DL-Ops Assignment-3 Report

TASK 1:

Objectives:

- Perform image classification on even-numbered classes (odd roll number) of CIFAR-10 dataset using a transformer model with following variations:
    - Use cosine positional embedding with six encoders and decoder layers with eight heads. Use relu activation in the intermediate layers.
    - Use learnable positional encoding with four encoder and decoder layers with six heads. Use relu activation in the intermediate layers.
    - Change the activation function to tanh in both cases and compare the performance.

Procedure:

- Install and import required packages.
- Check for GPU, and set batch size accordingly.
- Load the CIFAR-10 dataset, transform to Tensor and use normalization on the images.
- Enumerate the dataset and save the even-numbered indices in a list. Create Subsets with the indices list using Subset class from torch.utils.data
- Define train, test, accuracy function and curves plotting functions.
- To feed the image to the transformer, make patches of the image. Then flatten these patches. The patches are created using a convolution layer with kernel size and stride equal to patch size. The out channels of the convolution layer are equal to the embedding dimension.
- Define class for Cosine Positional Embedding to add the position information to each part of the sequence. It create a position vector that has length equal to number of patches and second dimension equal to the embedding size. The cosine function is used to encode even positions, while the sine function is used to encode odd positions.
- Define class for Learnable Positional Embedding to add the position information to each part of the sequence and the model learns the sequence. It create a position vector that has length equal to number of patches and second dimension equal to the embedding size which is randomly initialized and changes on every backpropagation of the network. The vector is created using nn.Parameter which enables it to learn. This allows the model to adapt the positional encoding to the specific characteristics of the data, and potentially capture more complex patterns in the positional information.
- Define a class for Multi-Layer Perceptron that uses GELU (Gaussian Error Linear Unit Function) sandwiched between two linear layers and a dropout layer after each of the linear layers. This MLP shall be used in the transformer block.
- Now define the class for multi-headed self-attention block (Attention class). It maps a query with a set of key and value pairs and output are all vectors. Reference.

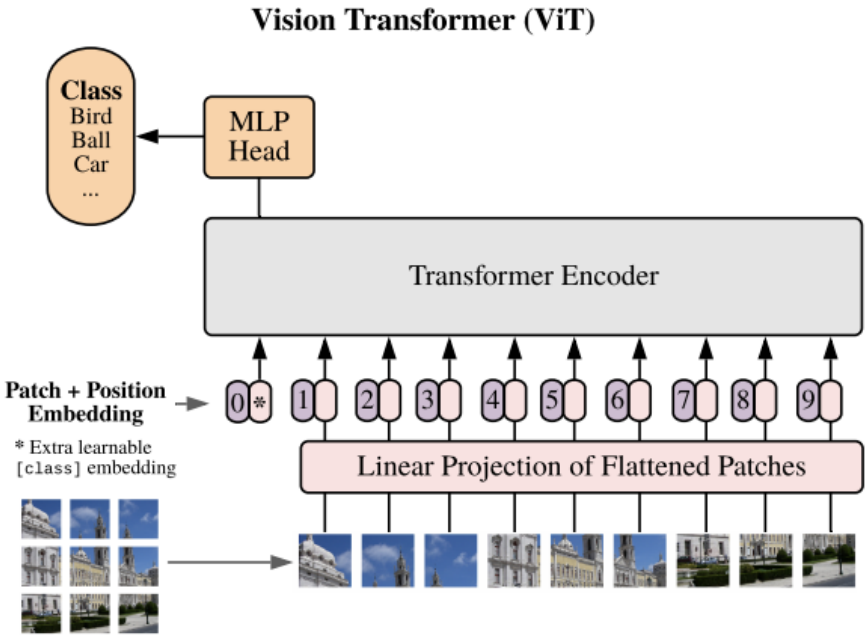
- Define the class for the Transformer block. The transformer block receives the embedded sequence and applies layer normalization. This is then fed to the attention block which divides it into queries, keys and value pairs. The output is then merged and passed to another layer normalization layer. The output of the layer norm is forwarded to the Multi-layered perceptron. All this happens in the forward function of the transformer block.
- Now create a Vision Transformer class (ViT) that packs all the above classes into one. An input batch of images is received which is converted into a sequence of flattened patches of image and then each sequence is embedded with positional embedding using either cosine positional embedding or learnable positional embedding. It is fed to a sequence of n number of transformer blocks and then to another layer norm layer. The output is then forwarded to a linear layer for classification.

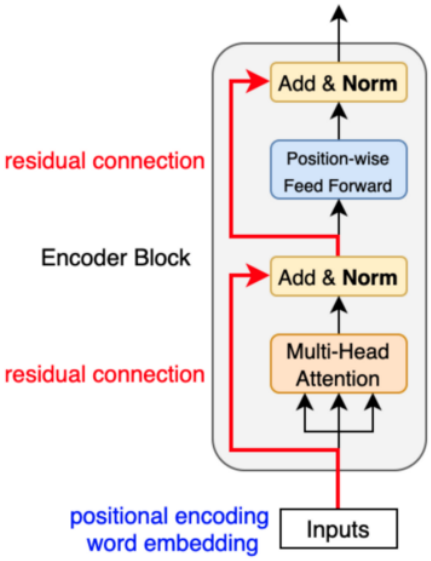
Results and Observations:

Hyperparameters (Common for all the models):

Batch size: 256
Optimizer: Adam
Learning Rate: 0.001
Epochs: 40
Criterion: Cross-Entropy Loss
Dropout on attention layer: 0.2
MLP ratio: 4
Embedding dimension: 768
Patch size: 16x16
Number of patches for an image: 4

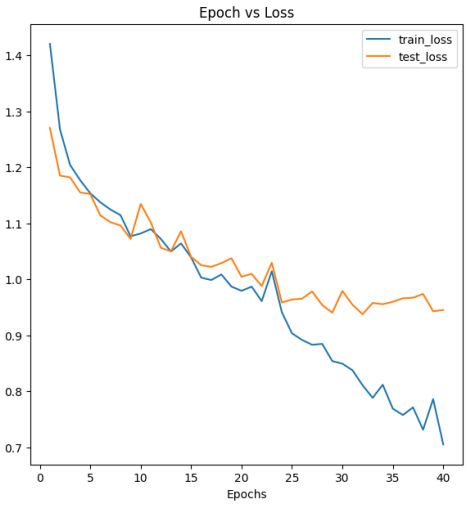
Model Architecture:

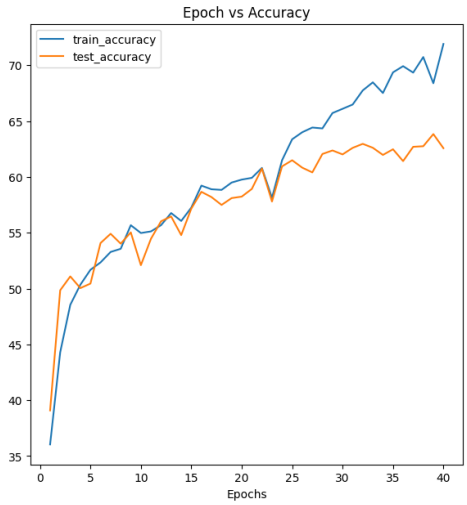




Training ViT with Cosine Positional Embedding, attention heads = 8, encoder blocks = 6, activation ReLU.

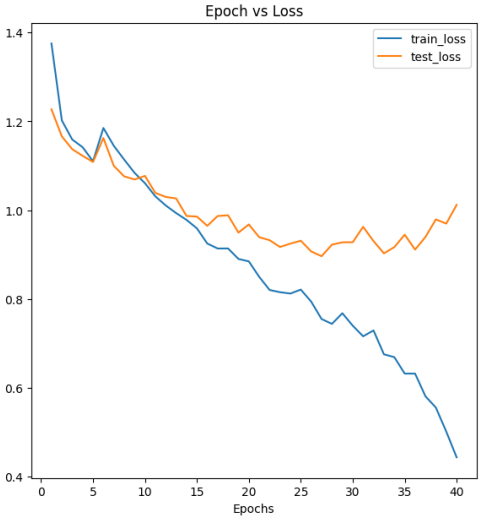
Epochs: 40 | Train Loss: 0.71 | Train Accuracy: 71.91 | Test Loss: 0.95 | Test Accuracy: 62.57

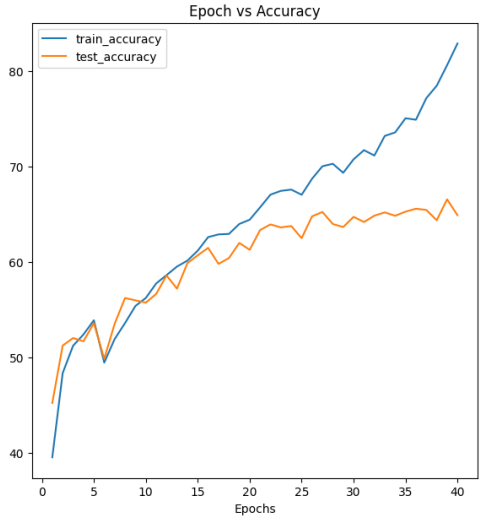




Training ViT with Learnable Positional Embedding, attention heads = 8, encoder blocks = 6, activation ReLU.

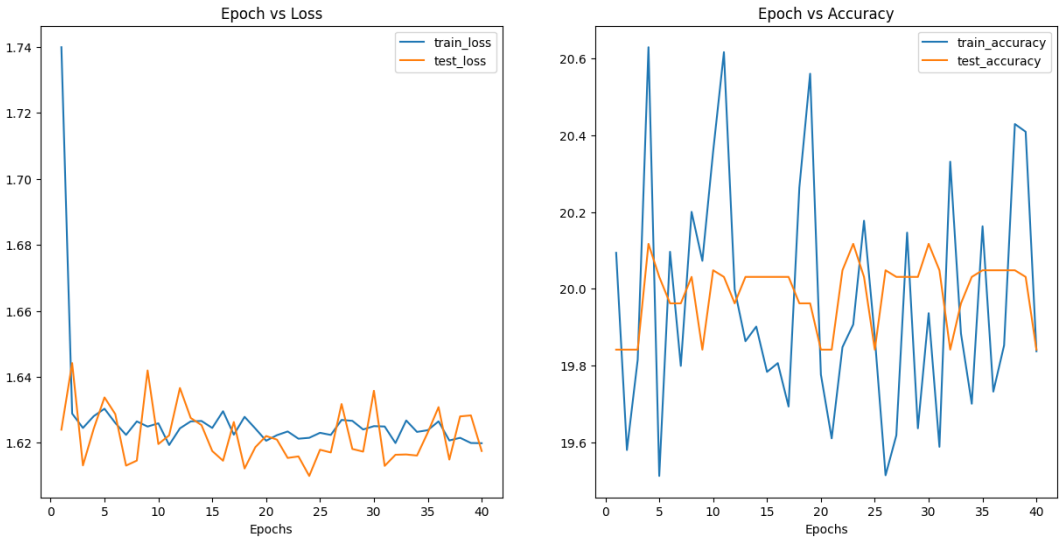
Epochs: 40 | Train Loss: 0.44 | Train Accuracy: 82.88 | Test Loss: 1.01 | Test Accuracy: 64.89





