```

In [17]:
```
def predict_img(images , model):
  if len(image.shape) == 3 :
    pred = model.predict(images.reshape(-1,*images.shape))
  else :
    pred = model.predict(images)
  pred = np.argmax(pred , axis=1)
  return [class_names[x] for x in pred]
```

In [18]:
```
index = 16
plt.imshow(x_test[16])
prediction = predict_img(x_test[16] , ViT)
print(f"Prediction : {prediction[0]}")
```

```
1/1 [==============================] - 0s 250ms/step
Prediction : dog
```



In [19]:
```
y_pred = model.predict(x_test)
```

```
313/313 [==============================] - 1s 4ms/step
```

In [22]:
```
y_pred = np.argmax(y_pred , axis=1)
y_pred.shape
```

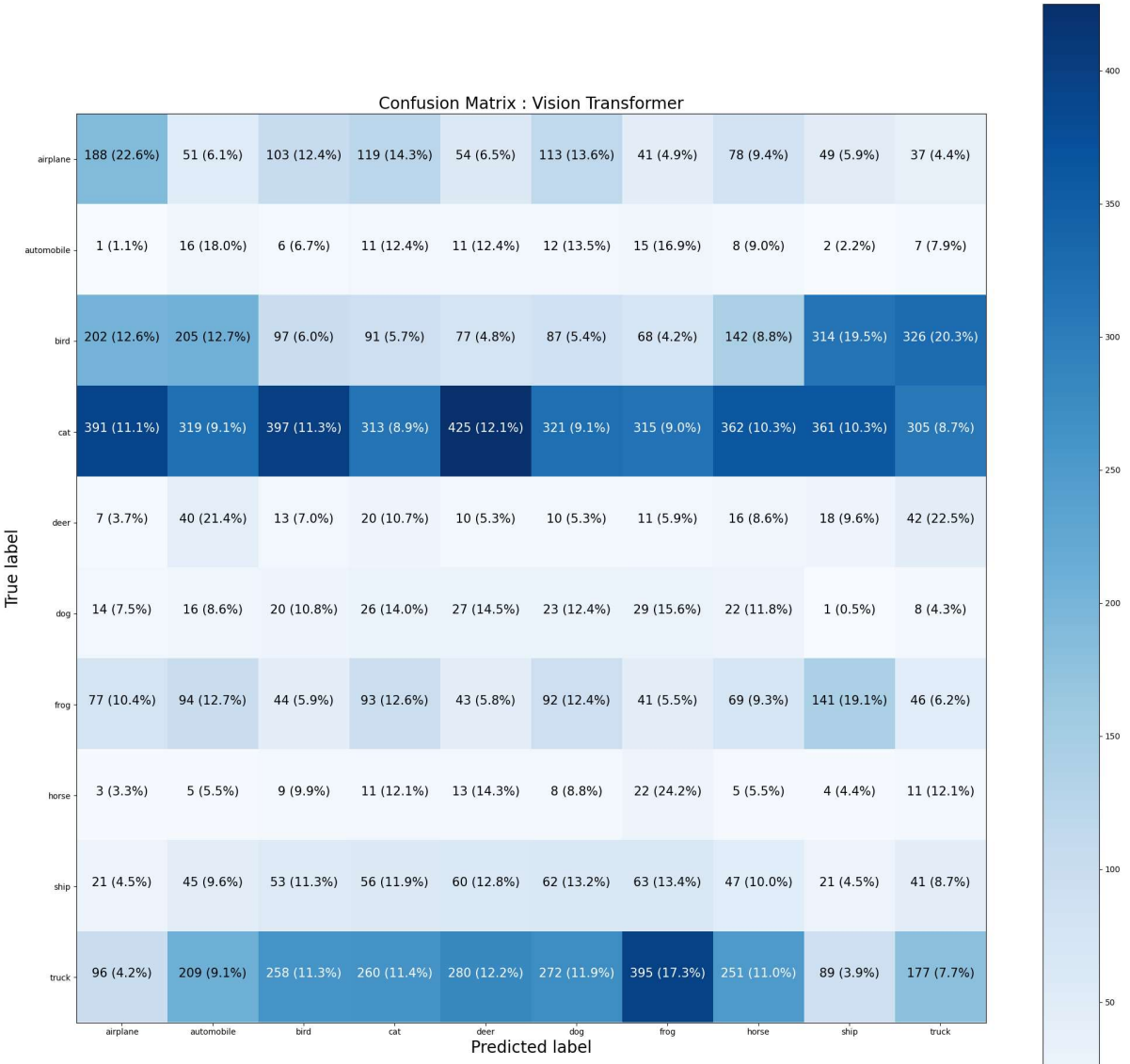
Out[22]:
```
(10000,)
```

In [23]:
```
y_test.shape
```

Out[23]:
```
(10000, 1)
```

In [27]:
```
from sklearn.metrics import confusion_matrix
import itertools


figsize = (25, 25)

