DL-Ops

Assignment 3 - Report

Debonil Ghosh
Roll No: M21AIE225
Executive MTech
Artificial Intelligence
Indian Institute of Technology, Jodhpur

Problem Statement:

Question 1 [50 marks]

Perform image classification on selected classes[check below in Note] of CIFAR-10 dataset using transformer model with following variations:

1. Use cosine positional embedding with six encoders and decoder layers with eightnheads. Use relu activation in the intermediate layers. Marks [20]
2. Use learnable positional encoding with four encoder and decoder layers with six heads. Use relu activation in the intermediate layers. Marks [20]
3. For parts (a) and (b) change the activation function in the intermediate layer from relu to tanh and compare the performance. Marks [10]

Note: Those who have
Even roll number : select odd classes
Odd roll number: select even classes
Reference Blog:
https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c
https://theaisummer.com/positional-embeddings/

Questions 2 [50 marks]

Based on the lecture by Dr. Anush on DLOPs, you have to perform the following experiments :

- Load and preprocessing CIFAR10 dataset using standard augmentation and normalization techniques [10 Marks]
- Train the following models for profiling them using during the training step [5 *2 = 10Marks]
  - Conv -> Conv -> Maxpool (2,2) -> Conv -> Maxpool(2,2) -> Conv -> Maxpool(2,2)
    - You can decide the parameters of convolution layers and activations on your own.
    - Make sure to keep 4 conv-layers and 3 max-pool layers in the order describes above.
  - VGG16
- After the profiling of your model, figure out the minimum change in the architecture that would lead to a gain in performance and decrease training time on CIFAR10 dataset as compared to one achieved before. [30 Marks]

You are free to use any tool/library that was discussed in the lecture to perform the above task. Describe your analysis and all your steps in detail, with relevant screenshots of the model's profiling in your report.

Question 1

Transformer

The Transformer is a deep learning architecture introduced in 2017, which has become a popular choice for various natural language processing tasks such as language translation, language modelling, and text classification.

Unlike traditional recurrent neural networks (RNNs) or convolutional neural networks (CNNs), the Transformer is based on a self-attention mechanism that allows it to capture long-term dependencies between words in a sequence without the need for recurrence or convolution.

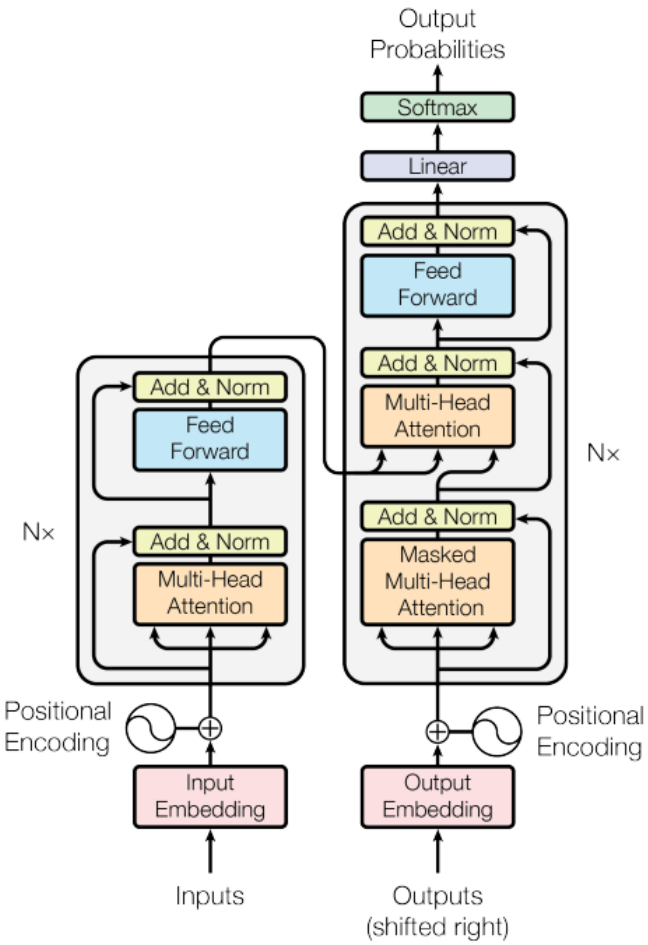


Figure 1: The Transformer - model architecture.

Source: Paper - Attention Is All You Need -2017

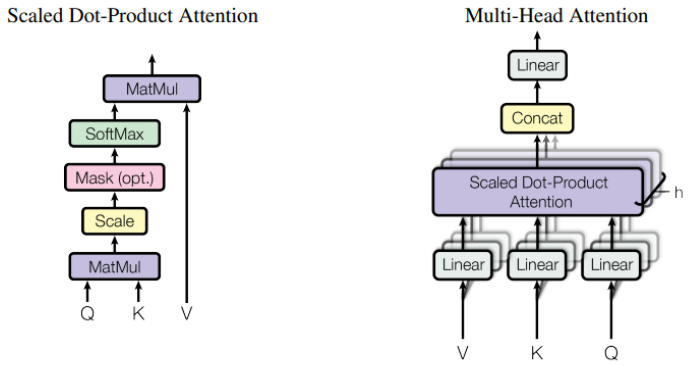
The Transformer architecture consists of an encoder and a decoder, each composed of multiple layers of self-attention and feedforward neural networks. The encoder takes the input sequence and generates a series of hidden representations that capture the meaning of each word in the context of the entire sequence. The decoder then uses these representations to generate the output sequence.

One of the key advantages of the Transformer is its ability to parallelize computation, which makes it much faster than traditional RNNs for long sequences. The Transformer has achieved state-of-the-art performance on various natural language processing benchmarks, and its architecture has also been adapted for other tasks such as image recognition and speech recognition.

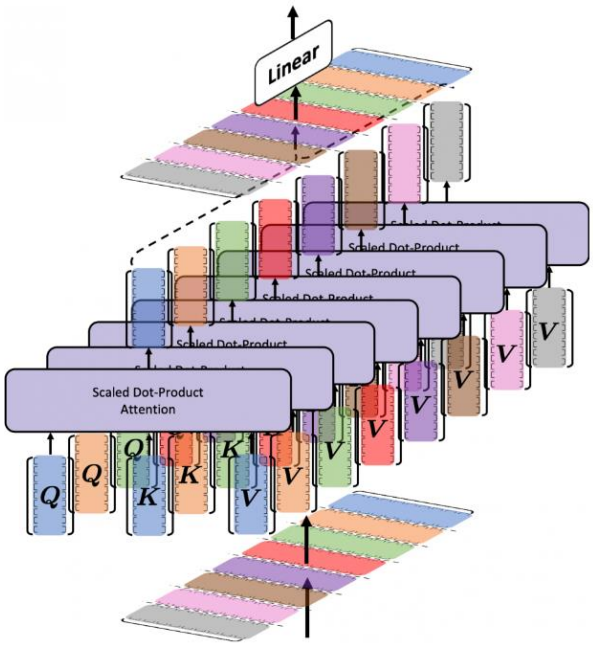
Multi Head Self Attention

Multi-Head Self-Attention is a key component of the Transformer model, which is a state-of-the-art neural network architecture widely used in natural language processing tasks.

Self-attention is a mechanism that allows the model to focus on different parts of the input sequence when processing each element in the sequence. In Multi-Head Self-Attention, this mechanism is applied multiple times in parallel, with each instance of attention called a "head".



In the Transformer model, the input sequence is first projected into three different representations: query, key, and value. Then, the self-attention mechanism is applied to these representations in parallel for each head, producing a set of output vectors. These output vectors are then concatenated and projected to obtain the final representation of the input sequence.

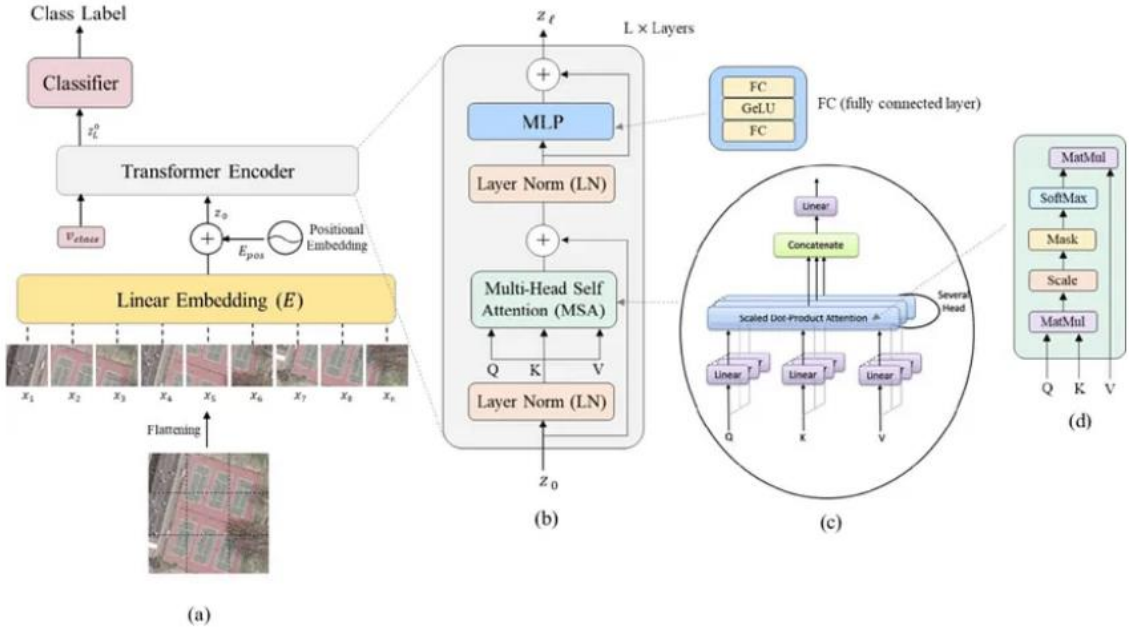


The use of multiple heads allows the model to capture different relationships between elements in the sequence, enhancing its ability to represent complex dependencies. Additionally, the model can learn to attend to different parts of the input sequence simultaneously, enabling it to capture both local and global contextual information.

Multi-Head Self-Attention has been shown to be highly effective in a wide range of natural language processing tasks, including machine translation, language modelling, and sentiment analysis.

Vision Transformer

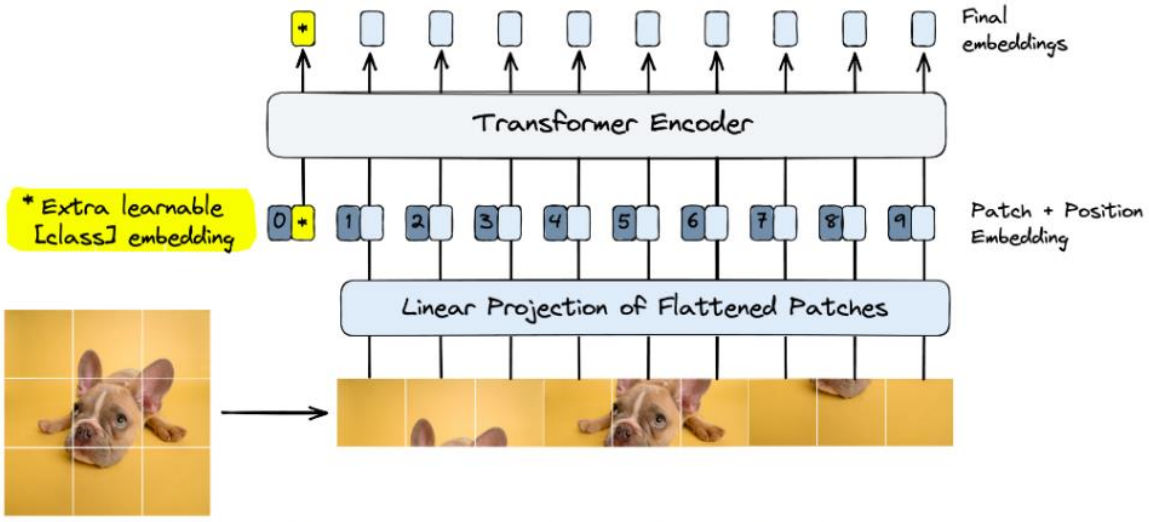
The Vision Transformer was introduced in a paper titled "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" by Dosovitskiy et al. in 2020. The paper proposed a new architecture for image classification tasks that replaces the convolutional layers of a traditional convolutional neural network with a series of transformer layers.



The architecture of the ViT with specific details on the transformer encoder and the MSA block. Keep this picture in mind. Picture from Bazi et. al.

The main idea behind the Vision Transformer is to represent the input image as a sequence of 2D patches, which are flattened into 1D vectors and then processed by the transformer layers. The authors showed that this approach can achieve state-of-the-art performance on several image classification benchmarks, including ImageNet, without using any convolutional layers.

One of the advantages of the Vision Transformer architecture is that it allows for greater flexibility and interpretability compared to traditional CNNs. In a CNN, the features learned by each layer are typically difficult to interpret, whereas the attention mechanisms used by transformers allow for more explicit feature extraction and selection. This can also make it easier to apply pre-trained models to other tasks or domains.



ViT process with the learnable *class embedding* highlight (left).

Source: https://www.pinecone.io/learn/vision-transformers/

However, the Vision Transformer also has some limitations. For example, it may not perform as well as CNNs on tasks that require spatial understanding or object localization, since it processes images as flat sequences of patches. The model also requires a large amount of training data and computational resources to achieve state-of-the-art performance.

Despite these limitations, the Vision Transformer represents a promising direction for deep learning in image classification, and ongoing research is exploring ways to improve its performance and extend its applications.

Positional Encoding

Positional encoding is a crucial component of the vision transformer model, which is a deep neural network architecture used for image classification and other computer vision tasks.

In the vision transformer, the input image is first divided into patches, each of which is treated as a separate "token" in the model's input sequence. However, unlike in natural language processing tasks, the position of each token (i.e., the patch's location within the image) contains important spatial information that must be preserved in order for the model to accurately classify the image.

To preserve positional information, the vision transformer adds a fixed set of learnable vectors, called positional encodings, to the patch embeddings before passing them through the transformer layers. These positional encodings contain information about the spatial location of each patch within the image, and are added to the token embeddings to provide the model with a sense of positional awareness.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

The positional encoding scheme used in the vision transformer is based on the sine and cosine functions, and is designed to capture both the position and frequency of the embeddings. This allows the model to learn to attend to different positions and frequencies within the input image, enabling it to capture complex patterns and features for image classification tasks.

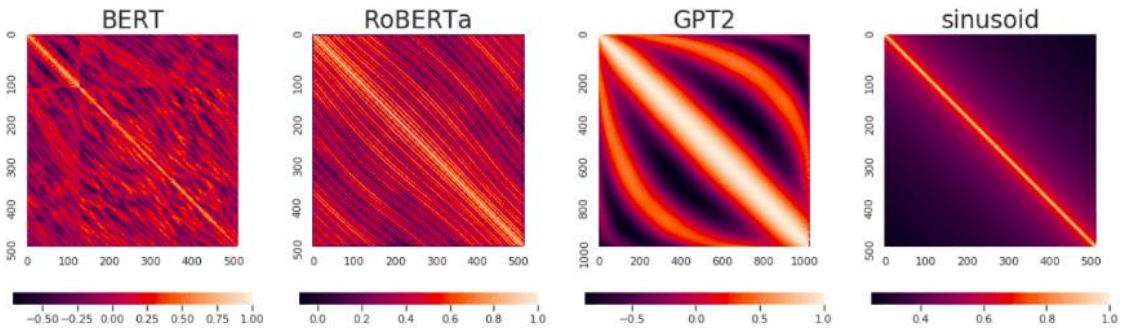
Overall, the positional encoding step in the vision transformer is a critical component for enabling the model to accurately classify images by preserving important spatial information within the input sequence.

Positional Embeddings

Positional embedding is a technique used in the Transformer model, a neural network architecture for natural language processing tasks.

In traditional neural networks, the order of input data does not matter because the input data is represented by a fixed-size vector. However, in natural language processing tasks, the order of words in a sentence is critical for understanding the meaning of the sentence. Therefore, the Transformer model includes positional embeddings to represent the position of each word in the input sequence.

The positional embedding is added to the input word embedding, which is a vector representation of each word in the input sequence. The positional embedding is a vector of the same dimension as the word embedding, and each element of the vector corresponds to a position in the input sequence. The values of the positional embedding are calculated based on the position of the corresponding word in the input sequence.



Position-wise similarity of multiple position embeddings. Image from Wang et Chen 2020

By adding the positional embedding to the word embedding, the Transformer model can differentiate between words based on their position in the input sequence, which allows it to capture the sequential information of the input text.

