CS771 Homework Assignment 2 Write-up

Moniek Smink
smink2@wisc.edu

Venkata Saikiranpatnaik
vbalivada@wisc.edu

Sourav Suresh
sourav.suresh@wisc.edu

1. Team Member Contributions

Name	Contribution
Moniek Smink *smink2@wisc.edu*	4: Training Deep NNs Bonus: Adv. Training
Venkata Saikiranpatnaik *vbalivada@wisc.edu*	3: Custom Convolutions
Sourav Suresh *sourav.suresh@wisc.edu*	5: Attention and Adversarial Samples

2. Overview

The main objectives of the assignment are three-fold. First, we implement our custom convolution in Python, see Section 3. Next, we experiment with training numerous variants of vision models, design a custom network architecture CustomNet, and compare its performance against the other models in Section 4. Finally, we test adversarial robustness of our model as well as examine an approach to increase robustness through adversarial training in Section 5.

3. Understanding Convolutions

3.1. Forward Pass

Training a CNN consists of a forward pass and a backward pass. First, we discuss the implementation of the forward pass. The high-level idea is summarized really well in the slide shown in Figure 1.

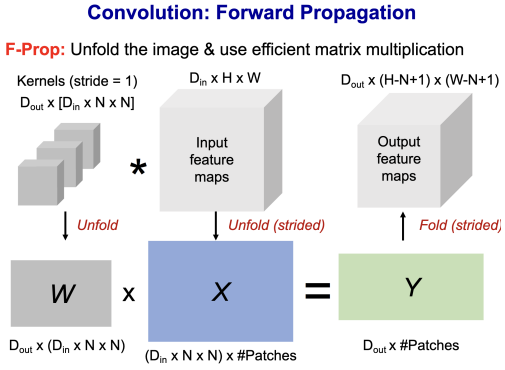

Figure 1. Forward Pass of the Convolution Operation

First, we unfold the input matrix. The weight matrix can easily be unfolded by viewing it as a two-dimensional matrix with dimensions $(C_o, C_i \times K \times K)$ since the high dimensional tensor has the trailing dimensions stored in a contiguous fashion. More importantly, the input is unfolded in a similar fashion by the `unfold` operation where we unfold the width first, then the height, and finally along the input channel dimension. This lines up with the weight view and we are now free to matrix multiply the "unfolded" weight matrix with the unfolded input matrix. A quick note: the input matrix also has the batch dimension but this is handled by the batched matrix multiply. In particular, the unfolded weight matrix has shape $(C_o, C_i \times K \times K)$ and the unfolded input matrix has dimension $(N, C_i \times K \times K, H_o \times W_o)$, where N is the batch dimension. The weight matrix is broadcasted along the first dimension and matrices are multiplied for each element in the batch giving an output dimension of $(N, C_o, H_o \times W_o)$. We now add the bias term but need to unsqueeze along the trailing dimension to broadcast the bias along both the batch dimension and the unfolded output grid dimension. Finally, we view the output as a folded tensor (do not need to fold it explicitly since there are no overlapping folds).

3.2. Backward Pass

Now, before discussing the backward pass, in the forward pass, we store the input in our context for later use in gradient computation. The high-level idea of the backward pass is neatly summarized in the slide shown in figures 2 and 3.

We use similar tricks as the forward pass to implement this. We view the output gradients as a $(N, C_o, H_o \times W_o)$ shaped tensor. We are required to compute the gradients w.r.t. input, weights, and biases but only when we need them, as indicated by `ctx.needs_input_grad`.

For computing the input gradient, we first treat weights as a two-dimensional tensor of shape $(C_o, C_i \times K \times K)$, matrix multiply with the unfolded output gradients, and finally fold the matrix product into the input dimension (N, C_i, H, W). An explicit fold is required here because of overlapping gradients. Now, to compute the weight gradient, we first transpose the input (ignoring the batch dimension) before matrix multiplying with the output gradients.

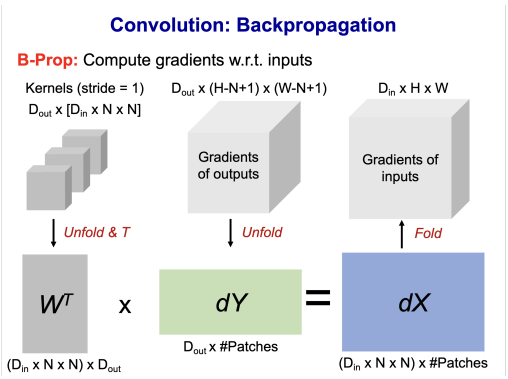

Figure 2. Gradients w.r.t. Inputs

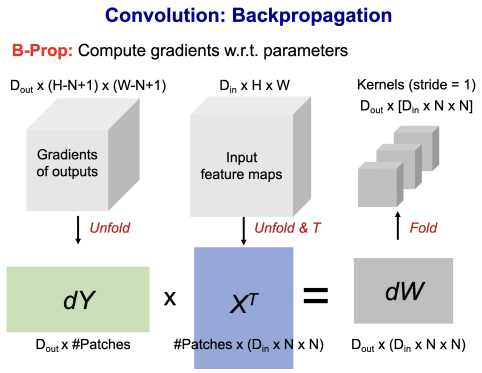

Figure 3. Gradients w.r.t. Weights

Finally, we sum the gradients along the batch dimension before viewing the gradient weights in their original dimension (C_o, C_i, K, K).

