BMI / CS 771 Homework Assignment 2
Convolution and Transformer Networks

Siddharth Baskar sbaskar2@wisc.edu

Karan Vikyath Veeranna Rupashree veerannarupa@wisc.edu

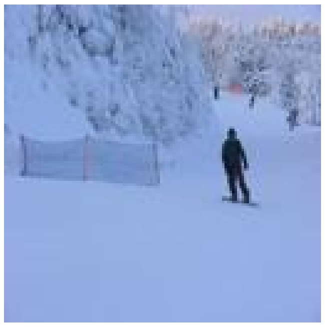
Anudeep Kumar kumar256@wisc.edu

1 Introduction

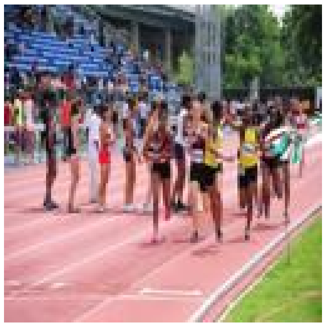
This assignment was focused on learning image classification especially scene recognition using some interesting networks namely Convolutional networks and Transformer neural networks. We used Pytorch for the task. We first trained Pytorch's Convolution Network and then replaced the forward and back-propagation with our own implementation. We implemented a Simple Vision Transformer. A saliency map was built to get the information regarding important part of image for classification. At the end adversarial models were generated that helped confuse the model.

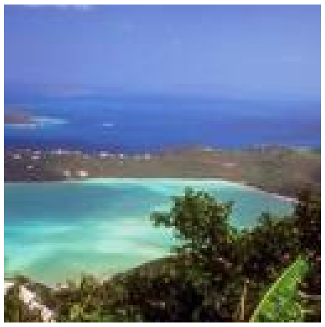
2 Dataset

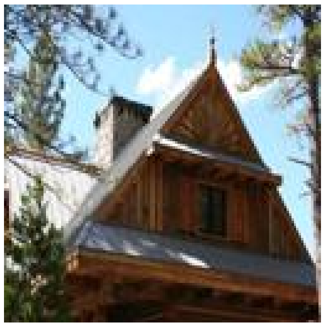
The Dataset used for this is the MiniPlaces Dataset from MIT. It consists of 120k images from 100 different categories Validation set draws from this and has 10k images


(a) Ski Slope

(b) Inside of a Bus


(c) Marathon


(d) Island


(e) House

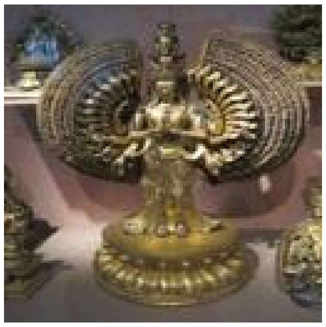

(f) Sculpture

Figure 1: Sample from Dataset

3 Understanding Convolution

This section helped us understand the technique of convolution. To bypass the complexity and exploit the ease of matrix multiplication the pytorch fold and unfold function is used.

3.1 Forward Propagation

We first calculated the height and width of the output by the formula $[\frac{H+2P-K}{S}]$ and same for width. The we use the unfold() function from pytorch. It was interesting to understand unfold function which created a tensor with the possible matrices based on given kernel and stride. After obtaining that, it was easy to do a matrix multiplication operation. We then folded it back to correct size using the fold operation.
We also save the unfolded features and weights for back propagation.

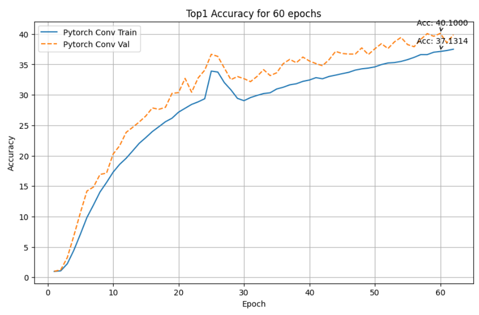
3.2 Backpropagation

We obtain the unfolded feature vector and weights from forward prop. We then unfold the output grad and multiply it after transposing to match the dimensions for matrix multiplication with weights we obtained in forward pass. This operation gave us the unfolded gradient of input. We fold it to get the gradient of input. Similarly the gradient of weight is obtained by multiplying input with gradient of output. Everything is folded to expected size and returned. At the end the gradient are summed over mini batches and reshaped to match the kernel size.