Positional Encoding vs Embedding

Embeddings are learned representations of input tokens that are used to encode semantic information about the input sequence. For example, in NLP tasks, word embeddings are learned representations of words that capture their semantic meaning and relationships to other words in the vocabulary. In computer vision tasks, image embeddings are learned representations of images that capture their features and patterns.

On the other hand, positional encoding is a technique used to preserve the spatial information of input tokens or patches. In NLP tasks, this is important because the position of words in a sentence can have a significant impact on their meaning and relationships to other words. In computer vision tasks, the position of image patches within the input image contains important spatial information that needs to be preserved for accurate classification.

In the vision transformer model, for example, the input image is divided into patches, each of which is treated as a separate token. To preserve the spatial information of these patches, the model applies positional encoding to the patch embeddings before passing them through the transformer layers.

In summary, while embeddings are used to capture semantic information about input tokens, positional encoding is used to preserve their spatial information, which is critical for accurate classification in certain tasks like computer vision.

Vision Transformer Architecture used for this assignment:

For this assignment , Vision Transformer Architecture and source code is highly influenced by the Blog https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c and the Github Repository: https://github.com/BrianPulfer/PapersReimplementations .

```
VisionTransformerClassifier(
  (linear_mapper): Linear(in_features=48, out_features=8, bias=True)
  (blocks): ModuleList(
    (0): VisionTransformerBlock(
      (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
      (mhsa): MultiHeadSelfAttentionBlock(
        (q_mappings): ModuleList(
          (0): Linear(in_features=1, out_features=1, bias=True)
          (1): Linear(in_features=1, out_features=1, bias=True)
          (2): Linear(in_features=1, out_features=1, bias=True)
          (3): Linear(in_features=1, out_features=1, bias=True)
          (4): Linear(in_features=1, out_features=1, bias=True)
          (5): Linear(in_features=1, out_features=1, bias=True)
          (6): Linear(in_features=1, out_features=1, bias=True)
          (7): Linear(in_features=1, out_features=1, bias=True)
        )
        (k_mappings): ModuleList(
          (0): Linear(in_features=1, out_features=1, bias=True)
          (1): Linear(in_features=1, out_features=1, bias=True)
          (2): Linear(in_features=1, out_features=1, bias=True)
          (3): Linear(in_features=1, out_features=1, bias=True)
```

```
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (v_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (activation_fn): Softmax(dim=-1)
    )
    (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=32, bias=True)
      (1): ReLU()
      (2): Linear(in_features=32, out_features=8, bias=True)
    )
  )
  (1): VisionTransformerBlock(
    (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mhsa): MultiHeadSelfAttentionBlock(
      (q_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (k_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (v_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
```

```
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (activation_fn): Softmax(dim=-1)
    )
    (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=32, bias=True)
      (1): ReLU()
      (2): Linear(in_features=32, out_features=8, bias=True)
    )
  )
  (2): VisionTransformerBlock(
    (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mhsa): MultiHeadSelfAttentionBlock(
      (q_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (k_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (v_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
```

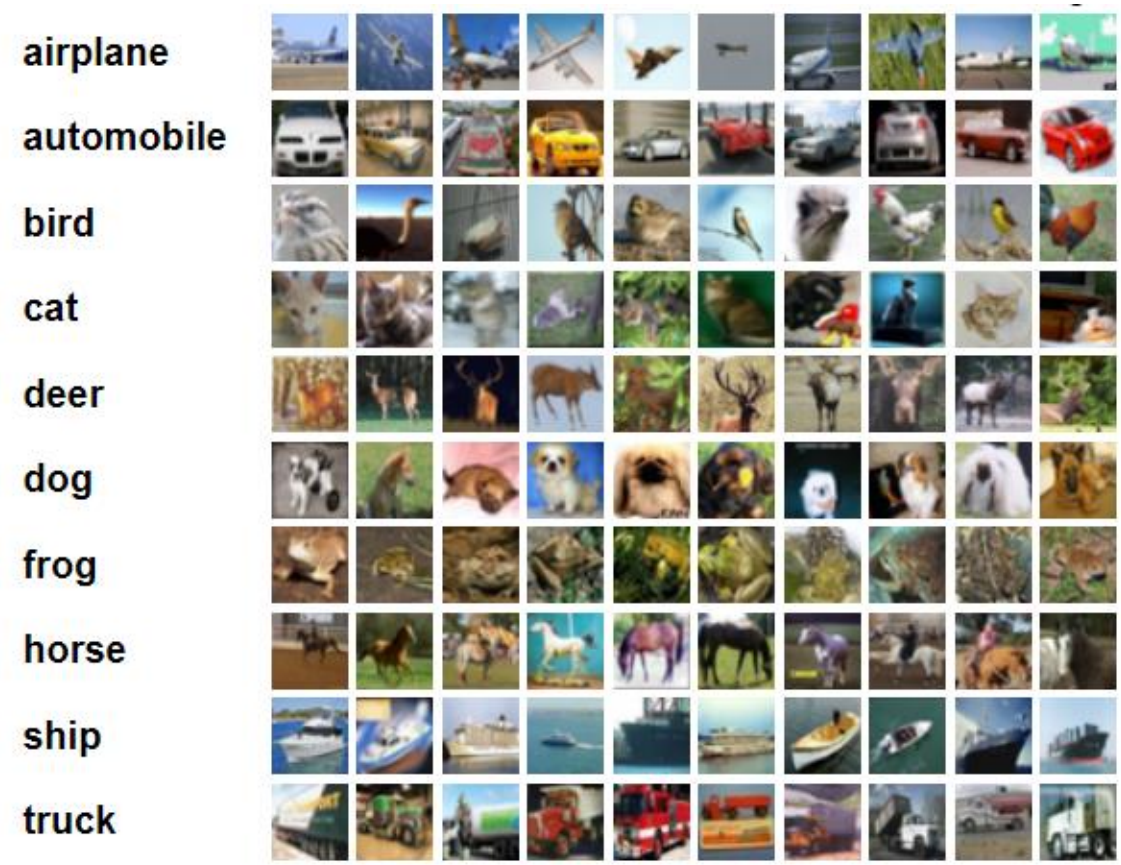
```
      )
      (activation_fn): Softmax(dim=-1)
    )
    (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=32, bias=True)
      (1): ReLU()
      (2): Linear(in_features=32, out_features=8, bias=True)
    )
  )
  (3): VisionTransformerBlock(
    (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mhsa): MultiHeadSelfAttentionBlock(
      (q_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (k_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (v_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (activation_fn): Softmax(dim=-1)
    )
    (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=32, bias=True)
      (1): ReLU()
```

```
      (2): Linear(in_features=32, out_features=8, bias=True)
    )
  )
  (4): VisionTransformerBlock(
    (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mhsa): MultiHeadSelfAttentionBlock(
      (q_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (k_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (v_mappings): ModuleList(
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, bias=True)
        (5): Linear(in_features=1, out_features=1, bias=True)
        (6): Linear(in_features=1, out_features=1, bias=True)
        (7): Linear(in_features=1, out_features=1, bias=True)
      )
      (activation_fn): Softmax(dim=-1)
    )
    (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=32, bias=True)
      (1): ReLU()
      (2): Linear(in_features=32, out_features=8, bias=True)
    )
  )
  (5): VisionTransformerBlock(
    (norm1): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
    (mhsa): MultiHeadSelfAttentionBlock(
      (q_mappings): ModuleList(
```

```
            (0): Linear(in_features=1, out_features=1, bias=True)
            (1): Linear(in_features=1, out_features=1, bias=True)
            (2): Linear(in_features=1, out_features=1, bias=True)
            (3): Linear(in_features=1, out_features=1, bias=True)
            (4): Linear(in_features=1, out_features=1, bias=True)
            (5): Linear(in_features=1, out_features=1, bias=True)
            (6): Linear(in_features=1, out_features=1, bias=True)
            (7): Linear(in_features=1, out_features=1, bias=True)
          )
          (k_mappings): ModuleList(
            (0): Linear(in_features=1, out_features=1, bias=True)
            (1): Linear(in_features=1, out_features=1, bias=True)
            (2): Linear(in_features=1, out_features=1, bias=True)
            (3): Linear(in_features=1, out_features=1, bias=True)
            (4): Linear(in_features=1, out_features=1, bias=True)
            (5): Linear(in_features=1, out_features=1, bias=True)
            (6): Linear(in_features=1, out_features=1, bias=True)
            (7): Linear(in_features=1, out_features=1, bias=True)
          )
          (v_mappings): ModuleList(
            (0): Linear(in_features=1, out_features=1, bias=True)
            (1): Linear(in_features=1, out_features=1, bias=True)
            (2): Linear(in_features=1, out_features=1, bias=True)
            (3): Linear(in_features=1, out_features=1, bias=True)
            (4): Linear(in_features=1, out_features=1, bias=True)
            (5): Linear(in_features=1, out_features=1, bias=True)
            (6): Linear(in_features=1, out_features=1, bias=True)
            (7): Linear(in_features=1, out_features=1, bias=True)
          )
          (activation_fn): Softmax(dim=-1)
        )
        (norm2): LayerNorm((8,), eps=1e-05, elementwise_affine=True)
        (mlp): Sequential(
          (0): Linear(in_features=8, out_features=32, bias=True)
          (1): ReLU()
          (2): Linear(in_features=32, out_features=8, bias=True)
        )
      )
    )
    (mlp): Sequential(
      (0): Linear(in_features=8, out_features=5, bias=True)
      (1): Softmax(dim=-1)
    )
)
```

CIFAR 10 Dataset:

The CIFAR-10 dataset consists of **60000 32x32 colour images** in **10 classes**, with 6000 images per class. There are **50000 training** images and **10000 test images**.



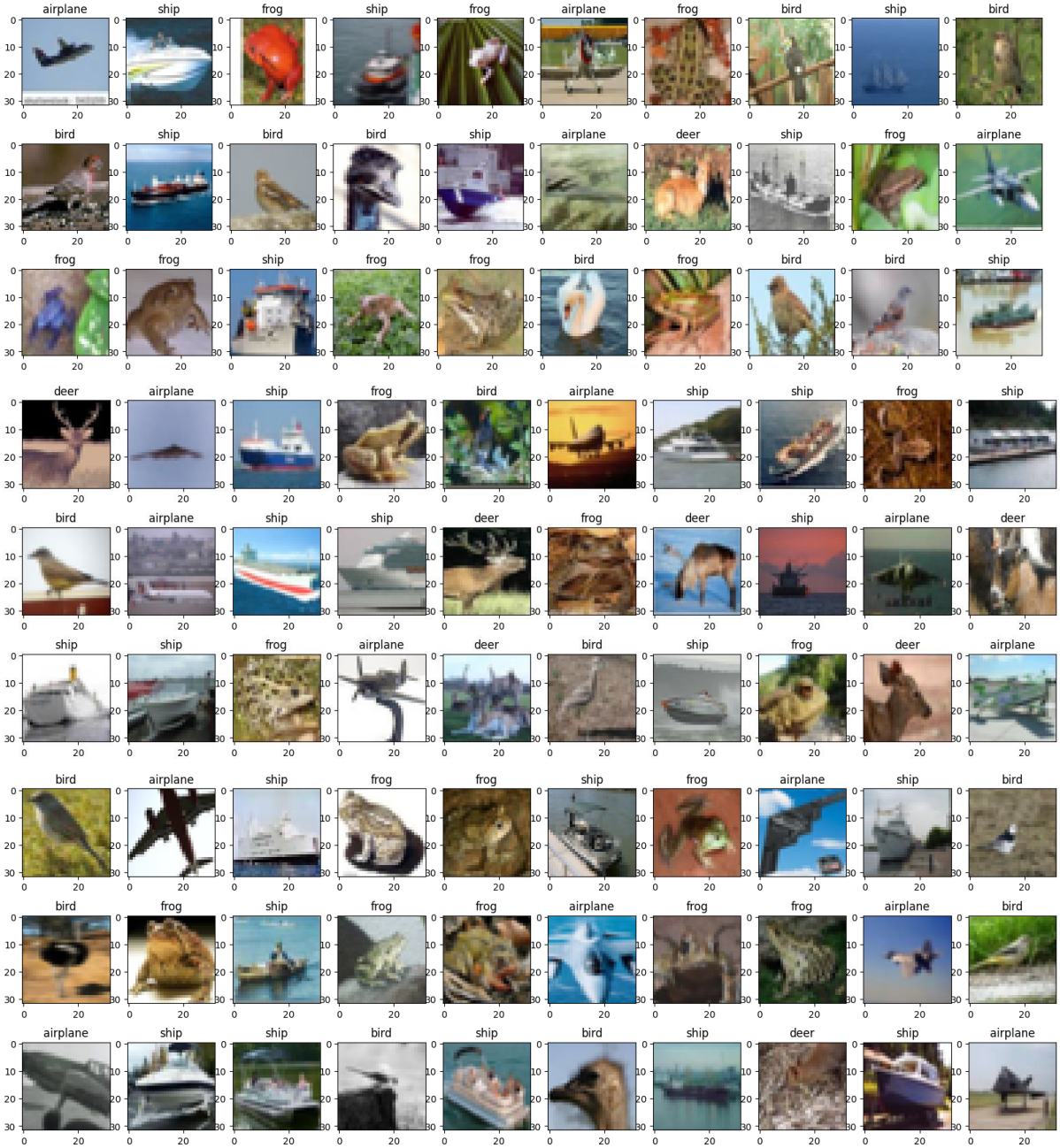
Torch Transformations Applied:

- Normalize to (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
- Transform to tensor

Using default PyTorch data-loader function with the dataset path ‘./data’ and used above mentioned transforms for pre-processing.

Selecting **even classes 0,2,4,6,8** as roll number is odd (**M21AIE225**)

Sample images from selected classes with their class labels:



Hyper Parameters Used:

Hyper Parameter	Value
Learning Rate	1e-4
Number of Epochs	10
Batch Size	32
Loss Function	CrossEntropyLoss
Optimizer	Adam

Slurm Script Used:

```
#!/bin/bash
#SBATCH --job-name=m21aie225_lab7   # Job name
#SBATCH --partition=gpu2          #Partition name can be test/small/medium/large/gpu #Partition
“gpu” should be used only for gpu jobs
#SBATCH --nodes=1                       # Run all processes on a single node
#SBATCH --ntasks=1                      # Run a single task
#SBATCH --cpus-per-task=4         # Number of CPU cores per task
#SBATCH --gres=gpu:1              # Include gpu for the task (only for GPU jobs)
#SBATCH --mem=16gb                      # Total memory limit
#SBATCH --time=90:00:00           # Time limit hrs:min:sec
#SBATCH --output=m21aie225_lab7_%j.log # Standard output and error log
#date;hostname;pwd


module load python/3.8

python3 M21AIE225_DLOps_Assignment_3_Q_1.py
```

Highlighted modifications done in the given template.