3.3. Testing

Verified the outputs and gradients against the vanilla Conv2d module provided by pytorch. Copied weights and biases from the in-built module to reproduce the results. Tested for $(0, 1), (0, 3), (3, 3)$ padding, stride pairs. Finally, we also ran `test_conv.py` for a sanity check, see Figure 4.

4. Designing and Training Deep Networks

4.0.1 Training

Unless otherwise noted, training was done using stochastic gradient descent with a batch size of 256, momentum of 0.9, and starting learning rate of 0.1 decayed according to a linear warm-up, cosine annealing schedule. A weight decay warm-up of 5 epochs and weight decay parameter of 1e-4 was used for regularization. Cross entropy loss was used to compare the predictions made by the model to the ground truth labels. Data was augmented during training to improve the generalization potential of the model. Images were randomly cropped, flipped, color-manipulated, scaled, and rotated during training. During validation, images were randomly scaled and center cropped. The Top-1 and Top-5 accuracy was computed at each epoch. A Top-k accuracy marks a prediction as correct if the ground truth label is in the model's top k guesses. Both the Top-1 and Top-5 accuracies are displayed because the Top-1 accuracy can be misleading in cases where class distributions are unequal. Training was done on a Google Cloud Virtual Machine equipped with a Nvidia Tesla T4 GPU.

```
torch ❱ python test_conv.py
Using device: cpu
Check Fprop ...
Fprop testing passed
Check Bprop ...
Bprop testing passed
Check nn.module wrapper ...
All passed! End of testing.
```

Figure 4. Sanity Check custom Conv

4.0.2 Dataset

Training and validation was done on the MiniPlaces dataset, a subset of the Places dataset [4]. This dataset was created by MIT with 120k images and 100 mutually exclusive classes of scenes such as 'auditorium', 'highway', and 'playground'. The training set consisted of 100k images with 1k images per class. The validation set consisted of 10k images mixed classes. The training set was used to minimize the loss function with stochastic gradient descent during training. The validation set was used during training to monitor the generalization progress and calculate the Top-1 and Top-5 accuracies for each model at each training step.

4.1. SimpleNet

A simple convolutional model, called SimpleNet, was trained for 60 epochs with default parameters. SimpleNet is made up of 7 convolutional layers with ReLU activation and occasional max pooling layers, with a final average pool and dense layer. The Top-1, Top-5, learning rate schedule, and training loss graphs for training are displayed in Figure 5. This model's Top-1 and Top-5 accuracy on the validation set was 44.880 and 73.630, respectively. During training, 3113 MiB of the GPU memory was used.

4.2. SimpleNetCustConv2D

A SimpleNet with custom-made 2D convolution operation as described in Section 3, called SimpleNetCustConv2D,

