

Deep Learning Report

DLOps Assignment -

3

Ayush Abrol
B20AI052

Question 01

Aim:

Perform image classification on selected classes of CIFAR-10 dataset using 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.
- For parts (a) and (b) change the activation function in the intermediate layer from Relu to Tanh and compare the performance.

My Roll Number: B20AI052 (even) - Selecting even classes

Reference Blog used:

<https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c>

Procedure:

- All the necessary imports such as numpy, pandas, torch, torchvision, tqdm, warnings, etc. were done.
- Added transformations such as Random Horizontal Flip, RandomAffine, converted the images to Tensor and normalized the images.
- Imported the CIFAR10 dataset from torchvision.datasets with batch_size = 32

```
Train data shape: (50000, 32, 32, 3)
Test data shape: (10000, 32, 32, 3)
Train data labels shape: (50000,)
Test data labels shape: (10000,)
```

- classes = {0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer', 5: 'dog', 6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'}
- Then, I selected all the even classes.
- Then, my data looked as:

```
Train data shape: (25000, 32, 32, 3)
Test data shape: (5000, 32, 32, 3)
Train data labels shape: (25000,)
Test data labels shape: (5000,)
```

- Visualization of selected classes:



- Created train and test loaders with shape of the data compatible with PyTorch module.
- Then, I set up my device to GPU.
- After that, I started creating the Transformer architecture with cosine positional embeddings with 6 encoders and decoder layers with 8 heads with ReLU activation function in the intermediate layers.
- Created a patchify function that takes in a batch of images and the desired number of patches to extract from each image. The function returns a tensor containing the patches for each image in the batch.
- Here is a brief overview of how the function works:
 - The function first checks if the input images are square. If they are not, it raises an assertion error.
 - It initializes a tensor to store the patches for each image in the batch.

- For each image in the batch, the function loops over each patch and extracts it from the image.
 - The patch is flattened and stored in the patches tensor.
 - Once all patches have been extracted for all images in the batch, the function returns the patches tensor.
- Then, I built a python function that generates the positional embeddings for a given sequence length and embedding dimension. Positional embeddings are used in transformer-based neural network architectures to encode the position of each element in the input sequence.
- Here is a brief overview of how the function works:
 - The function initializes a tensor to store the positional embeddings.
 - For each element in the sequence (represented by the rows of the tensor), the function loops over each embedding dimension (represented by the columns of the tensor).
 - The positional embedding value for the current element and embedding dimension is calculated using a formula that involves the current element index and the embedding dimension. Specifically, it uses a sinusoidal function with a period that varies based on the embedding dimension. If the embedding dimension is even, the period is determined by $(10000 ^ (j / d))$, where j is the current embedding dimension and d is the total embedding dimension. If the embedding dimension is odd, the period is determined by $(10000 ^ ((j - 1) / d))$.
 - The calculated positional embedding value is stored in the corresponding position in the tensor.
 - Once all positional embeddings have been calculated and stored, the function returns the tensor.
- Then created a class MyMSA for Multi-Head Self-Attention. The module takes as input a sequence of tokens and applies an attention mechanism on it using multiple heads. The input sequence is split into `n_heads` parts, where `n_heads` is a hyperparameter. For each head, the module applies three linear mappings to transform the input to query, key, and value vectors. It then calculates the dot-product similarity between the query and key vectors, applies softmax, and multiplies the result with the value vector to get the attention output for each head. The outputs for all the heads are concatenated and returned.