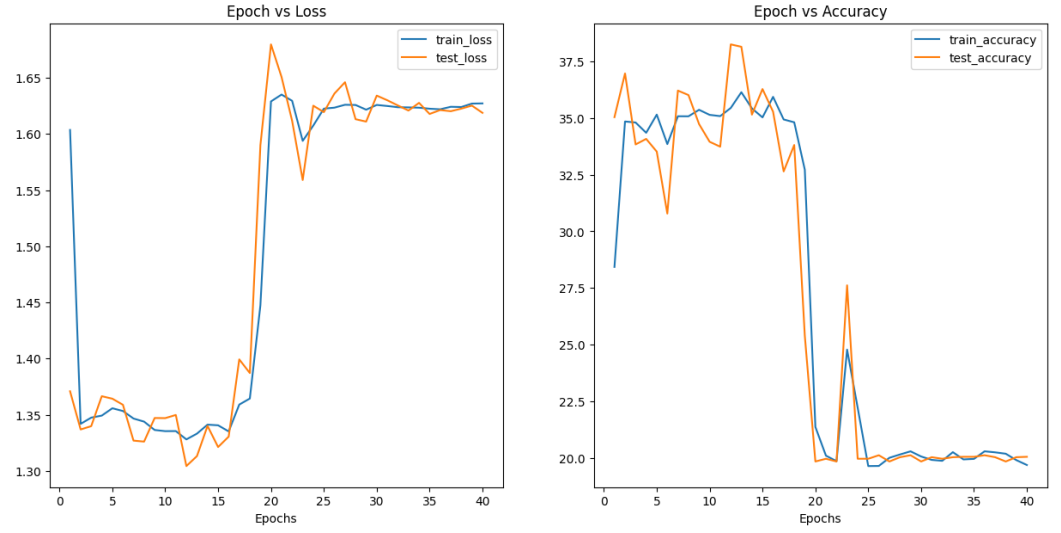
Training ViT with Cosine Positional Embedding, attention heads = 8, encoder blocks = 6, activation Tanh.

Epochs: 40 | Train Loss: 1.62 | Train Accuracy: 19.84 | Test Loss: 1.62 | Test Accuracy: 19.84



Training ViT with Learnable Positional Embedding, attention heads = 8, encoder blocks = 6, activation Tanh

Epochs: 40 | Train Loss: 1.63 | Train Accuracy: 19.69 | Test Loss: 1.62 | Test Accuracy: 20.05



Observations:

1. Model with learnable positional embedding and relu activation converges faster than the model with cosine position encoding and relu activation.
2. Both the models above slightly overfits the training data. This could be due to reduced training data size and the model complexity. We can help reduce the overfitting by increasing the dropout value for regularization or increase the number of attention heads. Increasing the number of attention heads can help improve the model's generalization by allowing it to capture more complex interactions between the input

tokens. We can also try setting the hyperparameter like patch size to a lower value. But doing this may increase the training time.

3. The training time of the model is much lower than the models like resnet.
4. It can be observed that vision transformer takes higher epochs to converge than traditional CNN models like resnet18. Although training the ViT for longer and using the best combination of hyperparameters may make outperform resnet model.
5. The models with Tanh activation underfit the training data. Tanh is a function that maps the input to the range [-1, 1]. It is generally more computationally expensive and can suffer from vanishing gradient problems, which may be the case here as it can be observed in case of model with learnable position embeddings where the accuracy in the initial stage increases but suddenly drops down.

TASK 2:

Objectives:

- Load and preprocess CIFAR10 dataset using standard augmentation and normalization techniques
- Train the following models for profiling them using during the training step
    - 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.

Procedure:

- Install and import required packages.
- Check for GPU availability.
- Import the dataset. Apply standard data augmentation and normalization techniques.
- Define the model class as given in the question.
- Profile this model during the training step.
- Import the vgg16 model and profile it during the training step.
- Make observations on the profile of the models.