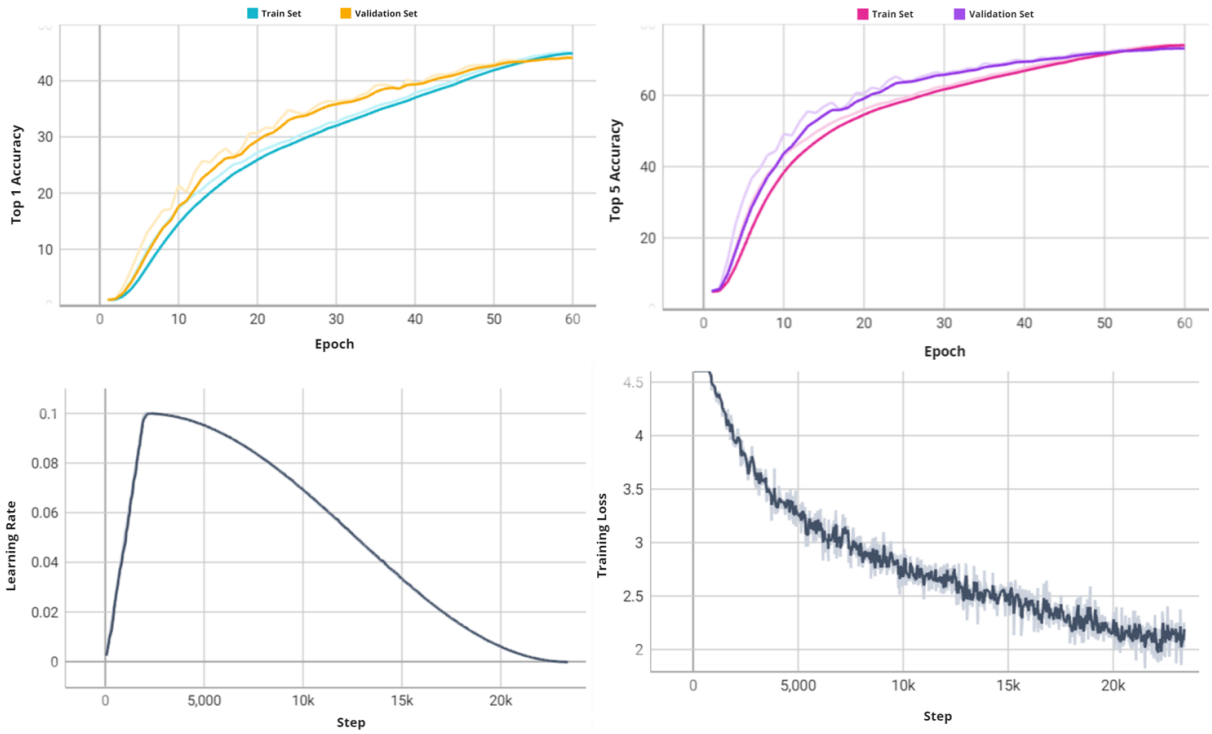

Figure 5. The Top-1 (top left), Top-5 (top right), learning rate schedule (bottom left), and training loss (bottom right) graphs for training SimpleNet.

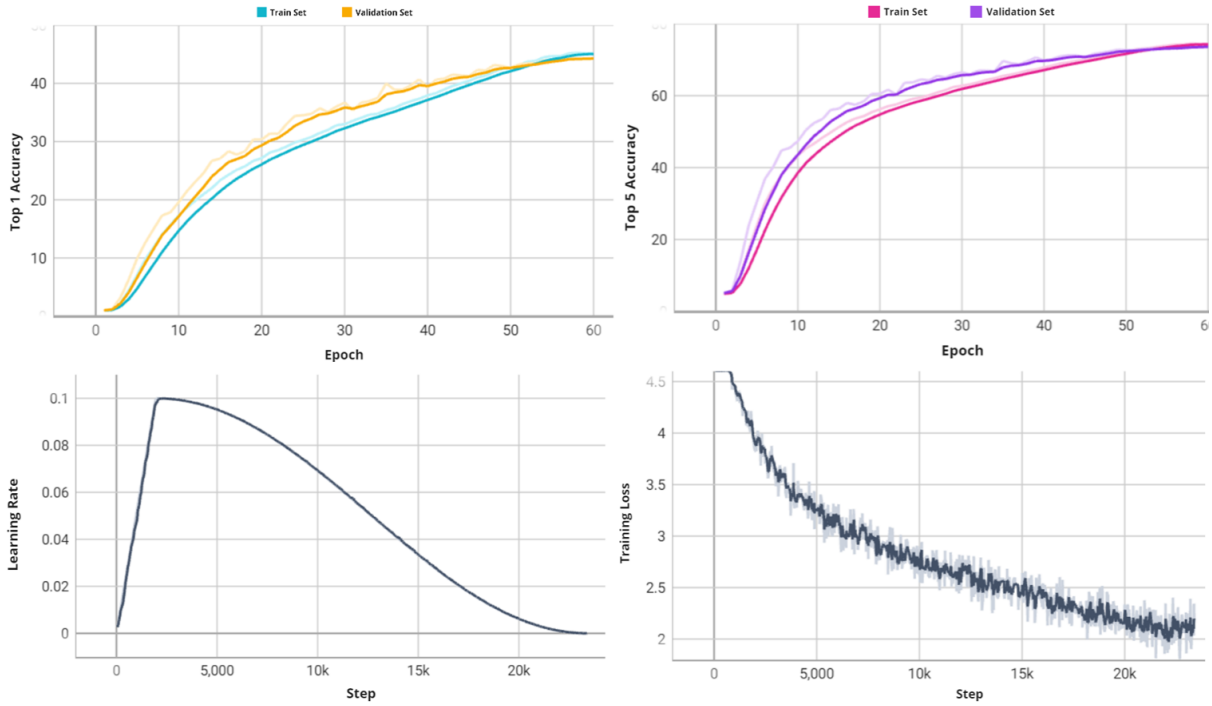

Figure 6. The Top-1 (top left), Top-5 (top right), learning rate schedule (bottom left), and training loss (bottom right) graphs for training SimpleNet with a custom Conv2D operation.

was trained for 60 epochs with default parameters. The Top-1, Top-5, learning rate schedule, and training loss graphs for training are displayed in Figure 6. This model's Top-1 and Top-5 accuracy on the validation set was 44.900 and 73.330, respectively. During training, 4881 MiB of the GPU memory was used.

We see that the accuracies for this model are very comparable to the original SimpleNet model with Pytorch's Conv2D operation, likewise, the rate of convergence seems to be very similar, as demonstrated in the training loss vs step graph shown in the figures. The only discernible difference we can see is the GPU memory used. Using nvidia-smi, we see that the average amount of GPU memory used during the traditional SimpleNet training was 3113 MiB. Meanwhile, the average GPU memory used during the SimpleNet training with the custom Conv2D operation was 4881 MiB. This is likely explained by some memory efficiency optimization Pytorch does in its Conv2D operation such as reducing intermediate variables or efficiently handling biases.

4.3. SimpleViT

A vision transformer model, called SimpleViT, was implemented. Transformers are commonly used on sequences of inputs like in NLP. A Vision Transformer classifies an image by separating an image into a series of patches, creating a patch embedding, adding a positional embedding, running this series into a transformer encoder, aggregating the patch-level outputs, and finally creating the class logits with an MLP head.

