DL-Ops Assignment-3 Report

- by D22CS051

Task: Implement different architectures on the CIFAR10 dataset using standard augmentation and
normalization techniques

Note: All the files are uploaded in the zipped folder, so there is no need to train the model again.

Objectives:

Question 1.

- Implement ViT from scratch
- 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.

Question 2.

- Load and preprocessing 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:

- The necessary libraries are imported, including PyTorch, NumPy, and torchmetrics. The device is also set to "cuda" if a GPU is available, otherwise it is set to "cpu".
- The CIFAR-10 dataset is downloaded and transformed using PyTorch's transforms module. Two subsets are created from the full train and test sets that only contain images of the classes 'airplane', 'bird', 'car', 'dog', and 'ship'. These subsets are then converted to PyTorch DataLoaders to facilitate feeding data to the model during training and evaluation.

- The model architecture, early stopping criteria, hyperparameters, loss function, and optimizer are defined.
- The script trains the model on various hyperparameter combinations, each combination being trained for a fixed number of epochs. The train function defined in engine.py is used to train the model.
- The results of each experiment are printed, including the training and validation loss and accuracy, the time taken to train the model, and the hyperparameters used.
- Finally, a plot of the training and validation accuracy curves is generated and saved to disk using the plot_curves function defined in plotting.py.
- Results:

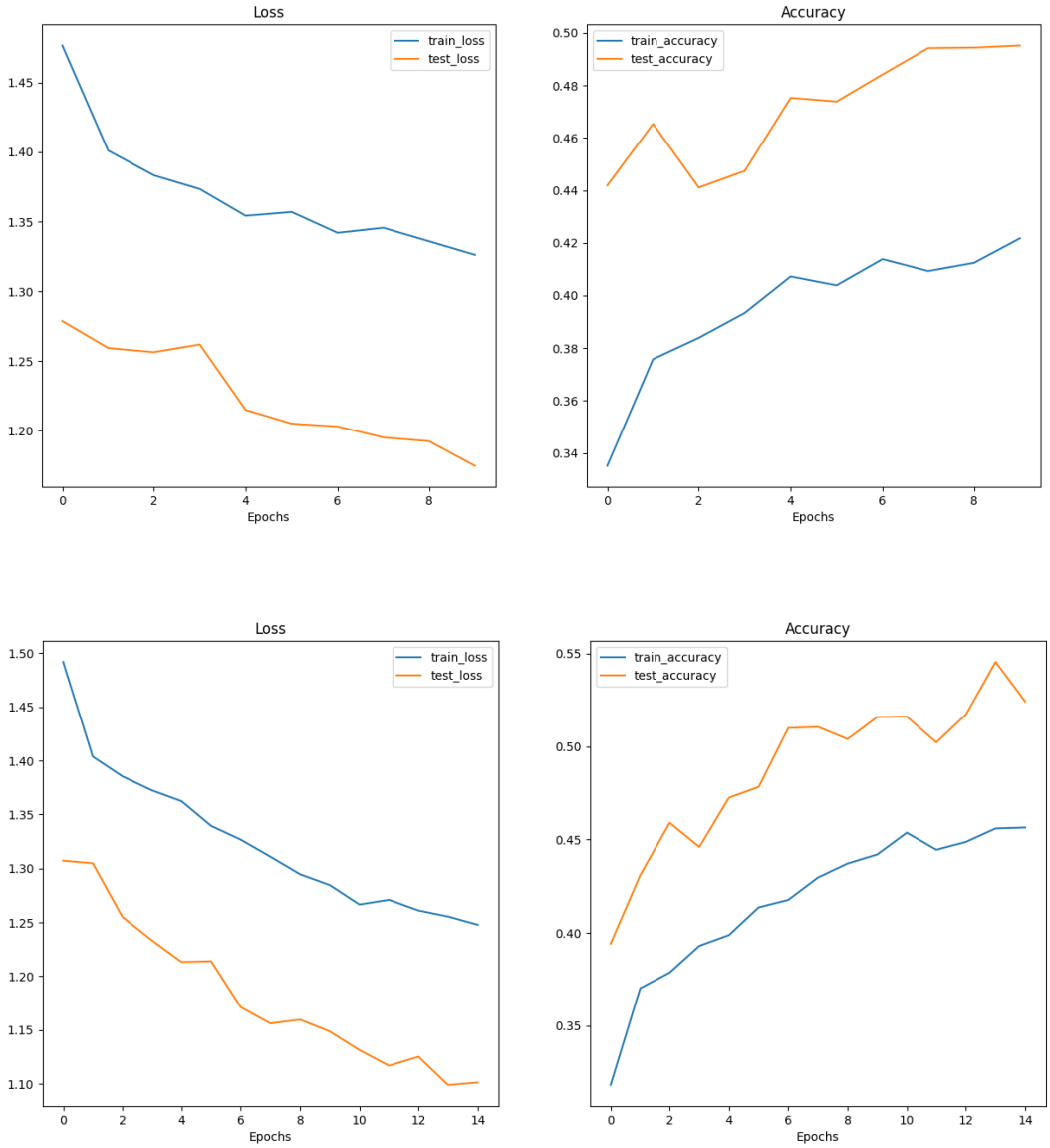

Fig: Training ViT on Cifar-10

Analysis:

- The above shows the training of the ViT arch. With Tanh & ReLU activation.
- From the Curve one can see that the ReLU converges better than Tanh activation despite skip connections in the architecture.
- Aslo one can observe the Tanh is train with 10 epochs, and ReLU is trained with 15 epochs but, this happed only because of hyperparms tunning, & also on comparison it is also clear that ReLU performs better.
- Also the training Time is also very high, and performance is relative to the present CNN arch.
- But it is also True that ViT will outperform the Best performing CNN Arch on a Large dataset like Image-Net and Cifar-100.
- ViT inference time is also very high but can be optimized using torch script and ONNX inference engines.
- The Size is ViT model is soo large that is not possible to run on mobile/embedded systems hence needs a server to host.
- Also, the initial Accuracies are far better then the cnns, the reason behind that is we are using only 5 class subsets of the original dataset.

Procedure:

- Import the required libraries such as PyTorch, Torchvision, NumPy, and others
- Define the device to use for computation, based on the availability of a GPU
- Load the CIFAR10 dataset and transform the data using a set of transformations
- Convert the data into torch dataloader to create an iterable object for training and testing data
- Import the CustomArch model and EarlyStopping from the respective Python modules
- Define a function to get an instance of the CustomArchModified model
- Set up hyperparameters for the model such as learning rate, betas, weight decay, and epochs
- Train the model with each combination of hyperparameters, using the train function from the engine module
- Compute and plot the loss and accuracy curves for each set of hyperparameters
- Print the time taken for training each model with different sets of hyperparameters.
- Do the Same for CustomArchModified.
- Repeated the same steps for VGG-16 Arch also.
- Results: All the analysis are given at the end of graphs