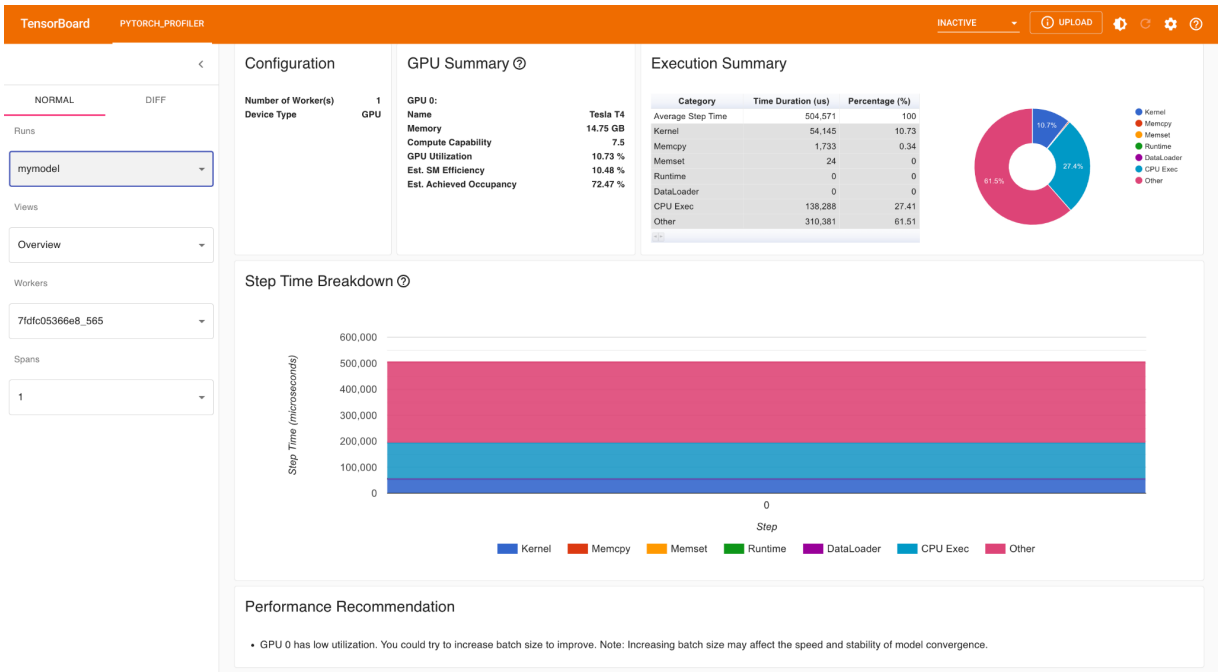
Results and Observations:

The models were profiled for 10 steps in a single epoch.

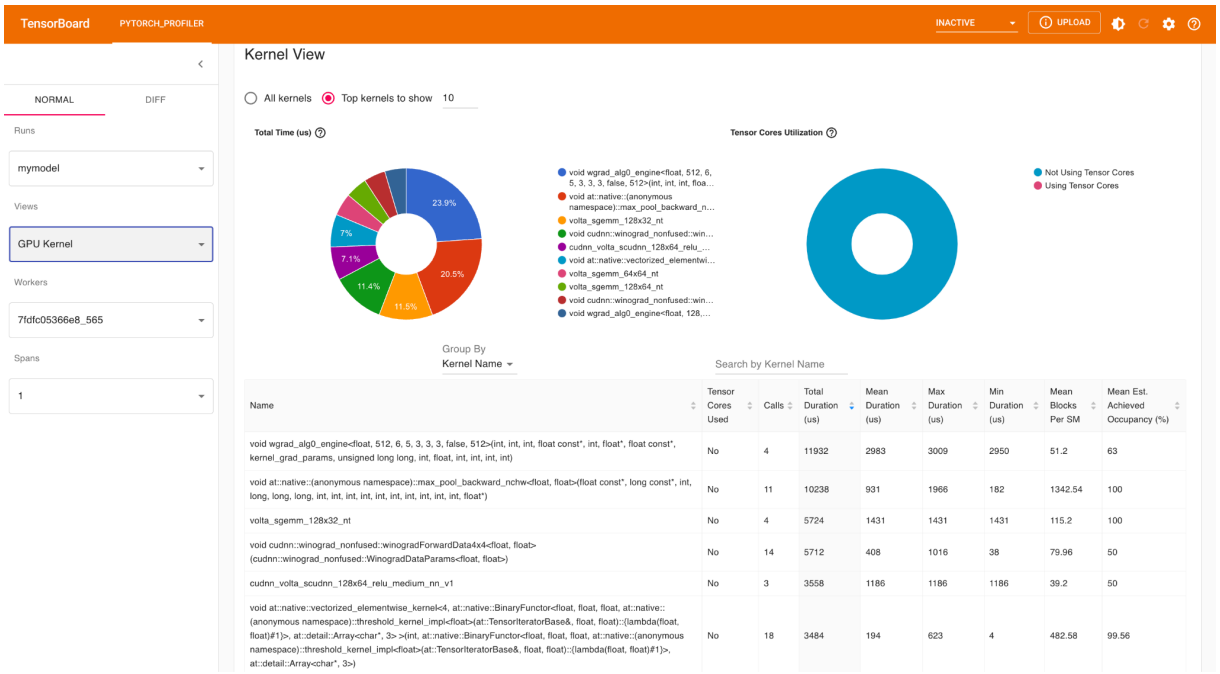
Below is the summary of the model profiling.

