3.1 Convolutional Neural Network

Code and experiments for this section are contributed by Jiangyi Liu.

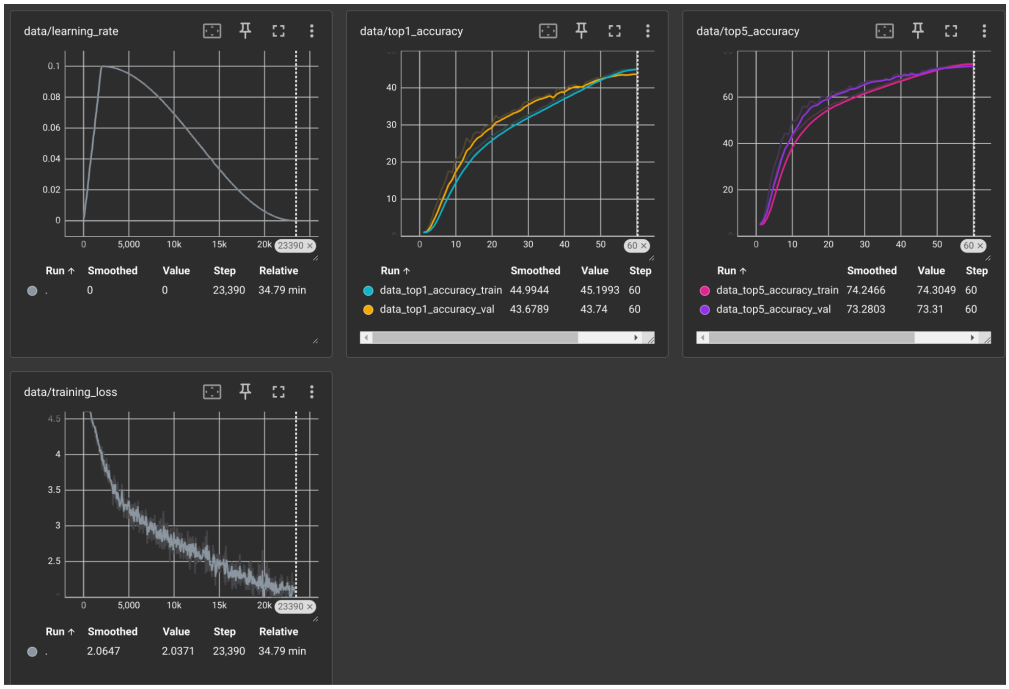
Section 3.1.1

For the implementation of convolution operator, I refered to the slides for Lecture 4. The code for forward propagation uses unfold to effectively turn "patches" in the image into column vectors. For backward propagation, I used the formulas on page 59 and 62 of the Lecture 4 slides. The calculated results should be reshaped or folded to confirm to dimensions of weight / input tensors. Note that since no minibatch dimension exists for the weight tensor, we take a sum on the first dimension to eliminate minibatch dimension in the gradient of weight. This is according to the derivation rules of multi-variable calculus.

Section 3.1.2

A Simple Convolutional Network

The curves are below. The evaluated results are: 44.35% accuracy when taking 1-top, and 73.89% accuracy when taking 5-top.

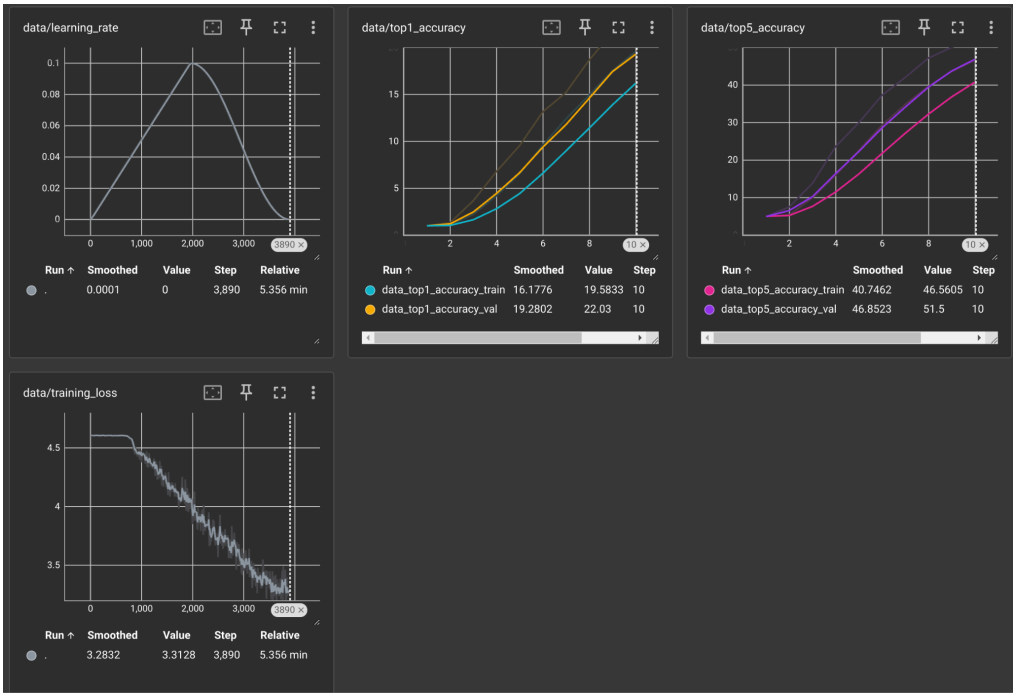


Train with Your Own Convolutions

When using our custom convolution operators, we observed:

