CS 771 Homework Assignment 2 Writeup

Name	Contribution
Debarshi Deka <ddeka@wisc.edu>	• Setting up GCP VM instances • Implemented CustomConv, Saliency maps and downloading dataset • Executed experiments • Maintaining GitHub and writing initial sessions in the report
Satya Sai Srinath Namburi <sgnamburi@wisc.edu>	• Setting up conda environment in VM • Implemented ViT, adversarial attack and others • Executed experiments • Write later sessions of the report, formulating results from TensorBoard

3.1 Understanding Convolutions

Both forward and backward propagation has been implemented using fold and unfold operations present in PyTorch.

Forward Operation:
Key Idea: Instead of convoluting weight tensor with input tensor, we unroll the tensors and perform a matrix multiplication.

- First unfold the input tensor based on stride, padding and kernel size to get the number of input patches ($I_{patches}$). Then unroll the weight matrix (W).
- Now, compute the matrix multiplication between the input patches and weight matrix i.e $W^T x\ I_{patches}$. Now, fold it using the output size to get the final output.

Backward Operation:
Key Idea: Backward propagation is just matrix multiplication between various elements in a systematic way.

- $\partial y/\partial X$: Unfold the grad_output tensor and reshape the weight matrix W. Then, $W^T xgrad\_output$ will be equal to $\partial y/\partial X$
- $\partial y/\partial W$: Unfold the grad_output tensor and reshape the input matrix. Then, $Input^T xgrad\_output$ will be equal to $\partial y/\partial W$

Results: Our implementation is able to pass the test cases provided in *test_conv.py*

3.2 Design and train a deep neural network

We have successfully downloaded the MiniPlaces Dataset using the shell script. Categorical Cross Entropy loss is used as it is a classification problem and a weight decay of 1e-4 is used with initial learning rate of 0.1 (default). We report the default provided top-1 and top-5 accuracies.

Simple convolutional Network:

We didn't modify anything in the code provided in SimpleNet.

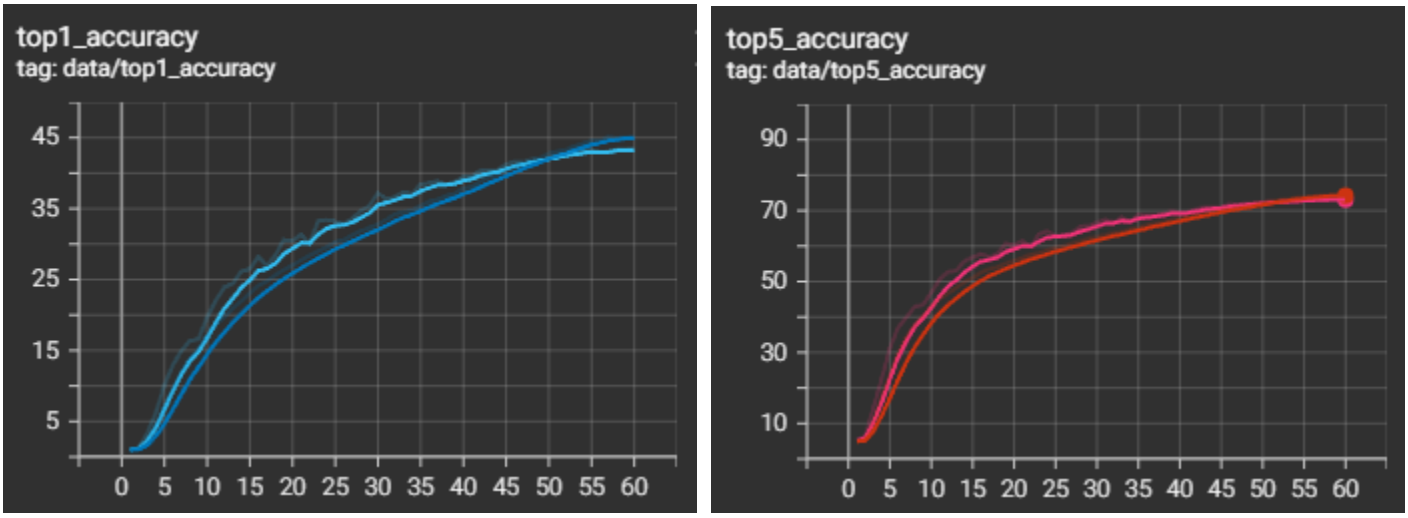

Fig: Top-1 and Top-5 train/val accuracies of SimpleNet with Pytorch Convolutions (Ran for 60 epochs)

Train with your own convolutions:

Instead of the pytorch under-the-hood convolutions, we passed our own built convolutions (from CustomConv2D) to observe the performance. We performed only 10 epochs due to time constraints.