Commands Used to submit batch:

[m21aie225@hpclogin ~]$ pwd

/iitjhome/m21aie225

[m21aie225@hpclogin ~]$ sbatch slurm_script.sh

Submitted batch job 40401

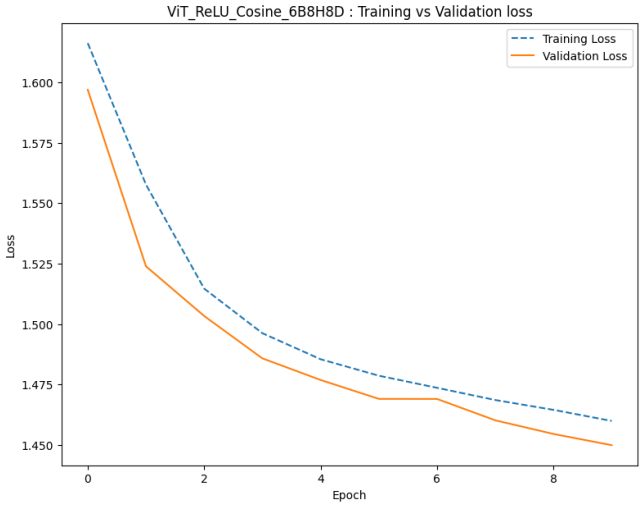
[m21aie225@hpclogin ~]$ squeue -j 40401

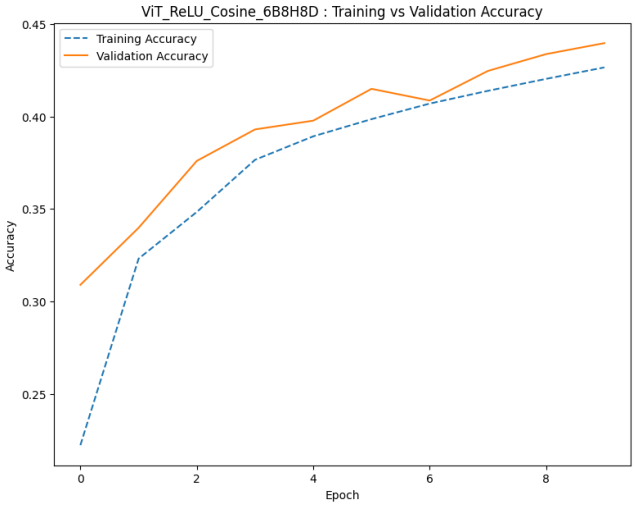
JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST(REASON)
40401	gpu2	m21aie22	m21aie22	R	0:16	1	gpu2

[m21aie225@hpclogin ~]$ cat m21aie225_lab7_40401.log

Model Training details

1. ViT with Cosine PE and ReLU Activation:



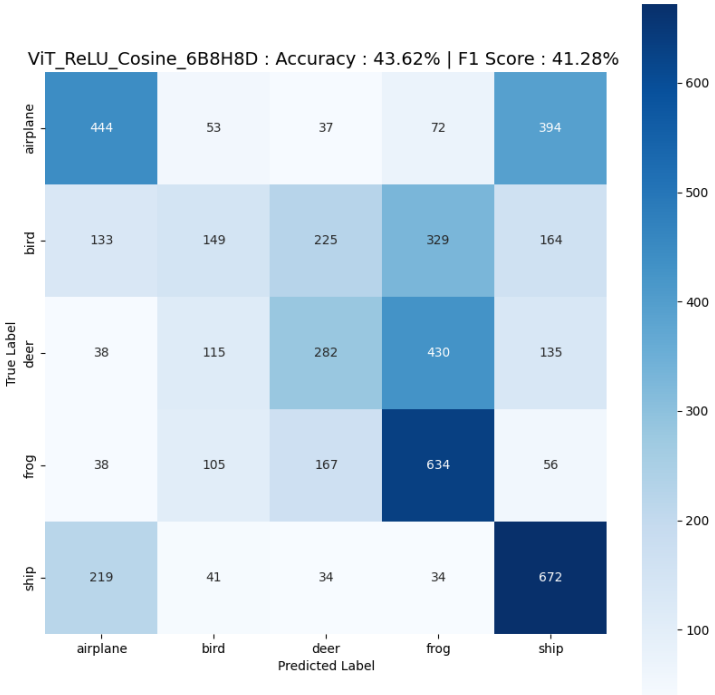


MODEL EVALUATION SUMMARY:
ACCURACY: 43.620%
F1 SCORE: 41.276%

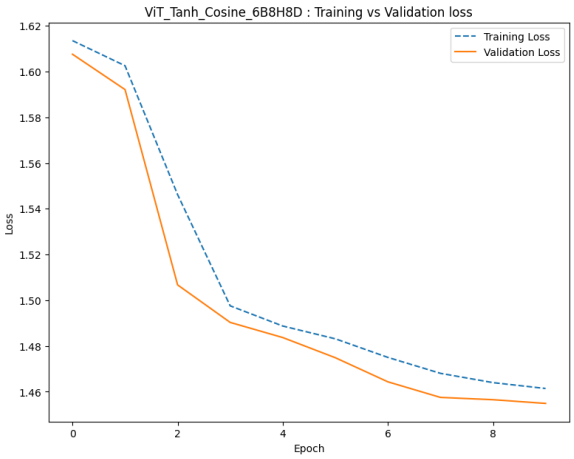
CLASS WISE ACCURACY SCORE:

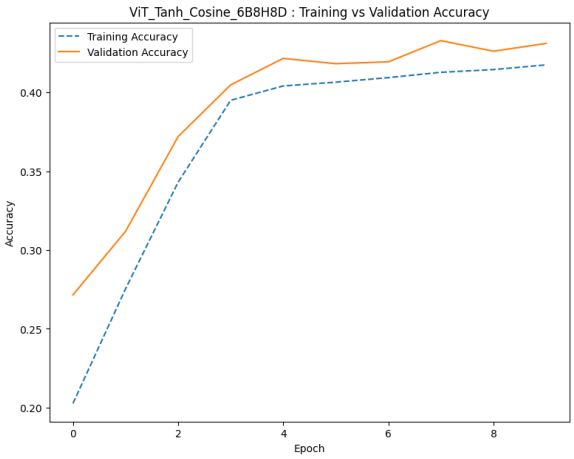
AIRPLANE	BIRD	DEER	FROG	SHIP
0.444	0.149	0.282	0.634	0.672

CONFUSION MATRIX:



2. ViT with Cosine PE and Tanh Activation:



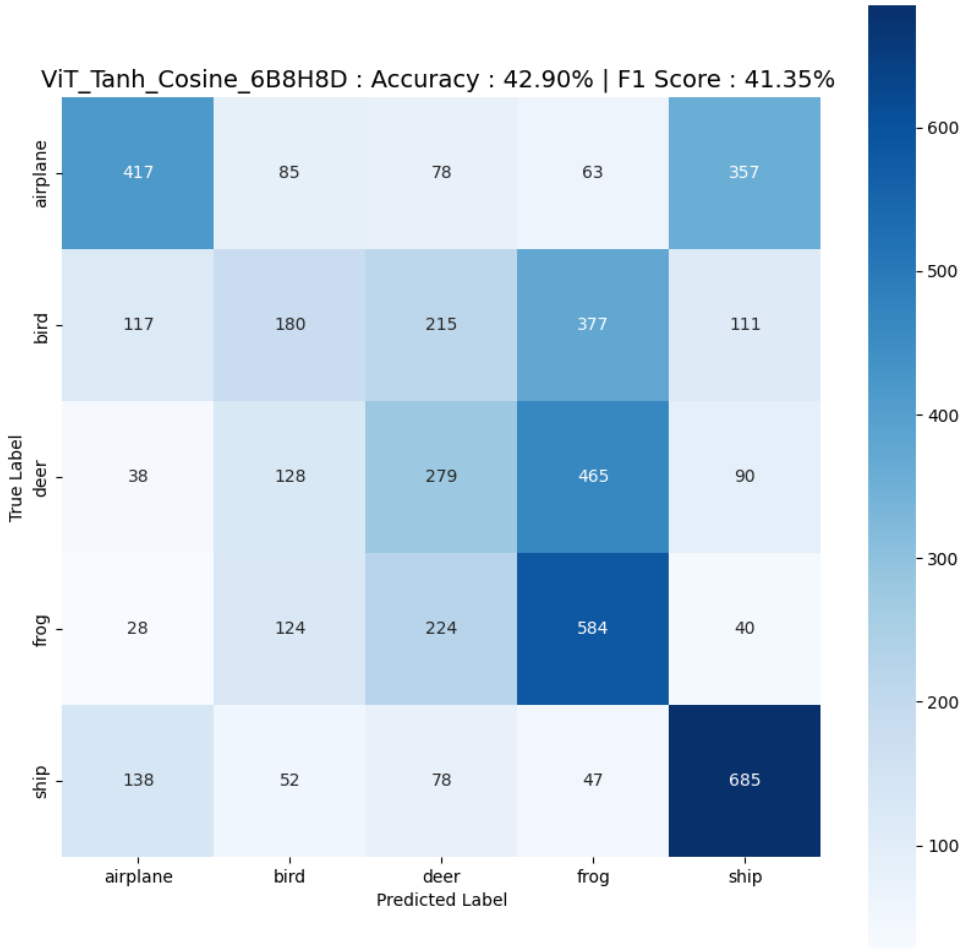


MODEL EVALUATION SUMMARY:
ACCURACY: 42.900%
F1 SCORE: 41.354%

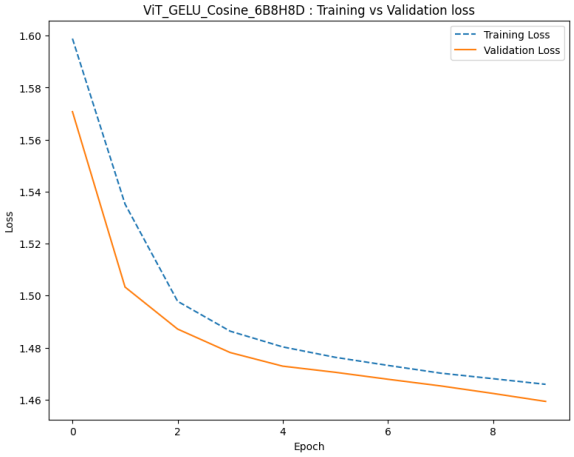
CLASSWISE ACCURACY SCORE:

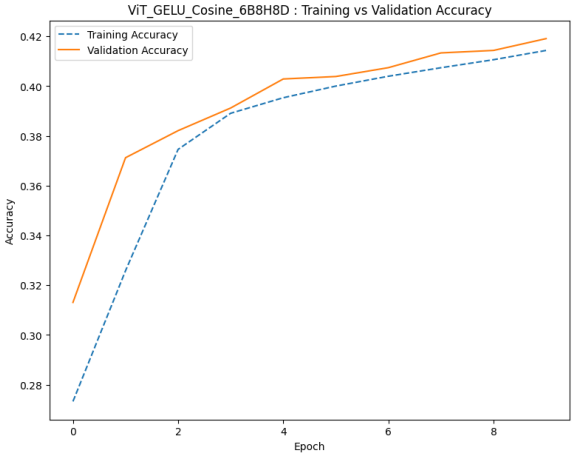
AIRPLANE	BIRD	DEER	FROG	SHIP
0.417	0.18	0.279	0.584	0.685

CONFUSION MATRIX:



3. ViT with Cosine PE and GELU Activation:



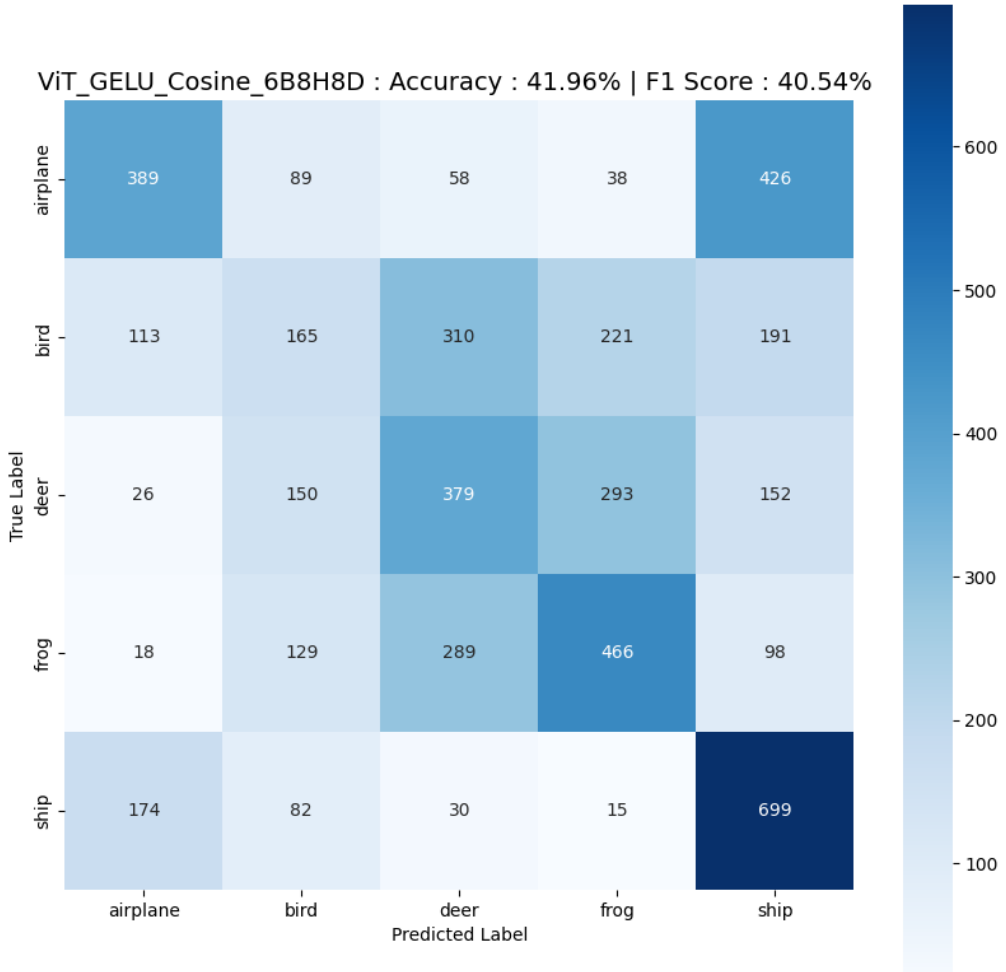


MODEL EVALUATION SUMMARY:
ACCURACY: 41.960%
F1 SCORE: 40.536%

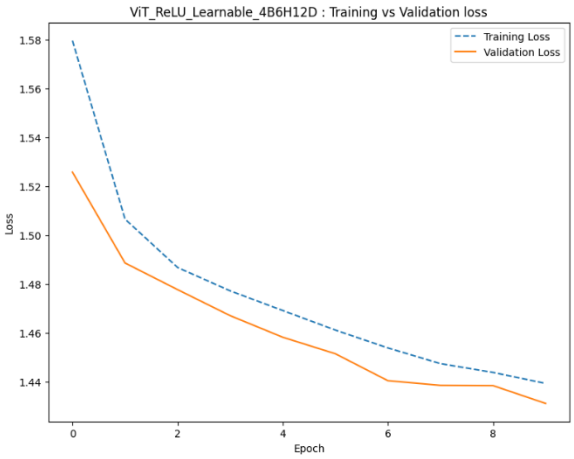
CLASSWISE ACCURACY SCORE:

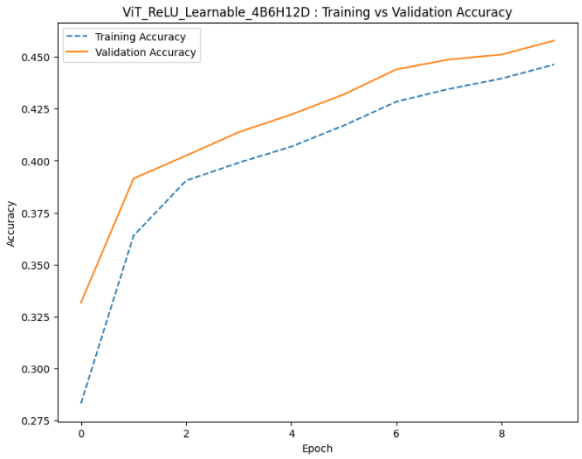
AIRPLANE	BIRD	DEER	FROG	SHIP
0.389	0.165	0.379	0.466	0.699

CONFUSION MATRIX:



4. ViT with Learnable and ReLU Activation:



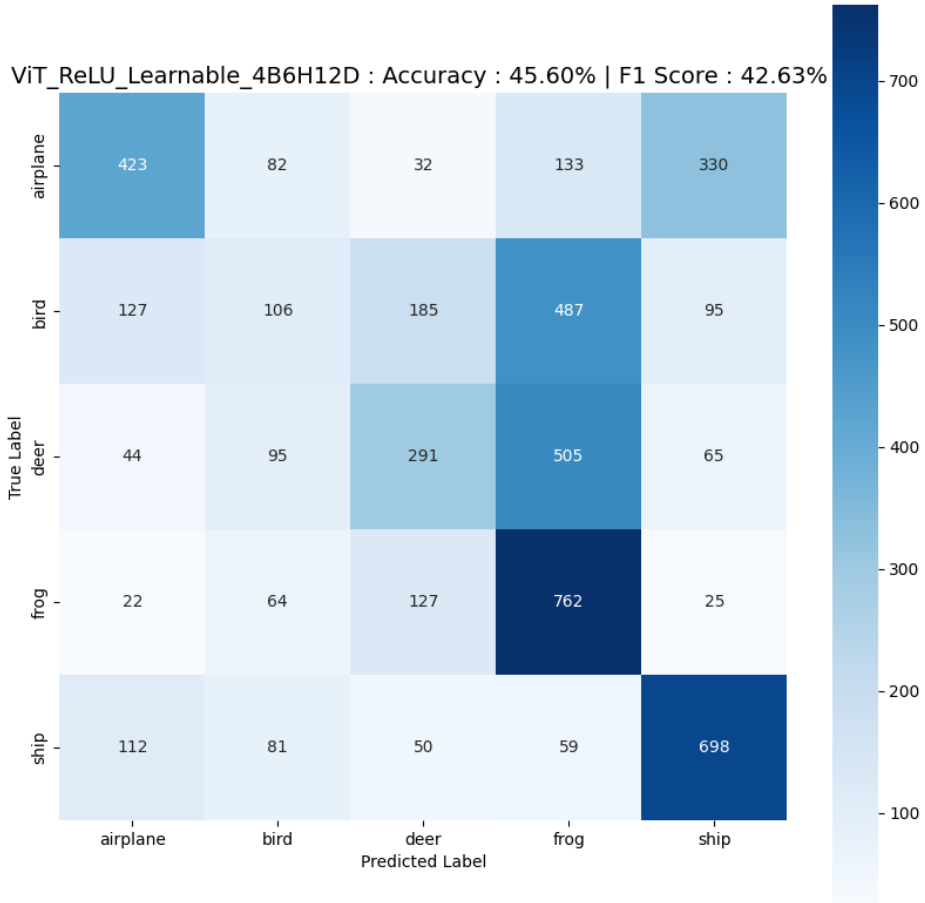


MODEL EVALUATION SUMMARY:
ACCURACY: 45.600%
F1 SCORE: 42.631%

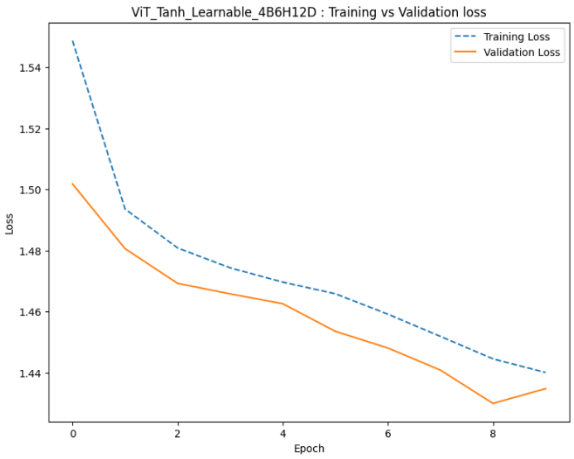
CLASSWISE ACCURACY SCORE:

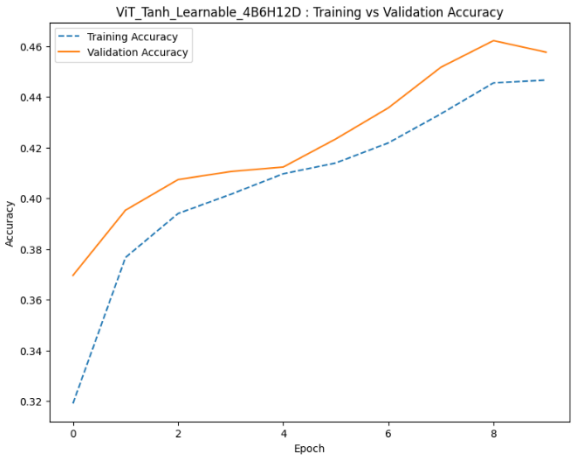
AIRPLANE	BIRD	DEER	FROG	SHIP
0.423	0.106	0.291	0.762	0.698

CONFUSION MATRIX:



5. ViT with Learnable and Tanh Activation:



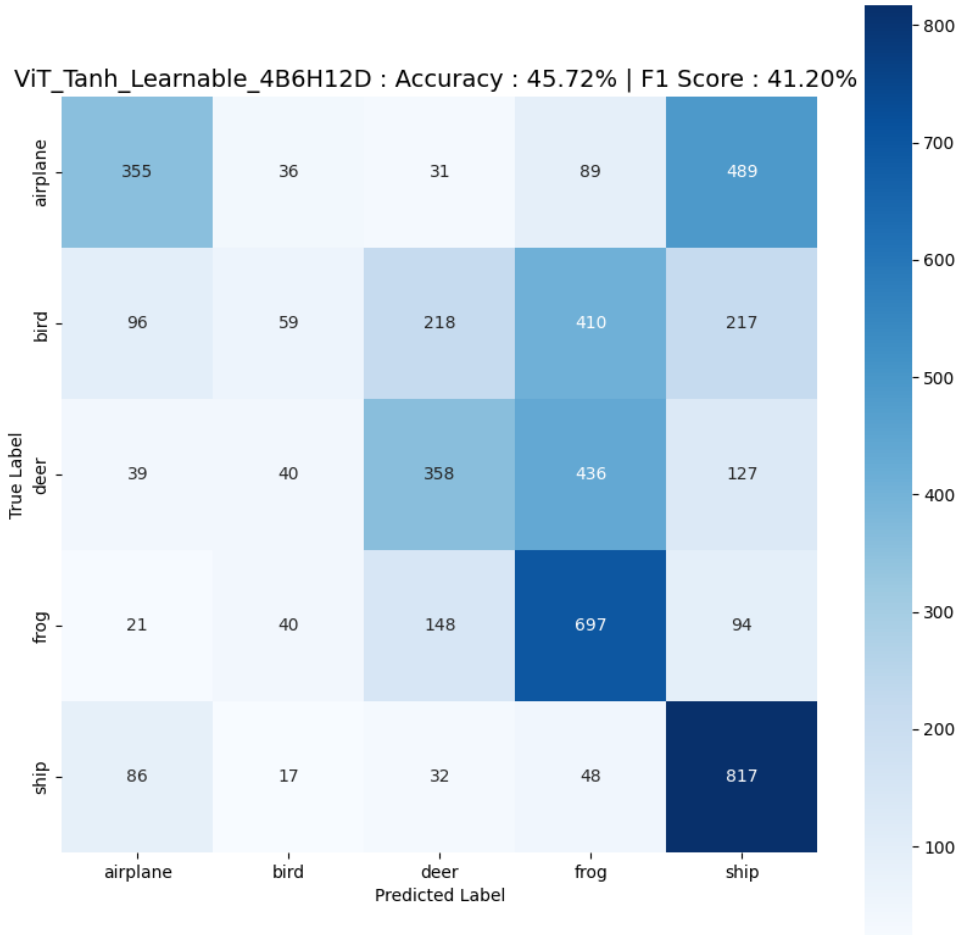


MODEL EVALUATION SUMMARY:
ACCURACY: 45.720%
F1 SCORE: 41.198%

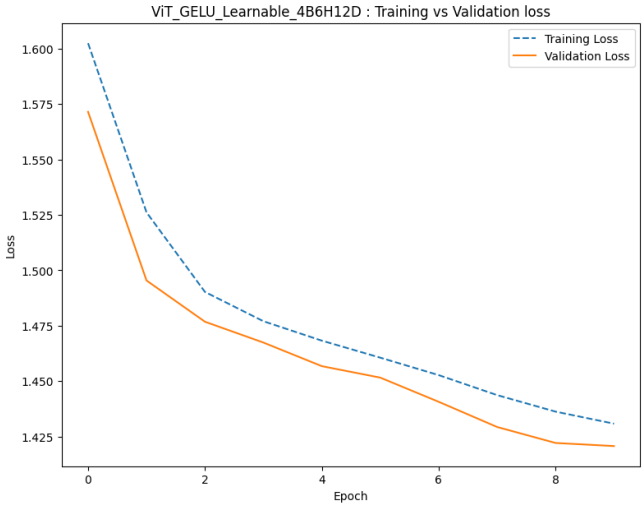
CLASSWISE ACCURACY SCORE:

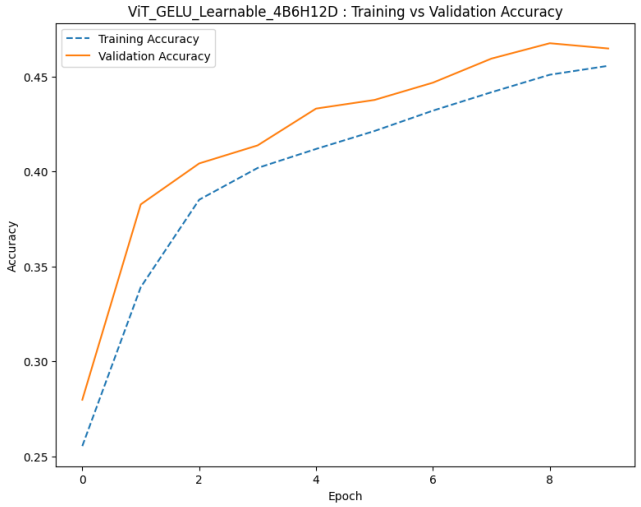
AIRPLANE	BIRD	DEER	FROG	SHIP
0.355	0.059	0.358	0.697	0.817

CONFUSION MATRIX:



6. ViT with Learnable and GELU Activation:



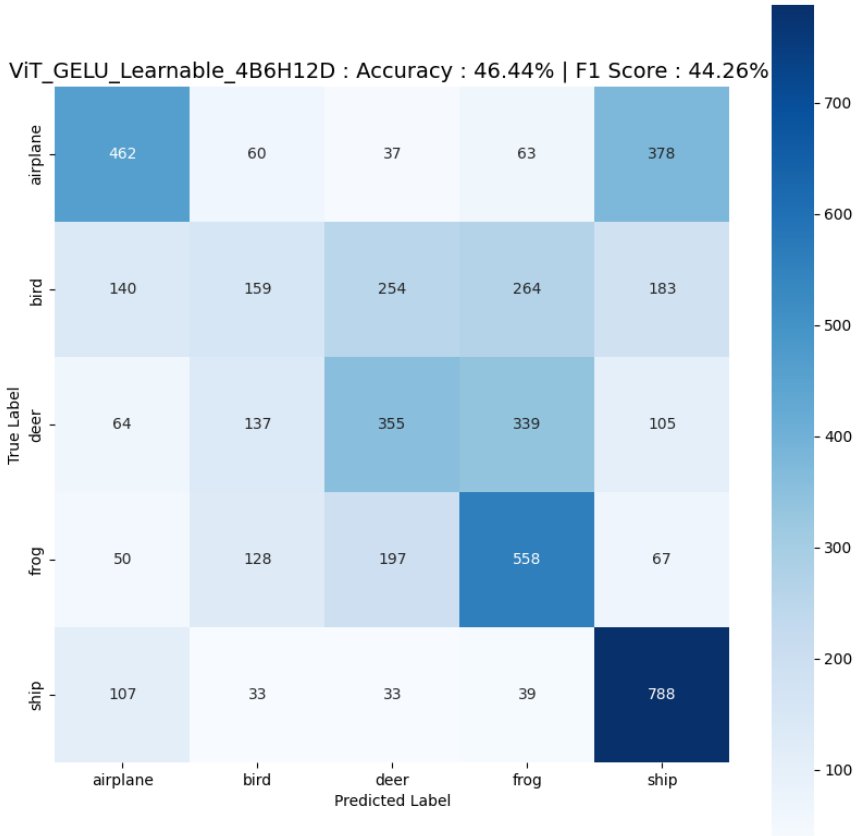


MODEL EVALUATION SUMMARY:
ACCURACY: 46.440%
F1 SCORE: 44.265%

CLASSWISE ACCURACY SCORE:

AIRPLANE	BIRD	DEER	FROG	SHIP
0.462	0.159	0.355	0.558	0.788

CONFUSION MATRIX:



Final Results:

Model Architecture	Positional Embed	Activation Fn	Training Time	Training Accuracy	Test Accuracy
ViT_ReLU_Cosine_6B8H8D	Cosine PE	ReLU	9355.130	0.427	0.440
ViT_Tanh_Cosine_6B8H8D	Cosine PE	Tanh	9638.374	0.418	0.431
ViT_GELU_Cosine_6B8H8D	Cosine PE	GELU	9658.270	0.414	0.419
ViT_ReLU_Learnable_4B6H12D	Learnable PE	ReLU	4596.729	0.446	0.458
ViT_Tanh_Learnable_4B6H12D	Learnable PE	Tanh	4555.459	0.447	0.458
ViT_GELU_Learnable_4B6H12D	Learnable PE	GELU	4587.071	0.456	0.465

The table shows the results of training different ViT models with different positional embedding types, activation functions, training time, training accuracy, and test accuracy.

Based on the above results, we can make the following observations:

1. Performance is better with learnable positional embeddings compared to cosine positional embeddings for all the models tested.
2. Among the activation functions, ReLU performing better than other activation functions on an average. Though highest performing model has activation function GELU, it's worst performer with Cosine Positional Embedding.
3. ViT with learnable positional embeddings and ReLU activation function seems to have the highest test accuracy (0.458), closely followed by ViT with learnable positional embeddings and Tanh activation function (0.458).
4. The training time is significantly lower for models with learnable positional embeddings compared to cosine positional embeddings.
5. All the models failed to identify the class Bird, and the class Ship has been identified by all the models better.

Overall, it appears that using learnable positional embeddings is more effective than using cosine positional embeddings in improving the performance of Vision Transformer. Additionally, the choice of activation function has a relatively minor impact on performance. The best model in terms of test accuracy was ViT with learnable positional embeddings and ReLU activation function.

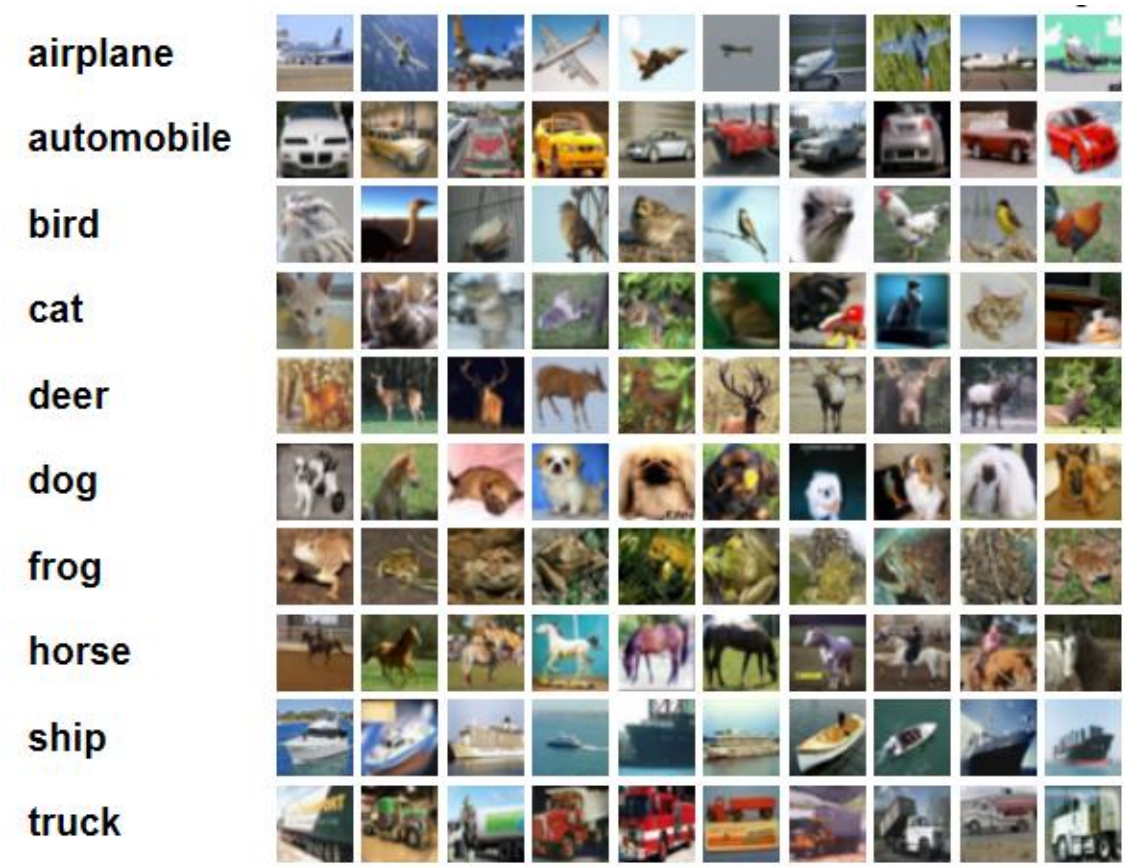
It is worth noting that the test accuracy of all models is relatively low, suggesting that there may be room for improvement in the model architecture or training process. Additionally, the training time for each model varies significantly, with some models taking almost twice as long as others despite achieving similar test accuracy. This may be due to differences in number of heads and blocks used in the model for training.

Question 2

1. Load and pre-processing CIFAR10 dataset using standard augmentation and normalization techniques:

CIFAR 10 Dataset:

The CIFAR-10 dataset consists of **60000 32x32 colour images** in **10 classes**, with 6000 images per class. There are **50000 training** images and **10000 test images**.



Torch Transformations Applied:

- Random Crop (in training only)
- Random Flip (in training only)
- Transform to tensor
- Normalize to (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

The dataset is first loaded and split into training and testing sets. The images are then augmented with various transformations to increase the size of the training dataset and improve the model's ability to generalize. These transformations include randomly flipping the images horizontally, randomly cropping them, and randomly adjusting the brightness and contrast.

After augmentation, the pixel values of the images are normalized so that they have a mean of 0 and a standard deviation of 1. This is done to make it easier for the model to learn from the data.

The final step is to convert the image data into PyTorch tensors, which are the preferred format for working with neural networks in PyTorch. The training and testing datasets are then returned and can be used to train and evaluate machine learning models.