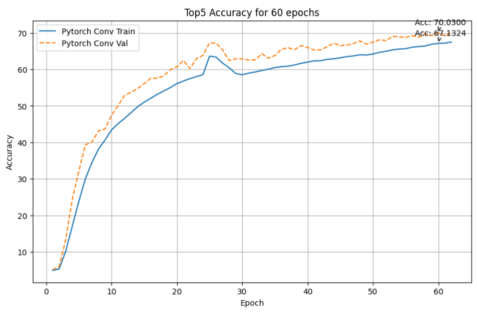
3.3 Design and Train the Deep Neural Network

3.3.1 A Simple Convolutional Network

The code was already given in student_code.py. We train the model for 60 epochs. The graphs are attached below. We obtained top-1 validation accuracy as 40.100% and top-5 accuracy as 70.030%



(a) Pytorch Top1 Accuracy



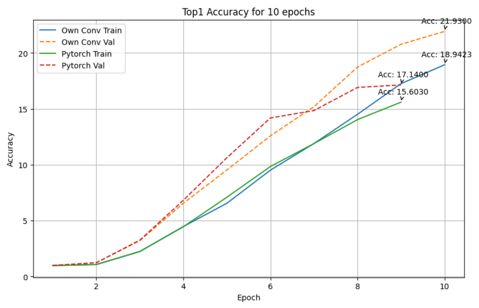
(b) Pytorch Top5 Accuracy

Figure 2: Accuracy For Pytorch Model

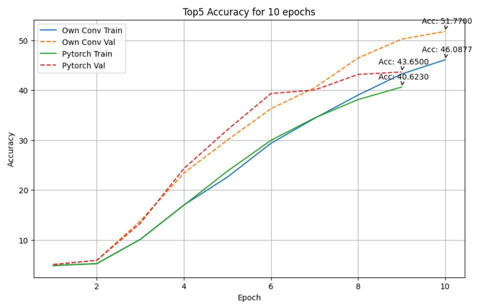
3.3.2 Train with your own Convolution

Here we replace the forward and backward propagation with the code that we wrote.
We trained our model for just 10 epochs. We can see the 10 epoch accuracy for both models below:

Acc/Model	Pytorch	Own
Top-1 Acc	20.260%	21.930%
Top-5 Acc	47.360%	51.770%



(a) Pytorch vs Custom Top1 Accuracy

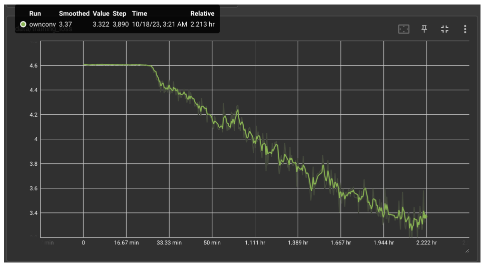


(b) Pytorch vs Custom Top5 Accuracy

Figure 3: Accuracy For 10 epoch Custom vs Pytorch Model

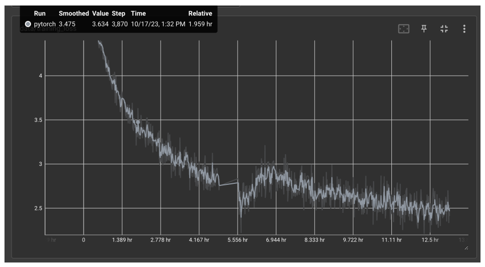
3.3.3 Time for Training

We can see in the figure attached Our own implementation completes 3890 steps in 2.213 hours and Pytorch implementation completes same steps in less time. Fig 4.c shows that the loss is comparative for both implementation