The patch embedding is done with a Conv2D layer where the kernel size is equivalent to the stride with a patch embedding dimension parameter. The positional embedding is initialized to the image size and is later learned through training. The transformer encoder is made up of a series of transformer blocks utilizing interleaving local and global self-attention, layer norms, and MLP heads as described in [1]. Self-attention layers transform an input sequence using representations of the sequence called queries, keys, and values. Global self-attention allows for long-distance interactions within the image. Local self-attention allows for smaller-distance interactions within an image but reduces a lot of the computational complexity that comes with global self-attention. Interleaving global and local self-attention allows the model to learn long- and short-distance relationships while still maintaining reasonable computational complexity. By default, the transformer encoder contains four transformer blocks with the first and third utilizing local self-attention. After the transformer encoder, in order to aggregate the patch-level outputs, global average pooling is done. Final normalization and dense layers create the final output logits.

SimpleViT was trained for 90 epochs using an AdamW optimizer [2] with a decreased starting learning rate of 0.01 and increased weight decay parameter of 0.05 to allow for slower and more regularized training. All other parameters remained the same as the defaults. The Top-1, Top-5, learning rate schedule, and training loss graphs for training are displayed in Figure 7. This model's Top-1 and Top-5 accuracy on the validation set was 43.920 and 73.600, respectively. During training, 2117 MiB of the GPU memory was used.

4.4. CustomNet

To improve its performance, SimpleNet was augmented using batch normalization and skip connections, resulting in CustomNet. Batch normalization allows for simpler and more stable training of deep learning models as it makes the optimization landscape significantly smoother [3]. Skip connections help to preserve information and gradients that might otherwise be lost or diluted by passing through multiple layers, allowing for better training and deeper models without overfitting. The architecture of CustomNet is detailed further below.

The first two convolutional blocks of SimpleNet were kept, each of these blocks includes three convolutional layers, ReLU activations after each convolution layer, and a 2D max pooling layer. In addition, at the end of each of these convolution blocks, 2D batch normalization layers were added.

After these first two convolutional blocks, a series of four convolutional blocks, called 'skip blocks' were added that have skip connections over them. Each skip block has three convolutional layers, ReLU activations, and a 2D batch normalization layer at the end.

After the skip blocks, the last convolutional block and adaptive max pooling layer remained unchanged from SimpleNet. Finally, instead of the one fully connected layer in SimpleNet, two fully connected layers bridged by a ReLU activation and 1D batch normalization were added to finish the model. A diagram of CustomNet's architecture is shown in Figure 8.

CustomNet was trained for 60 epochs with default parameters. The Top-1, Top-5, learning rate schedule, and training loss graphs for training are displayed in Figure 9. This model's Top-1 and Top-5 accuracy on the validation set was 49.690 and 78.700, respectively. During training, 4277 MiB of the GPU memory was used.

4.5. ResNet18

A pre-trained ResNet18 model was fine-tuned for 60 epochs with the default parameters. The Top-1, Top-5, learning rate schedule, and training loss graphs for training are displayed in Figure 10. This model's Top-1 and Top-5 accuracy on the validation set was 53.250 and 80.230, respectively. During training, 3192 MiB of the GPU memory was used.