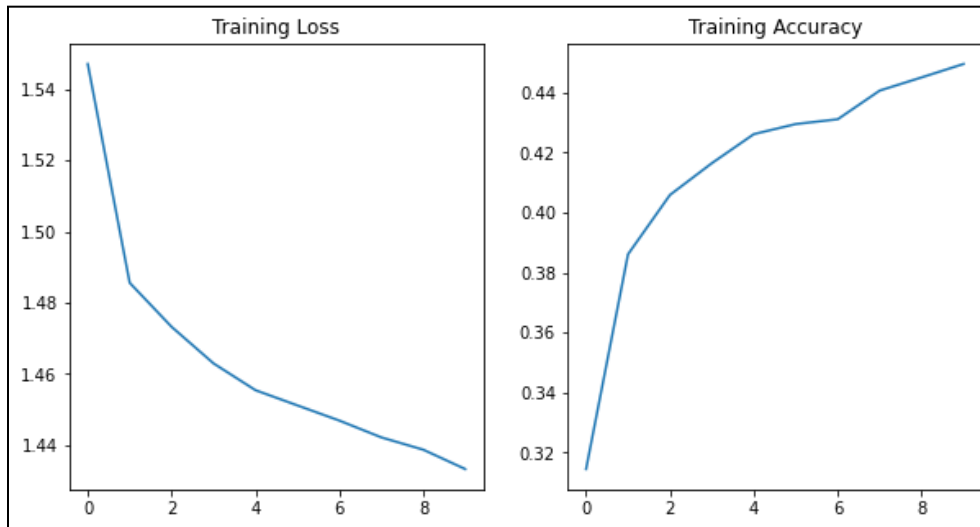
- Then, I implemented a class of the ViT (Vision Transformer) block. It is a module that consists of a multi-head self-attention layer (MyMSA) and a multi-layer perceptron (MLP) layer. The input tensor is first normalized using a LayerNorm layer, passed through the MyMSA layer to obtain the attention output, and then the residual output of the MyMSA layer is normalized again using another LayerNorm layer. Finally, the output is passed through the MLP layer and added back to the input tensor to produce the final output of the block.
- The constructor takes three arguments: `hidden_d`, `n_heads`, and `mlp_ratio`. `hidden_d` is the dimension of the input tensor, `n_heads` is the number of attention heads in the MyMSA layer, and `mlp_ratio` is the ratio of the hidden dimension of the MLP layer to the hidden dimension of the input tensor.
- The forward method takes an input tensor `x` and applies the ViT block to it, returning the final output tensor.
- Finally, wrote MyViT Class which includes:
 - Linear mapper: This takes in the image patches and maps them to a hidden dimension.
 - Learnable classification token: This is a learnable parameter that is added to the tokenized patches.
 - Cosine positional embedding: This is a fixed set of learnable parameters that are added to the tokenized patches.
 - Transformer encoder blocks: These are self-attention blocks that process the tokenized patches.
 - Classification MLP: This is a multi-layer perceptron that takes the final classification token and maps it to the output dimension, which is the number of classes.
- The MyViT class takes in the input shape `chw`, which is a tuple of (C, H, W), where C is the number of channels, and H and W are the height and width of the input image. `n_patches` is the number of patches to divide the image into, and `n_blocks` is the number of transformer blocks to use. `hidden_d` is the hidden dimension of

the transformer, and `n_heads` is the number of heads used in the multi-head self-attention. `out_d` is the output dimension, which is the number of classes in the classification task.

- The forward method of the `MyViT` class takes in the input image `images`, which has shape `(N, C, H, W)`, where `N` is the batch size. The input image is divided into patches using the `patchify` function. The patches are then passed through the linear mapper and concatenated with the classification token. The positional embedding is added to the resulting tensor, and then the transformer blocks process the tensor. Finally, the classification token is extracted from the output tensor and passed through the MLP to get the output class probabilities.
- Instantiated the model object with the following arguments:
 - `Chw = (3, 32, 32)`
 - `n_patches = 8`
 - `n_blocks = 2`
 - `n_hidden = 6`
 - `n_heads = 8`
 - `output_classes = 5`
- Used Adam optimizer with learning rate=0.001 and used `CrossEntropyLoss`.
- Then, trained the transformer model on the CIFAR10 dataset.

```
Training started ...
Epoch: 1, Loss: 1.5471, Accuracy: 31.45%
Epoch: 2, Loss: 1.4856, Accuracy: 38.61%
Epoch: 3, Loss: 1.4732, Accuracy: 40.59%
Epoch: 4, Loss: 1.4630, Accuracy: 41.65%
Epoch: 5, Loss: 1.4554, Accuracy: 42.62%
Epoch: 6, Loss: 1.4512, Accuracy: 42.96%
Epoch: 7, Loss: 1.4469, Accuracy: 43.12%
Epoch: 8, Loss: 1.4422, Accuracy: 44.07%
Epoch: 9, Loss: 1.4387, Accuracy: 44.51%
Epoch: 10, Loss: 1.4333, Accuracy: 44.96%
Training finished!
```

- Plotted the training loss and training accuracies epoch-wise.