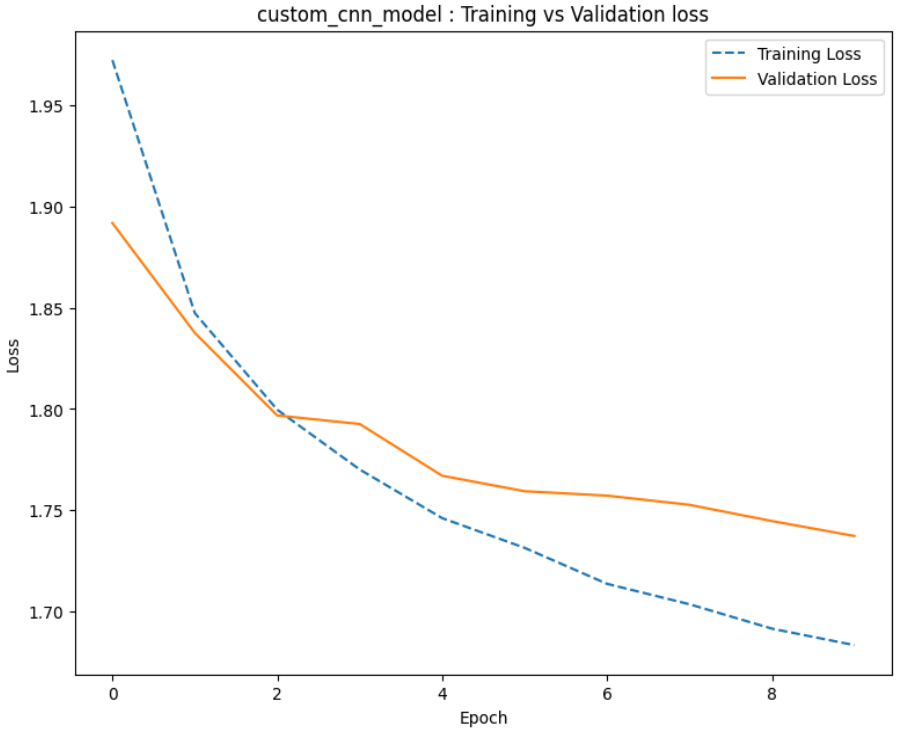
2.Train the following models for profiling them using during the training step [5 *2 =10Marks]

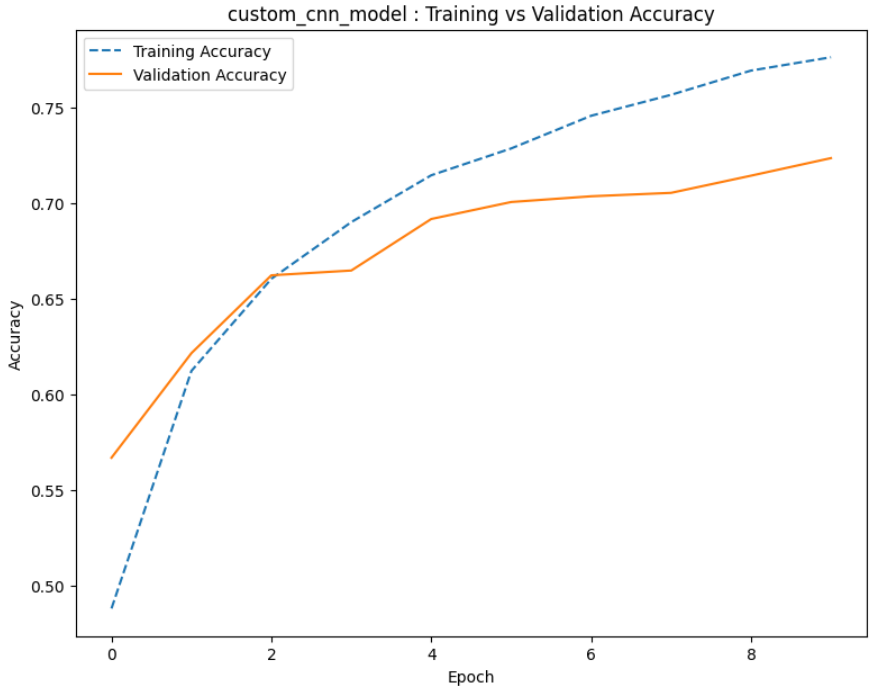
Custom Image Classifiers Architecture:

```
CustomImageClassifier(
  (conv_layers): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU()
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (13): ReLU()
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (17): ReLU()
  )
  (fc_layers): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Linear(in_features=4096, out_features=1024, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.1, inplace=False)
    (4): Linear(in_features=1024, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=10, bias=True)
    (7): Softmax(dim=1)
  )
)
```

Training of Custom Model:

Epoch: 1 (0m 34s) Training Loss: 1.972, Test Loss: 1.892, Training acc: 0.49, Test acc: 0.57,
Epoch: 2 (0m 26s) Training Loss: 1.847, Test Loss: 1.838, Training acc: 0.61, Test acc: 0.62,
Epoch: 3 (0m 24s) Training Loss: 1.800, Test Loss: 1.797, Training acc: 0.66, Test acc: 0.66,
Epoch: 4 (0m 29s) Training Loss: 1.770, Test Loss: 1.793, Training acc: 0.69, Test acc: 0.66,
Epoch: 5 (0m 25s) Training Loss: 1.746, Test Loss: 1.767, Training acc: 0.71, Test acc: 0.69,
Epoch: 6 (0m 22s) Training Loss: 1.731, Test Loss: 1.759, Training acc: 0.73, Test acc: 0.70,
Epoch: 7 (0m 21s) Training Loss: 1.714, Test Loss: 1.757, Training acc: 0.75, Test acc: 0.70,
Epoch: 8 (0m 22s) Training Loss: 1.704, Test Loss: 1.753, Training acc: 0.76, Test acc: 0.71,
Epoch: 9 (0m 21s) Training Loss: 1.692, Test Loss: 1.745, Training acc: 0.77, Test acc: 0.71,
Epoch: 10 (0m 22s) Training Loss: 1.683, Test Loss: 1.737, Training acc: 0.78, Test acc: 0.72,
Training completed in 4m 10s Training Loss: 1.683, Test Loss: 1.737, Training acc: 0.78, Test acc: 0.72,



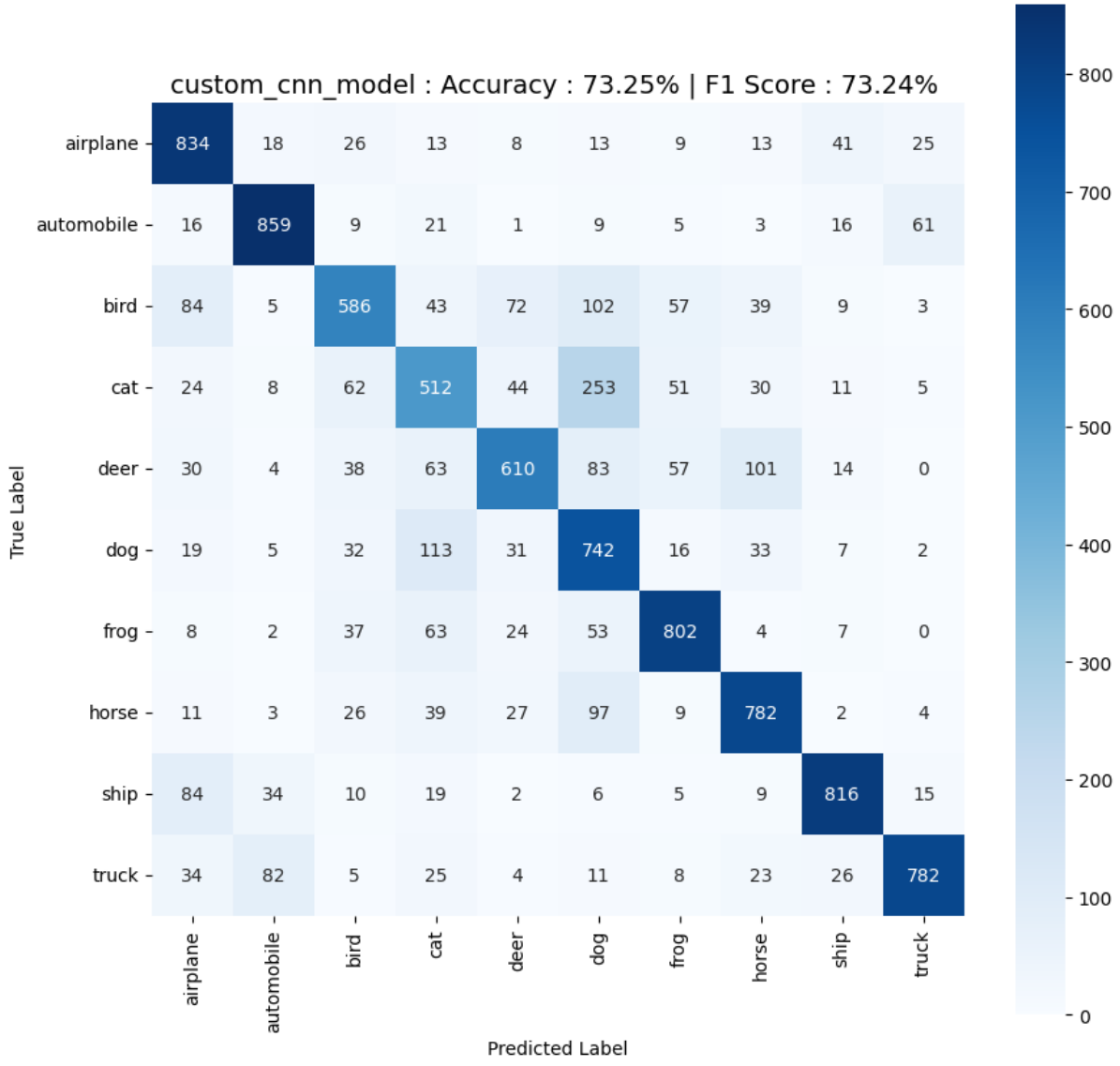


ACCURACY: 73.250%
F1 SCORE: 73.241%

CLASSWISE ACCURACY SCORE:

AIRPLANE	AUTOMOBILE	BIRD	CAT	DEER	DOG	FROG	HORSE	SHIP	TRUCK
0.834	0.859	0.586	0.512	0.61	0.742	0.802	0.782	0.816	0.782

CONFUSION MATRIX:

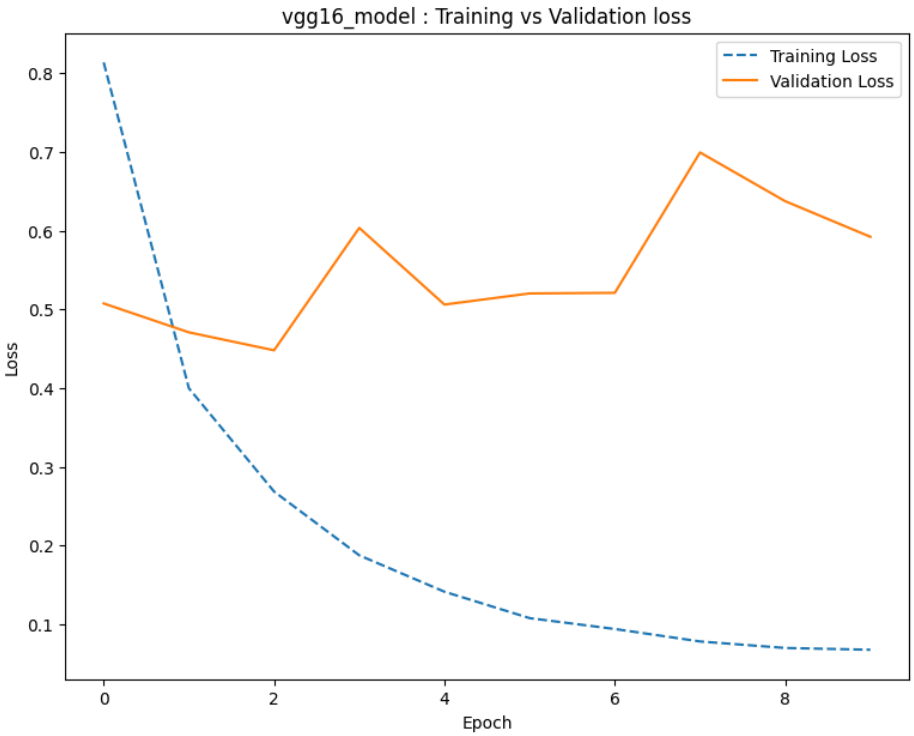


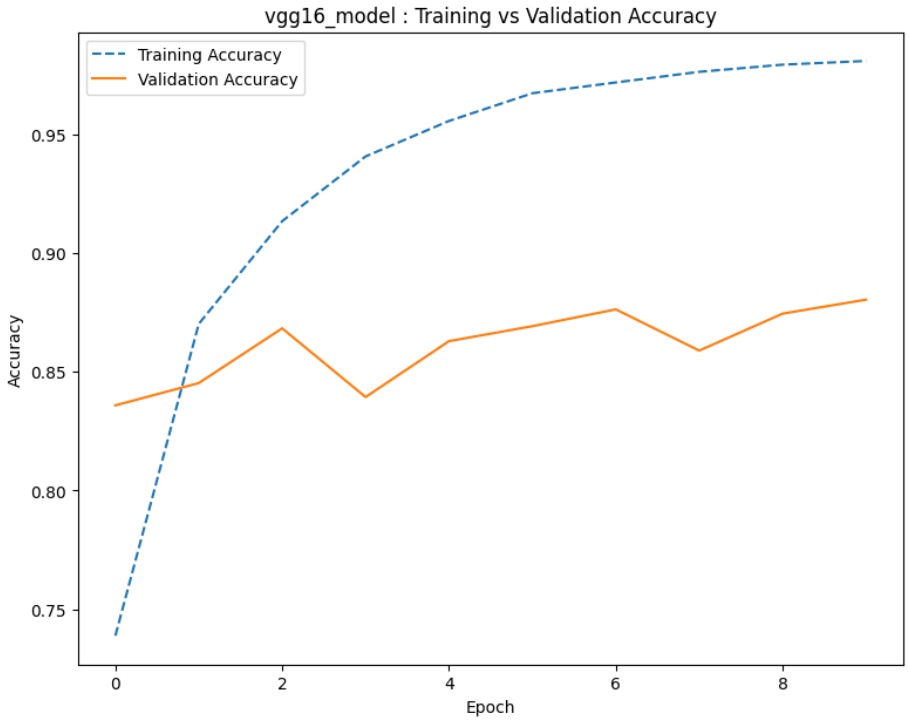
VGG 16 Model Architecture:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Training of VGG16 Model:

Epoch: 1 (2m 5s) Training Loss: 0.814, Test Loss: 0.508, Training acc: 0.74, Test acc: 0.84,
Epoch: 2 (2m 3s) Training Loss: 0.400, Test Loss: 0.471, Training acc: 0.87, Test acc: 0.85,
Epoch: 3 (2m 3s) Training Loss: 0.269, Test Loss: 0.448, Training acc: 0.91, Test acc: 0.87,
Epoch: 4 (2m 3s) Training Loss: 0.188, Test Loss: 0.604, Training acc: 0.94, Test acc: 0.84,
Epoch: 5 (2m 3s) Training Loss: 0.141, Test Loss: 0.506, Training acc: 0.96, Test acc: 0.86,
Epoch: 6 (2m 4s) Training Loss: 0.108, Test Loss: 0.520, Training acc: 0.97, Test acc: 0.87,
Epoch: 7 (2m 3s) Training Loss: 0.094, Test Loss: 0.521, Training acc: 0.97, Test acc: 0.88,
Epoch: 8 (2m 3s) Training Loss: 0.078, Test Loss: 0.700, Training acc: 0.98, Test acc: 0.86,
Epoch: 9 (2m 3s) Training Loss: 0.070, Test Loss: 0.638, Training acc: 0.98, Test acc: 0.87,
Epoch: 10 (2m 3s) Training Loss: 0.068, Test Loss: 0.592, Training acc: 0.98, Test acc: 0.88,
Training completed in 20m 39s Training Loss: 0.068, Test Loss: 0.592, Training acc: 0.98, Test acc: 0.88,