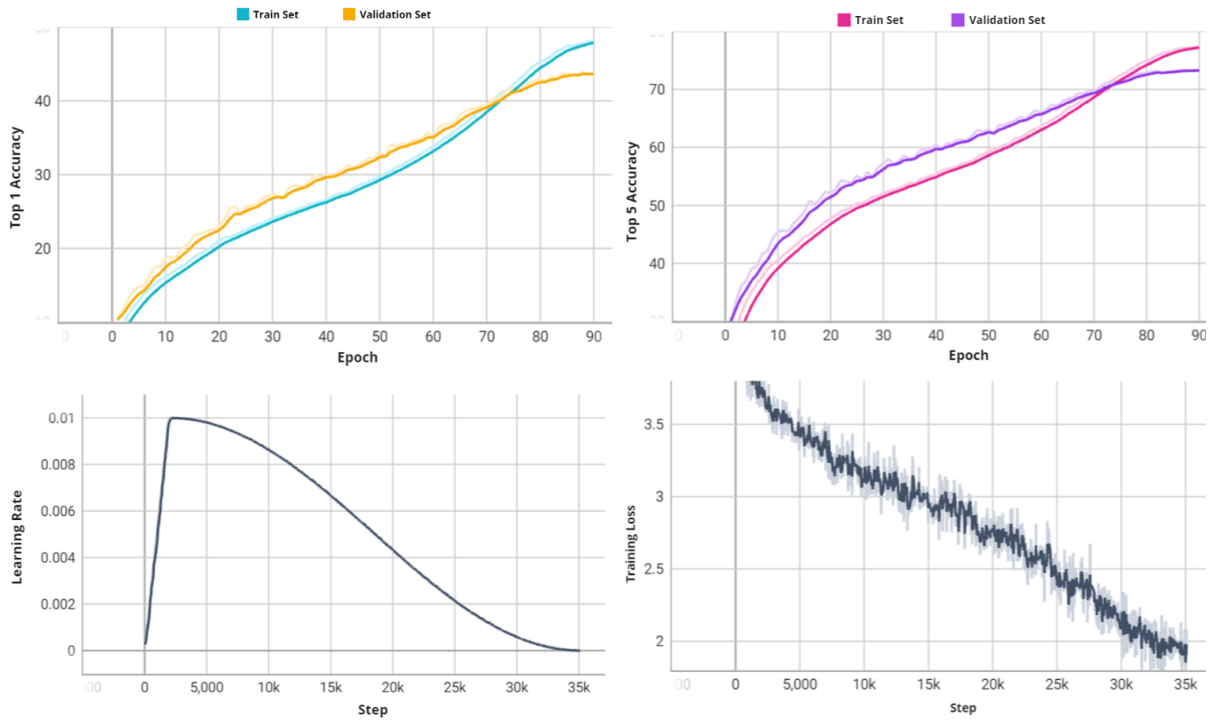

Figure 7. The Top-1 (top left), Top-5 (top right), learning rate schedule (bottom left), and training loss (bottom right) graphs for training SimpleViT.

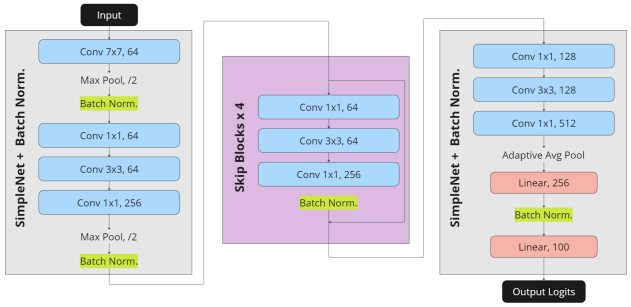

Figure 8. The architecture of CustomNet.

4.6. Discussion: comparing the models

Model Architecture	Accuracy Top-1	Accuracy Top-5	Epochs	GPU Mem Used (MiB)
SimpleNet	44.880	73.630	60	3113
SimpleNet-CustConv2D	44.900	73.330	60	4881
SimpleViT	43.920	73.600	90	2117
CustomNet	49.690	78.700	60	4277
ResNet18	53.250	80.230	60	3192

Table 1. The final Top-1 and Top-5 accuracies as well as some training metrics for models presented in this paper.

In Table 1, we see that the Top-1 and Top-5 accuracies of SimpleNet with and without custom convolutions and the SimpleViT model were extremely similar. Meanwhile, the accuracies of ResNet18 and CustomNet were higher, likely due to their more complex architecture. Furthermore, we see the effect of batch normalization and skip connections in the accuracy jump between SimpleNet and CustomNet.

It is interesting that SimpleViT was actually the smallest user of GPU memory out of the models, likely due to the smaller number of parameterized layers compared to the other models. Meanwhile, CustomNet had the largest GPU memory use even though it technically has less layers in total than ResNet18. This is likely due to some memory