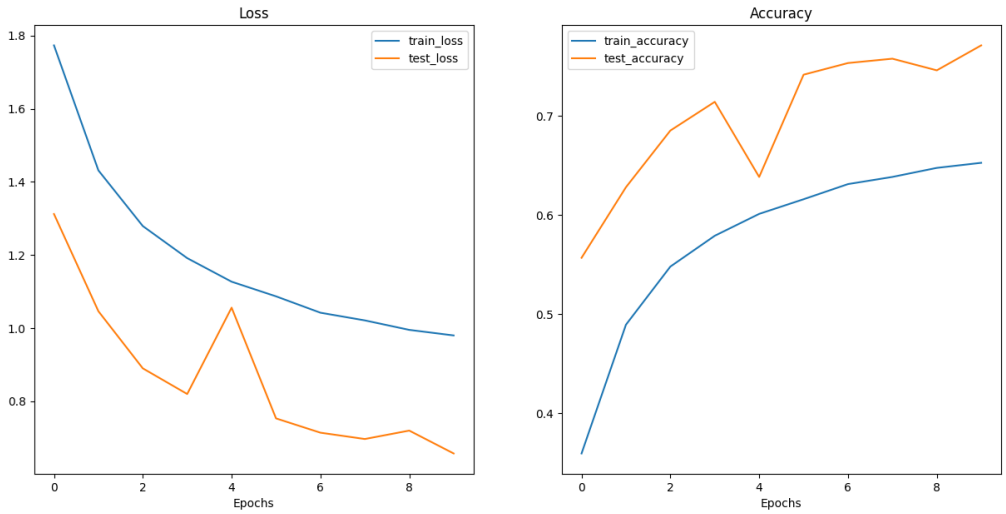

Fig:- Training CustomArch

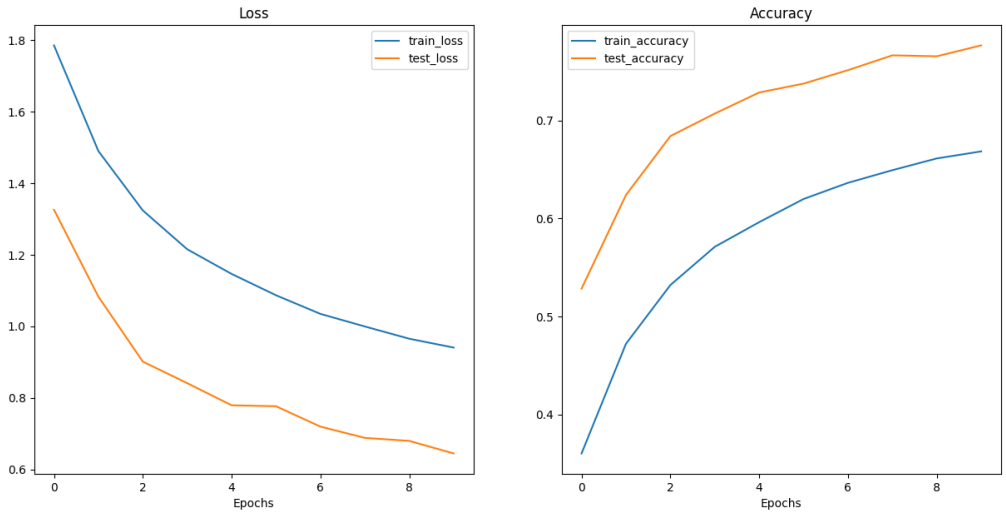

Fig: Training CustomArchModified

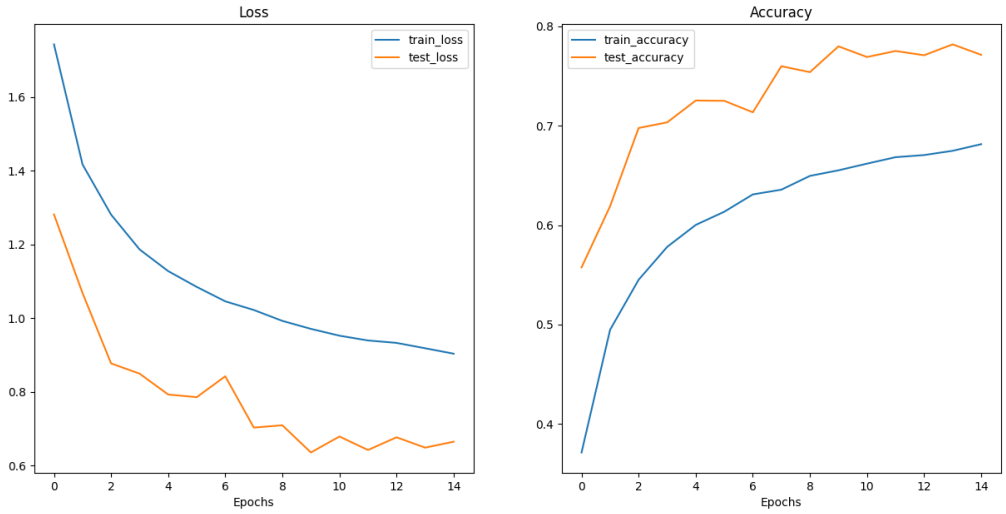

Fig: Training Custom Arch

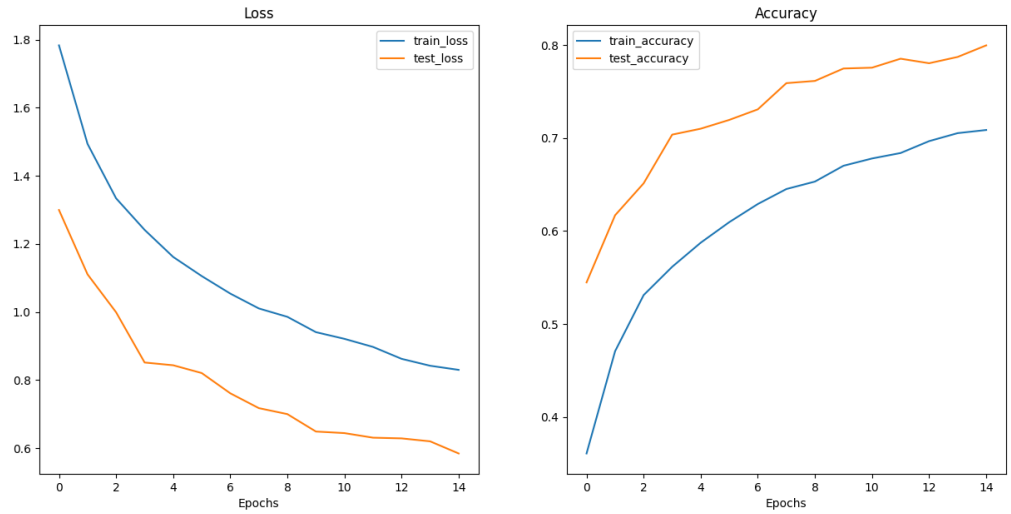

Fig: Training CustomArchModified

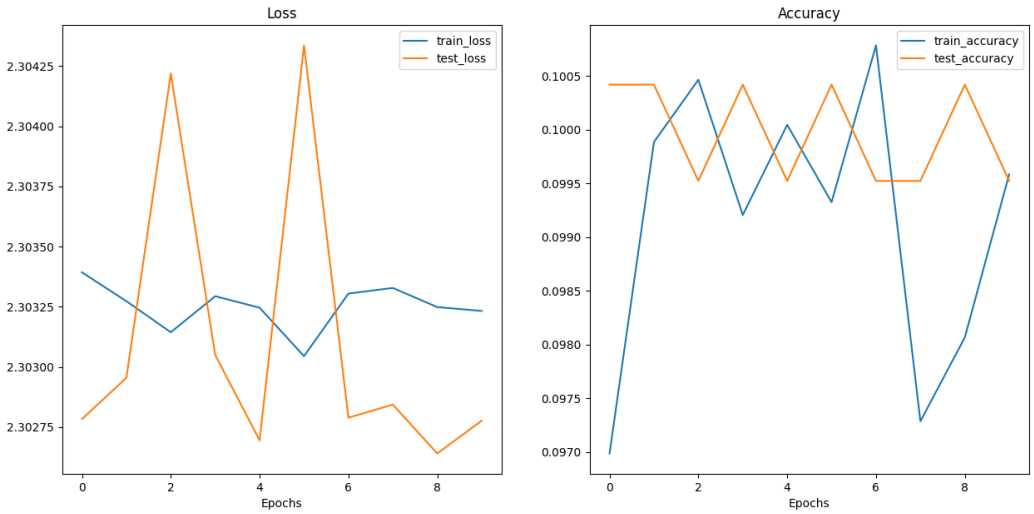

Fig: Training VGG-16

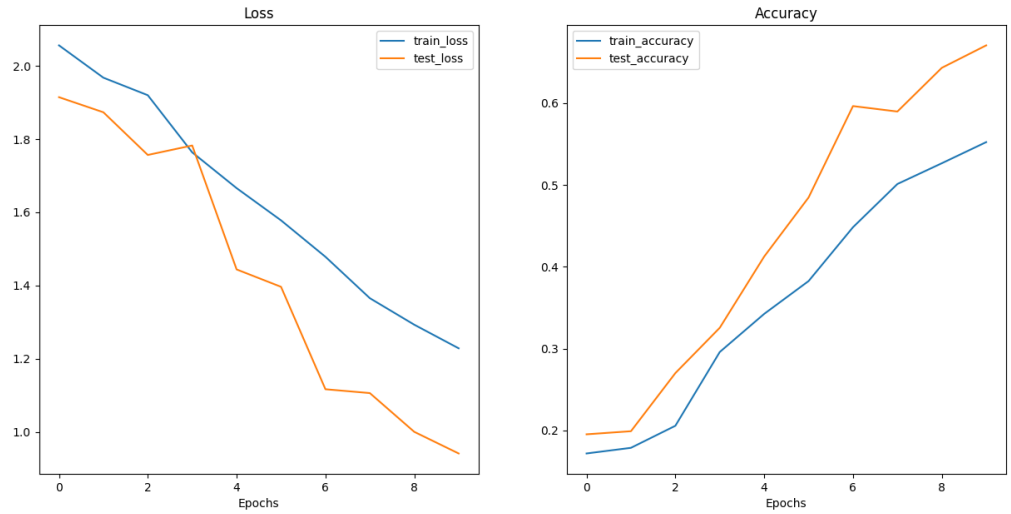

Fig: Training VGG-16 Modified

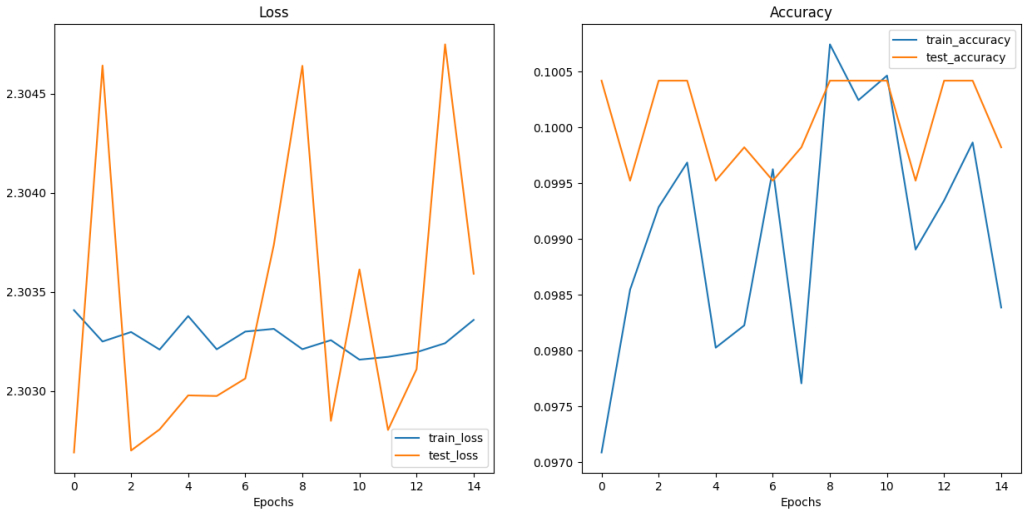

Fig: Training VGG-16

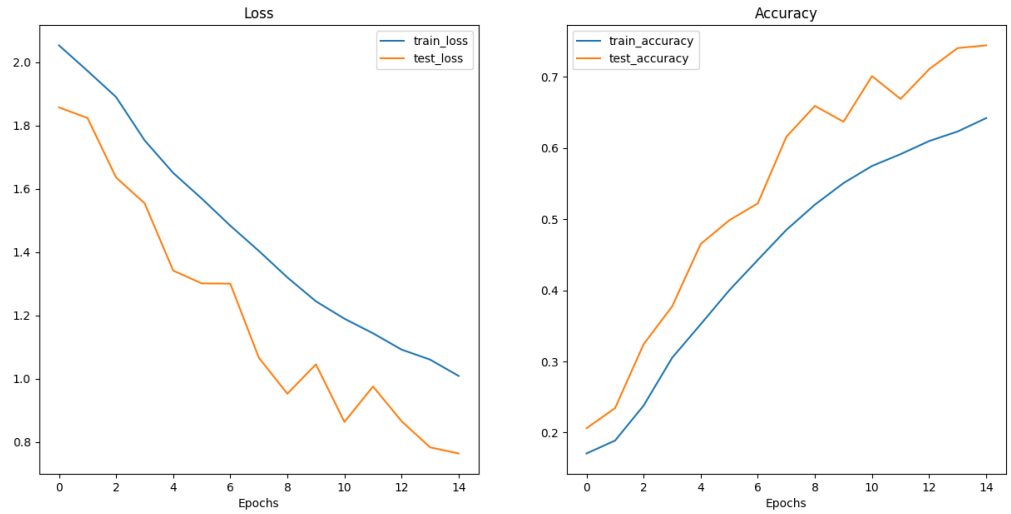

Fig: Training VGG-16 Modified

Analysis:

- By Modifying the Arch (CustomArch + VGG) the Training time has been reduced to around 31s for the 10 epoch run and 51-56s for 15 epoch run, which does not seem very significant but when training for 100 or 1000 epochs one can observe significant differences.