- Tested the model on the test dataset:

Accuracy of the network on the 5000 test images: 45 %

- Then, I moved on to implement learnable positional encoding with four encoder and decoder layers with six heads. Using relu activation in the intermediate layers.
- Wrote a function `get_learnable_positional_encodings` which:

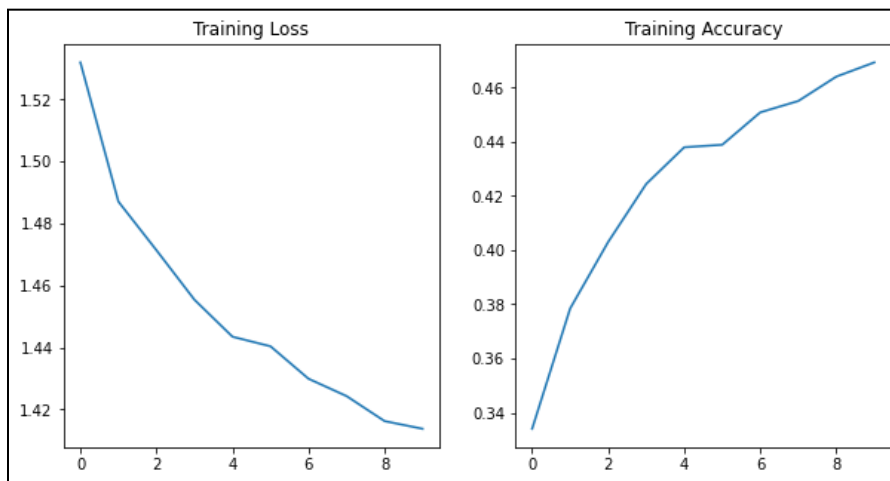
Returns learnable positional encodings for a given sequence length and hidden dimension d . It uses the `nn.Parameter` function to create a parameter tensor for the relative positional encodings. The values of the parameter tensor are randomly initialized using a normal distribution with mean 0 and standard deviation scale, where scale is calculated as $d^{-0.5}$.

The function supports the option of specifying the number of heads for multi-head attention. If heads is `None` or `True`, a shared set of relative positional encodings is returned for all attention heads. If heads is an integer greater than 1, a separate set of relative positional encodings is returned for each attention head.

- Then, trained a new model with learnable positional encodings with four encoder and decoder layers with 6 heads.

```
Training started ...  
Epoch: 1, Loss: 1.5320, Accuracy: 33.40%  
Epoch: 2, Loss: 1.4871, Accuracy: 37.84%  
Epoch: 3, Loss: 1.4714, Accuracy: 40.30%  
Epoch: 4, Loss: 1.4554, Accuracy: 42.44%  
Epoch: 5, Loss: 1.4434, Accuracy: 43.80%  
Epoch: 6, Loss: 1.4403, Accuracy: 43.89%  
Epoch: 7, Loss: 1.4299, Accuracy: 45.08%  
Epoch: 8, Loss: 1.4243, Accuracy: 45.50%  
Epoch: 9, Loss: 1.4162, Accuracy: 46.40%  
Epoch: 10, Loss: 1.4137, Accuracy: 46.93%  
Training finished!
```

- Then, plotted the training loss and training accuracy curves wrt no. of epochs.

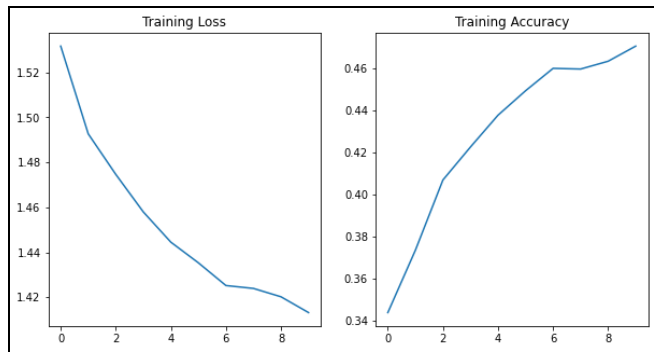


- Then, test accuracy of the model came out to be;

Accuracy of the network on the 5000 test images: 46 %

- Then, trained 2 new models as:
 - Tanh activation with cosine positional embeddings.