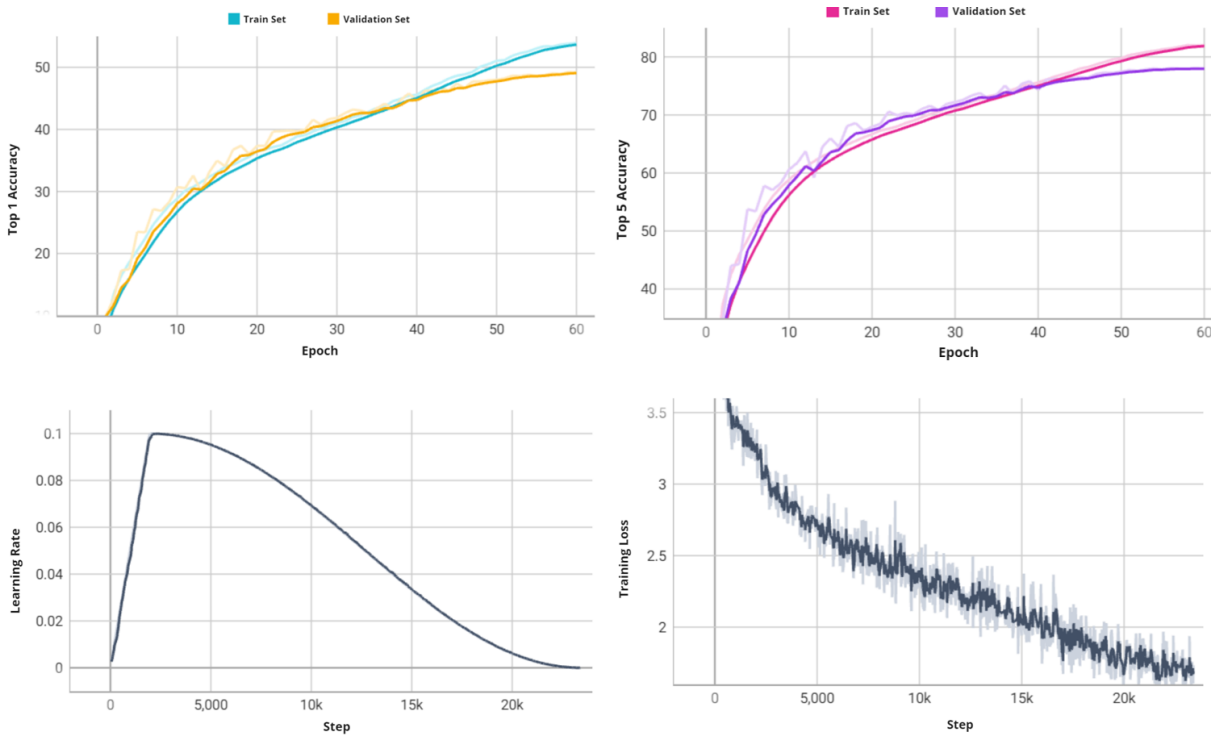

Figure 9. The Top-1 (top left), Top-5 (top right), learning rate schedule (bottom left), and training loss (bottom right) graphs for training CustomNet.

optimization Pytorch's version of the skip blocks have and because our skip blocks contain three convolutional layers per block instead of only two in traditional ResNet18.

For the differences in Top-1 and Top-5 accuracies seen throughout training for each model please see Figure 11. Through this figure, we can see that the accuracies for each model at first significantly increase as parameters get optimized and plateau as training goes on. We can see at which point each model starts to overfit as the accuracy for the training set significantly overtakes the accuracy for the validation set.

Curiously, we see that the final train and validation accuracies are fairly close together for most models except for the ResNet18 model where both the Top-1 and Top-5 accuracy for the training set is far higher than the validation set. This likely indicates the ResNet18 model was significantly overfitted to the training set. This is likely because we are fine-tuning the ResNet18 model instead of training it from scratch.

When comparing the curves for SimpleNet and CustomNet, we see that even though CustomNet has significantly more layers than SimpleNet, it is able to converge significantly faster (the point where the training accuracy overtakes the validation accuracy is earlier for CustomNet). This is likely due to the use of batch normalization and skip connections in CustomNet.

5. Attention and Adversarial Samples

5.1. Saliency Maps

Saliency Maps provide us with crucial insights to infer more about how the images are understood. We explore their creation and significance with respect to CNN interpretations. In Figure 12, we showcase Saliency Maps generated over SimpleNet model predictions. Gradients are enabled at the input, and the forward pass through the model yields the Saliency Map. The pixel with the maximum gradient in three channels defines a pixel's saliency, visualized as a red patches on the image.

With Figure 12, Saliency Maps reveal how models prioritize image elements. In the top row, we can see how it prioritized to detect buildings, elements in office room, amphitheatre etc., Also, if we look in the bottom row, to detect a corridor it just paid attention to the pathway and ignored the walls, doors etc., We can also see a corner case in the bottom right image which depicts a valley, where the model failed to predict this without getting much attention. This was predicted as a Sand/Dessert perhaps considering the color and texture. But overall we can see a decent understanding of images by the SimpleNet model.

Therefore, Saliency Maps captures how our model perceives the given images and also how it can be leveraged to understand and tweak the model to solve misclassifications.