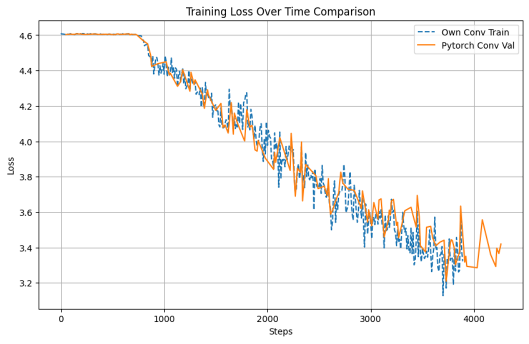
Run	Smoothed	Value	Step	Time	Relative
● ownconv	3.37	3.322	3,890	10/18/23, 3:21 AM	2.213 hr

(a) Own Convulution Loss vs Time



Run	Smoothed	Value	Step	Time	Relative
● pytorch	3.475	3.634	3,870	10/17/23, 1:32 PM	1.959 hr

(b) Pytorch Implementation Loss vs Time

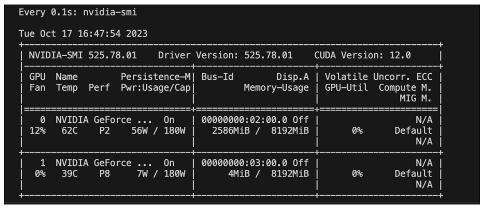


(c) 10 Epoch Loss comparison

Figure 4: Loss and Time Comparison

3.3.4 Memory

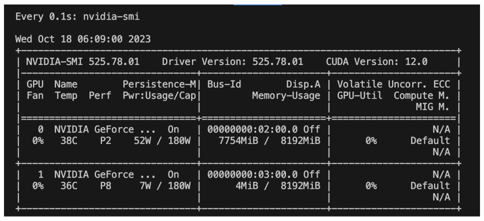
The memory usage for Pytorch and Own convolution has a drastic difference. Pytorch model took 2586/8192MB while own convolution took 7754/8192 MB.



```
Every 0.1s: nvidia-smi

Tue Oct 17 16:47:54 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.78.01    Driver Version: 525.78.01    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:02:00.0 Off |                  N/A |
| 12%   62C    P2    56W / 180W |   2586MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  On   | 00000000:03:00.0 Off |                  N/A |
|  0%   39C    P8     7W / 180W |      4MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

(a) Pytorch Memory Usage



```
Every 0.1s: nvidia-smi

Wed Oct 18 06:09:00 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.78.01    Driver Version: 525.78.01    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:02:00.0 Off |                  N/A |
|  0%   38C    P2    52W / 180W |   7754MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  On   | 00000000:03:00.0 Off |                  N/A |
|  0%   36C    P8     7W / 180W |      4MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

(b) Own Memory Usage

Figure 5: Memory Usage Comparison

4 Vision Transformer

4.1 Architecutre

The `SimpleViT` class delineates a straightforward implementation of the Vision Transformer (ViT) architecture utilizing PyTorch, as described in the paper "Exploring Plain Vision Transformer Backbones for Object Detection." Initially, the architecture extracts embeddings from the input image via a patch embedding mechanism, executed through a convolution operation by setting the stride equal to the kernel size in the `PatchEmbed` component. Should the `use_abs_pos` parameter be set to `True`, learnable positional embeddings are added, enhancing the model's ability to recognize spatial hierarchies in the data.

Following the embedding phase, the architecture propels the data through a series of transformer blocks, orchestrated in a `nn.ModuleList`. Each block in this sequence is a concoction of a batch normalization layer, an attention layer, and another batch normalization layer. In the attention layer, the architecture computes projections for Query (Q), Key (K), and Value (V) before melding Q and K through a softmax non-linearity, which is then multiplied by V, succeeded by a linear projection, as captured by the equation:

$$\text{SA}(X) = \text{softmax}(XW_Q(XW_K)^T)XW_VW_O$$

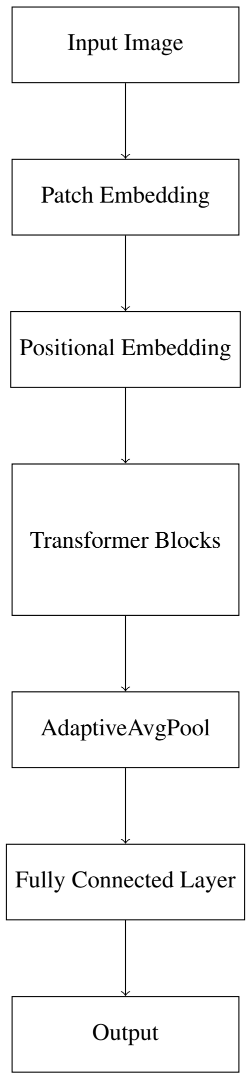
where $W_Q, W_K, W_V,$ and W_O are learnable weights.