- The VRAM usage (4.93GiB) is larger than the original version (2.46GiB). This is probably due to the fact that the PyTorch convolution operator is in-place, saving memory usage.
- The training speed and loss curve are similar.

We list the curves below. The evaluation result is 22.53% accuracy when taking 1-top, and 51.96% accuracy when taking 5-top. Note that this result is similar to the original version, because we only trained the model for 10 epochs, not 60.



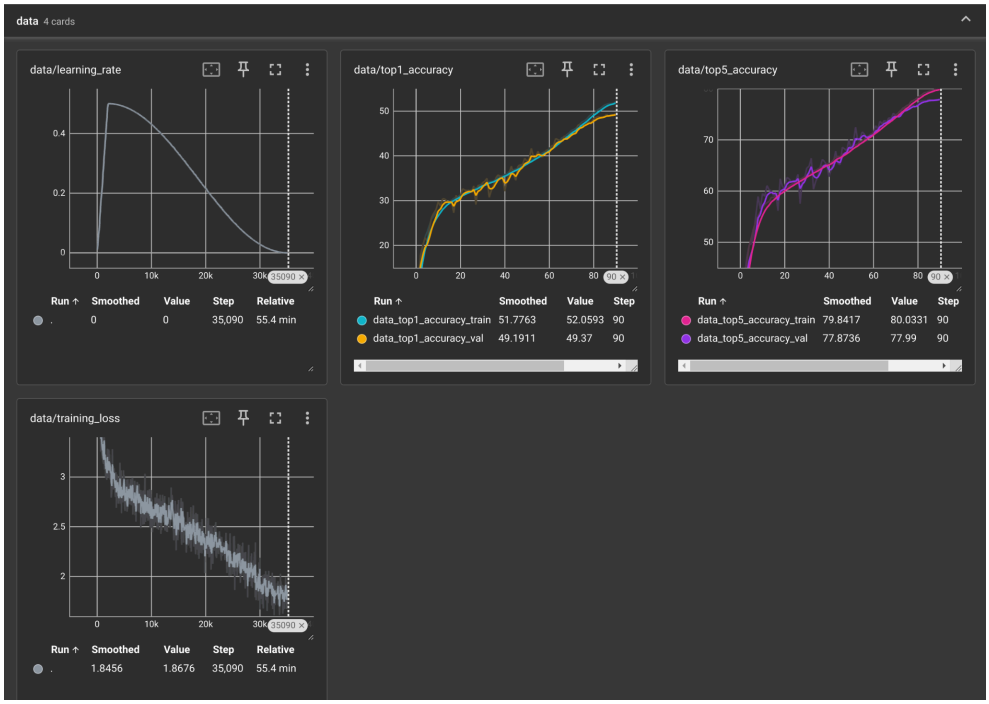
Design Your Own Convolutional Network

The following attempts were made, after several attempts:

- 3 batch-norm operators are inserted after conv1/2/3 blocks. (design change)
- Learning rate and epochs are changed to lr=0.5, epochs=90. (training scheme)