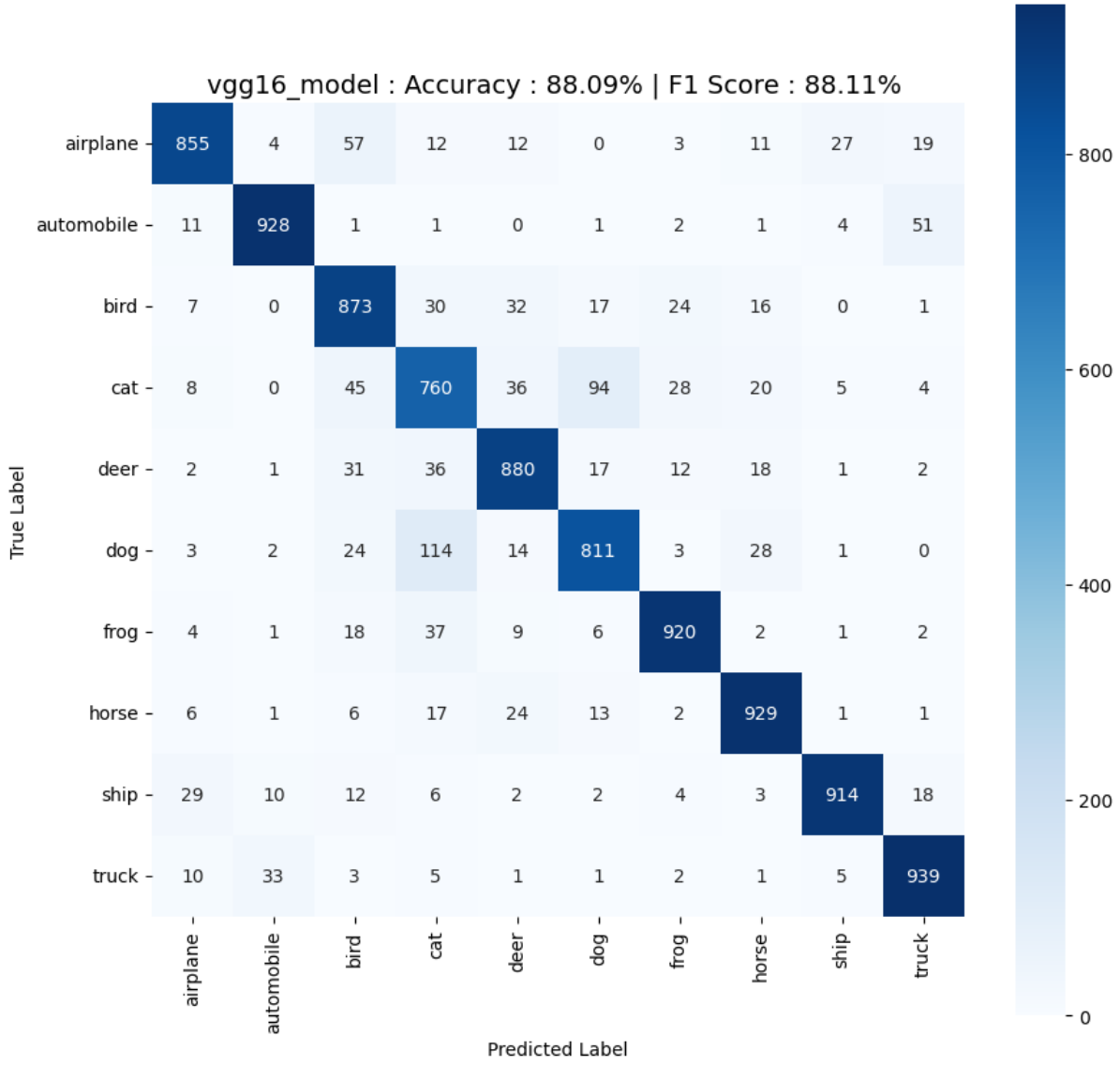


Accuracy: 88.090%
F1 Score: 88.109%

Classwise Accuracy Score:

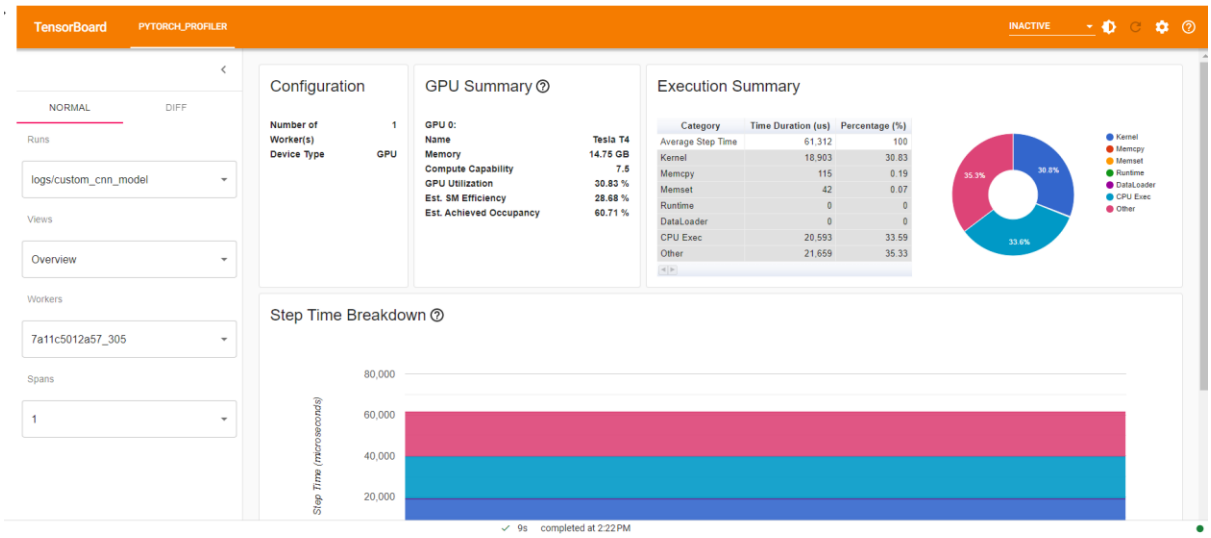
airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0.855	0.928	0.873	0.76	0.88	0.811	0.92	0.929	0.914	0.939

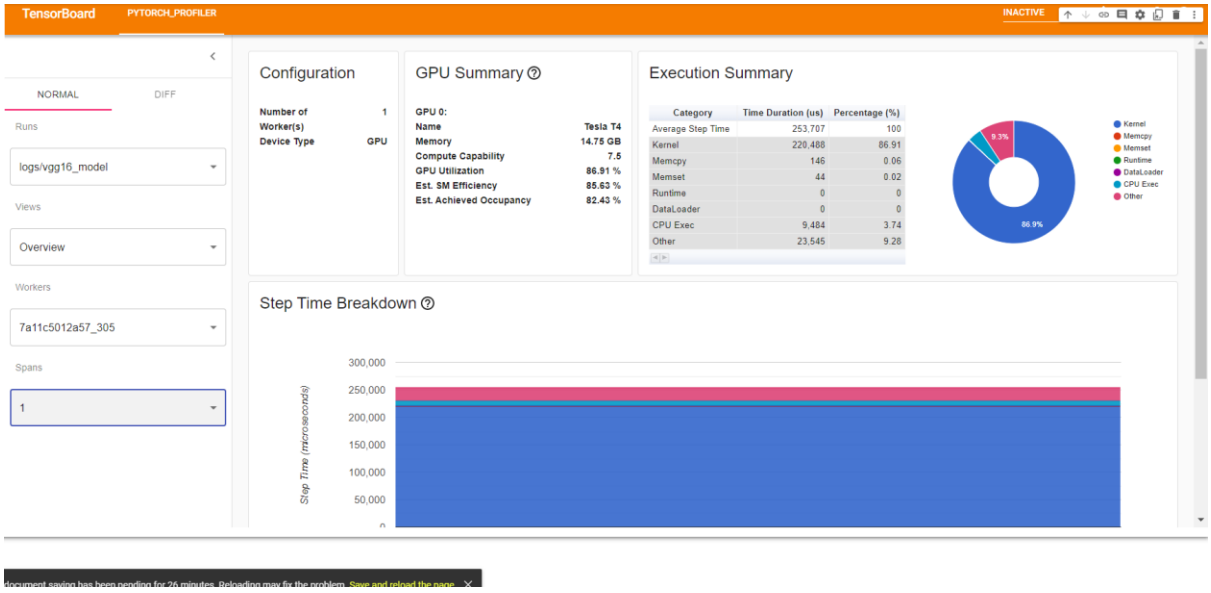
Confusion Matrix:



3. After the profiling of your model, figure out the minimum change in the architecture that would lead to a gain in performance and decrease training time on CIFAR10 dataset as compared to one achieved before.

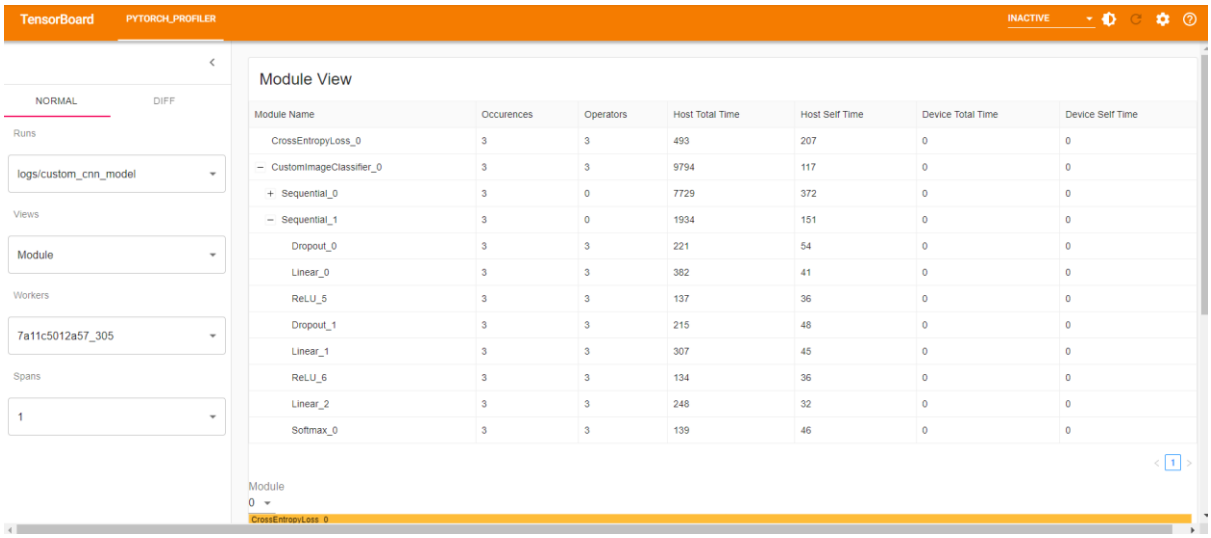
Over view of both model:

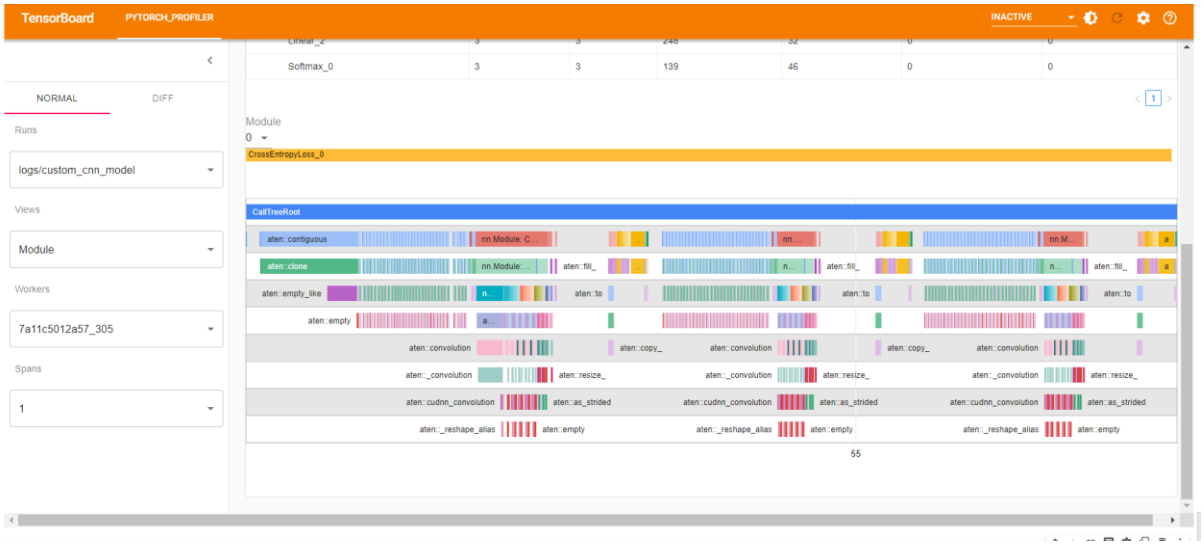


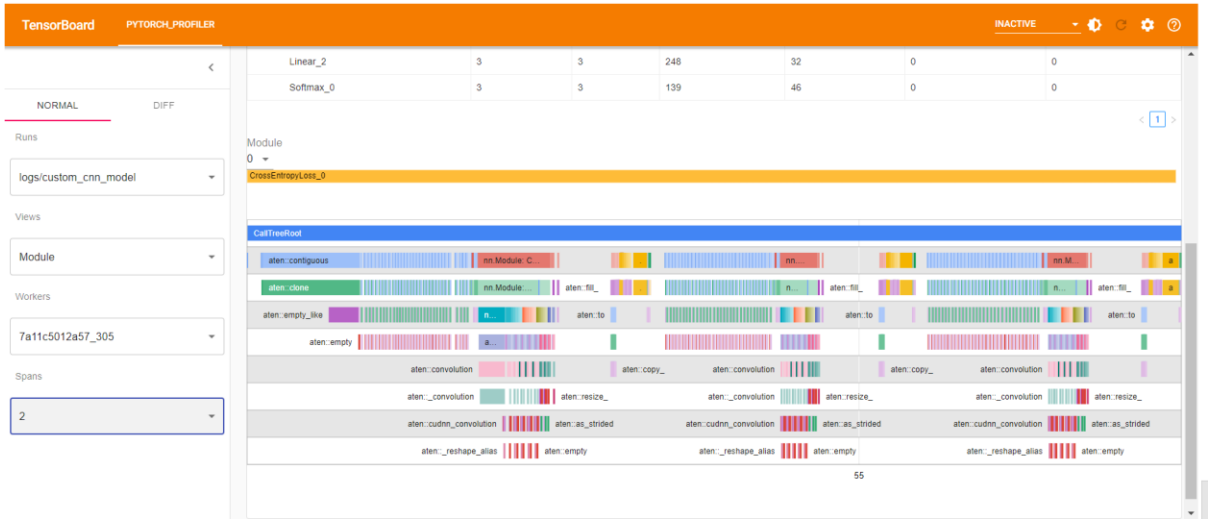


Module Wise Time Break up:

Custom Model:







From the above graphs, it is found that convolution layers (specially first one) is taking most of the time.