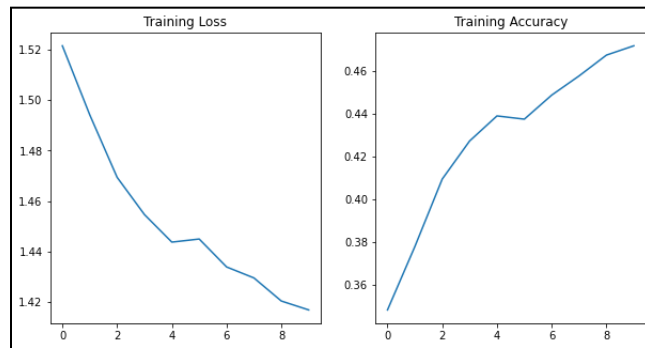
```
Training started ...
Epoch: 1, Loss: 1.5317, Accuracy: 34.37%
Epoch: 2, Loss: 1.4927, Accuracy: 37.34%
Epoch: 3, Loss: 1.4747, Accuracy: 40.69%
Epoch: 4, Loss: 1.4580, Accuracy: 42.26%
Epoch: 5, Loss: 1.4445, Accuracy: 43.78%
Epoch: 6, Loss: 1.4353, Accuracy: 44.94%
Epoch: 7, Loss: 1.4252, Accuracy: 46.01%
Epoch: 8, Loss: 1.4239, Accuracy: 45.98%
Epoch: 9, Loss: 1.4202, Accuracy: 46.35%
Epoch: 10, Loss: 1.4132, Accuracy: 47.07%
Training finished!
```



Accuracy of the network on the 5000 test images: 46 %

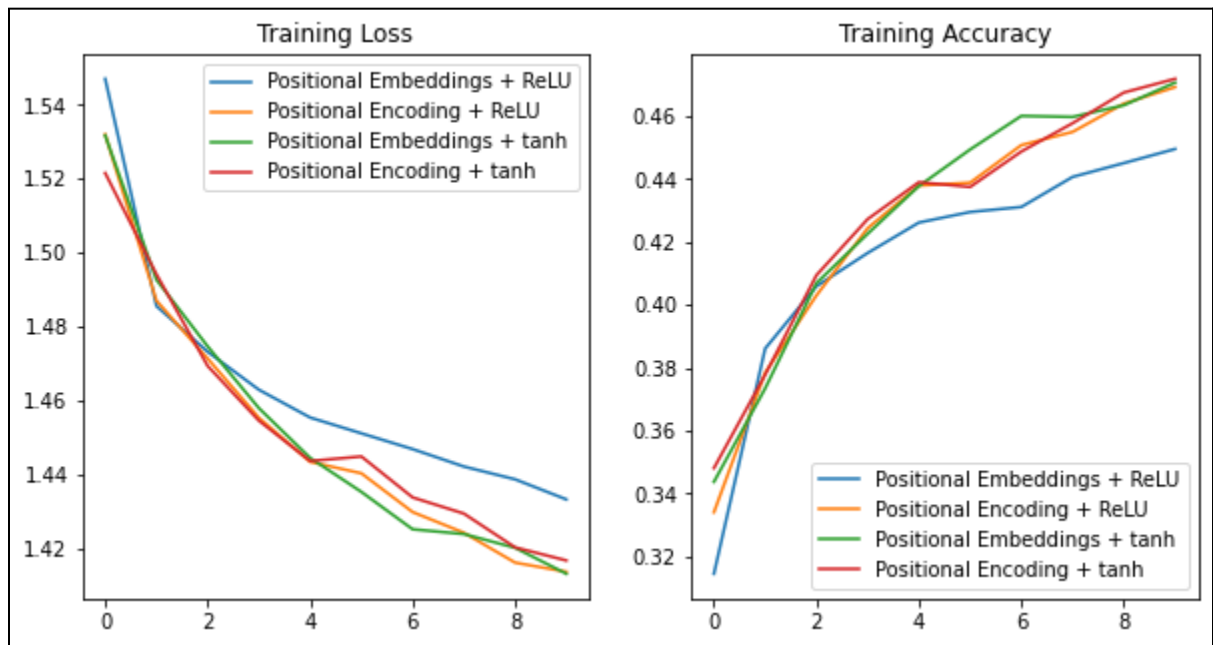
- Tanh activation with learnable positional encodings.

```
Training started ...
Epoch: 1, Loss: 1.5216, Accuracy: 34.81%
Epoch: 2, Loss: 1.4942, Accuracy: 37.78%
Epoch: 3, Loss: 1.4694, Accuracy: 40.94%
Epoch: 4, Loss: 1.4546, Accuracy: 42.73%
Epoch: 5, Loss: 1.4437, Accuracy: 43.90%
Epoch: 6, Loss: 1.4449, Accuracy: 43.75%
Epoch: 7, Loss: 1.4338, Accuracy: 44.88%
Epoch: 8, Loss: 1.4295, Accuracy: 45.78%
Epoch: 9, Loss: 1.4203, Accuracy: 46.76%
Epoch: 10, Loss: 1.4168, Accuracy: 47.19%
Training finished!
```



