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.

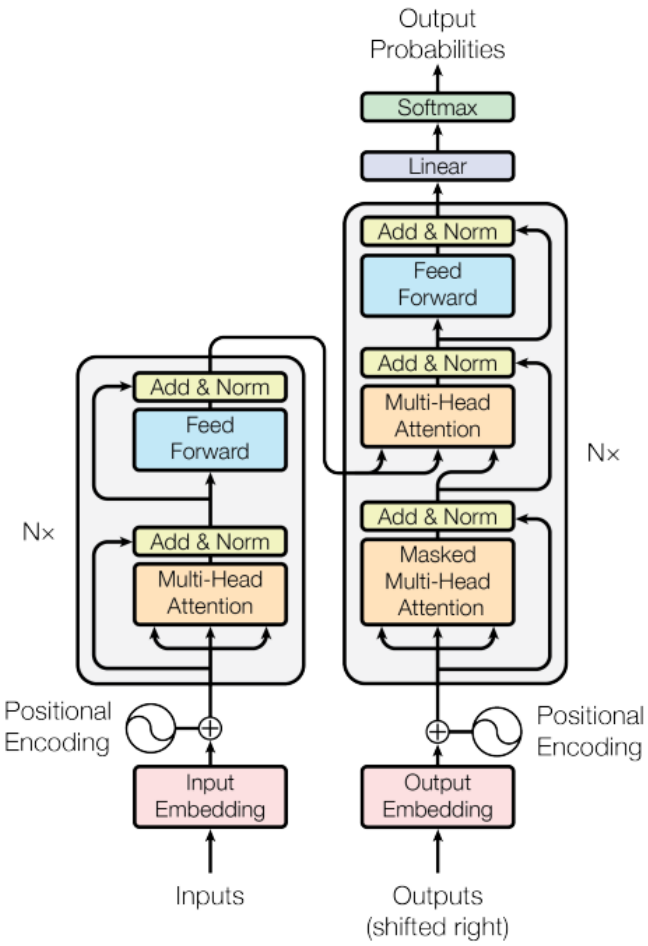


Figure 1: The Transformer - model architecture.

Source: Paper - Attention Is All You Need -2017

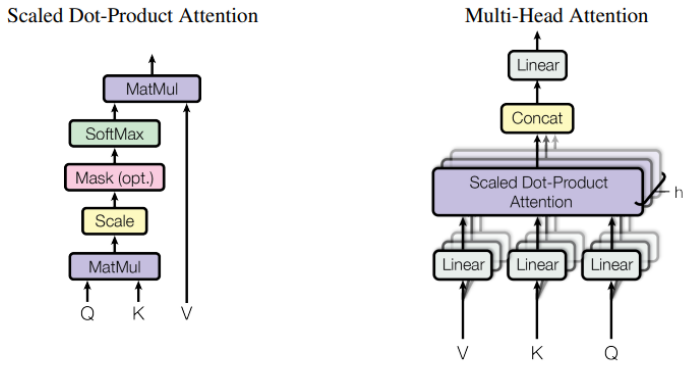
The Transformer architecture consists of an encoder and a decoder, each composed of multiple layers of self-attention and feedforward neural networks. The encoder takes the input sequence and generates a series of hidden representations that capture the meaning of each word in the context of the entire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

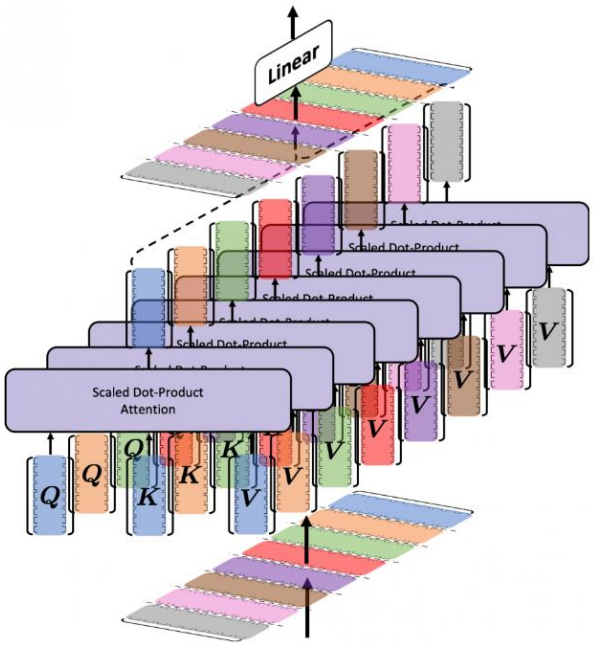
Multi Head Self Attention

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.