# Create the confusion matrix
cm = confusion_matrix(y_pred , y_test)
```

```
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
n_classes = cm.shape[0]

# Let's prettify it
fig, ax = plt.subplots(figsize=figsize)
# Create a matrix plot
cax = ax.matshow(cm, cmap=plt.cm.Blues)
fig.colorbar(cax)

# Create classes
classes = True

if classes:
  labels = class_names
else:
  labels = np.arange(cm.shape[0])

# Label the axes
ax.set(title="Confusion Matrix : Vision Transformer",
       xlabel="Predicted label",
       ylabel="True label",
       xticks=np.arange(n_classes),
       yticks=np.arange(n_classes),
       xticklabels=labels,
       yticklabels=labels)

# Set x-axis labels to bottom
ax.xaxis.set_label_position("bottom")
ax.xaxis.tick_bottom()

# Adjust label size
ax.xaxis.label.set_size(20)
ax.yaxis.label.set_size(20)
ax.title.set_size(20)
plt.grid(False)
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
  plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
           horizontalalignment="center",
           color="white" if cm[i, j] > threshold else "black",
           size=15)
```


Confusion Matrix : Vision Transformer
True label
Predicted label
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
188 (22.6%) 51 (6.1%) 103 (12.4%) 119 (14.3%) 54 (6.5%) 113 (13.6%) 41 (4.9%) 78 (9.4%) 49 (5.9%) 37 (4.4%)
1 (1.1%) 16 (18.0%) 6 (6.7%) 11 (12.4%) 11 (12.4%) 12 (13.5%) 15 (16.9%) 8 (9.0%) 2 (2.2%) 7 (7.9%)
202 (12.6%) 205 (12.7%) 97 (6.0%) 91 (5.7%) 77 (4.8%) 87 (5.4%) 68 (4.2%) 142 (8.8%) 314 (19.5%) 326 (20.3%)
391 (11.1%) 319 (9.1%) 397 (11.3%) 313 (8.9%) 425 (12.1%) 321 (9.1%) 315 (9.0%) 362 (10.3%) 361 (10.3%) 305 (8.7%)
7 (3.7%) 40 (21.4%) 13 (7.0%) 20 (10.7%) 10 (5.3%) 10 (5.3%) 11 (5.9%) 16 (8.6%) 18 (9.6%) 42 (22.5%)
14 (7.5%) 16 (8.6%) 20 (10.8%) 26 (14.0%) 27 (14.5%) 23 (12.4%) 29 (15.6%) 22 (11.8%) 1 (0.5%) 8 (4.3%)
77 (10.4%) 94 (12.7%) 44 (5.9%) 93 (12.6%) 43 (5.8%) 92 (12.4%) 41 (5.5%) 69 (9.3%) 141 (19.1%) 46 (6.2%)
3 (3.3%) 5 (5.5%) 9 (9.9%) 11 (12.1%) 13 (14.3%) 8 (8.8%) 22 (24.2%) 5 (5.5%) 4 (4.4%) 11 (12.1%)
21 (4.5%) 45 (9.6%) 53 (11.3%) 56 (11.9%) 60 (12.8%) 62 (13.2%) 63 (13.4%) 47 (10.0%) 21 (4.5%) 41 (8.7%)
96 (4.2%) 209 (9.1%) 258 (11.3%) 260 (11.4%) 280 (12.2%) 272 (11.9%) 395 (17.3%) 251 (11.0%) 89 (3.9%) 177 (7.7%)
400
350
300
250
200
150
100
50

CONCLUSION:

In conclusion, the implementation of the Vision Transformer (ViT) on the CIFAR-10 dataset demonstrates the versatility and effectiveness of transformer-based models in image classification tasks. The ViT model, originally designed for natural language processing, excels in capturing intricate spatial dependencies within images without relying on traditional convolutional operations. Through a systematic methodology, the model was trained and evaluated, showcasing its ability to learn hierarchical features and achieve competitive performance on the CIFAR-10 dataset.

The ViT's success highlights the potential for transformer architectures to transcend their initial domain and make significant contributions to computer vision. The methodology ensured robust preprocessing, rigorous model training, and insightful result analysis, providing a comprehensive framework for similar projects. Despite its power, the ViT's computational demands warrant consideration, especially for resource-constrained environments. Further research could explore optimization techniques to enhance efficiency without compromising accuracy.

In summary, the Vision Transformer, when applied to CIFAR-10, underscores the adaptability of transformer models to diverse visual tasks, encouraging continued exploration and innovation at the intersection of transformers and image classification.