BMI / CS 771 Homework Assignment 2
Convolution and Transformer Networks

Apoorva Kumar
akumar255@wisc.edu

Vaibhav Nitnaware
nitnaware@wisc.edu

Varun Kaundinya
kaundinya@wisc.edu

November 2, 2022

This assignment is about building convolution and Transformer neural networks for image classification. We have designed and trained different types of deep networks for scene recognition using PyTorch — an open-source deep learning package. A saliency map is also built from the learned network by identifying important image regions for classification. Finally, some adversarial samples are generated to confuse the built model.

1 Dataset

For this assignment, we will be using the MiniPlaces DataSet from MIT. This dataset has 120K images from 100 different categories. Here are some samples from the dataset.

(a) ski slope

(b) kindergarten classroom

(c) volcano

(d) golf course

(e) living room

(f) mountains

Figure 1: Samples from Dataset

Its a very diverse dataset containing samples from both outdoors like **Figure 1a, 1c, 1d, 1f** and indoors like **Figure 1b, 1e**.

The validation set also draws from the same sample and contains 10000 images.

2 Understanding Convolution

In this implementation, we use the fold and unfold operations in PyTorch to simulate convolution. Doing this allows us to utilize matrix multiplication optimization, which is more efficient than brute force convolution operation.

2.1 Forward Propagation

We find the output height and width of the output layer using the convolution kernel size, padding, strides, and input dimensions. We first `Unfold` the input and kernel matrices and then calculate the output by matrix multiplication. Since this multiplication is performed on two unfolded matrices, the result is an unfolded matrix output. We then add the channel bias by creating a so-called unfolded form of the bias vector for easy matrix addition. Finally, fold the output back to the initially calculated output shape using the `Fold` operation.

It also requires caching the unfolded input tensors, unfolded and original kernels matrix, and bias for the backward pass. The above processing is done on mini-batches of input images.

2.2 Backward Propagation

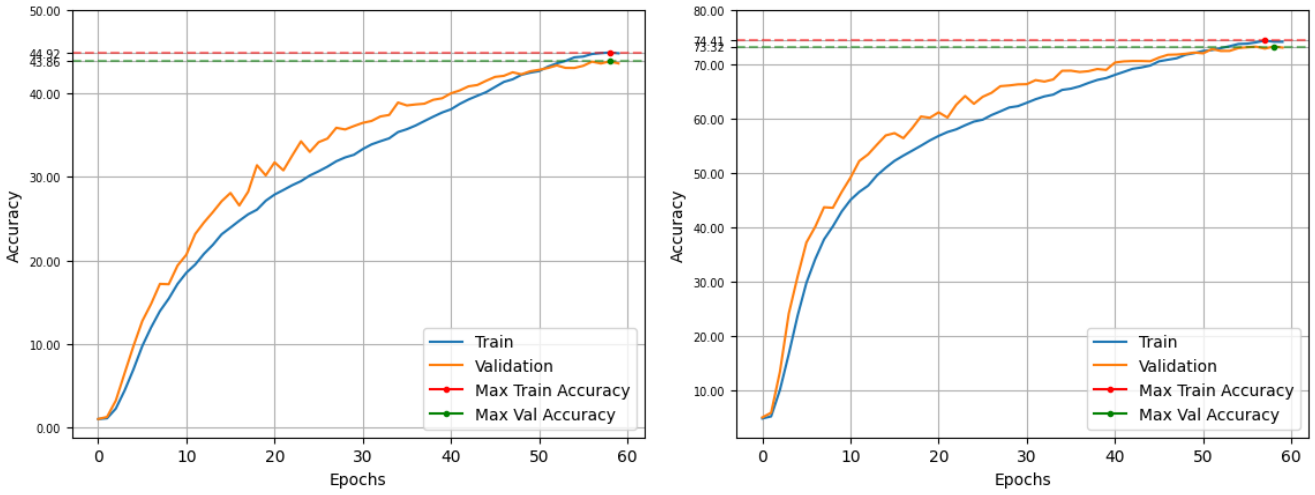
Like the forward propagation performed, we use the fold unfold operations to simulate the back-propagation of errors and gradients in the convolution layer and perform weight updates. First, we retrieve the saved parameters from the cache (from the forward pass). We calculate the gradients w.r.t the input by performing matrix multiplication on the unfolded kernel tensor and the transpose of the unfolded output gradients. It is then folded back into the same dimension as the initial input to the layer.

The gradient w.r.t weights is calculated by performing matrix multiplication between the transposed unfolded grad outputs and unfolded input tensor. The gradients are then summed across the mini-batch and reshaped to the kernel size.