Accuracy of the network on the 5000 test images: 45 %

- Then, I compared the results of all the 4 models trained.



Question 02

Aim:


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
- Train the following models for profiling them using during the training step
 1. Conv -> Conv -> Maxpool (2,2) -> Conv -> Maxpool(2,2) -> Conv -> Maxpool(2,2)
 - We can decide the parameters of convolution layers and activations on our own.
 - Made sure to keep 4 conv-layers and 3 max-pool layers in the order described above.
 2. VGG16
- After the profiling of our model, figured 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:

- All the necessary imports such as numpy, pandas, torch, torchvision, tqdm, warnings, etc. were done.
- Added transformations such as Random Horizontal Flip, RandomAffine, converted the images to Tensor and normalized the images.
- Imported the CIFAR10 dataset from torchvision.datasets with batch_size = 128
- classes = {0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer', 5: 'dog', 6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'}
- Setup the device to GPU
- Created a CNN architecture according to the question's requirements:

```
CNN(  
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (conv4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (fc1): Linear(in_features=2048, out_features=500, bias=True)  
  (fc2): Linear(in_features=500, out_features=10, bias=True)  
)
```

- 
- Imported tensorboard and torch_tb_profiler libraries.
 - Implemented a training function for training a PyTorch model using the Adam optimizer and cross-entropy loss. It also includes profiling using PyTorch's built-in profiler to collect performance metrics during training.
 - The input parameters of the function are:
 - model: the PyTorch model to be trained.
 - model_name: a string indicating the name of the model for saving the profiler output.
 - The function first moves the model to the GPU and initializes the optimizer and loss function. It then enters a loop over 10 epochs, where it trains the model on the training data and evaluates it on the testing data.
 - Within each epoch, it sets the model to train mode, initializes counters for accuracy and loss, and iterates over the training data in batches. For each batch, it computes the outputs of the model, calculates the loss using the cross-entropy loss function, computes gradients and updates the model parameters using the Adam optimizer. It also computes the accuracy of the model on the training data for that batch.
 - After training, it sets the model to evaluation mode, and evaluates the model on the test data using the same process. It prints the training and testing loss and accuracy for each epoch.
 - Finally, it returns a dictionary stats containing the training and testing loss and accuracy for each epoch, which can be used for further analysis.