Modified Custom Model:

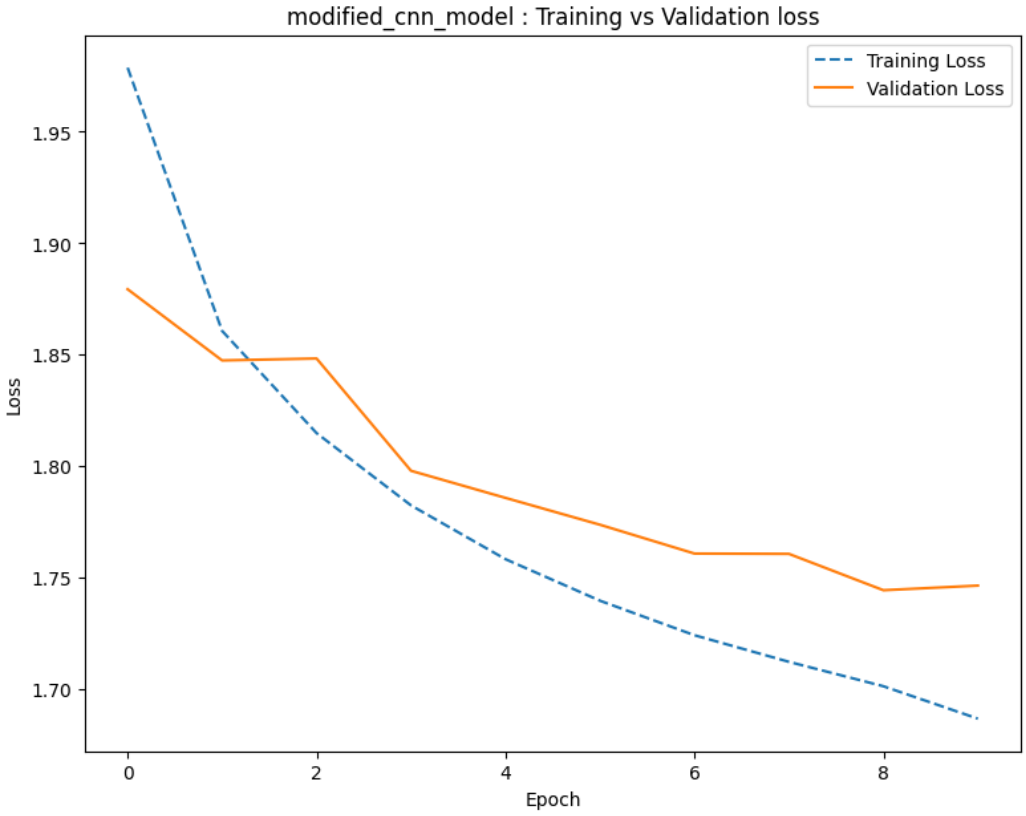
```
ModifiedCustomImageClassifier(
  (conv_layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
  )
  (fc_layers): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Linear(in_features=4096, out_features=1024, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.1, inplace=False)
    (4): Linear(in_features=1024, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=10, bias=True)
    (7): Softmax(dim=1)
  )
)
```

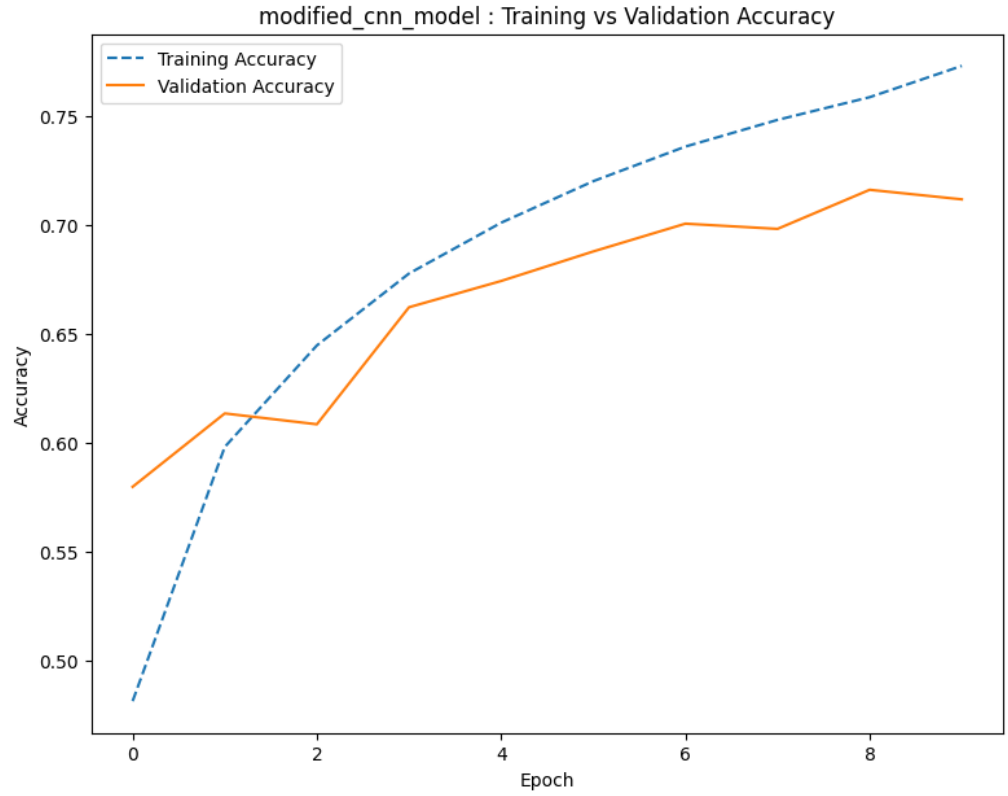
[Removed first convolution layer from first custom model]

Training:

Epoch: 1 (0m 33s) Training Loss: 1.964, Test Loss: 1.899, Training acc: 0.50, Test acc: 0.56,
Epoch: 2 (0m 22s) Training Loss: 1.846, Test Loss: 1.839, Training acc: 0.61, Test acc: 0.62,
Epoch: 3 (0m 21s) Training Loss: 1.798, Test Loss: 1.801, Training acc: 0.66, Test acc: 0.66,
Epoch: 4 (0m 21s) Training Loss: 1.772, Test Loss: 1.777, Training acc: 0.69, Test acc: 0.68,
Epoch: 5 (0m 21s) Training Loss: 1.748, Test Loss: 1.776, Training acc: 0.71, Test acc: 0.68,
Epoch: 6 (0m 20s) Training Loss: 1.731, Test Loss: 1.773, Training acc: 0.73, Test acc: 0.69,
Epoch: 7 (0m 21s) Training Loss: 1.717, Test Loss: 1.762, Training acc: 0.74, Test acc: 0.70,
Epoch: 8 (0m 21s) Training Loss: 1.705, Test Loss: 1.743, Training acc: 0.76, Test acc: 0.72,
Epoch: 9 (0m 21s) Training Loss: 1.693, Test Loss: 1.746, Training acc: 0.77, Test acc: 0.71,
Epoch: 10 (0m 20s) Training Loss: 1.682, Test Loss: 1.740, Training acc: 0.78, Test acc: 0.72,
Training completed in 3m 46s Training Loss: 1.682, Test Loss: 1.740, Training acc: 0.78, Test acc: 0.72,

Training Time Reduced from 4 min 10 sec to 3 min 46s **[10% reduced]**



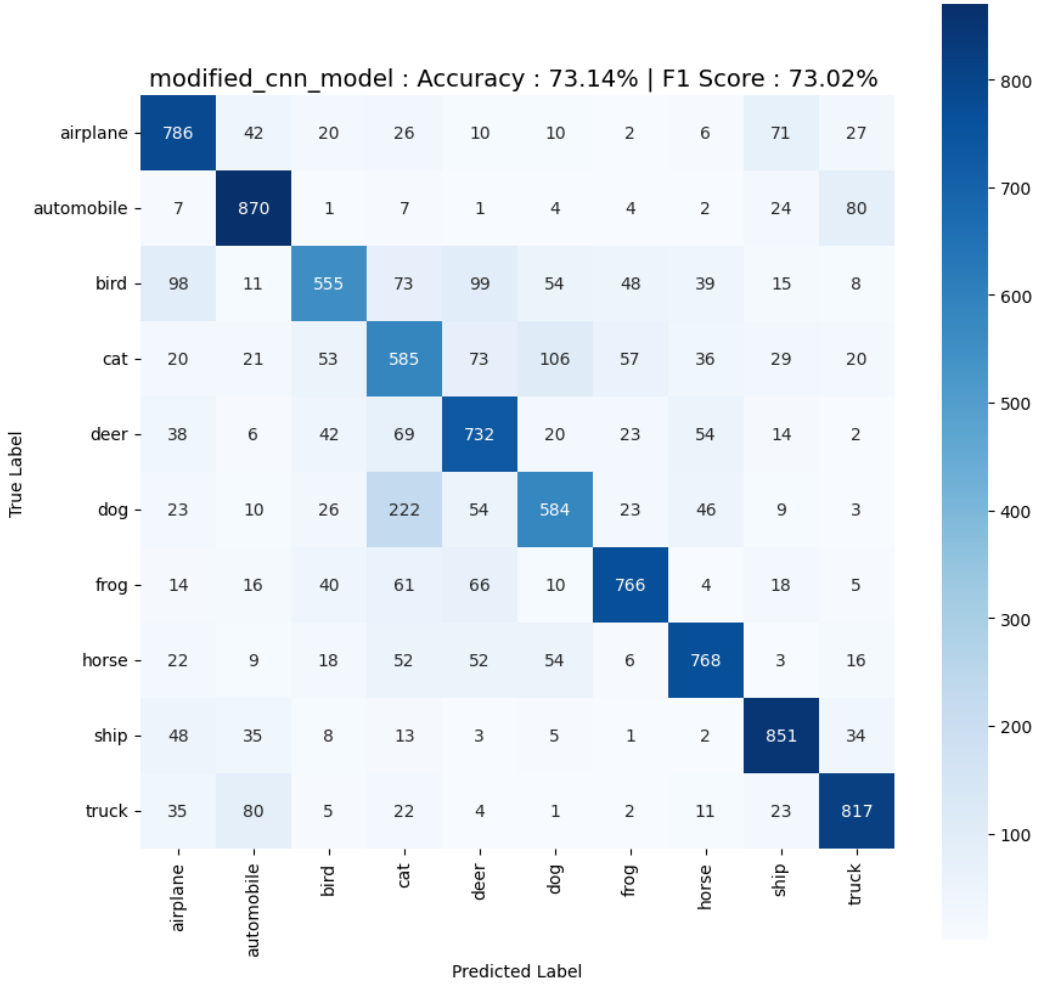


Accuracy: 73.140% [Reduced by 0.15 % only]
F1 Score: 73.02% [Reduced by 0.3 % only]

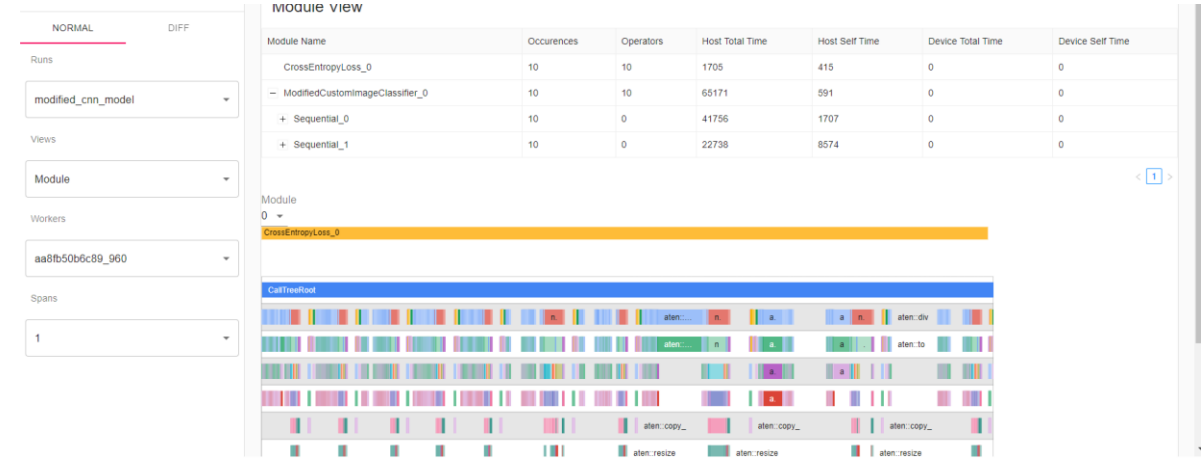
Classwise Accuracy Score:

airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0.786	0.87	0.555	0.585	0.732	0.584	0.766	0.768	0.851	0.817

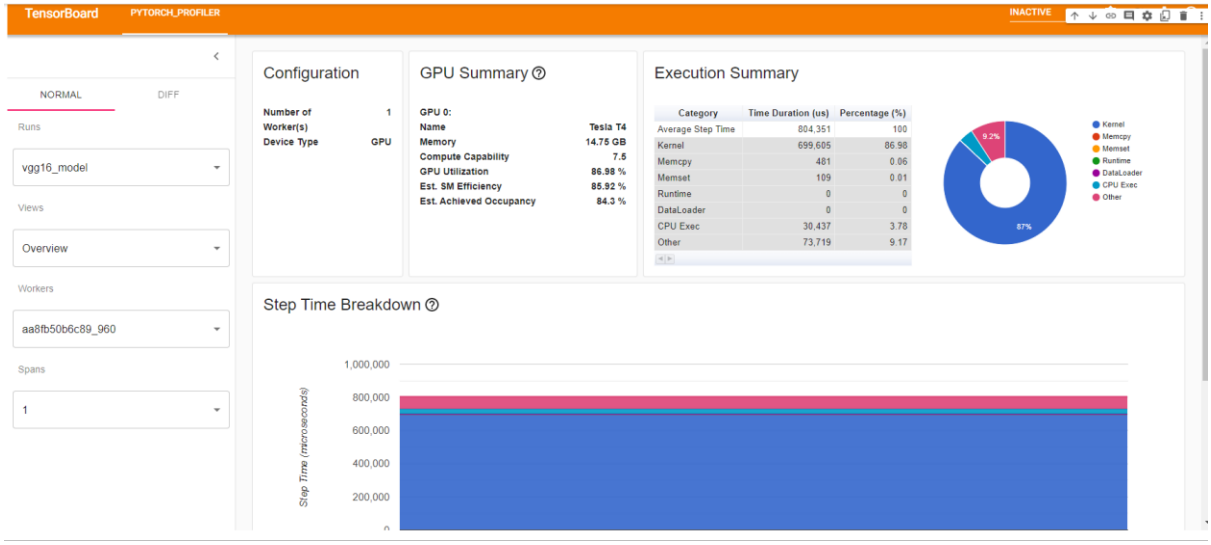
Confusion Matrix:

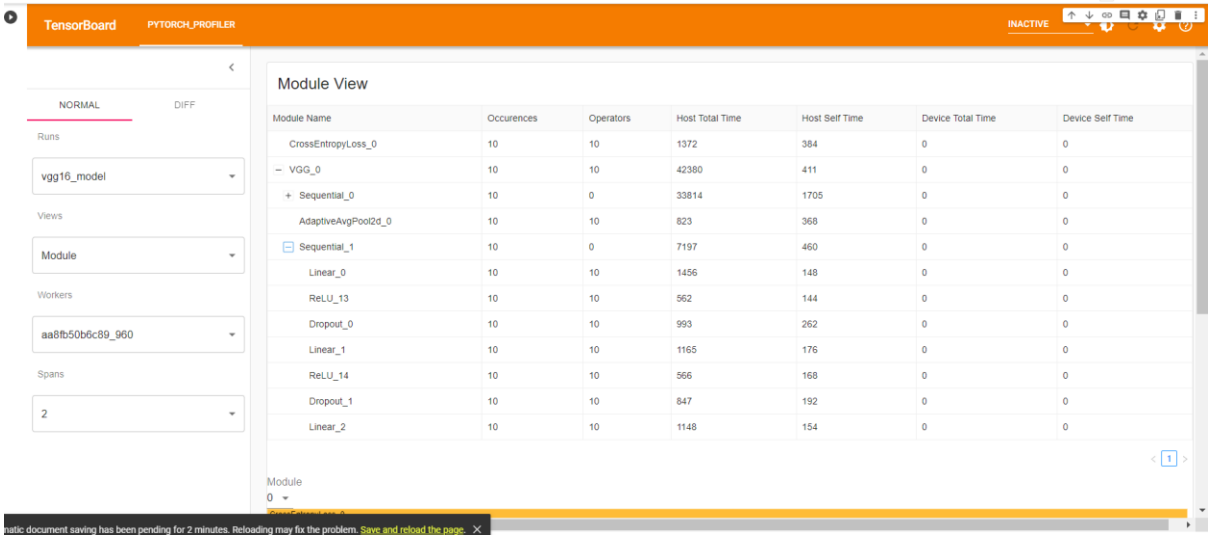


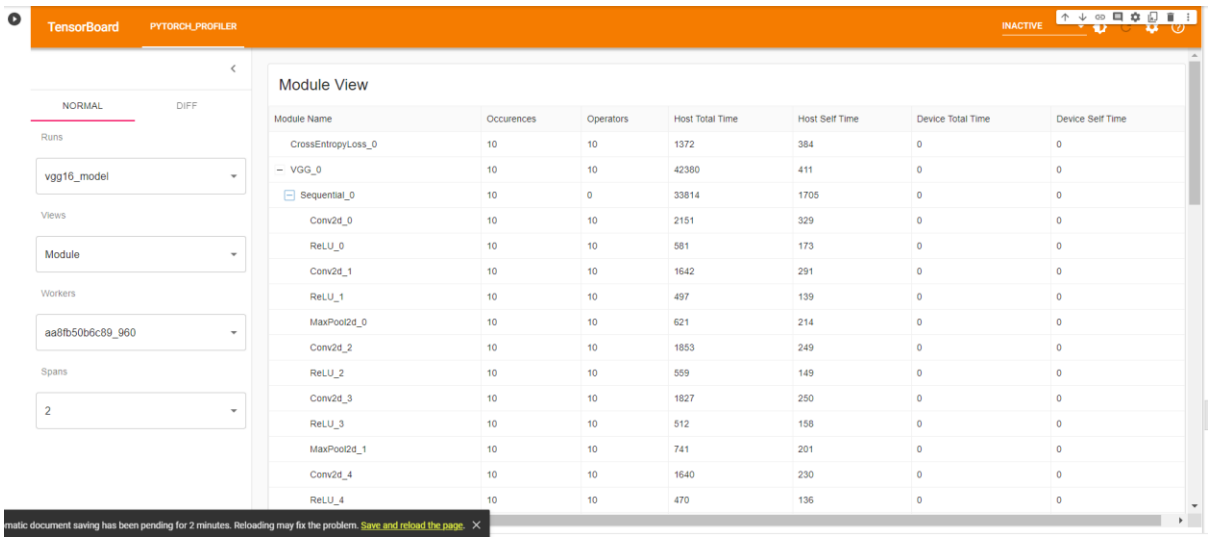
Modified Custom Model Profile:



VGG 16 Model:

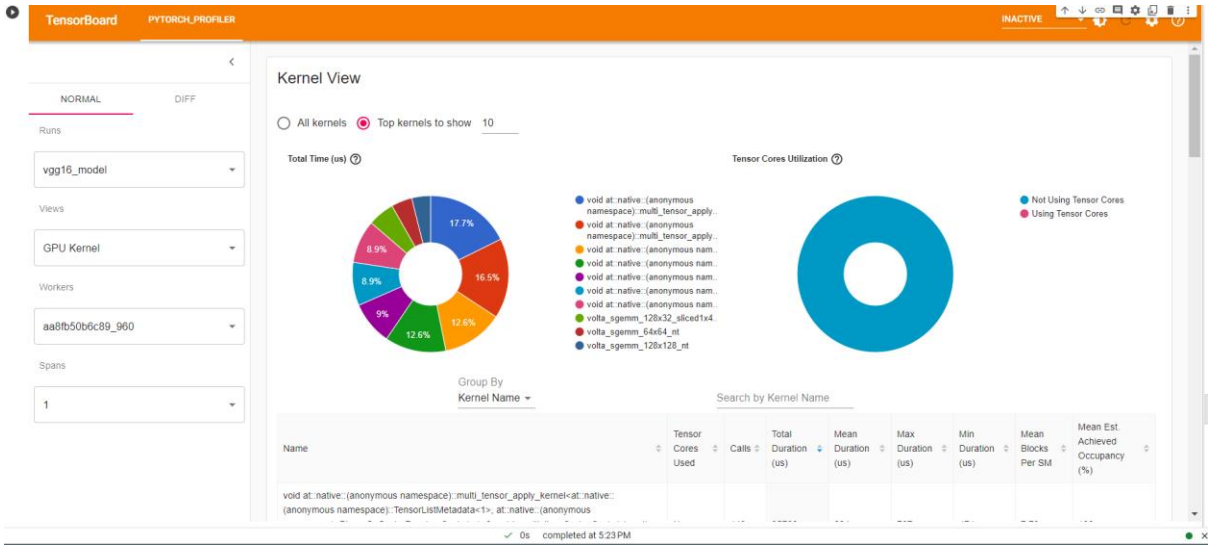

TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
vgg16_model
Views
Overview
Workers
aa8fb50b6c89_960
Spans
1
Configuration
Number of Worker(s) 1
Device Type GPU
GPU Summary
GPU 0:
Name Tesla T4
Memory 14.75 GB
Compute Capability 7.5
GPU Utilization 86.98 %
Est. SM Efficiency 85.92 %
Est. Achieved Occupancy 84.3 %
Execution Summary
Category | Time Duration (us) | Percentage (%)
Average Step Time 804,351 100
Kernel 699,605 86.98
Memcpy 481 0.06
Memset 109 0.01
Runtime 0 0
DataLoader 0 0
CPU Exec 30,437 3.78
Other 73,719 9.17
9.2%
87%
Kernel
Memcpy
Memset
Runtime
DataLoader
CPU Exec
Other
Step Time Breakdown
Step Time (microseconds)
1,000,000
800,000
600,000
400,000
200,000
0


TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
vgg16_model
Views
Module
Workers
aa8fb50b6c89_960
Spans
2
Module View
Module Name | Occurences | Operators | Host Total Time | Host Self Time | Device Total Time | Device Self Time
CrossEntropyLoss_0 10 10 1372 384 0 0
VGG_0 10 10 42380 411 0 0
Sequential_0 10 0 33814 1705 0 0
AdaptiveAvgPool2d_0 10 10 823 368 0 0
Sequential_1 10 0 7197 460 0 0
Linear_0 10 10 1456 148 0 0
ReLU_13 10 10 562 144 0 0
Dropout_0 10 10 993 262 0 0
Linear_1 10 10 1165 176 0 0
ReLU_14 10 10 566 168 0 0
Dropout_1 10 10 847 192 0 0
Linear_2 10 10 1148 154 0 0
Module
0
matic document saving has been pending for 2 minutes. Reloading may fix the problem. Save and reload the page.


TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
vgg16_model
Views
Module
Workers
aa8fb50b6c89_960
Spans
2
Module View
Module Name | Occurences | Operators | Host Total Time | Host Self Time | Device Total Time | Device Self Time
CrossEntropyLoss_0 10 10 1372 384 0 0
VGG_0 10 10 42380 411 0 0
Sequential_0 10 0 33814 1705 0 0
Conv2d_0 10 10 2151 329 0 0
ReLU_0 10 10 581 173 0 0
Conv2d_1 10 10 1642 291 0 0
ReLU_1 10 10 497 139 0 0
MaxPool2d_0 10 10 621 214 0 0
Conv2d_2 10 10 1853 249 0 0
ReLU_2 10 10 559 149 0 0
Conv2d_3 10 10 1827 250 0 0
ReLU_3 10 10 512 158 0 0
MaxPool2d_1 10 10 741 201 0 0
Conv2d_4 10 10 1640 230 0 0
ReLU_4 10 10 470 136 0 0
matic document saving has been pending for 2 minutes. Reloading may fix the problem. Save and reload the page.

Module wise Breakdown

Kernel View



Top 10 Operator

After observing above, decided to remove 2nd convolution layer from VGG16 model. Also removed some nodes from last two layers of classifier layer.

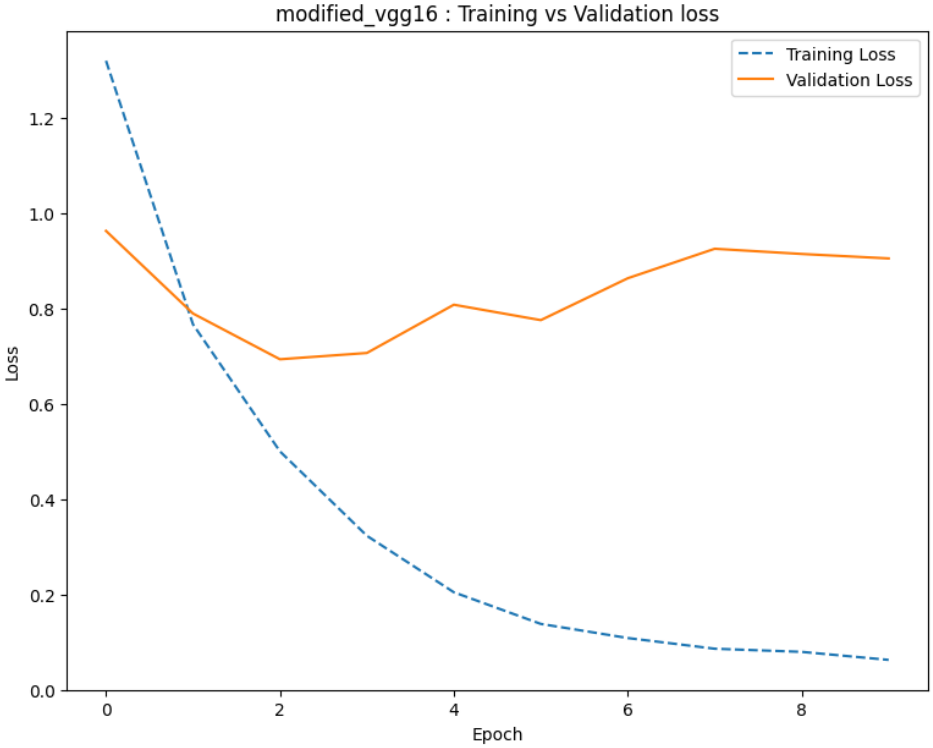
Modified VGG 16 model Architecture:

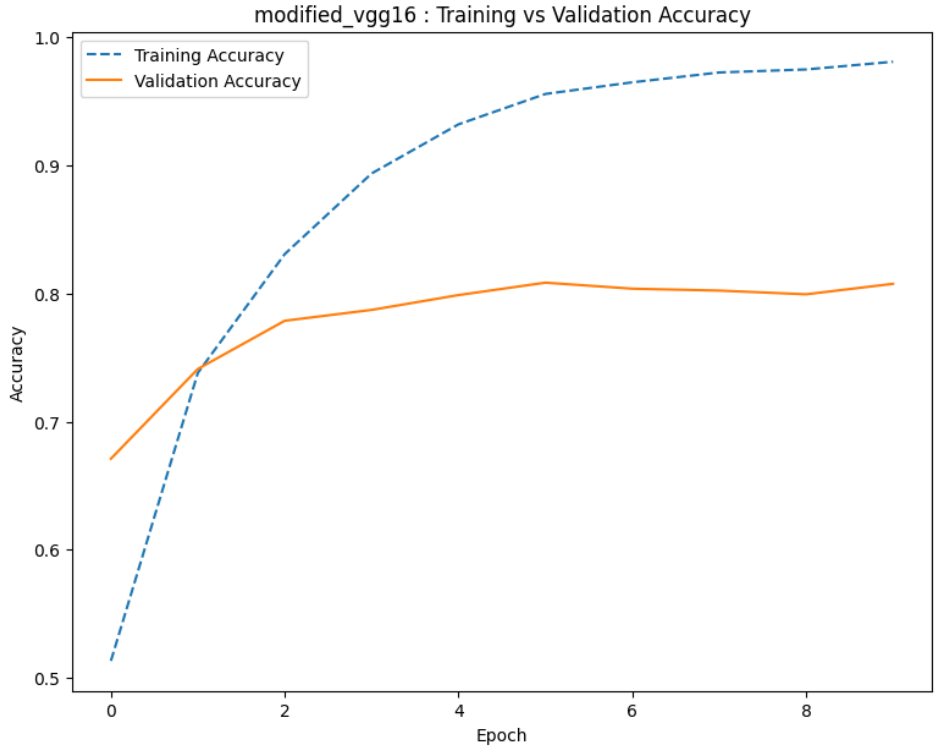
```
ModifiedVGG16Classifier(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU(inplace=True)
    (17): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): ReLU(inplace=True)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=4096, out_features=256, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.1, inplace=False)
    (6): Linear(in_features=256, out_features=10, bias=True)
  )
)
```

Training:

```
Epoch: 1 (2m 7s)  Training Loss: 1.319,  Test Loss: 0.962,  Training acc: 0.51,  Test acc: 0.67,
Epoch: 2 (1m 47s)  Training Loss: 0.766,  Test Loss: 0.789,  Training acc: 0.74,  Test acc: 0.74,
Epoch: 3 (1m 47s)  Training Loss: 0.500,  Test Loss: 0.693,  Training acc: 0.83,  Test acc: 0.78,
Epoch: 4 (1m 47s)  Training Loss: 0.323,  Test Loss: 0.706,  Training acc: 0.89,  Test acc: 0.79,
Epoch: 5 (1m 47s)  Training Loss: 0.204,  Test Loss: 0.808,  Training acc: 0.93,  Test acc: 0.80,
Epoch: 6 (1m 47s)  Training Loss: 0.138,  Test Loss: 0.775,  Training acc: 0.96,  Test acc: 0.81,
Epoch: 7 (1m 47s)  Training Loss: 0.108,  Test Loss: 0.863,  Training acc: 0.96,  Test acc: 0.80,
Epoch: 8 (1m 46s)  Training Loss: 0.086,  Test Loss: 0.925,  Training acc: 0.97,  Test acc: 0.80,
Epoch: 9 (1m 46s)  Training Loss: 0.079,  Test Loss: 0.914,  Training acc: 0.97,  Test acc: 0.80,
Epoch: 10 (1m 46s) Training Loss: 0.063,  Test Loss: 0.905,  Training acc: 0.98,  Test acc: 0.81,
Training completed in 18m 13s  Training Loss: 0.063,  Test Loss: 0.905,  Training acc: 0.98,  Test acc: 0.81,
```

Training Time Reduced from 20 min 39 sec to 18 min 13s [**14% reduced**]



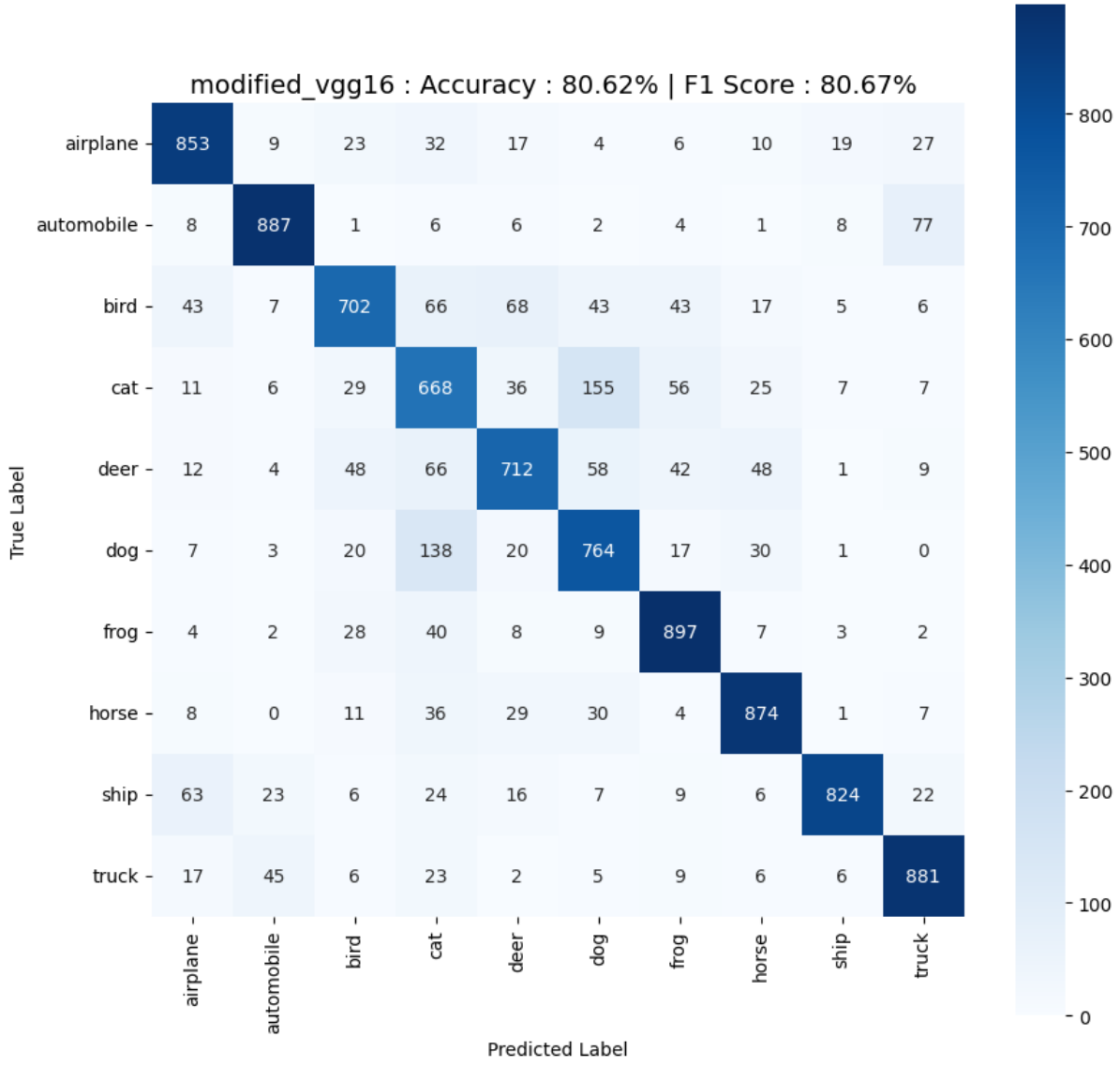


Accuracy: 80.620% [Reduced by 9 %]
F1 Score: 80.674% [Reduced by 9 %]

Classwise Accuracy Score:

airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0.853	0.887	0.702	0.668	0.712	0.764	0.897	0.874	0.824	0.881

Confusion Matrix:



Conclusion:

In case of Custom CNN Model, after profiling model pruning was successful, as after pruning training time of the model has been reduced by 10% without any significant loss (0.15%) in accuracy. Whereas in case of VGG16, model pruning has reduced training time by 14% but model performance also reduced by 9%.

References:

1. Lectures given by Respected Professors, Dr Anush Shankaran
2. Help sessions given by TAs
3. Papers:
    a. "Attention is all you need" (Vaswani, Ashish & Shazeer, Noam & Parmar, Niki & Uszkoreit, Jakob & Jones, Llion & Gomez, Aidan & Kaiser, Lukasz & Polosukhin, Illia. (2017))
    b. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", by Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby
4. Blogs:
    a. https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c
    b. https://theaisummer.com/positional-embeddings/
    c. https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634
    d. https://www.pinecone.io/learn/vision-transformers/
    e. And many other