3 Design and Train a Deep Neural Network

3.1 A Simple Convolutional Network

Code for this was provided in the `SimpleNet` Section. This model is trained for 60 epochs on GPU machine over cloud (GCP). The *top-1 validation* accuracy of **43.86%** and *top-5 validation* accuracy of **73.32%** was recorded.



(a) Top 1 Accuracy

(b) Top 5 Accuracy

Figure 2: Training Accuracy of Simple Convolution Network with Pytorch Convolution Layer

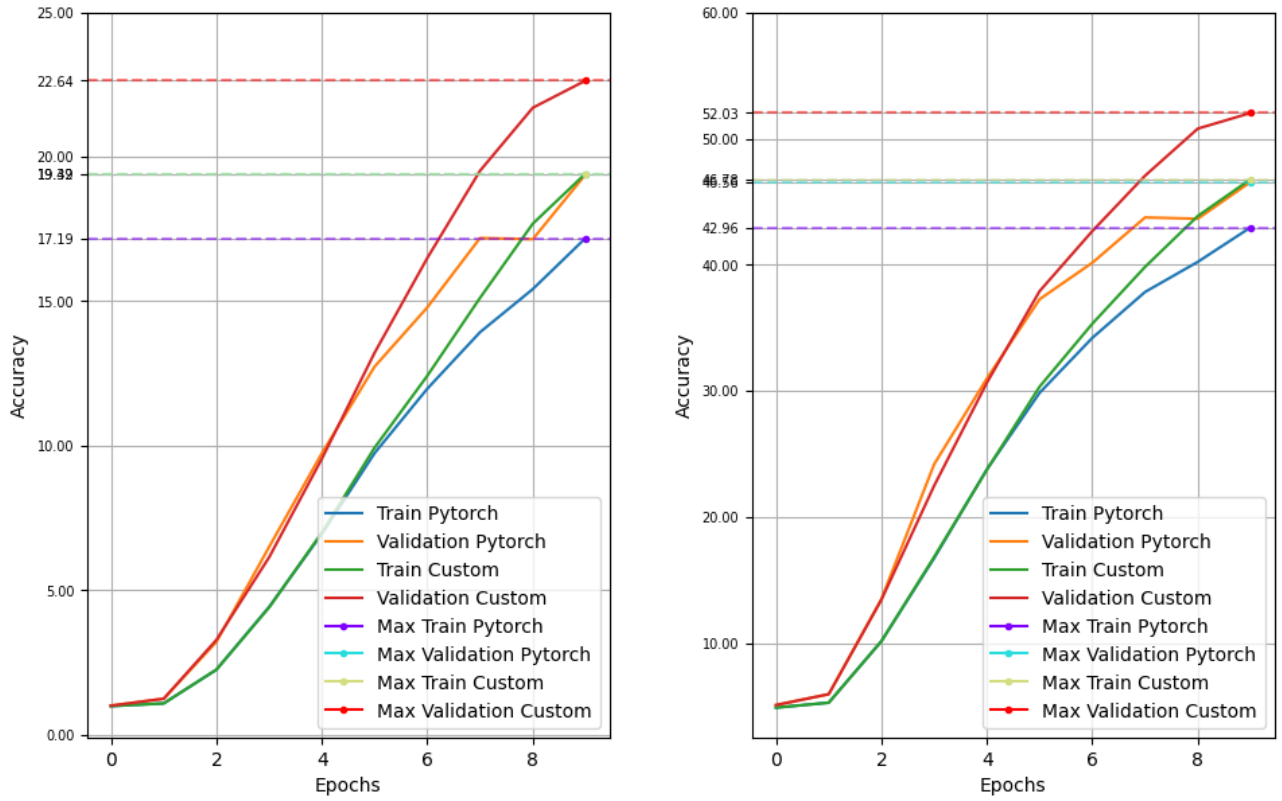
Looking at the accuracy curves in **Figure 2**, we can observe that the graph hasn't plateaued yet, so the training is yet to be completed, and we haven't reached the point that will mark the end of training and the beginning of over-fitting. If we continue training the model for more epochs, the accuracy should improve with lower loss.

3.2 Train With your own Convolutions

For this, we trained the model with the same configuration as before, except the convolutions were replaced by our Custom Convolutions that we designed earlier.

3.2.1 Training Accuracy

A *top-1 validation accuracy* of **22.64%** was observed after 10 epochs. When compared to the PyTorch implementation, after 10 epochs, a *top-1 validation accuracy* of **19.34%** was observed. The same metrics for *top-5 validation accuracy* was observed to be **52.03%** vs **46.56 %** (PyTorch). Therefore we can confidently say that the `CustomConvolution` doesn't have a drop in accuracy vs the PyTorch implementation and performs as intended. The accuracy is shown in **Figure 3**.