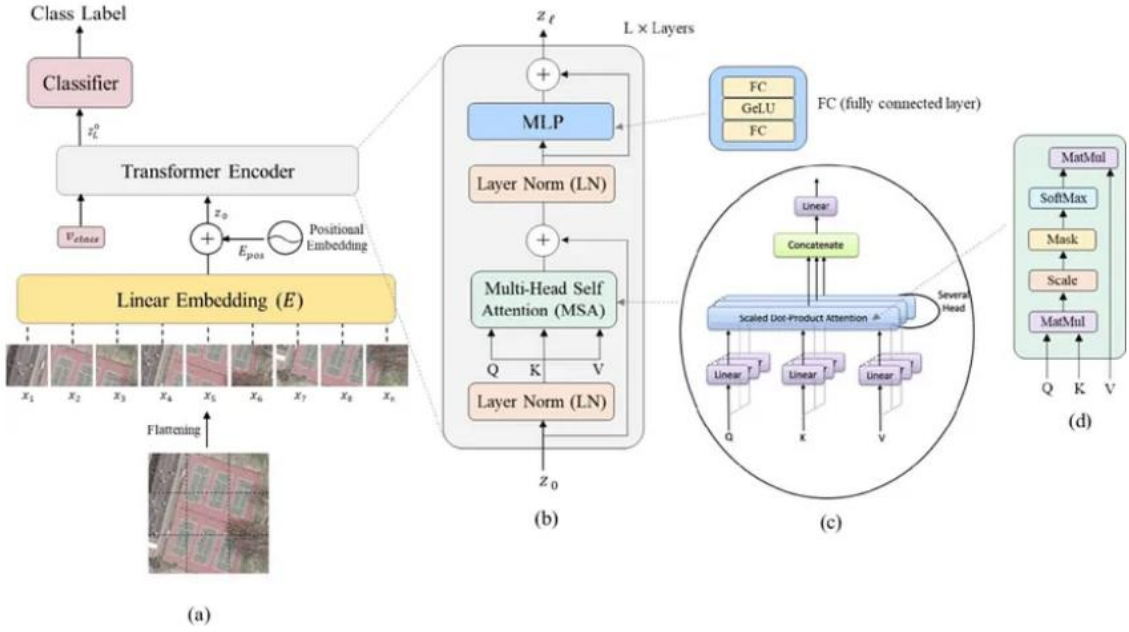


The use of multiple heads allows the model to capture different relationships between elements in the sequence, enhancing its ability to represent complex dependencies. Additionally, the model can learn to attend to different parts of the input sequence simultaneously, enabling it to capture both local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

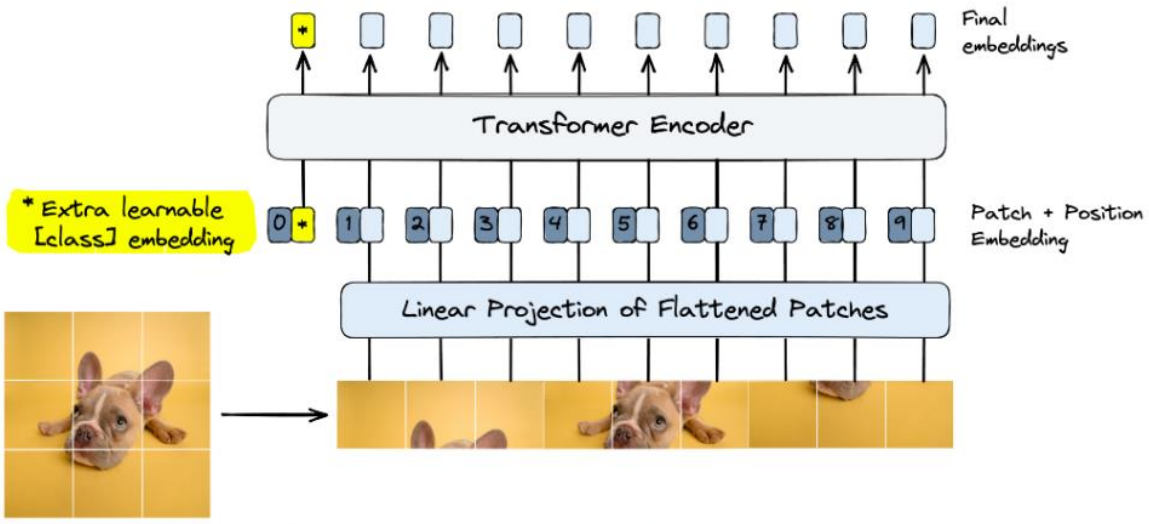
The Vision Transformer was introduced in a paper titled "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" by Dosovitskiy et al. in 2020. The paper proposed a new architecture for image classification tasks that replaces the convolutional layers of a traditional convolutional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

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.