The architecture also entertains a `window_size` parameter to orchestrate local and global attention across patches. In the absence of this parameter, global attention is bestowed upon all patches by default. Conversely, when specified, the `window_block_indexes` parameter designates which blocks employ local attention, facilitating a blend of local and global attention. This amalgamation aims to pare down computational complexity while sustaining a vast receptive field. Moreover, a drop path setting is invoked within the transformer blocks, simulating drop-out to mitigate overfitting, as dictated by the `drop_path_rate` parameter.

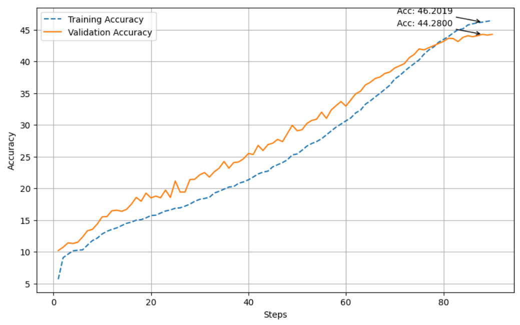
Subsequent to the transformer block sequence, the architecture channels the output through an average pooling layer, epitomized by `nn.AdaptiveAvgPool3d`. This layer is instrumental in reducing the dimensionality of the data while encapsulating essential information. The pooled output is then ushered into a fully connected layer, materialized by `nn.Linear`, paired with a softmax classifier to predict class labels. The output dimensions of this layer echo the number of classes specified. This streamlined architecture is concocted to furnish a robust yet simplified backbone for object detection tasks, as illustrated in Figure 6a.

4.2 Training and Inference

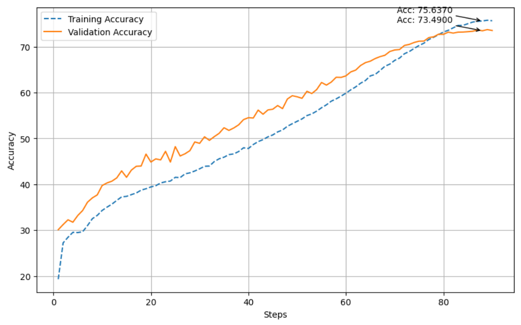
The model was trained for 90 epochs with a learning rate of 0.01 and a weight decay of 0.05. It achieved a top-1 validation accuracy of **44.28%** and top-5 validation accuracy of **73.49%** were recorded. These numbers are comparable to the accuracy recorded for simple CNN. Figure 5 depicts the training for the transformer which shows the plot for epochs vs accuracy.



(a) Simple Vision Transformer Architecture



(b) Top-1 accuracy



(c) Top-5 accuracy

Figure 6: Simple Vision Transformer

5 Design your own Network: Custom ViT - An Inception style model

5.1 Architecutre

The *CustomViT* is an innovative adaptation of the Vision Transformer (ViT) architecture that incorporates an “Inception Style” design. At its core, it processes the input image through multiple parallel pathways with varying patch resolutions, akin to the multi-scale feature processing in the Inception architecture.

During initialization, several hyperparameters are set, including the input image size, the number of output classes, patch sizes, and the embedding dimension. Notably, this architecture supports both absolute and window-based positional embeddings. The latter allows for local attention mechanisms within specified window sizes in designated transformer blocks.

The architecture comprises two parallel processing blocks: one for 16x16 patches and another for 4x4 patches. Both blocks employ patch embedding layers that convert image patches into embeddings. When using absolute positional embeddings, these embeddings are then added with the respective positional embeddings. The transformed embeddings are then passed through a series of transformer blocks of depth = 8 and number of attention heads = 6.

After processing the image through both pathways, the features are adaptively average pooled and concatenated. This combined feature vector then passes through a linear head to produce the final class predictions.