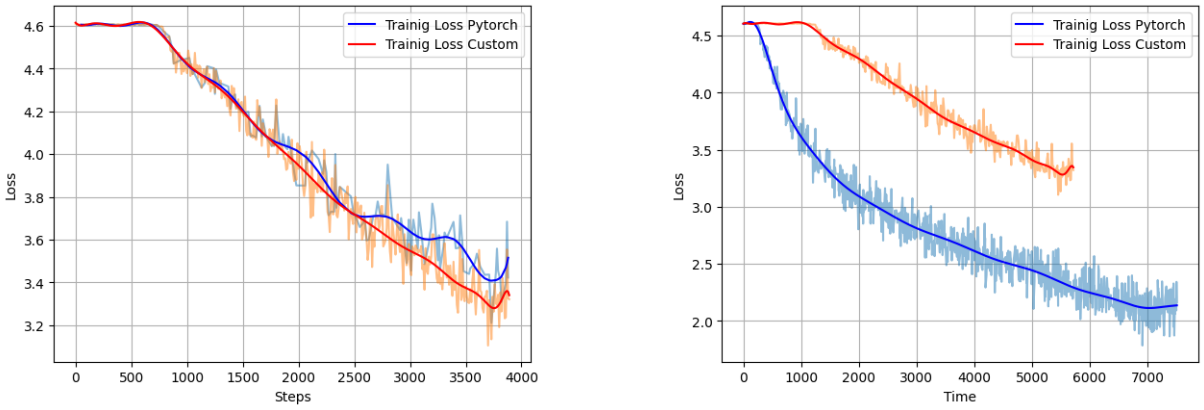
(a) Top 1 Accuracy

(b) Top 5 Accuracy

Figure 3: Training Accuracy of Simple Convolution Network with Custom Convolution Layer

3.2.2 Training Time

We can observe a time comparison in **Figure 4b**, which shows the time taken by the Custom Convolution model to run 10 epochs vs the Pytorch model to run 60 epochs. We observed a massive difference in this metric; the PyTorch model had completed 42 epochs in the time it took our model to complete 10 epochs. We infer that PyTorch uses internal optimization to perform efficient training for convolution layers.



(a) Training Loss after 10 epochs

(b) Total Training Time comparison

Figure 4: Loss of Simple Convolution Network with Custom Convolution Layer

3.2.3 Memory

In terms of memory consumption, we found the PyTorch model took approximately ∼**40%** (3079/7680 MiB) while the custom convolution model takes ∼**85%** (6574/7680 MiB). We infer that the PyTorch model uses memory optimization which reduces the runtime consumption by more than half.

4 A Simple Vision Transformer

In this, the code for a Vision Transformer blocks was provided and the model was put together by us.

4.1 Architecture

We first extract the embeddings from the input image using patching, done via a convolution operation by setting the stride equal to the kernel size. We add the learnable positional embeddings to the training method if passed as a parameter.

Then we apply a sequence of transformer blocks consisting batch norm layer, followed by an attention layer, followed by another batch norm layer. In the attention layer, we first calculate the projections across Q, K, and V. After which, we combine the Q and K using a softmax non-linearity multiplied by V, followed by linear projection.

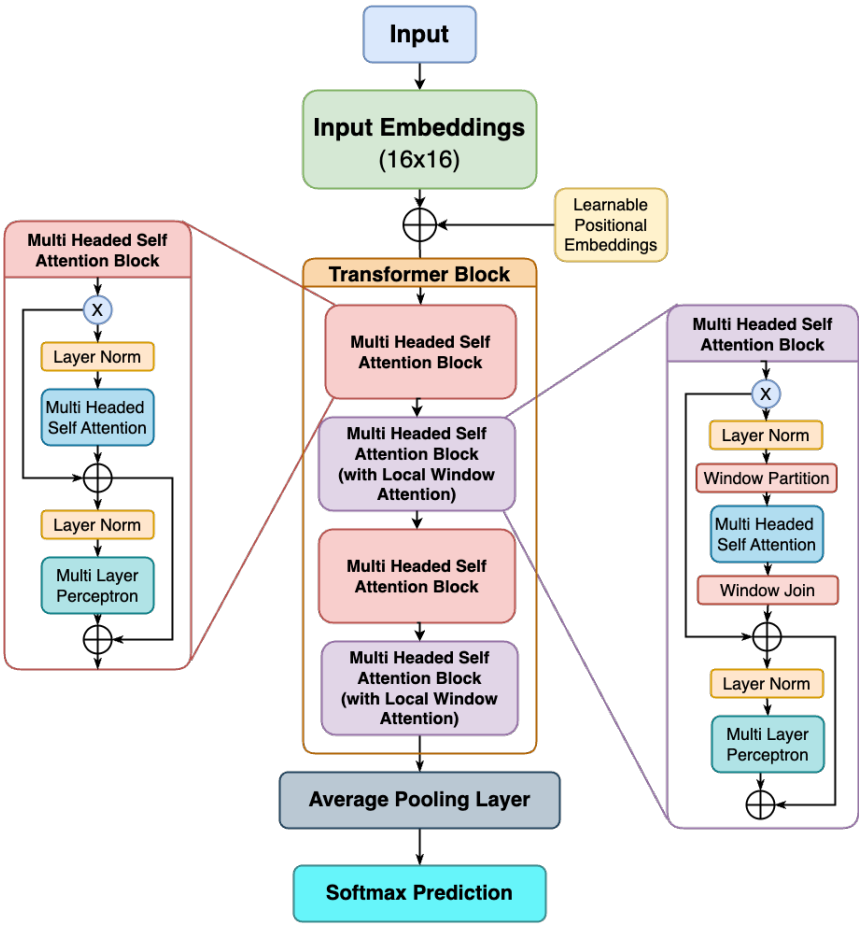
$$SA(X) = softmax(XW^Q(XW^K)^T)XW^VW^O \tag{1}$$