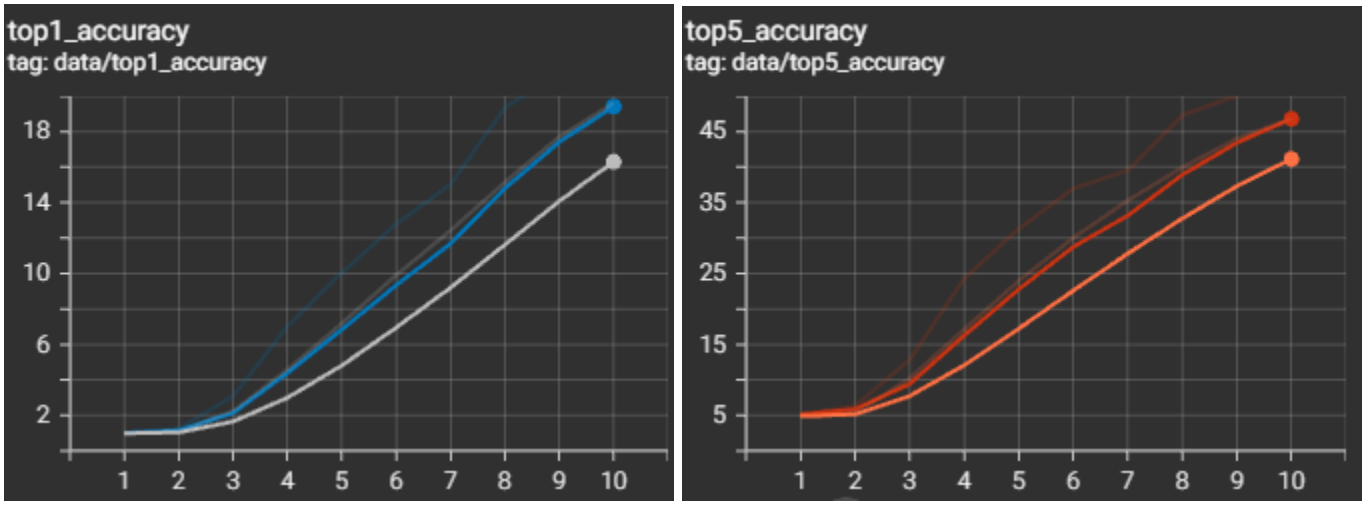

Fig: Top-1 and Top-5 train/val accuracies of SimpleNet with CustomConv2D (Ran for 10 epochs)

Results:

1. The CustomConv2D achieved a (smoothed) top-1 val accuracy of 19.41 (as compared to 20 i.e val top-1@10epochs when using Pytorch version).
2. The CustomConv2D achieved a (smoothed) top-5 val accuracy of 46.8 (as compared to 47 i.e val top-5@10epochs when using Pytorch version). So, it means that the CustomConv2D is almost perfect when it comes to computation
3. It took around 2min/epoch (appx total 3hrs for 90 epochs) vs 2.8min/epoch (appx 28min for 10 epochs) which means that the Pytorch version of convolution is much faster compared to CustomConv2D because of under-the-hood cpp implementations
4. The memory consumption is also much higher in CustomConv2D (around 3600MiB) compared to the Pytorch version (around 3100MiB) which can be attributed to optimizations done in Pytorch.

Implement Vision Transformer (ViT):

We implemented a standard ViT with default parameters (i.e a patch embedding block → add positional encodings → 4 transformer blocks). The PatchEmbed and Transformer blocks are used from *custom_blocks.py*

Results:

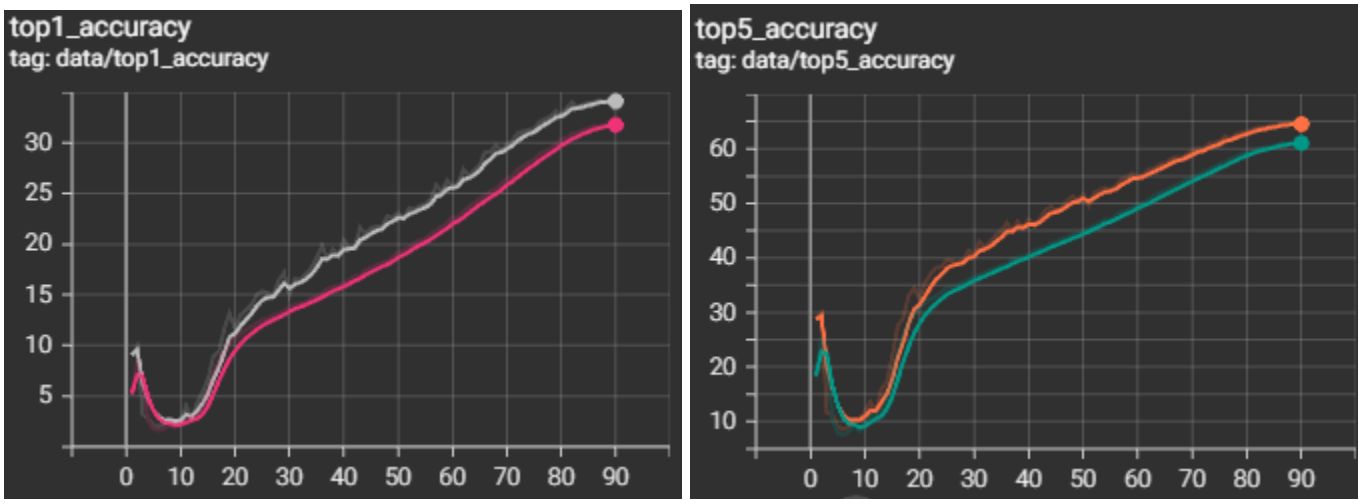

Fig: Top-1 and Top-5 train/val accuracies of ViT (Ran for 90 epochs)

1. ViTs achieve an accuracy of ~34% which is significantly lower compared to SimpleNet. A part of this can be attributed to inductive bias as ViTs are not made for image related tasks.
2. It took around 4hrs to run for 90 epochs (compared to 3hrs/90 epochs for SimpleNet)

Design your own network:

We added Batch Normalization layers (BatchNorm2D) after each convolution operation in SimpleNet. Our intuition is that batch normalization reduces the covariance (domain shift dependency) and thus increases the accuracy of the network.

Results:

1. As expected, the val top-1@10epochs accuracy of Batch Norm network went to 31% as opposed to 20% from a normal SimpleNet.
2. It took an additional 3min to run Batch Normalization network (28min/10epochs for SimpleNet vs 31min/10epochs for SimpleNet+BatchNorm)