In essence, this custom Vision Transformer captures multi-scale features from the input image, leveraging the strengths of both the transformer and inception-style architectures. The design aims to combine the global reasoning capabilities of transformers with the multi-scale processing of the Inception network.

5.1.1 Changes in main.py

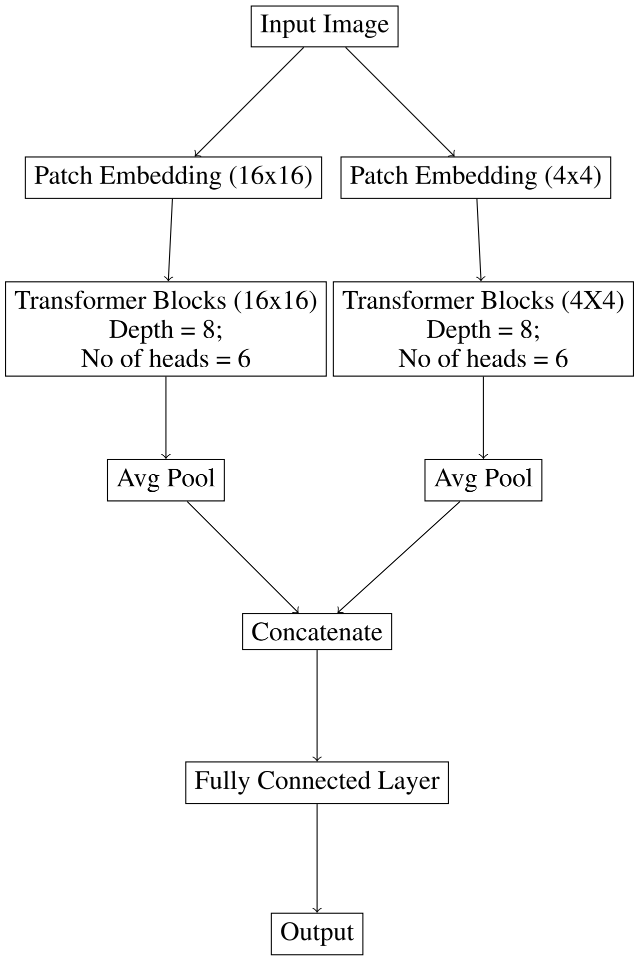
To test out Own model we made some modifications to the `main.py` to facilitate easier changes. The changes are mentioned below.

- Added Argument Parser option to directly access out model using the option `-use-custommodel` [Line 137-138]
- Added extra condition in model chooser to choose out custom model [Line 177-178]
- Added custom model to the condition choosing optimizer [Line 191]
- Added custom model to clipping gradients condition when training [Line 414]

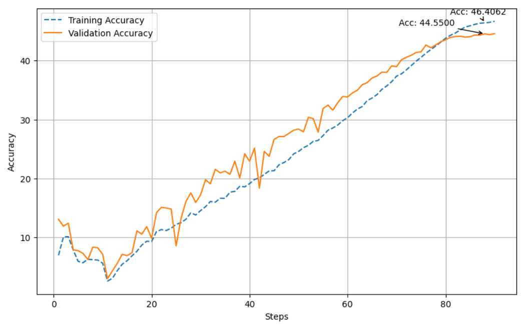
We thoroughly checked to ensure that these modifications don’t break any previous functions.

5.2 Training and Inference

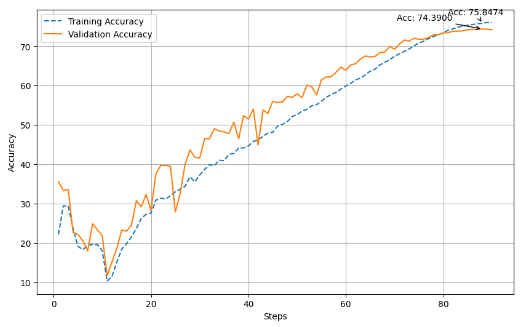
The model was trained for 90 epochs with a learning rate of 0.01 and a weight decay of 0.05. It achieved a top-1 validation accuracy of **44.55%** and top-5 validation accuracy of **74.39%** were recorded. A slightly higher accuracy was achieved than the Simple ViT. Figure 7 depicts the training for the transformer which shows the plot for epochs vs accuracy.