We also use the window size parameter to decide which patches require local and global attention. If this is not part of the parameters, all the patches will use global attention by default. Else the mentioned window block indexes will have local attention configured. In doing so, we combine local and global attention to reduce computational complexity while preserving a large receptive field. A drop path setting is also invoked to simulate drop-out in a vision transformer.

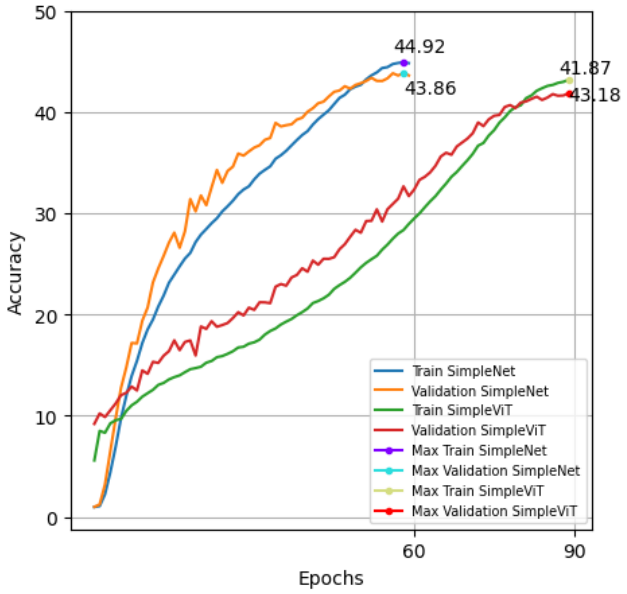
Post this transformer block, the output is processed through an average pooling layer to reduce the dimension and capture the required information to be processed. This is then passed to a fully connected layer with a softmax classifier with output dimensions equal to the number of classes. The complete model is shown in **Figure 5a**.

4.2 Training and Inference

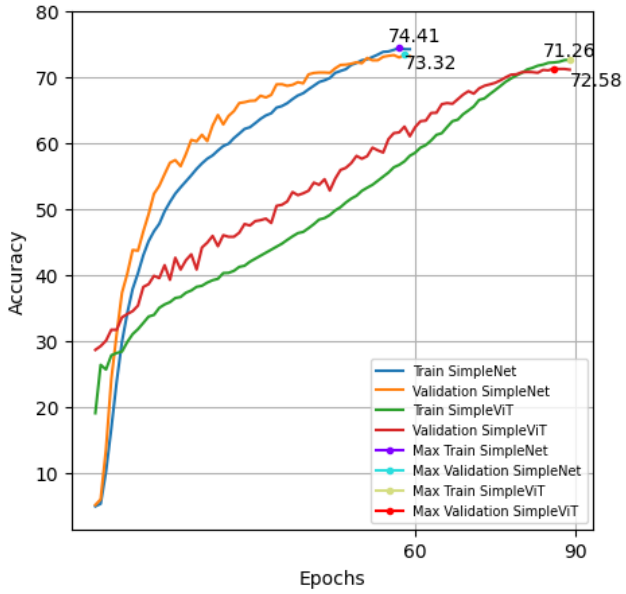
We trained the model for 90 epochs with a learning rate of 0.01 and a weight decay of 0.05. The *top-1 validation accuracy* of **43.18%** and *top-5 validation accuracy* of **72.58%** were recorded. The results of the final training epochs for the Transformer can be seen in **Figure 5**. We find these numbers comparable to the accuracy recorded for simple CNN and can also be inferred from the figure.