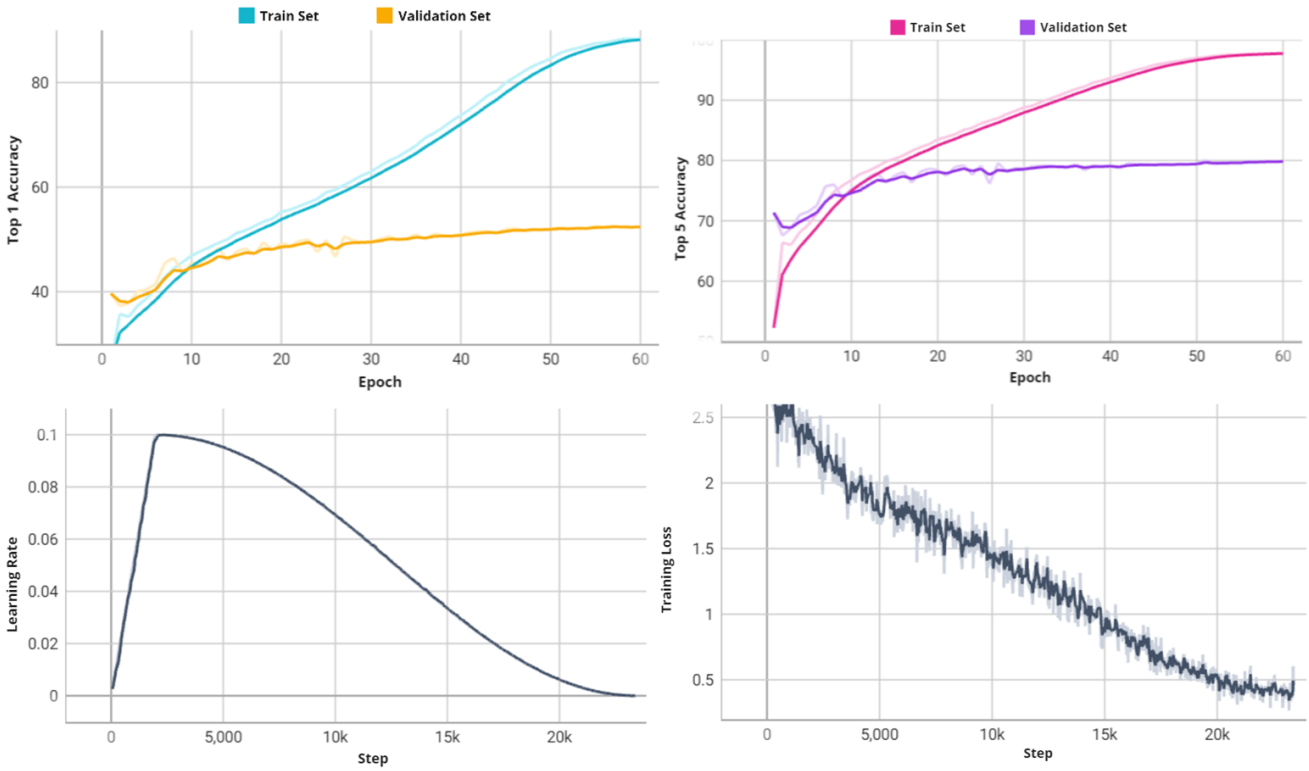

Figure 10. The Top-1 (top left), Top-5 (top right), learning rate schedule (bottom left), and training loss (bottom right) graphs for fine-tuning ResNet18.

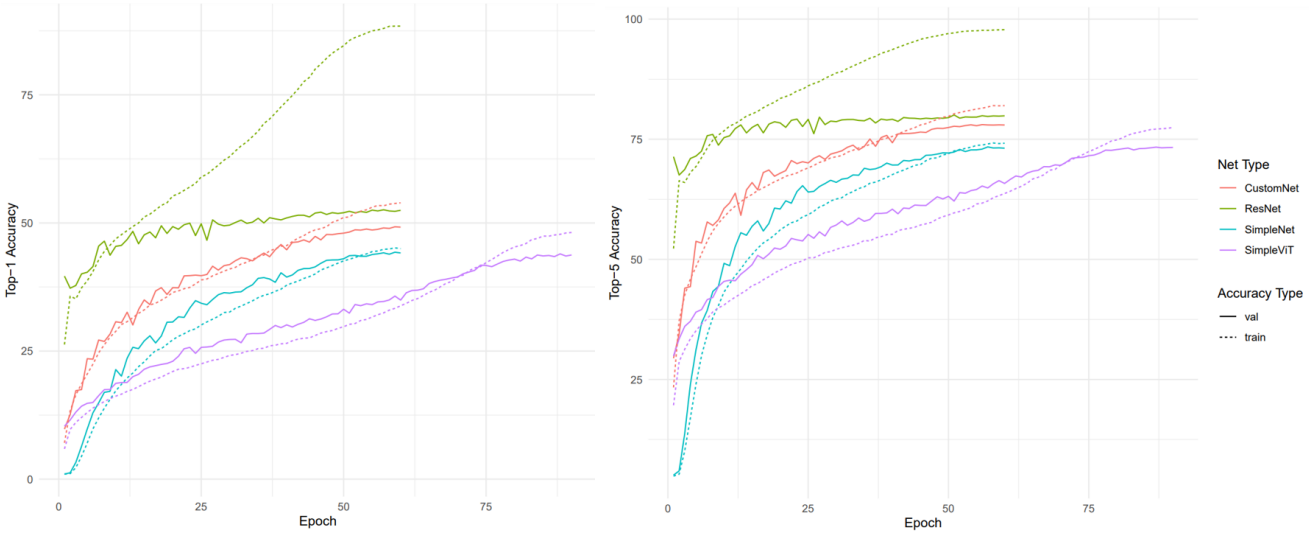

Figure 11. The Top-1 (left) and Top-5 (right) training and validation accuracies computed while training the models presented in this paper.

5.2. Adversarial Samples

The key idea behind these adversarial samples is to purposefully introduce small perturbations that cause misclassification. We accomplish this by performing a projected gradient on an input image to minimize the loss with respect to the least confident incorrect label predicted by the model.

Specifically, the attack algorithm clones the input image to enable gradient computation while preventing accumulation across iterations. In each iteration, we first perform a

forward pass through the model to obtain the logit outputs. We determine the least confident incorrect label based on the confidence scores. The loss is calculated between the model predictions and this chosen incorrect target label.

To ensure the perturbation satisfies the l-infinity norm constraint, we clip the updated image to be within an epsilon neighborhood of the original image. This projects the perturbed image back into the acceptable epsilon ball around the original input. We repeat this process for a specified number of steps, detaching the image from the computational graph after each step to avoid gradient explosion.

When validated against a SimpleNet model which was trained for 60 epochs from Section 4, adding perturbation to images caused a significant drop in top-1 accuracy to 11.2 and top-5 accuracy to 34.15 with default settings(num_steps=10, step_size=0.01, epsilon=0.1). The adversarial samples are shown in Figure 13. The perturbations are not directly visible to the naked eye. We further tried to increase these configuration variables individually and the results have been captured with 2. Among these 3 increased configurations results the number of steps increasing from 10 to 50, we observed the model was attacked significantly, reducing its Top-1 accuracy to 8.03 and Top-5 accuracy to 25.34. Increasing the number of steps allowed more iterations to accumulate larger perturbations, generally making the attack more successful. With Figure 14, we can observe slight irregularities added within the images thereby making the attack stronger.