(a) Custom Vision Transformer: Inception Style Architecture



(b) Top-1 accuracy

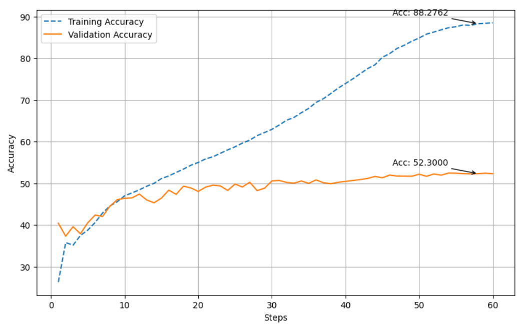


(c) Top-5 accuracy

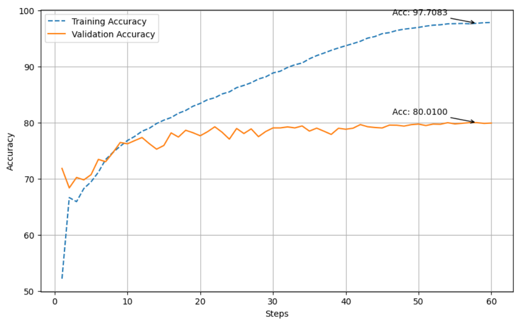
Figure 7: Custom Network

6 Fine-Tune a Pre-trained model

After being pre-trained, the Resnet18 model showcased notable improvements over its 60-epoch training, achieving a top-1 validation accuracy of **52.3%** and a top-5 validation accuracy of **80%**. This enhancement is evident when compared to the results from a basic CNN and our unique model, as depicted in Figure 8. An early stabilization in the validation curve for ResNet suggests the possibility of over-optimization and a looming risk of overfitting. These findings highlight the power of pre-trained models, especially when tailored for tasks that are closely related. Remarkably, by tweaking just one layer, the model managed to outclass several of its complex counterparts.



(a) Top-1 accuracy



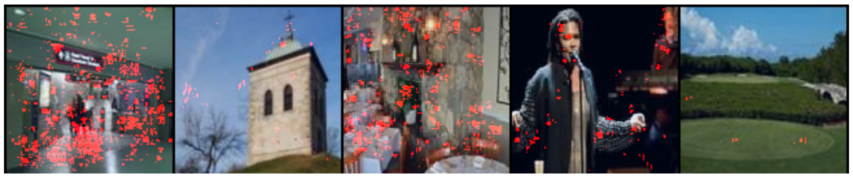
(b) Top-5 accuracy

Figure 8: Pre-trained model: ResNet18

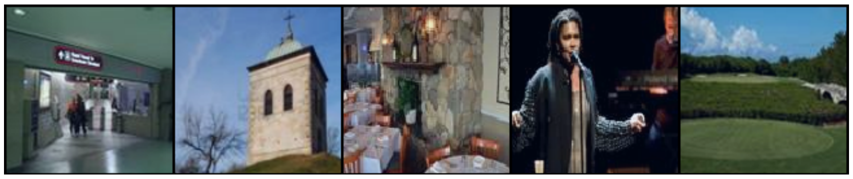
7 Attention and Adversarial Samples

7.1 Saliency Maps

Saliency maps depicts the features or regions that have the most influence on the training of a model and the model's output. Fig 9(a) depicts the highlighted features. In the method that was used to generate the saliency maps first the model parameters are frozen. This is done to prevent the model from updating the weights during subsequent operations so as to assess the impact of the input on the model's output without altering the model itself.



(a) Saliency Map imposed Images



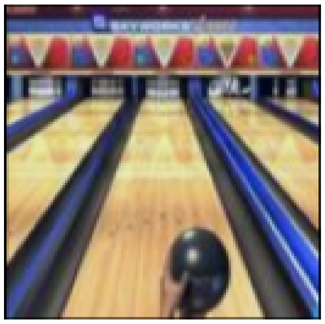
(b) Original Images

Figure 9: Saliency Maps

Then, the model is switched to evaluation mode so that it can compute saliency not while training. The code then passes the input through the model to obtain predictions.