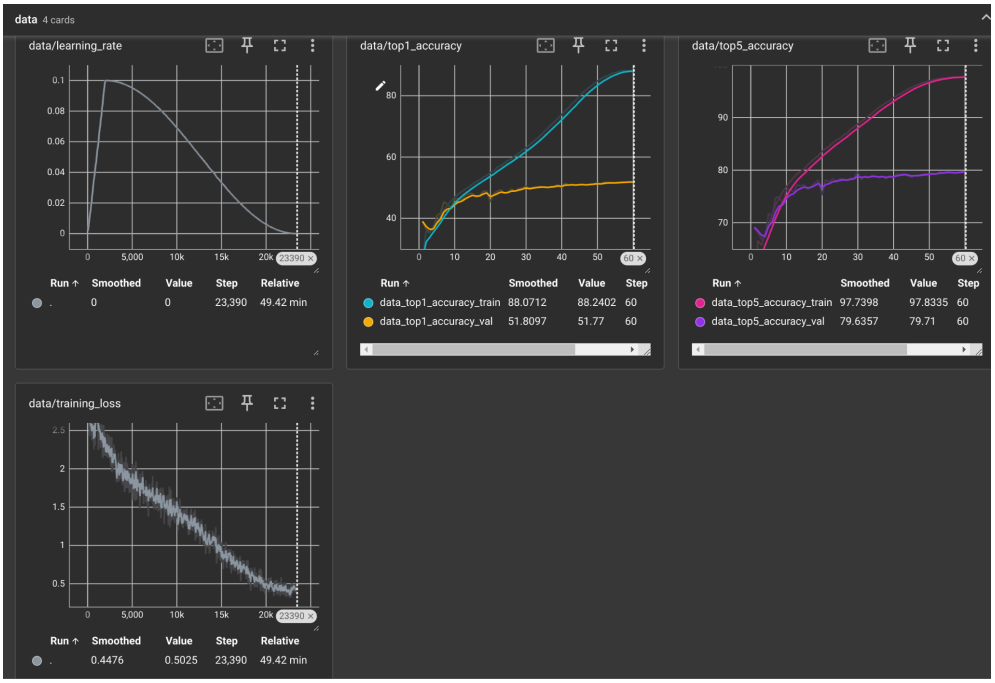
We changed the training scheme because we noted that the accuracy and loss doesn't converge after 60 epochs with original training parameters.

The curves are listed below. We observed a significant performance boost, with 49.73% accuracy for top-1 and 78.33% accuracy for top-5.



Fine-Tune a Pre-trained Model

The fine-tuning results for resnet-18 are listed below. With Resnet-18, we got 52.90% accuracy for top-1 and 80.26% accuracy for top-5.



Comparison between ResNet and Our Best Model

- For accuracy curves, the pre-trained ResNet starts from around 40% and 70%, while our best model starts from 0%. ResNet's finetune quickly converges, while our best model's accuracy curves still don't converge after 90 epochs
- The train loss of ResNet is lower than that of our best model.
- The accuracy curves (both 1-top and 5-top) from ResNet are both around 2% higher than those of our best model.

3.2 Vision Transformers

Code and experiments for this section are contributed by Venkata Abhijeeth Balabhadruni.
In the first part of this section, we have filled the parts of code in **Attention** class to complete the self attention block. After that, we reviewed the **TransformerBlock** code and understood how to use it to do local self attention which significantly reduces the runtime complexity.

Then in the next part we have designed and implemented a vision transformer in the class **SimpleViT.**

Model Design :

The following is our design for the vision transformer we implemented :