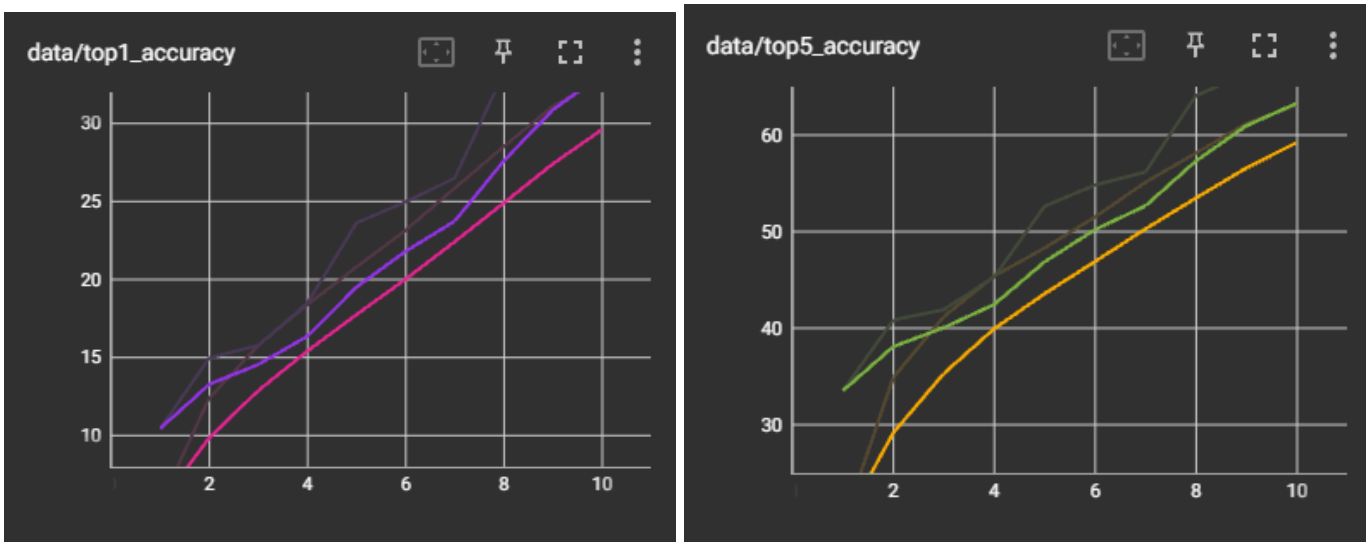

Fig: Top-1 and Top-5 train/val accuracies of SimpleNet + BatchNormalization (Ran for 10 epochs)

Finetune a pre-trained model:

We fine-tuned the ResNet18 with all the default parameters for 60 epochs

Results:

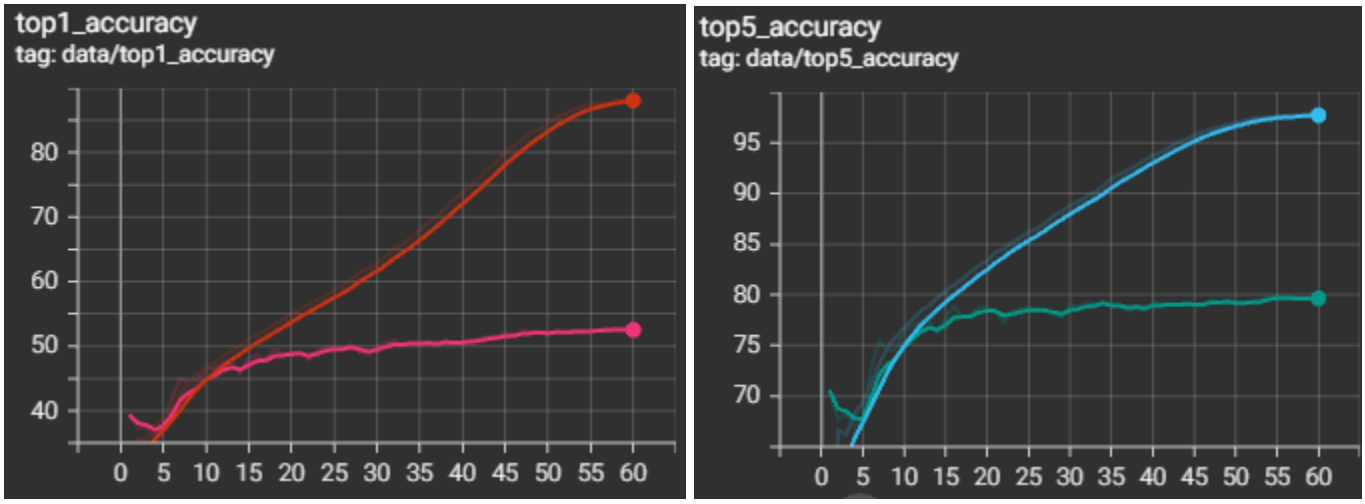

Fig: Top-1 and Top-5 train/val accuracies of fine tuned ResNet model

The val top-1 accuracy is very high ~88% which beats all the previous models and is attributed to

1. Complex model (18 layers) which has residual connections, batch normalization layers and has inductive bias, pretrained on ImageNet
2. More number of epochs (even for 10 epochs, it achieved a val accuracy of ~40% which is pretty decent)

Note: More details about the comparison and train/val accuracies for top-1/top-5 are in the below table

Model	**Train accuracy (top 1%, top 5%)**	**Validation accuracy (top 1%, top 5%)**
SimpleNet (60 epochs)	43.32, 73.07	44.97, 74.23
SimpleNet + CustomConv (10 epochs)	16.3, 41.14	19.41, 46.8
ViT (90 epochs)	31.76, 61.02	34.1, 64.49
ResNet, finetune (60 epochs)	88.08, 97.72	52.5, 79.65
SimpleNet + BatchNorm (Own Network) (10 epochs)	29.6, 59.21	33.1, 63.28

Table 1: Summarizing the average train and val top-1, top-5 accuracies across multiple models in multiple conditions