(a) Model Diagram Vision Transformer



(b) Top 1 Accuracy



(c) Top 5 Accuracy

Figure 5: Vision Transformer

5 Design Your Own Network

5.1 Model Architecture

Taking inspiration from the Inception Model [4], we decided to build a model similar to the inception but replacing the convolution blocks with transformer blocks. Our intuition behind this model was to capture different patch-sized blocks through each transformer block. Once they pass through the transformer blocks, we have computed the local and global attention for patches of different dimensions, allowing the model to learn attention across different patch sizes. We processed the input with three different embedding layers in parallel, each dividing the input in patch sizes of 4,8,16, respectively; respective outputs were then passed to different transformer blocks whose strides were scaled up/down based on patch size. These outputs (after average pooling) were stacked and passed through the fully connected layer as before. The model has been visualized in **Figure 6**.

5.1.1 Modifications to `main.py`

To test out Own model we made some modifications to the `main.py` to facilitate easier changes. The changes are mentioned below.

- Added Argument Parser option to directly access out model using the option `--use-custommodel` [Line 122-123]
- Added extra condition in model chooser to choose out custom model [Line 172-173]
- Added custom model to the condition choosing optimizer [Line 184-196]
- Added custom model to clipping gradients condition when training [Line 377-381]

We thoroughly checked to ensure that these modifications don't break any previous functions.