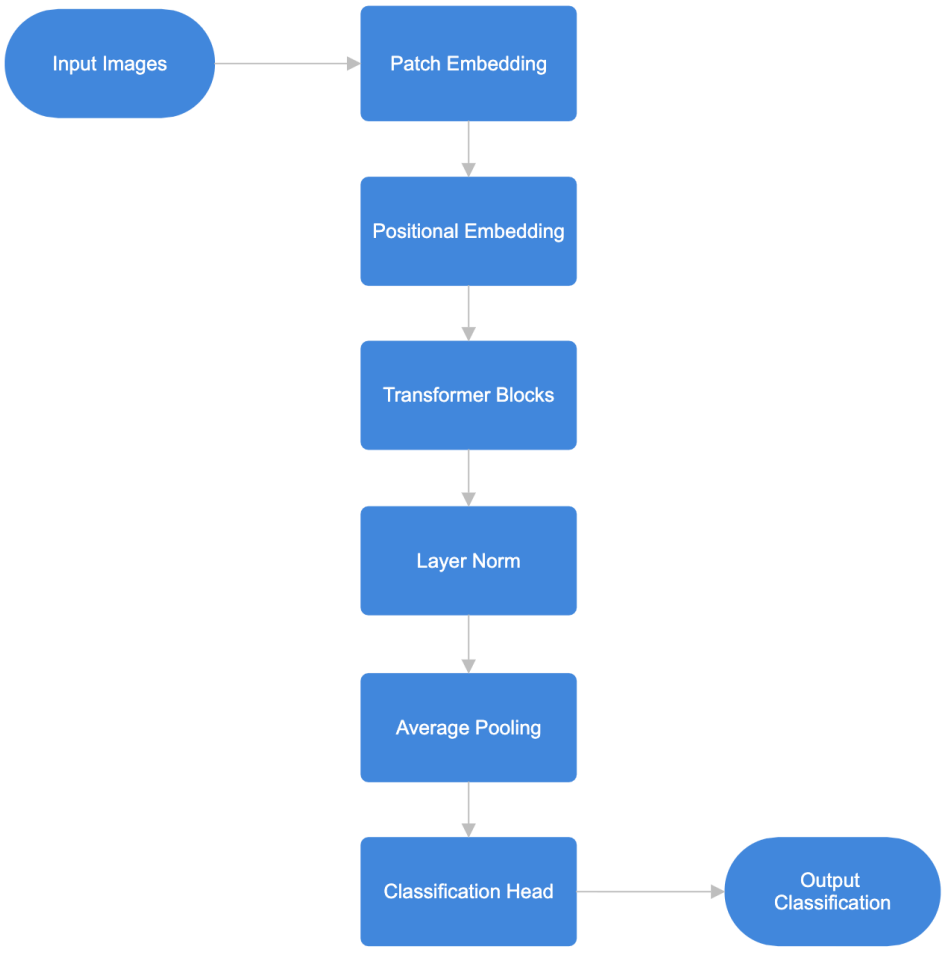


Figure 1 : Architecture of SimpleViT

SimpleViT takes a batch of 256 images as input. (Each image has 3 channels and 128x128 pixel size).

Patch Embedding Layer:

The input images are divided into non-overlapping patches of size 16x16. Each patch is flattened into a vector and then linearly projected into a higher-dimensional space (embedding dimension: 192). This process effectively transforms the image into a sequence of patch tokens. Tensor shape after this layer will be [256, 8, 8, 192]

Positional Embeddings:

Positional embeddings are added to the patch embeddings to retain positional information, as self-attention lacks inherent spatial awareness. This ensures that the spatial relationships between patches are preserved. Tensor shape after this layer will be [256, 8, 8, 192]

Transformer Blocks:

The core of the model comprises multiple Transformer blocks, where each block includes:

- Multi-Head Window-Based Self-Attention (MSA): Each patch embedding is attended to by other patches based on similarity scores. Multiple heads allow the model to focus on different aspects of the relationships between patches. Attention is limited to smaller local windows for some blocks, reducing computational complexity.
- Layer Normalization: Applied before each sub-layer to stabilize the training.
- Feedforward Network (MLP): A two-layer fully connected network with a GELU activation that processes the output of the attention mechanism. The hidden layer dimension is scaled by a factor (mlp_ratio=4.0) of the embedding dimension.
- Residual Connections: Skip connections help maintain stable gradient flow.

Tensor shape after this layer will be [256, 8, 8, 192]

Layer Normalization (Post-Transformer Blocks):

After the sequence passes through the Transformer blocks, another layer normalization step is applied across the embedding dimension. Tensor shape after this layer will be [256, 8, 8, 192]

Average Pooling:

The patch embeddings are pooled into a single feature vector for each image by averaging the sequence along spatial dimensions. Tensor shape after this layer will be [256, 192]

Classification Head:

The pooled feature vector is passed through a fully connected layer, which serves as the classification head. This produces output logits of shape (B, num_classes), representing the predicted class scores for each image. Tensor shape after this layer, i.e the output size is [256, 100]

Training Scheme :

Command used : *python ./main.py ../data --epochs=90 --wd 0.05 --lr 0.01 --use-vit*

The model is trained for 90 epochs, Increasing the number of epochs allows the model to train for a longer duration, which helps it to better learn from the data and improve its performance. However, it also increases the risk of overfitting if the model becomes too tailored to the training data.

Here, a weight decay value of 0.05 is used, which prevents the model from learning overly large weight values that could lead to overfitting. It acts as a regularization technique to keep the model generalized and prevent it from memorizing the training data too closely.

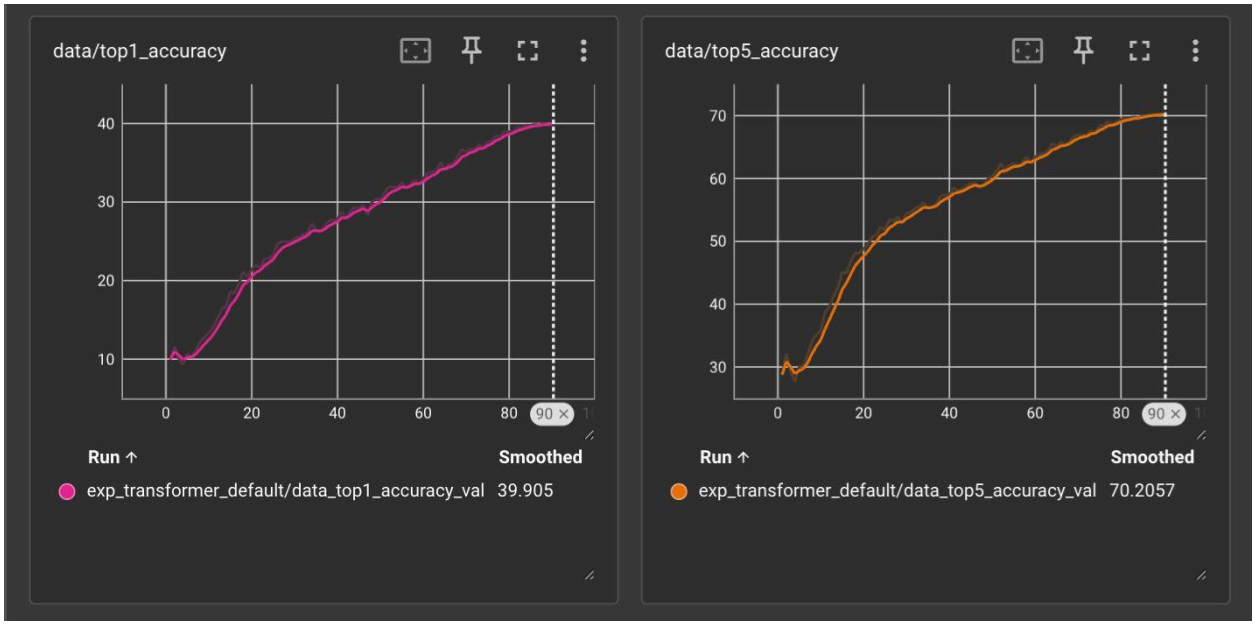
A learning rate of 0.01 is relatively moderate, allowing the model to make steady progress without overshooting optimal solutions.