- The Accuracis also gets hurt when we save time hence the accuracy drop can be observed from 2% to 7%, but going further we have to optimize the tradeoff b/w them.
- We have archived Better Accrccy than base models but then train time has not dropped significantly as per the 1st case the drop is around 10 to 13 sec for 10 epoch runs and about 22 to 28 sec for 15 epoch runs.
- The Reason behind this is simply the model learns better when having more parameters to learn but the learning takes time also, hence the training time is directly proportional to # of parameters, but here we have reduced the non-learnable parameters which reduces the computation time, resulting in less training time and also more steady and a little higher accuracies then base models.
- Here the non-learnable computation is MAXPOOL, hence only one max pool with appropriate parameters can give the same output shape, which results in better results & less time than applying the max pool after every conv-block.
- The Main task of the Maxpool layer is to reduce the size of the image/feature vector, hence one max pooling layer at the end will do the task.
- Also for our task VGG-16 is Overkill, as it has very large numbers of conv layers, for the cifar 10 data set we will not be requireing those many conv layers.

References:

- https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c
- https://theaisummer.com/positional-embeddings/
- https://www.learnpytorch.io/08_pytorch_paper_replicating/
- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/docs/stable/onnx.html
- https://pytorch.org/tutorials/recipes/torchscript_inference.html
- https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html
- https://pytorch.org/docs/master/profiler.html
- https://numpy.org/doc/stable/reference/index.html
- https://torchmetrics.readthedocs.io/en/stable/
- https://stackoverflow.com/
- https://github.com/facebookresearch/mae/issues/30
- https://mchromiak.github.io/tag/vit.html
- https://huggingface.co/docs/transformers/model_doc/vit
- https://arxiv.org/abs/2010.11929v2
- https://paperswithcode.com/paper/an-image-is-worth-16x16-words-transformers-1#code
- DL-OPS Class videos.