After that, loss is computed based on the model's softmax output and the most confident class. By backpropagating this loss, we compute the gradients with respect to the input data. Then, the code calculates the saliency map by taking the absolute value of the gradients with respect to the input data using and returns it.

Sometimes there is also a failure in the creation of these saliency maps where the features are not highlighted as shown in Figure 10


(a) Original Image

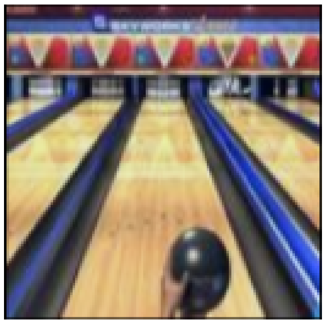
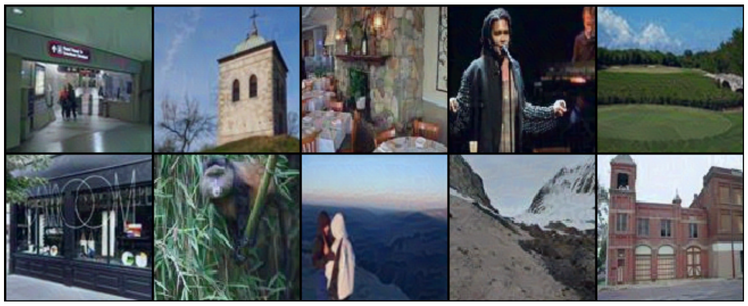

(b) Failed Case Image

Figure 10: Failure Case of Saliency Map

7.2 Adversarial Samples

The attack strategy is based on the concept that a model tries to reduce its loss by following the gradient descent path. In other words, it moves in the direction of the gradient that minimizes the loss. Therefore, when we move in the opposite direction of the gradient with respect to the input image, we introduce a slight perturbations in the input. This perturbation has the effect of maximizing the loss and, in turn, confusing the model.


(a) Images with perturbations

(b) Original Image

Figure 11: Adverserial Sample Generation

This method starts by freezing the model's weights. This is done because the goal is to make the model robust to adversarial perturbations, and allowing the model's weights to change during this process would make it more challenging to train and interpret the results.

The code then runs in an iterative loop where, in each step, it computes adversarial perturbations on the input data and updates it. This iterative approach allows the perturbations to gradually steer the input data in a direction that is both adversarial and realistic.

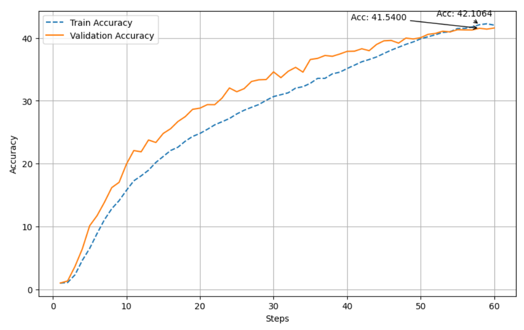
Then the adversarial image is clamped to a specified range and detached. Clamping ensures that the perturbations don't exceed certain bounds, which is essential for maintaining a realistic appearance. Detaching prevents further gradient flow from the adversarial image, isolating it from subsequent training steps.

Then the loss is computed and the model backpropagates this loss to update the adversarial example. By doing this, the model is effectively learning to defend against adversarial attacks during training. The adversarial perturbations that maximize this loss are making the model more resilient to similar attacks at test time.

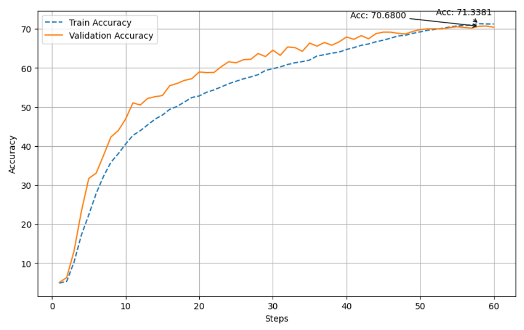
The examples of adversarial samples can be seen in Figure 11. It can be observed in fig 11(A) that the perturbations are not visible to the naked eye but is rather detected by the model. This was validated when a drop in top-1 validation accuracy to **36.210** and top-5 validation accuracy to **61.980** was observed.