Experiments :

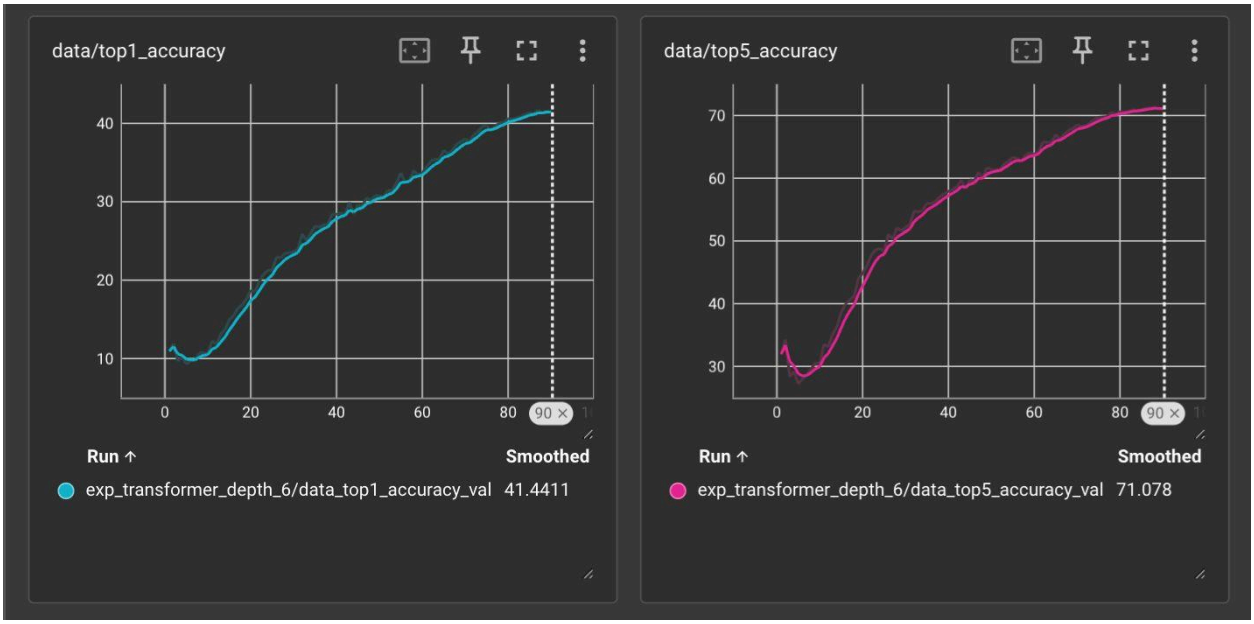
Default parameters for **SimpleViT** Model :

Input image size = 128,
Number of object categories = 100,
Patch size = 16,
Number of input image channels = 3,
Patch embedding dimension = 192,
Depth of ViT = 4,
Number of attention heads in each ViT block = 4,
Ratio of mlp hidden dim to embedding dim = 4.0,
Learnable bias to query, key, value = True,
Stochastic depth rate = 0.1,
Normalization layer = nn.LayerNorm,
Activation layer = nn.GELU,
Absolute positional embeddings = True,
Window size for local attention blocks = 4,
Indexes for blocks using local attention = (0, 2)

First we have run the transformer model with the given default parameters, and ran it with the above mentioned training scheme. With this configuration we got an top1_accuracy of 39.905% and top5_accuracy of 70.2057%. The following graphs show number of epochs vs accuracy :



Then, we have increased the depth of ViT from 4 to 6, keeping the other parameters and the training scheme same, to see if it's able to improve the model performance. We got an top1_accuracy of 41.44% and top5_accuracy of 71.078%. The following graphs show number of epochs vs accuracy :



Increasing the number of transformer blocks from 4 to 6 led to a slight improvement in both top-1 and top-5 accuracy, suggesting that deeper models can potentially learn more expressive features.