My custom model:

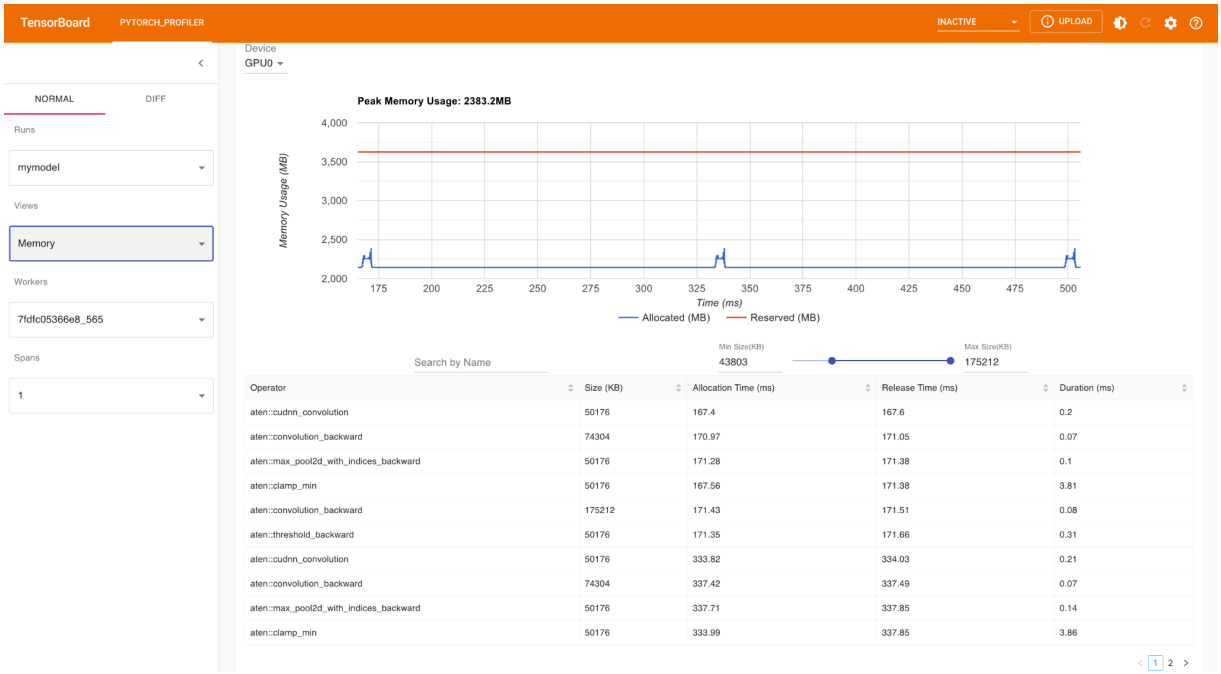
Overview



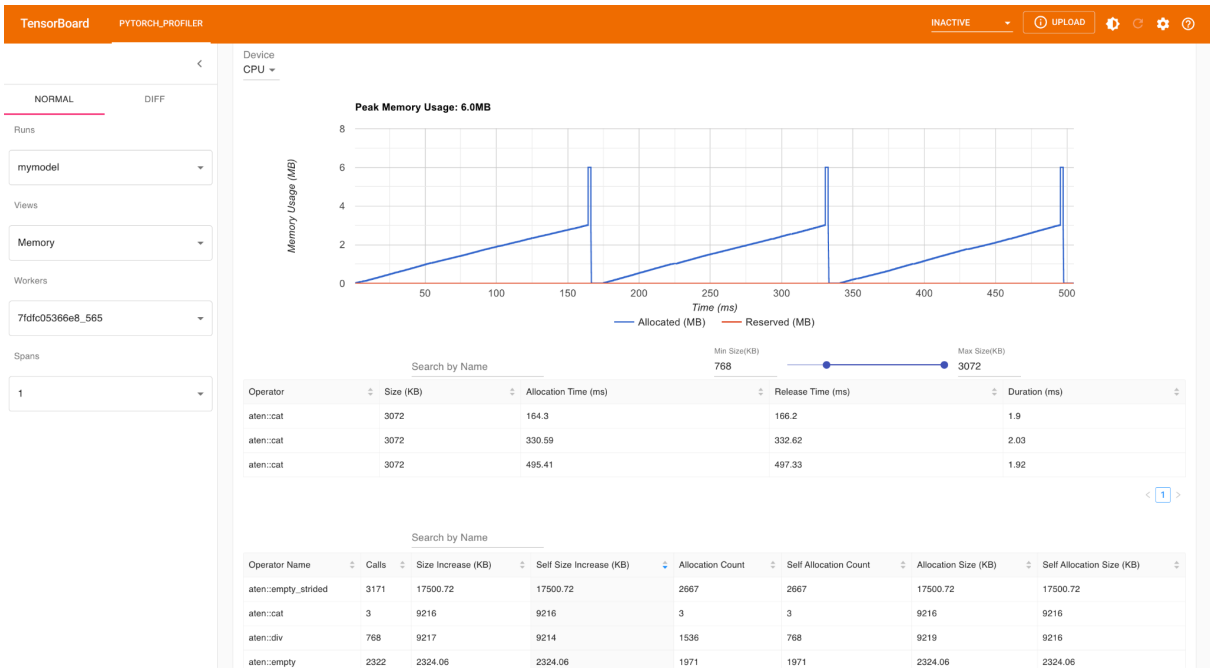
GPU Kernel Utilization



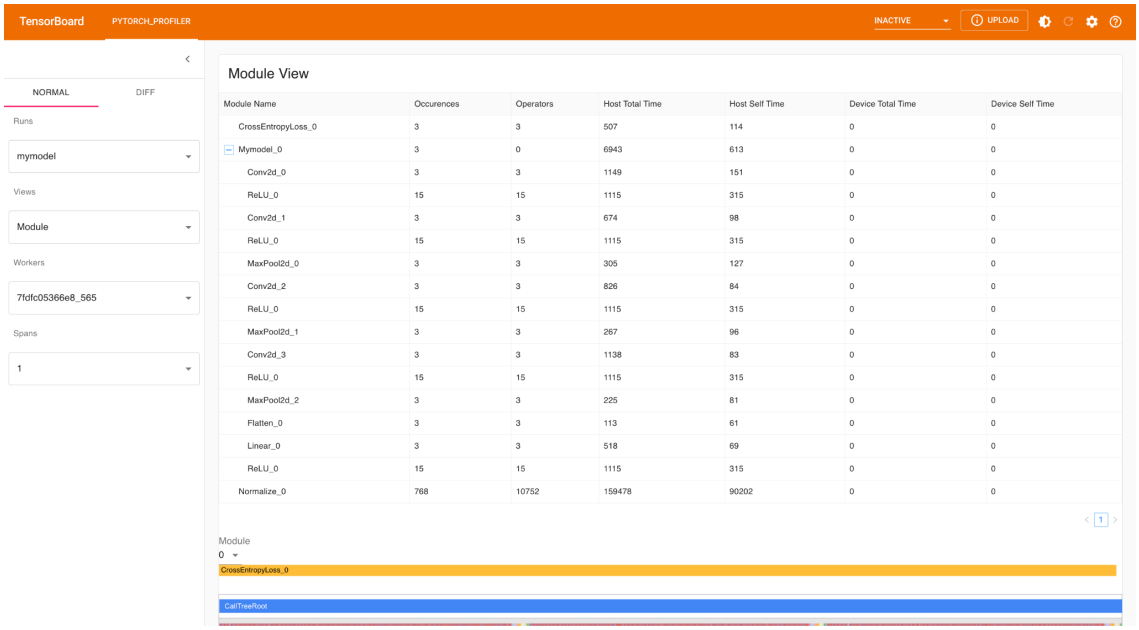
Memory - GPU



Memory - CPU



Module view



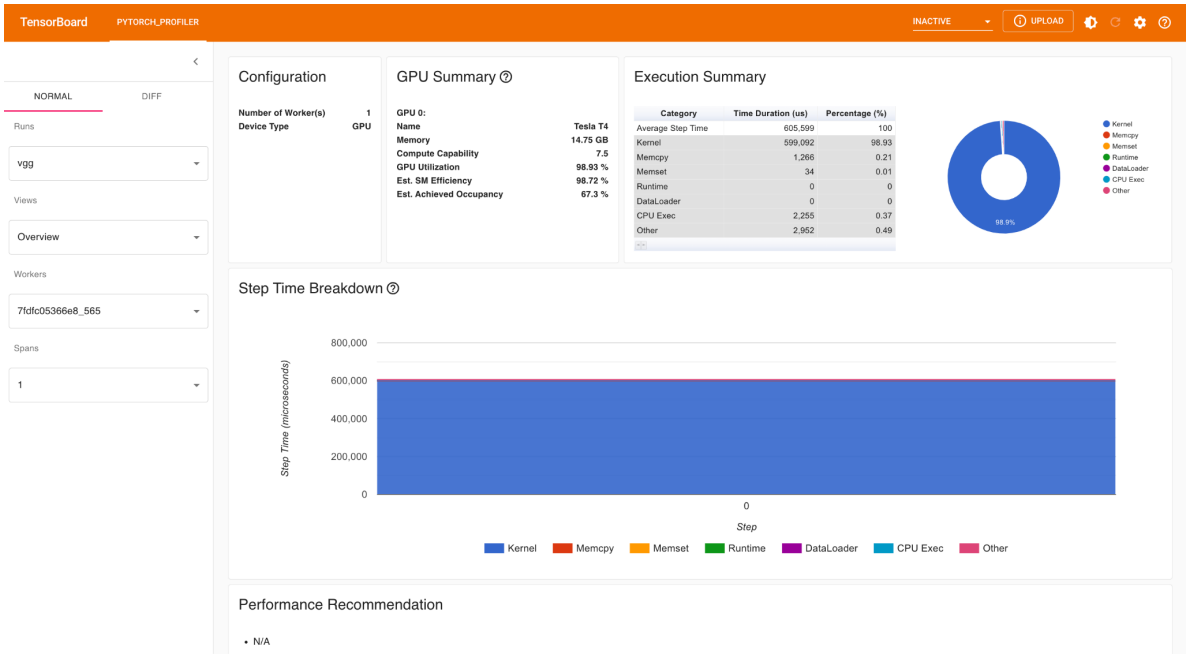
- GPU utilization is very low (10.73%) for the model.
- About 27% of the average step time is spent on CPU execution.
- Normalize_0 module takes a very high amount of time as compared to the rest.
- CPU memory utilization is low because all the tensors are preloaded on the GPU memory.