8 Adversarial Training

In this experiment, we utilized the SimpleNet architecture to train a Convolutional Neural Network (CNN) with adversarial samples, employing the Projected Gradient Descent Adversarial (PDA) attack function to generate adversarial examples. The number of steps was reduced to 5, while adding a new argument(adversarial training = True) to the forward function of SimpleNet, to prevent the model from going into an infinite loop while creating adversarial samples.The training was conducted over 60 epochs, and the performance was assessed using both top-1 and top-5 accuracy metrics. The figures below illustrate the training progression for both accuracy metrics over the 60 epochs. These numbers are comparable to the training obtained withoit adversarial training in Figure 2.



(a) Top-1 accuracy



(b) Top-5 accuracy

Figure 12: SimpleNet wtih Adversarial Training

The subsequent table delineates the enhanced validation accuracy against PDA attacks achieved through adversarial training. The results exhibit a noteworthy improvement in the model's resilience to adversarial attacks, underscoring the efficacy of adversarial training in fortifying the CNN against malicious input perturbations. Through this adversarial training paradigm, the SimpleNet model has demonstrated an augmented capability to withstand PDA attacks, thereby contributing to the broader endeavor of fortifying machine learning models against adversarial threats.

Validation Accuracy	Without Adversarial Training	With Adversarial Training
Top-1	36.21%	40.78%
Top-5	61.98%	69.40%

9 Results

Model Name	**Epoch**	**Top 1 Accuracy**		**Top 5 Accuracy**	
		Training	**Validation**	**Training**	**Validation**
Simple CNN Pytorch	60	44.92	43.86	74.41	73.32
Simple CNN Pytorch	10	17.19	19.34	42.96	46.56
Simple CNN Custom	10	19.42	22.64	46.78	52.03
Simple ViT	90	46.20	44.28	75.63	73.49
Custom Model	90	46.41	44.55	75.85	74.39
PreTrained ResNet	60	88.28	52.30	97.70	80.01
Adverserial Attack	–	–	36.21	–	61.98
Adverserial Training	60	42.10	41.52	71.34	70.68
Adverserial Attack with Adv Training	–	–	40.78	–	69.40

10 Conclusion

This assignment has contributed a lot in developing a deeper understanding of Convolution Neural Networks, Vision Transformer and Saliency maps which gave an insight of how the model understands things by highlighting important pixels.
We started with implementing forward and backward propagation from scratch. This helped us get a better grasp of internal math of the network. We then compared our implementation with Pytorch's implementation which had shed light on how much optimized Pytorch actually is compared to basic implementation. The Memory, Training Time and Loss comparison clearly backs the above statement.
Next was getting under the hood of Vision transformers. Playing around with different parameters showed the performance changes. We implemented a custom model for scene classification on MiniPlace Dataset. We compared this result against pre-trained ResNet and the findings were reported.

Then, we implemented Saliency Maps which was used to help understand which features from the image is detected by the model during its functioning. This was done by highlighting the specific pixels that are important during this process. After that, adversarial images were generated that usually provide a threat to the model as it fools it due to the perturbations introduced in the image. We found that the accuracy generated by the model when these adversarial samples were fed into it went down as compared to the initial accuracy. Finally, the model was trained on adversarial samples and it was observed that the model was much more resilient and had higher resiliency to withstand the PDA attacks.

11 Teammate Contributions

Anudeep Kumar	**Karan Vikyath Veeranna Rupashree**	**Siddharth Baskar**
Custom Convolution Code	SimpleViT Training	Saliency Maps
Training SimpleNet own Conv	Custom Model Design	Adversarial Attack
Training SimpleNet with Pytorch wit	Resnet Training	Adversarial Training

References

[1] Convolution Matrix Multiplication

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In ICLR, 2015.

[3] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In ICLR, 2014.

[4] K. Simonyan and A. Zisserman. Very deep convolutional networks for largescale image recognition. In ICLR, 2015

[5] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In ICLR, 2018.

[6] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In International Conference on Learning Representations, 2018.