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(
```

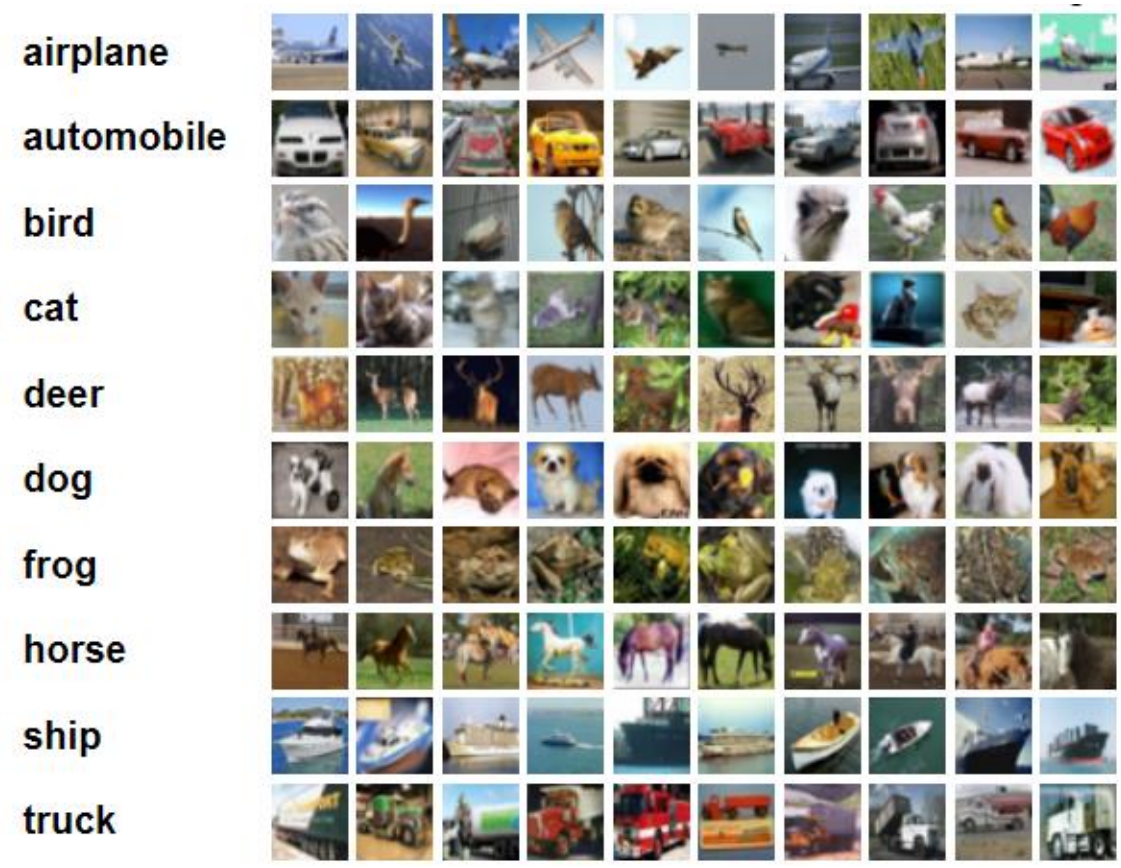
```
        (0): Linear(in_features=1, out_features=1, bias=True)
        (1): Linear(in_features=1, out_features=1, bias=True)
        (2): Linear(in_features=1, out_features=1, bias=True)
        (3): Linear(in_features=1, out_features=1, bias=True)
        (4): Linear(in_features=1, out_features=1, 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**.