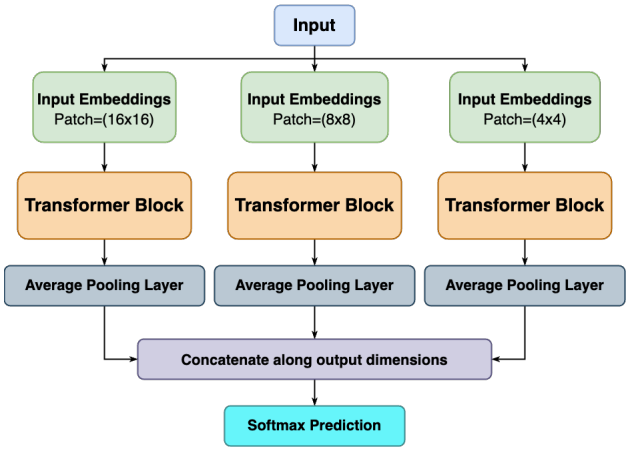


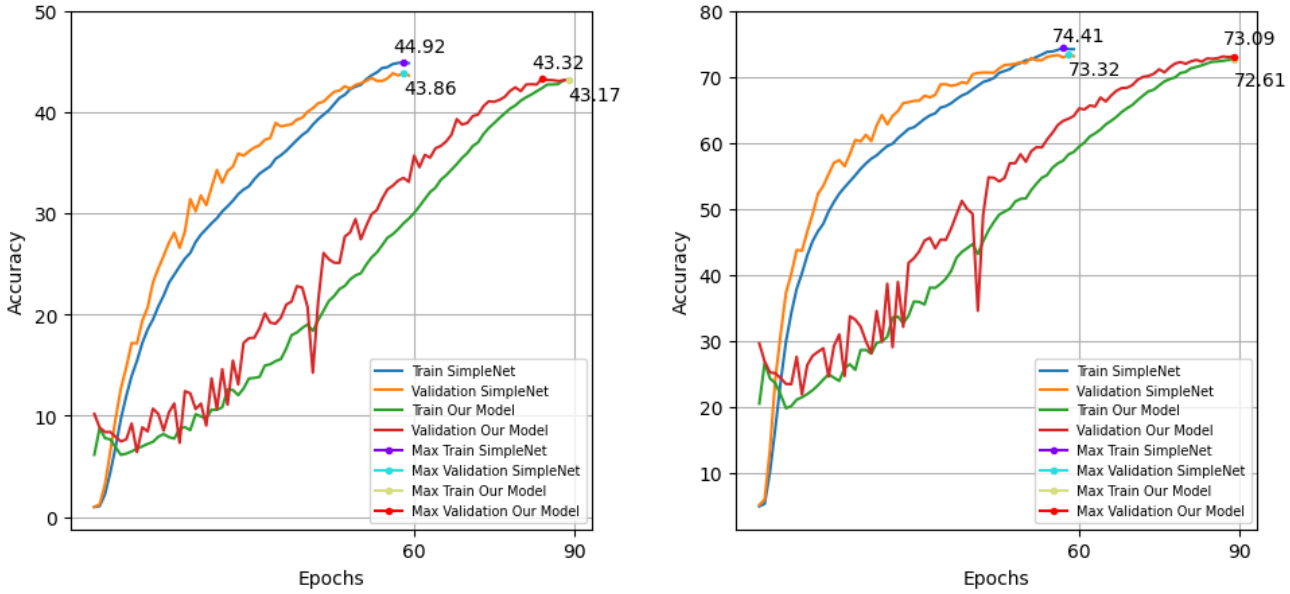
Figure 6: Model Diagram Custom Model

5.2 Training

The model was trained for 90 epochs with a *learning rate* of **0.01** and a *weight decay* of **0.05**. The *top-1 validation accuracy* of **43.32%** and *top-5 validation accuracy* of **73.09%** were recorded. The accuracy can be observed in **Figure 7**.

5.3 Inference

We find the accuracy numbers to be comparable to the accuracy recorded for simple CNN and the simple Vision Transformer. We notice something interesting when we observe the loss curves in **Figure 8**. Firstly, we see an increase in the loss with very high fluctuations, even though the accuracy seems to increase. Thus, we can infer



(a) Top 1 Accuracy

(b) Top 5 Accuracy

Figure 7: Training Accuracy of Custom Model

that, while the model can predict the classes well, it cannot differentiate very well between them, and the prediction probabilities of the actual class are very close to some other false classes. Secondly, the loss eventually converges to that of the simple Vision Transformer, which essentially points out that for this dataset, the change in patch size doesn't achieve any extra information capture.

6 Fine-Tune a Pre-trained Model

We use the inbuilt `resnet18` model in `torchvision`, which is pre-trained on the ImageNet dataset. We replace the last fully-connected layer to accept an input feature vector of 512 dimensions and output the classification vector of 100 dimensions.

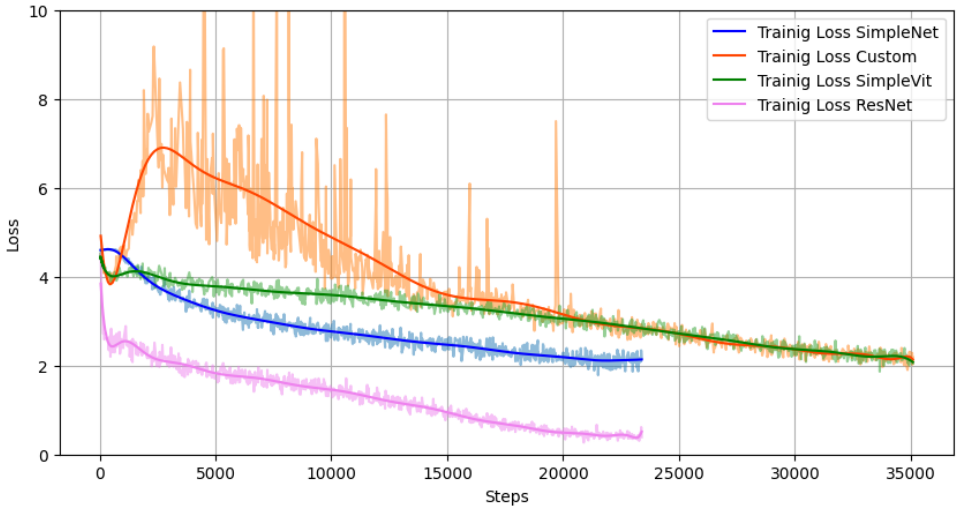
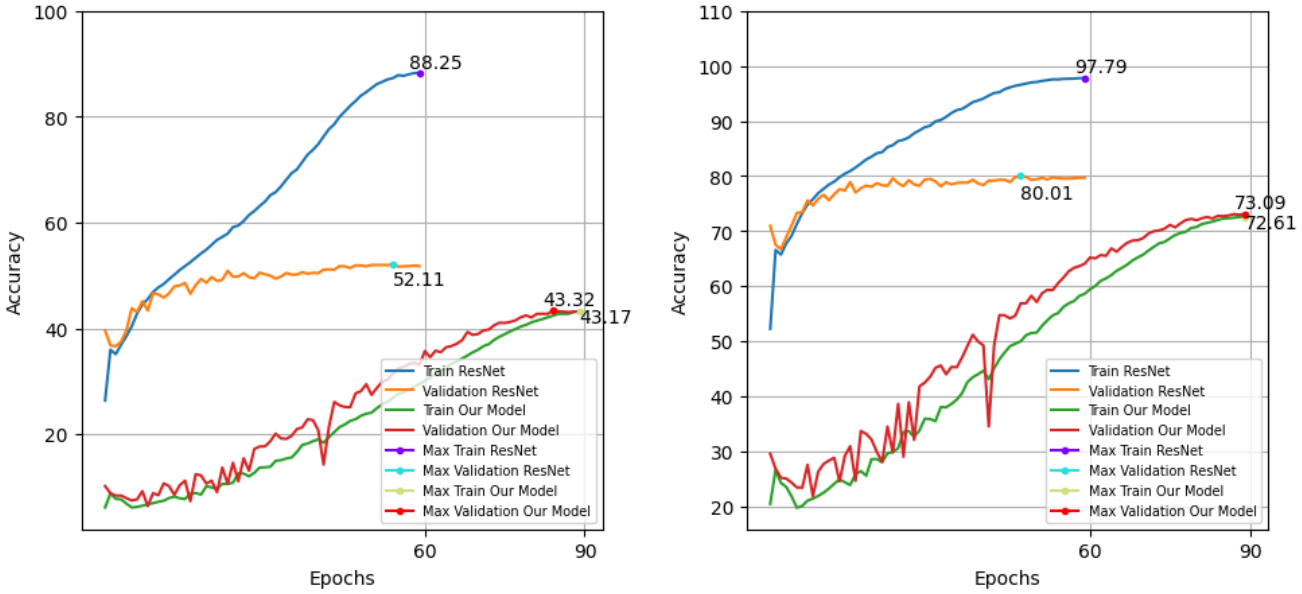