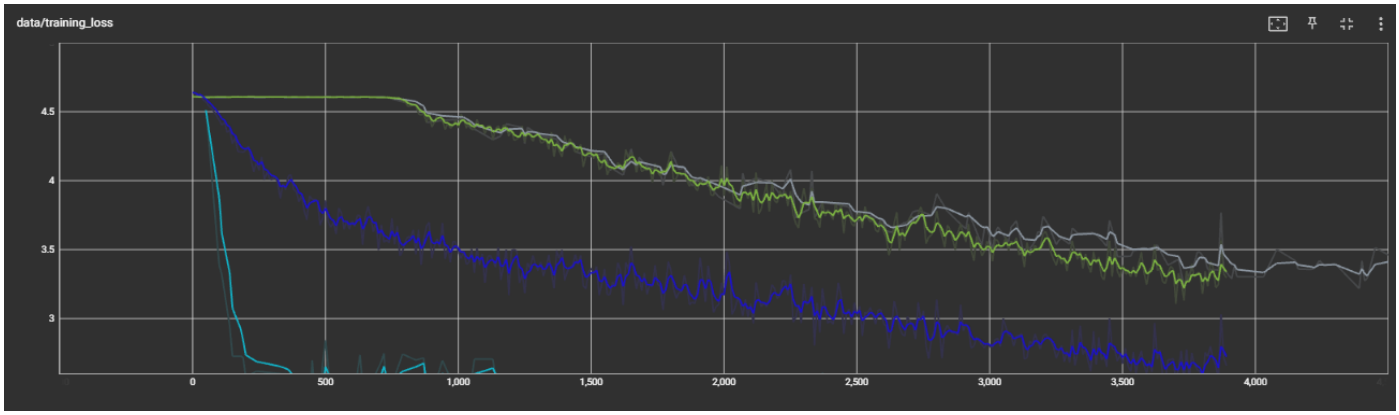

Fig: Training loss for various models: (Color represents the curve) - Resnet18, SimpleNet + Batch Norm, SimpleNet, SimpleNet with CustomConv (Avoiding ViT due to scaling issues)

3.3 Attention and Adversarial Samples

Saliency Maps:

We used the best SimpleNet model to generate Saliency Maps. These feature visualizations help us to find which spatial locations trigger the classification outcome, thus providing the insights about the models understanding of the input image.

Results:

1. The validation accuracy@1 is 44.29 and acc@5 is 73.86 (we are not training anything here)

2. We can see that our network didn't pay proper attention to important pixels and focused on something else (for example, the bottom right corner image) where the model is focusing on background grass instead of the firecans.
3. As it's just validation, it took around 10 sec to generate the saliency maps on GPU.

By understanding on what the model is looking at, we can build better models, clean data in an efficient way (a way of explainability)

Image, From top left: Original Image, Adversarial Image, Saliency Maps

Adversarial Samples:

We generated Adversarial samples using PGDAttack. We take the input, pass it to the model and the prediction of least confident class is used as a proxy for the incorrect label. We compute the Cross Entropy loss between the predicted label and the incorrect proxy label. Using the computed loss, we calculate the gradients for the input tensor. The sign of the gradient along with the step size is used to perturb the denormalized input tensor.
Then, we compute the delta as the difference between the output and the denormalized input and clamp the values based on the ϵ value. The generated delta is summed up with the denormalized input to generate the output. We clamp the values of the output to a predefined range, normalize the tensor and repeat this iterative process for a given number of steps. The final output is used to attack the model.

Results:

1. We can observe that the samples produced are classified as the same as the original ones by us humans but the models can be fooled thus can be regarded as adversarial samples.
2. It took around 10min to generate Adversarial samples using GPU
3. When we increase the number of iterations with ε as constant, the test accuracy reduces. The same way, when we increase ε, with the number of iterations as constant, the test accuracy reduces.
4. As both ε and number of iterations has a inverse relationship, we played around the parameters to understand the influence of these two variables on validation accuracy and is summarized in Table 2
5. We can perform stronger attacks by increasing ε and number of iterations

	Accuracy @ top 1	**Accuracy @ top 5**	**Time taken to execute**
10 iterations, $\varepsilon = 0.1$	2.49	13.2	2min
20 iterations, $\varepsilon = 0.05$	2.52	13.99	4min
40 iterations, $\varepsilon = 0.01$	28.16	56.3	7min