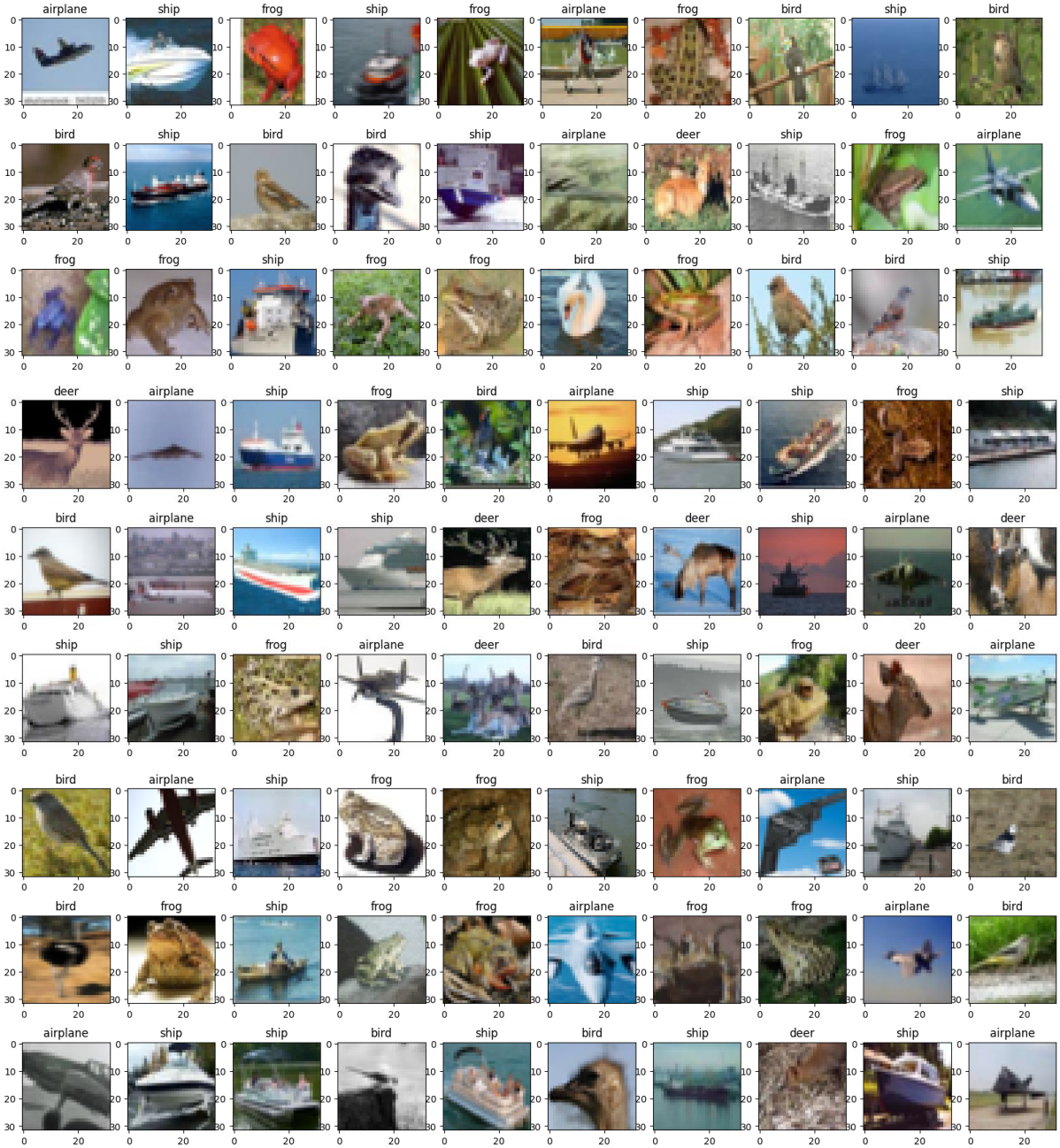
Torch Transformations Applied:

- 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

Final Results:

model	Positional Embed	Activation Fn	Training Time	Training Accuracy	Test Accuracy
ViT_ReLU_Cosine_6B8H8D	Cosine PE	ReLU	8780.814	0.342	0.345
ViT_Tanh_Cosine_6B8H8D	Cosine PE	Tanh	8732.586	0.345	0.346
ViT_GELU_Cosine_6B8H8D	Cosine PE	GELU	8782.595	0.345	0.352
ViT_ReLU_Learnable_4B6H12D	Learnable PE	ReLU	4594.395	0.405	0.408
ViT_Tanh_Learnable_4B6H12D	Learnable PE	Tanh	4581.365	0.357	0.358
ViT_GELU_Learnable_4B6H12D	Learnable PE	GELU	4629.802	0.343	0.352

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 results, it appears that the ViT models with learnable positional embeddings and ReLU or GELU activation functions achieved the highest test accuracy. The ViT model with learnable positional embeddings and ReLU activation function achieved the highest test accuracy of 0.408, while the ViT model with learnable positional embeddings and GELU activation function achieved a test accuracy of 0.352.

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 model size, training data, or hardware used for training.