 - Trained the CNN Model:

```

100%|██████████| 391/391 [00:20<00:00, 19.20it/s]
Epoch: 3, Train Loss: 0.9260671343035101, Train Accuracy: 0.6724
Epoch: 3, Test Loss: 0.790056586265564, Test Accuracy: 0.7262

100%|██████████| 391/391 [00:20<00:00, 19.13it/s]
Epoch: 4, Train Loss: 0.7945665310106009, Train Accuracy: 0.7215
Epoch: 4, Test Loss: 0.7500615957417066, Test Accuracy: 0.7439

100%|██████████| 391/391 [00:23<00:00, 16.90it/s]
Epoch: 5, Train Loss: 0.7210229296818413, Train Accuracy: 0.7462
Epoch: 5, Test Loss: 0.6576734898211081, Test Accuracy: 0.7766

100%|██████████| 391/391 [00:22<00:00, 17.50it/s]
Epoch: 6, Train Loss: 0.6609776976620755, Train Accuracy: 0.7672
Epoch: 6, Test Loss: 0.6275909422319147, Test Accuracy: 0.7846

100%|██████████| 391/391 [00:20<00:00, 19.12it/s]
Epoch: 7, Train Loss: 0.6154347982857843, Train Accuracy: 0.7847
Epoch: 7, Test Loss: 0.6663246147240265, Test Accuracy: 0.7751

100%|██████████| 391/391 [00:20<00:00, 19.16it/s]
Epoch: 8, Train Loss: 0.5739971953432274, Train Accuracy: 0.8005
Epoch: 8, Test Loss: 0.5685162155688563, Test Accuracy: 0.8061

100%|██████████| 391/391 [00:45<00:00, 8.54it/s]
Epoch: 9, Train Loss: 0.5441160349894667, Train Accuracy: 0.8082
Epoch: 9, Test Loss: 0.5379513011703009, Test Accuracy: 0.8132

100%|██████████| 391/391 [00:33<00:00, 11.82it/s]
Epoch: 10, Train Loss: 0.5190929659187337, Train Accuracy: 0.8188
Epoch: 10, Test Loss: 0.5508139578601982, Test Accuracy: 0.8104