The minimum change in the architecture that would lead to a gain in performance and a decrease in training time:

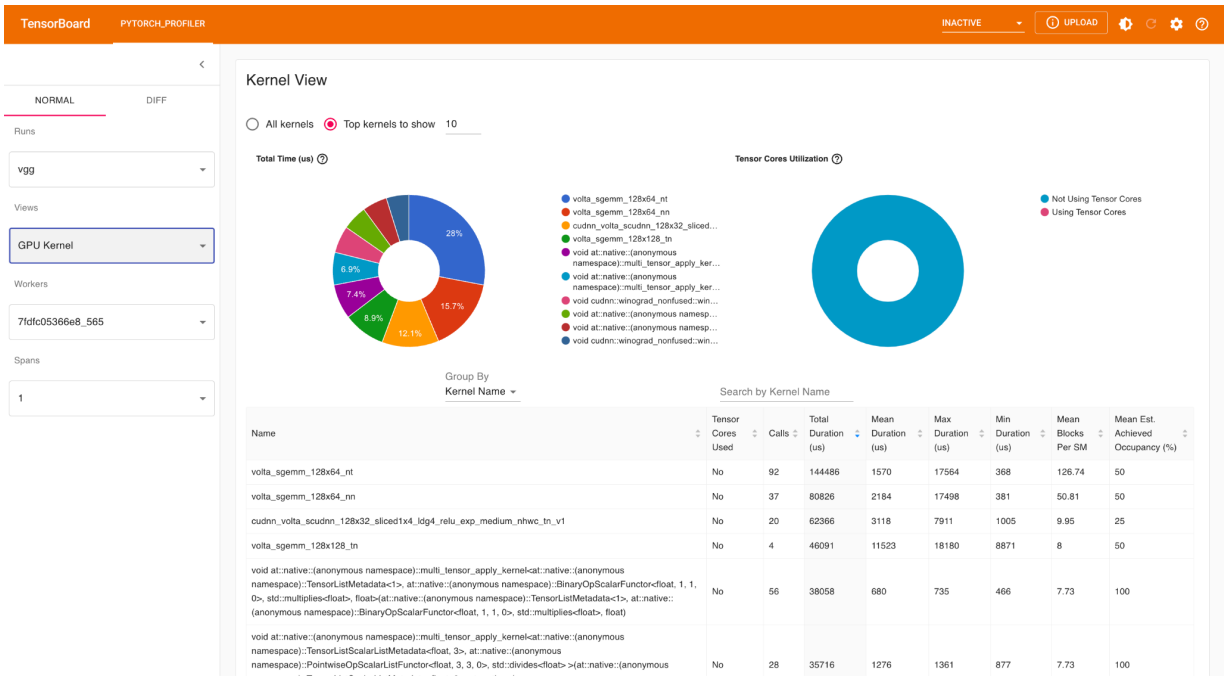
- We can remove the second max-pooling layer and set the kernel size of the final maxpool layer such that the number of parameters remains the same. Maxpool is a non-learnable computation. It keeps the important features while removing the ones that are not important. Using too many maxpool layers can reduce the amount of information available to the subsequent layers and limit the model's capacity to learn complex features. Removing a maxpool layer may make the model converge faster, in lesser number of epochs.

VGG-16 model:

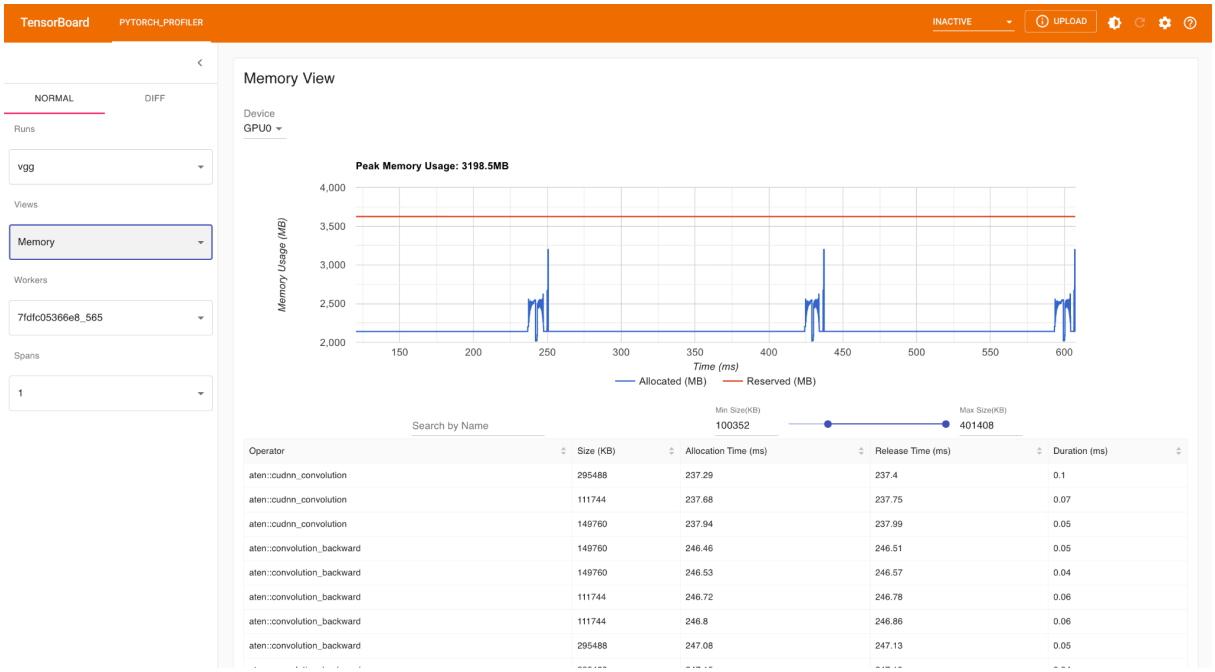
Overview



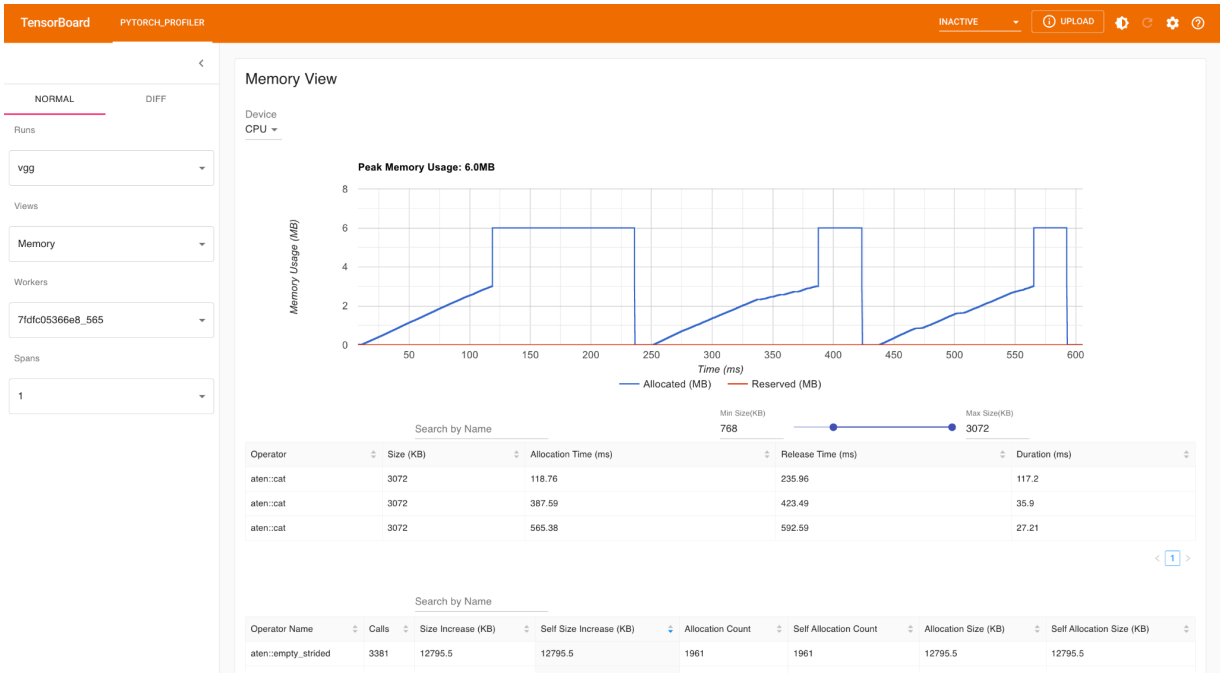
GPU Kernel



Memory GPU



Memory CPU

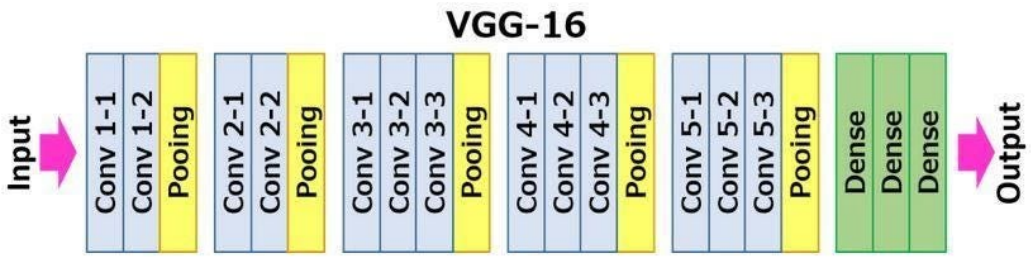


Module View

- GPU utilization is very good (98.93%) for the model.
- Only 0.37% of the average step time is spent on CPU execution.
- Normalize_0 module takes a very high amount of time as compared to the rest.
- CPU memory utilization is low because all the tensors are preloaded on the GPU memory.

The minimum change in the architecture that would lead to a gain in performance and a decrease in training time:

- Vgg is already a well-optimized model.



- We can further improve the performance and reduce training time by introducing skip connections or residual connections. This will help to improve the gradient flow and make it converge in lesser number of epochs.

References:

PyTorch 2.0 documentation
Vision Transformers from Scratch (PyTorch): A step-by-step guide | by Brian Pulfer | MLearning.ai | Medium
How Positional Embeddings work in Self-Attention (code in Pytorch) | AI Summer

[1706.03762] Attention Is All You Need
[1607.06450] Layer Normalization
Optimize PyTorch Performance for Speed and Memory Efficiency (2022) | by Jack Chih-Hsu Lin | Towards Data Science
Vision Transformer for Image Classification
Vision Transformer in PyTorch
Pytorch Transformers from Scratch (Attention is all you need)
Vision_transformer
Chat GPT