With the increase in all the three configuration variables(num_steps=50, step_size=0.1, epsilon=10), we can see images got perturbed significantly. As in Figure 15 we can observe the perturbations with a naked eye. This led to reducing the Top-1 accuracy to almost 1.16 and the Top-5 accuracy to 6.06.

We also evaluated with most likely prediction (most confident label) in which case we maximized the output and we can see the comparison between the least likely v/s most likely prediction results in Figure 16. The complete evaluation metrics can be found in Table 2 which shows Top-1 Accuracy and Top-5 Accuracy for both settings(least likely & most likely) with all the configuration variable changes.

	Least Likely		Most Likely	
(num_step/ step_size/ ϵ)	Top-1	Top-5	Top-1	Top-5
10/ 0.01/ 0.1	11.21	34.15	9.0	35.700
$\uparrow$ **50**/0.01/0.1	8.03	25.34	5.23	19.88
10/ $\uparrow$ **0.1**/0.1	17.27	40.4	12.86	34.13
10/ 0.01/ $\uparrow$ **10**	11.21	34.15	9.0	35.70
10/ $\uparrow$ **0.1**/ $\uparrow$ **10**	1.84	8.93	1.59	7.08
$\uparrow$ **50**/ $\uparrow$ **0.1**/ $\uparrow$ **10**	1.16	6.06	1.2	5.45

Table 2. The Top-1 and Top-5 accuracies with different configuration settings for both least likely and most likely selections

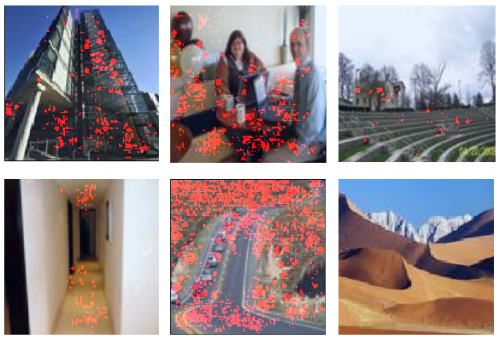

Figure 12. Saliency Maps

Figure 13. Adversarial Samples (with default settings)

Figure 14. Adversarial Samples (with the increase in Number of Steps)

5.3. Bonus: Adversarial Training

In order to improve model robustness to adversarial samples, models can be trained using these perturbed samples. To demostrate this, the forward step in the SimpleNet architecture was modified be able to produce adversarial samples as described in Section 5.2 during the training of SimpleNet if a parameter was set. Because the creation of adversarial samples is time consuming, the number of perturbation steps was decreased from 10 to 5 while the perturbation step size and epsilon parameters were not changed. An unmodified SimpleNet model was pre-trained for 30 epochs. This

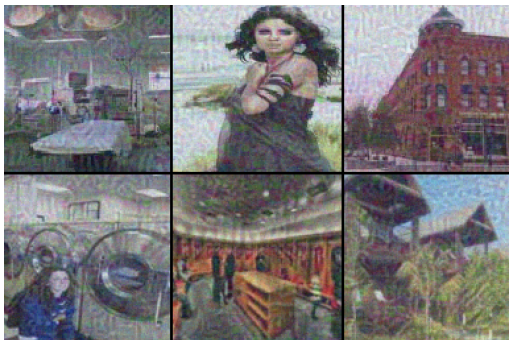

Figure 15. Adversarial Samples (with the increase in No. Steps + Step Size + Epsilon)

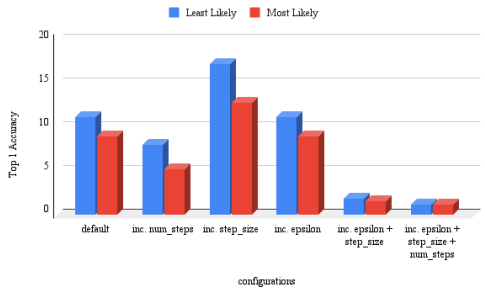

Figure 16. Top-1 & Top-5 Accuracy comparision of least likely v/s most likely selection with different configurations

unmodified SimpleNet model was further trained with adversarial samples for 10 epochs creating AdversarialSimpleNet. Another copy of the 30 epoch unmodified SimpleNet was further trained for 10 epochs without adversarial samples for comparison. The Top-1 and Top-5 accuracy of both of these models when being attacked by adversarial samples are shown in Table 3. We see that adversarial training allows AdversarialSimpleNet to significantly overtake SimpleNet when being attacked while still retaining relatively competitive accuracy when not being attacked.

	Regular Accuracy		Attacked Accuracy	
Model	Top-1	Top-5	Top-1	Top-5
SimpleNet	33.610	63.510	14.510	38.190
AdverSimpleNet	29.320	58.550	**23.600**	**52.330**

Table 3. The Top-1 and Top-5 accuracies with adversarial or regular images for SimpleNet models trained with and without adversarial samples.

References

[1] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. 4

[2] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. 4

[3] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2018. 4

[4] 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. 2