```

- Then, Imported the pretrained VGG16 model from torchvision.datasets and fine-tuned it for CIFAR10 dataset.

```

100%|██████████| 391/391 [01:23<00:00, 4.66it/s]
Epoch: 3, Train Loss: 1.0733815936176367, Train Accuracy: 0.6305
Epoch: 3, Test Loss: 0.9927021242395232, Test Accuracy: 0.6578

100%|██████████| 391/391 [01:24<00:00, 4.61it/s]
Epoch: 4, Train Loss: 0.8940036734351722, Train Accuracy: 0.7083
Epoch: 4, Test Loss: 0.7320887574666664, Test Accuracy: 0.7665

100%|██████████| 391/391 [01:24<00:00, 4.61it/s]
Epoch: 5, Train Loss: 0.7871910249790572, Train Accuracy: 0.7479
Epoch: 5, Test Loss: 0.6357861391351193, Test Accuracy: 0.7954

100%|██████████| 391/391 [01:24<00:00, 4.65it/s]
Epoch: 6, Train Loss: 0.7013016081679507, Train Accuracy: 0.7808
Epoch: 6, Test Loss: 0.6457263548162919, Test Accuracy: 0.8074

100%|██████████| 391/391 [01:24<00:00, 4.65it/s]
Epoch: 7, Train Loss: 0.671995295511792, Train Accuracy: 0.7901
Epoch: 7, Test Loss: 0.6093429210065286, Test Accuracy: 0.8113

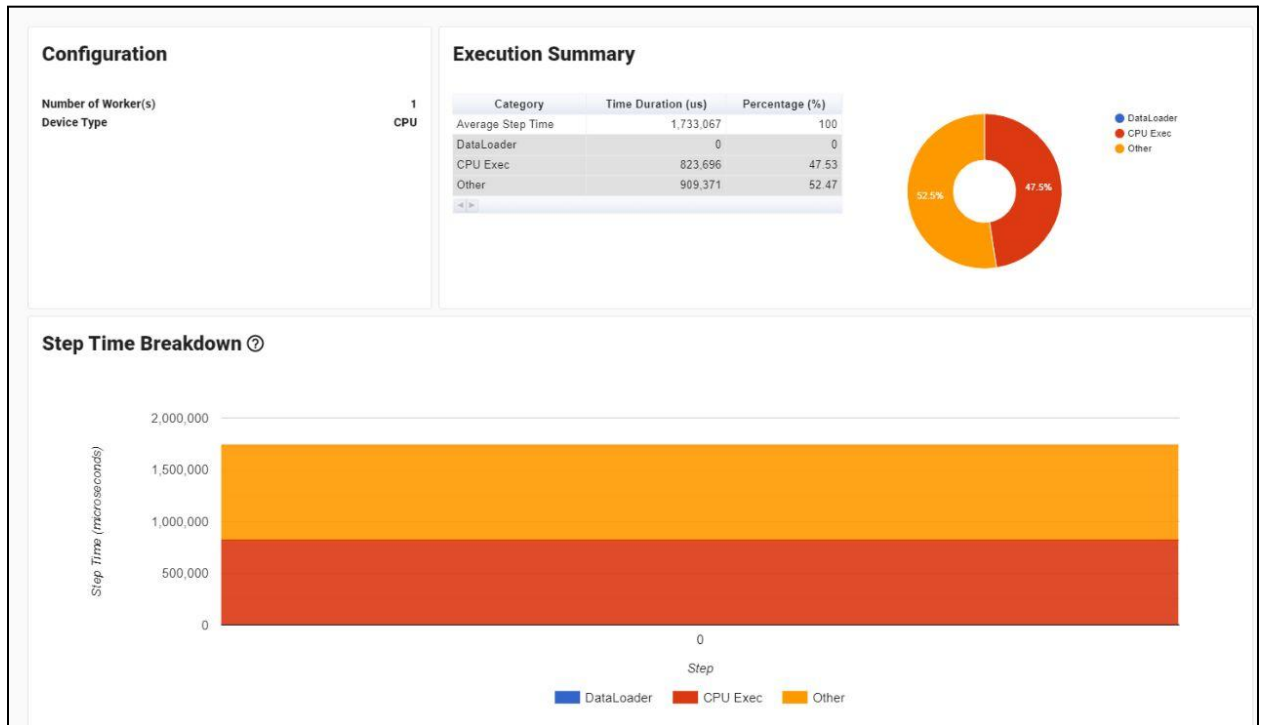
100%|██████████| 391/391 [01:23<00:00, 4.66it/s]
Epoch: 8, Train Loss: 0.6226440493560508, Train Accuracy: 0.8064
Epoch: 8, Test Loss: 0.5588809542263611, Test Accuracy: 0.8339

100%|██████████| 391/391 [01:24<00:00, 4.65it/s]
Epoch: 9, Train Loss: 0.6065432282207567, Train Accuracy: 0.8132
Epoch: 9, Test Loss: 0.5740932433665553, Test Accuracy: 0.8215


100%|██████████| 391/391 [01:24<00:00, 4.65it/s]
Epoch: 10, Train Loss: 0.596957706581906, Train Accuracy: 0.818
Epoch: 10, Test Loss: 0.5905067600026915, Test Accuracy: 0.8188

```

- CNN Tensorboard Screenshot:



- Then, figured out the minimum changes in CNN and VGG16 models so as to get a gain in performance and decrease training time on CIFAR10 dataset as compared to one achieved before.
- CNN Updated:
 - Replacing the fourth convolutional layer with a global average pooling layer. This modification can reduce the number of parameters in the model and prevent overfitting, thereby potentially improving its performance and reducing its training time.
 - The modified CNN_GAP model replaces the fourth convolutional layer with an adaptive average pooling layer (self.gap) with output size of (1, 1). This pooling layer averages the output feature maps from the third convolutional layer into a single feature vector for each input image. This feature vector is then passed through a fully connected layer (self.fc) to produce the final output predictions.
 - By replacing the fourth convolutional layer with a global average pooling layer, we can reduce the number of parameters in the model and prevent overfitting, which can lead to improved performance and reduced training



time on the CIFAR10 dataset. However, it's important to note that the performance of the modified CNN_GAP model may vary depending on factors such as the learning rate, the choice of optimizer, and the size of the dataset.

- VGG16 updated:
 - Modifying the number of filters in each convolutional layer and adjust the number of neurons in the fully connected layers accordingly. Specifically, we can reduce the number of filters in each convolutional layer to reduce the number of parameters and speed up training, while maintaining the structure of the network.
 - Below is the modified VGG16 model that reduces the number of filters by a factor of 4.