Figure 8: Training Losses



(a) Top 1 Accuracy

(b) Top 5 Accuracy

Figure 9: Training Accuracy of ResNet

We trained the model for 60 epochs. The *top-1 validation accuracy* of **52.11%** and *top-5 validation accuracy* of **80.01%** was recorded. We find a significant improvement in the pre-trained `resnet18` model as compared to both the simple CNN and our Custom Model in less number of epochs. The results can be seen in **Figure 9**. Another thing we inferred is that the validation curve for ResNet plateaus at a very early stage, which may signify over-training and eventually may even result in over-fitting. This shows how robust a pre-trained model can be when the required task is a derivative of a broader task. By simply replacing one layer in the model, we were able to outperform other complex models.

7 Attention and Adversarial Samples

7.1 Saliency Maps

in **Figure 10**, we can see the Saliency Maps of some images and how they can capture information in images.

We first enable the gradients at the input by setting the `requires_grad` flag for input tensor to `True` to ensure that we capture the gradients of the loss w.r.t to the input. We also disable the gradients for all model parameters so as to not collect the unnecessary gradients in the model graph.

Then, to obtain the saliency map, we first perform a forward pass of the input tensor through the given model. Based on the softmax prediction, the most confident class, i.e., the class with maximum softmax output, is chosen as the expected class, and the loss is computed for the given image.

The maximum gradients of loss w.r.t the input image will represent the importance of the pixel in the model. Since each pixel is composed of three channels, the final step is to find the absolute maximum among the three channel values to find a single saliency value for the pixel. We represent the saliency values corresponding to each pixel of the image in the form of a heat map which describes the level of contribution of each pixel towards making a final prediction by the model.

Let's discuss some specific examples to have some insight into how these might be capturing features.

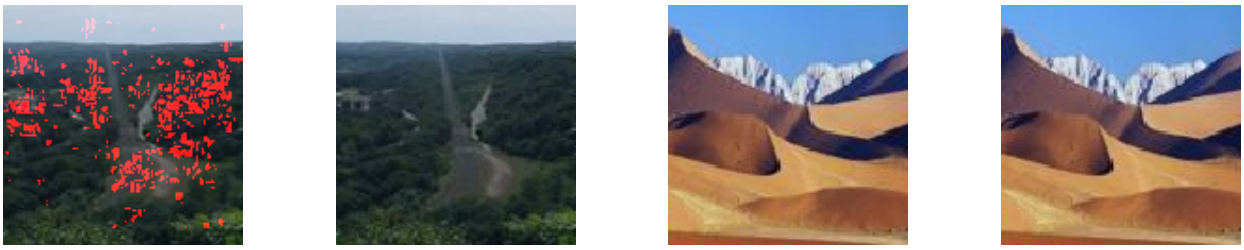
(a) Original Images (b) Saliency Map imposed Images

Figure 10: Saliency Maps

(a) ski slope (b) shed

Figure 11: Saliency Map Information Capture

In **Figure 11**, we can see how the model can capture important information in the images to help its classification while neglecting redundant or useless information. In **Figure 11a**, we can see how the snow is used as the primary source of information to classify the scene, neglecting the very prominent humans. Similarly, as shown in **Figure 11b**, the model only chooses the shed, which is the only object of importance, to make the final prediction.