However, the improvement was minimal, indicating that increasing model depth alone might not be sufficient to significantly boost performance. Further optimization, such as tuning other hyperparameters (e.g., learning rate, weight decay) or using advanced techniques to generate even more training data with data augmentation, might be necessary for a more substantial performance gain.

3.3 Adversarial Samples

Code and experiments for this section are contributed by Sujay Chandra Shekara Sharma.

In the first part of this section, we focussed on generating adversarial samples with the help of the **PGDAttack**. For the implementation of the **PGDAttack,** I mainly referred to the slides from lectures 9-10 and the instructions in the assignment readme.

We first clone the input image and then generate the adversarial sample from the cloned image iteratively over several steps. The model's logits (raw output scores) are calculated for each iteration of the adversarial sample. The label corresponding to the least confident prediction is then identified and used as a proxy for the incorrect label. We then compute the loss and the

gradients w.r.t the input. Finally, the sample is perturbed using the **Fast Gradient Sign Method (FGSM)** and the result is clipped to be within an **epsilon** range of the input image.

After implementing the **PGDAttack**, we tested it out on the **SimpleNet** model while varying the attack parameters. The results obtained are as follows:

Row	Test Scenario	Top-1 Validation Accuracy	Top-5 Validation Accuracy
1	SimpleNet (No Attack)	43.6789	73.2803
2	SimpleNet (10 steps, 0.1 epsilon, 0.01 alpha)	39.970	64.890
3	SimpleNet (20 steps, 0.05 epsilon, 0.01 alpha)	40.270	66.380
4	SimpleNet (100 steps, 0.01 epsilon, 0.01 alpha)	42.880	72.700
5	SimpleNet (20 steps, 0.1 epsilon, 0.01 alpha)	39.520	62.730
6	SimpleNet (20 steps, 1 epsilon, 0.01 alpha)	39.050	61.130
7	SimpleNet (10 steps, 0.2 epsilon, 1 alpha)	9.790	24.260

As you can see from the table, running the attack with the default hyperparameters (10 steps, 0.1 epsilon, 0.01 alpha) led to a **~4%** drop in **top-1** validation accuracy and a **~9%** drop in **top-5** validation accuracy.

After this, we tried to increase the number of steps in the attack while simultaneously decreasing the epsilon bound. This led to the attack being less effective as shown in rows **3** and **4**. This

seems to indicate that the error-bound (epsilon) has more of an impact compared to increasing the number of steps in the attack beyond 10.

We then tried to increase both the number of steps in the attack while also increasing the epsilon bound (row **6**). Our expectation was that this should lead to a significant drop in accuracy. However, the attack was only slightly more effective compared to using the default hyperparameters. This made us realize that the step size or alpha was too small for an increase in epsilon to be impactful.

Upon modifying alpha to 1 from 0.01 and increasing epsilon to 0.2, we managed to get a single-digit validation accuracy for top-1.

Below are some of the images that we obtained after the adversarial attack:

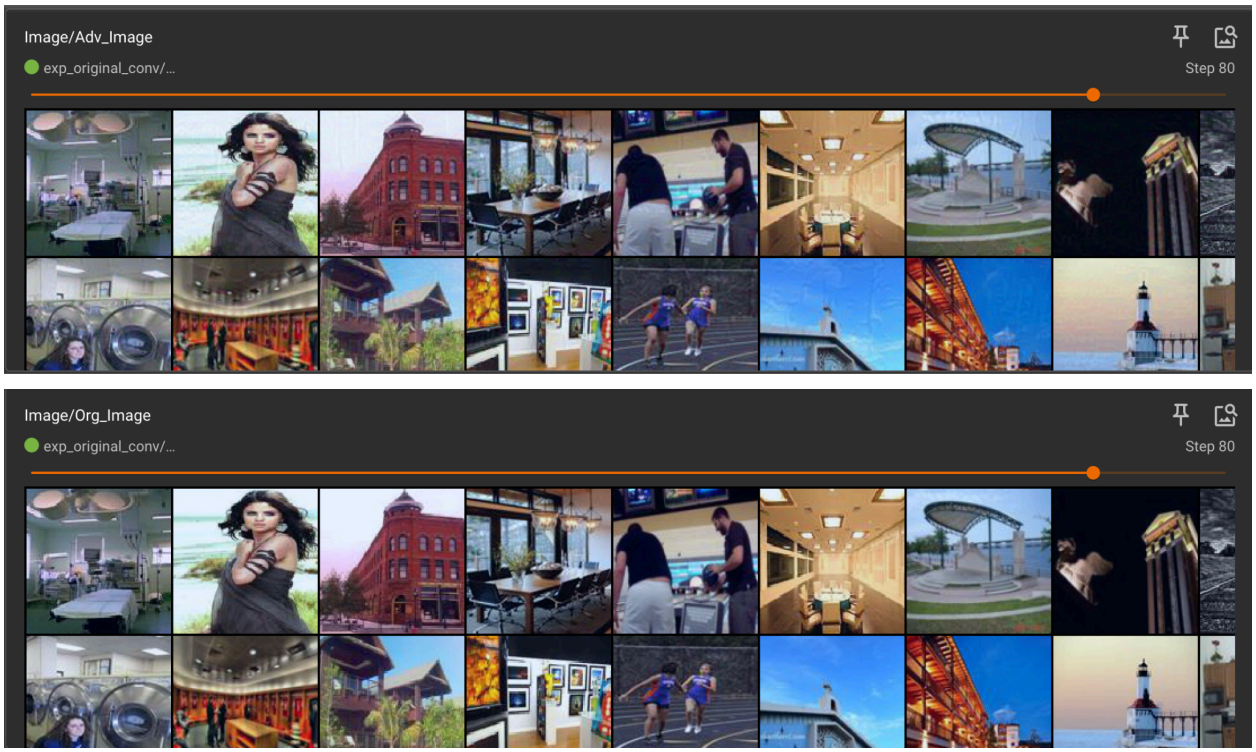


Figure 1: Adversarial images compared to the original images for the attack on SimpleNet with default parameters (10 steps, 0.1 epsilon, 0.01 alpha)

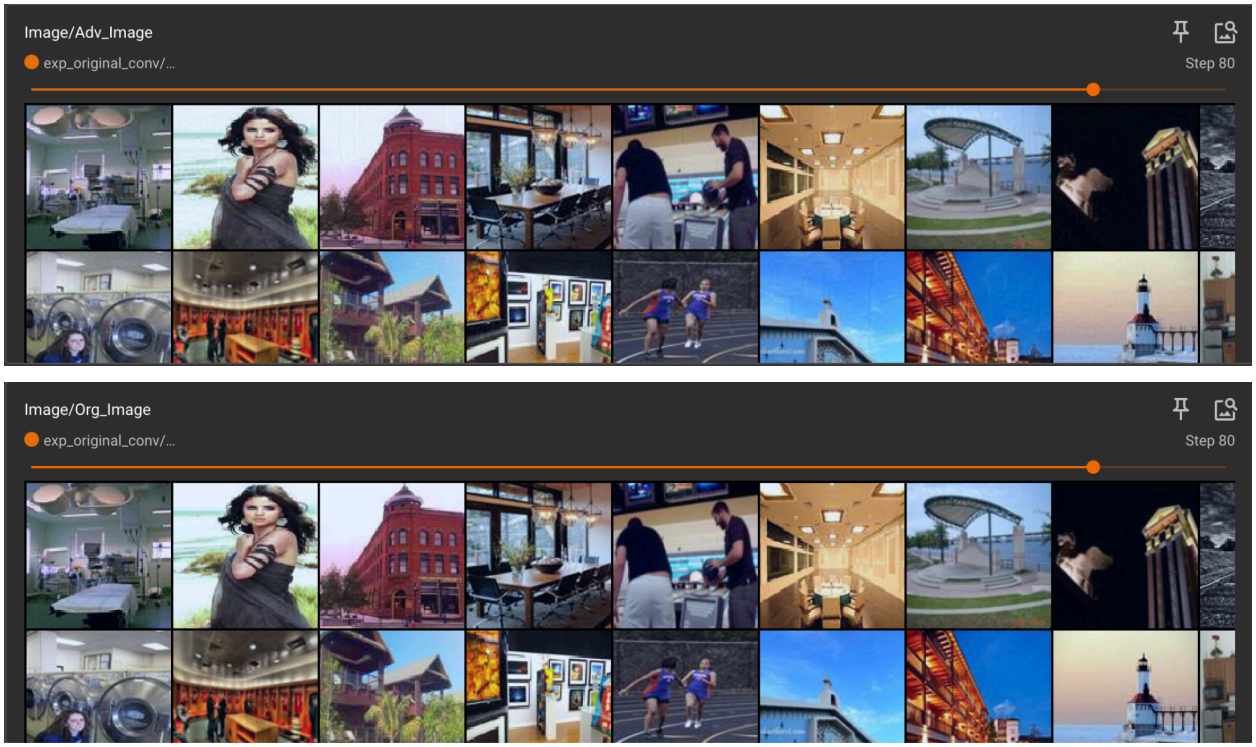


Figure 2: Adversarial images compared to the original images for the attack on SimpleNet while increasing the number of steps but reducing the error bound (20 steps, 0.05 epsilon, 0.01 alpha)