(a) valley 1 (b) valley 2

Figure 12: Saliency Map Failure Cases

In **Figure 12**, we observe an interesting case where we pick two images of the same classes and see that the model can capture information in the first image. In contrast, it misses the same in the second image. Hence, we can safely assume that the model has misclassified the second image.

Thus we conclude that Saliency Maps are an excellent way to capture how our model perceives our image and how

it can be used to understand and tweak the model to solve misclassifications.

7.2 Adversarial Samples

(a) Original Images

(b) Perturbed Images

Figure 13: Adverserial Sample Generation

The reasoning behind the attack leverages the idea that the model follows the path along gradient descent to minimize the loss. Thus, by following the direction of the gradient w.r.t to the input image (opposite to the direction of gradient descent), we essentially introduce a small perturbation in the input image, which maximizes the loss and confuses the model.

Firstly, we clone the input image and enable the gradients on the clone while disabling the gradients in the original input tensor and the model parameters so as to prevent the accumulation of the gradients. Then, we find the softmax predictions of the model for the given input image. The least confident label is picked up, which we shall mark as the "incorrect" label. We then calculate the gradient along the incorrect label, and take the sign of it, this will mark the direction of our perturbation for one step. It is then scaled by the step size, and finally clamp this in the range of $-\epsilon$ to $+\epsilon$ (where ϵ represent the bounds of perturbation). This will give us the final perturbation which is then added to the image. The above mentioned process is run for a given number of iteration where the output from previous step is used as input for the next step. After all the steps we returned the perturbed image.

$$x_{t+1} = \max\{\min\{x_t + \alpha sign(\nabla_x \mathcal{L}(\theta, x_t, l)), x + \epsilon\}, x - \epsilon\} \tag{2}$$

where, x_t is output at step t

l is least confident label

θ is the model

x is the original input to be perturbed

The perturbations are not directly visible to the naked eye but you can see slight irregularities in the Perturbed Images but nothing which makes the image unrecognisable. Now when we validated these images with the simple CNN model trained in Pytorch and observed a drop in *top-1 validation accuracy* to **39.2%** and *top-5 validation accuracy* to **64.89%**

8 Results

You can find the concise results of all the experiments below:

Model Name	**Epoch**	**Top 1 Accuracy**		**Top 5 Accuracy**		**Final Minimum Loss**
		Training	Validation	Training	Validation	
Simple CNN Pytorch	60	44.92	43.86	74.41	73.32	1.78
Simple CNN Pytorch	10	17.19	19.34	42.96	46.56	3.21
Simple CNN Custom	10	19.42	22.64	46.78	52.03	3.10
Simple ViT	90	41.87	43.18	71.26	72.58	1.86
Custom Model	90	43.17	43.32	72.61	73.09	1.9
PreTrained ResNet	60	88.25	52.11	97.79	80.01	0.27
Adverserial Attack	–	–	39.2	–	64.89	–

9 Conclusion

In this assignment, we first looked in depth at how convolution works and designed our own implementation from scratch. Specifically, we dissected deep neural networks in just two functions: forward pass and backward propagation. We then learned about different parameters for measuring the performance of a model like training accuracy, training time and memory usage. We then contrasted our Convolution Neural Network implementation with PyTorch's inbuilt implementation and concluded that there is much room for improvement to optimize training time and memory. Next, we reviewed the implementation of a Vision Transformer and learned how to design and train a model based on Vision Transformer and how tweaking the different hyper-parameters affect the runtime performance of the model. We also experimented with and designed a custom model with the hope that it could better perform at the task of scene classification using the MiniPlaces dataset. This was compared against a pre-trained ResNet model and provided the findings. We also implemented Saliency Maps which helped us better understand the functioning of the model by looking at what it sees by highlighting the degree of importance of pixels based on their contribution to the prediction task. Finally, we saw the threats to a deep neural network model by generating adversarial samples that fool the model with deceptive data. This led us to understand better how to make such models more robust and defend against adversarial attacks.

10 Teammate Contributions

Apoorva Kumar	Vaibhav Nitnaware	Varun Kaundinya
Custom Convolution Code	Custom Convolution Code	Training SimpleNet Pytorch
Training SimpleNet Custom	Training SimpleNet Pytorch	SimpleVit Code
SimpleVit Code	SimpleVit Training	Custom Model Design
Custom Model Design	Custom Model Design	Custom Model Code
Saliency Map Code & Generation	Custom Model Code	Custom Model Training
Adversarial Code & Generation	ResNet Training	Adversarial Code & Generation
Respective Graphs and Sections in Report		

11 References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[2] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2017.

[3] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps, 2013.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[5] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.