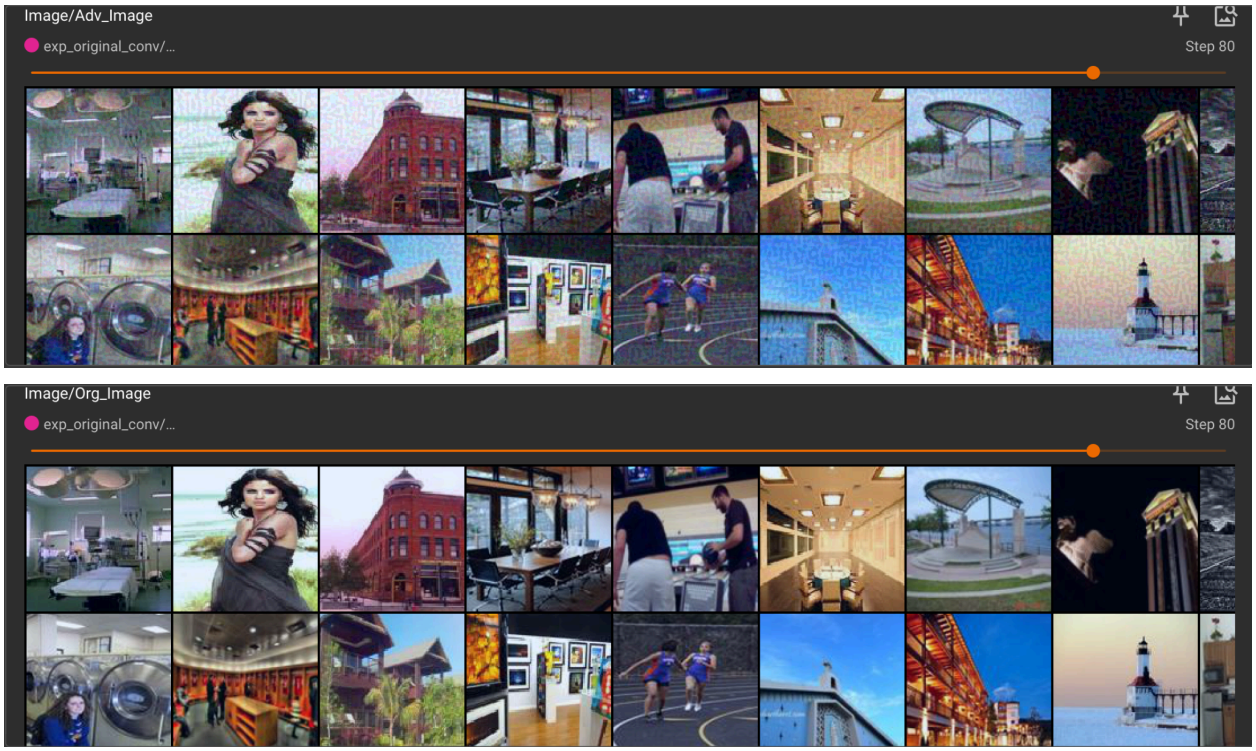


Figure 3: Adversarial images compared to the original images for the attack on SimpleNet with increased step size (10 steps, 0.2 epsilon, 1 alpha)

As you can observe, it is very difficult to find a difference between the original images and adversarial samples in the case of Figures **1** and **2** corresponding to the attacks in rows **2** and **3** in the table.

However, while the attack with the increased step size (row **7**) was able to achieve single-digit accuracies for the model, there is a difference between the original images and the adversarial samples as seen in Figure **3**.

In the second part of this section, we attempted to design the adversarial training defense strategy and explore whether it is effective against **PGDAttacks**. In order to implement this defense strategy, we modified the forward function of **SimpleNet** to generate adversarial samples during the training phase. However, we didn't want our training dataset to consist of only adversarial samples, so we designed it so that adversarial samples are generated only **20%** of the time during training. Due to the time taken to train, we tweaked the **PGDAttack** to only perform **5** perturbations per sample and we only trained the **SimpleNet** model for **10** epochs. We then compared its performance to a **SimpleNet** model trained for **10** epochs without adversarial training. The results are in the table below:

Training Scenario	**Top-1 Validation Accuracy (Original)**	**Top-5 Validation Accuracy (Original)**	**Top-1 Validation Accuracy (During Attack)**	**Top-5 Validation Accuracy (During Attack)**
SimpleNet - 10 epochs without adversarial samples	22.28	51.68	14.38	37.58
SimpleNet - 10 epochs with adversarial samples	20.75	50.30	19.12	46.03

As seen in the table, incorporating adversarial samples during training leads to much better performance against **PGDAttacks**. We see that the Top-1 accuracy improves by **~5%** and Top-5 accuracy improves by **~9%** when under attack if we use adversarial training. However, it appears to come at the cost of a slightly lower original accuracy after 10 epochs of training.

Contributions :

3.1 -> done by Jiangyi Liu
3.2 -> done by Abhijeeth
3.3 -> done by Sujay