3.1 Understand Convolutions

Forward Pass: Unfold the weights with stride and kernel size equal to kernel size. This moves over the entire width of the weight in a single hop, and each 3D tensor becomes a row of the output matrix. Unfold the input with the expected parameters, so each input becomes a column of the unfolded output. Multiply these two matrices, and fold with stride and kernel size of 1, so each 'pixel' is placed one at a time into the output blocks.
Backward Pass: Weight Gradient: Unfold the output gradient with stride and kernel size of 1, so that each 'pixel' in the input blocks goes one at a time to become a row of the output matrix. Transpose the unfolded input (to calculate the derivative w.r.t. weight), unfold using view to a 5D tensor, with the first dimension propagating the gradient back to each item in the batch, multiply by output blocks. Input Gradient: Permute input weights (to calculate derivative w.r.t. input), multiply with unfolded output gradient, re-fold to original input size.

3.2 Design and Train a Deep Neural Network

1) A Simple Convolutional Network: Training curves for the simple convolutional network are shown in **Figure 1**. Final top-1 accuracy was 44.130 and top-5 was 73.110, and training took ~2.5 hours for 60 epochs, and 22 minutes for 10 epochs.

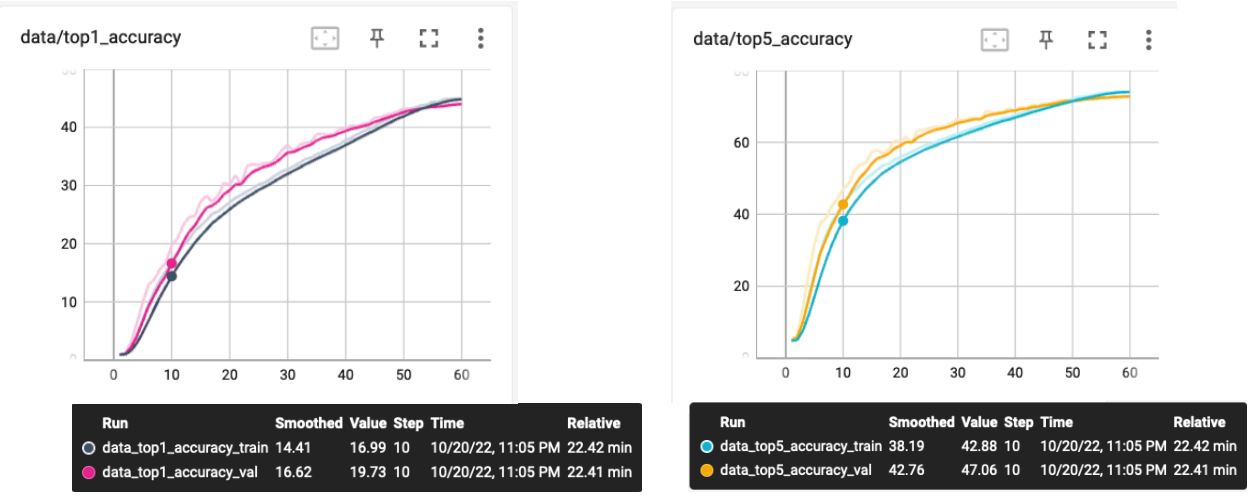

Figure 1: Training Curves for a simple convolutional network with default convolutions. Hover points were chosen at 10 epochs (Step value of 10) in order to compare to custom convolutions.

2) Train with Your Own Convolutions: Training curves for custom convolutions are shown in **Figure 2**. After 10 epochs, training reached top-1/5 accuracy of 22.4/51.82 and took 25 minutes. Compared to the default convolutions, the accuracy was somewhat higher, but within expectation of differences in training runs (~3%), while time was slightly slower (25 vs 22 minutes). Memory was not displayed in this tensorboard visualization. As mentioned, differences in accuracy should be due to differences in initialization or training order. Differences in time and

memory are probably due to perfectly optimized torch code, for example, not storing any intermediate variables, and/or more efficient functions for calculating the same output.

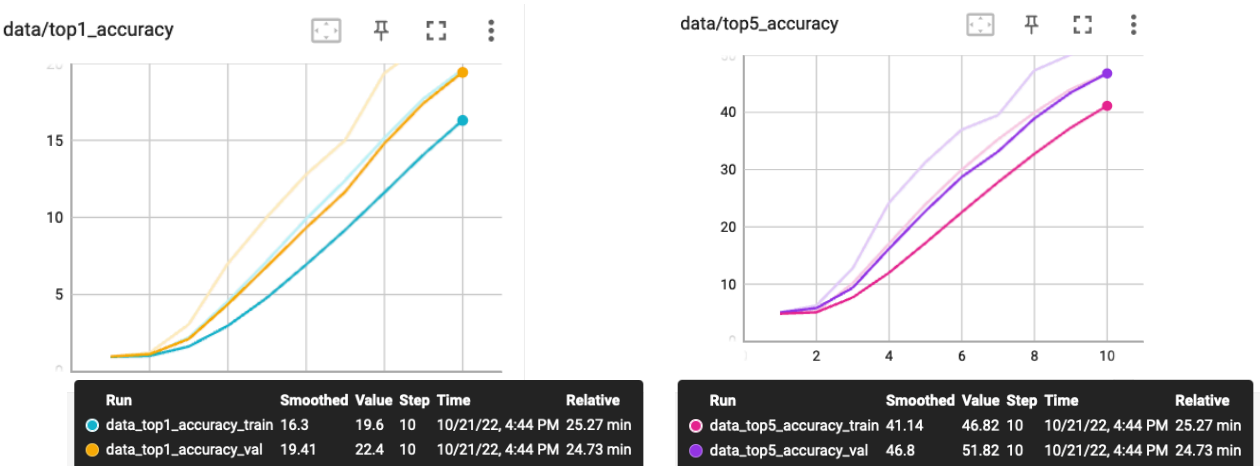

Figure 2: Training Curves for a simple convolutional network with custom convolutions. Training was fairly comparable to default convolutions. No memory consumption is available.

3) Implement a Vision Transformer: Our implementation consisted of a patch embedding with kernel size = (patch_size, patch_size), followed by a sequence of length depth of Transformer Blocks, with drop_path specified by the dpr array, and window size of size 0 if the current layer was not specified in the window_block_indexes array, and the specified window size otherwise. The Transformer blocks were followed with a classification head consisting of an Average Pool, Layer Norm, and Linear layer. An interesting observation was that the Layer Norm was essential to model performance; the model could not reach higher than 20% accuracy without it. Training was performed with the default configurations, and tensorboard visualizations are shown in **Figure 3.**

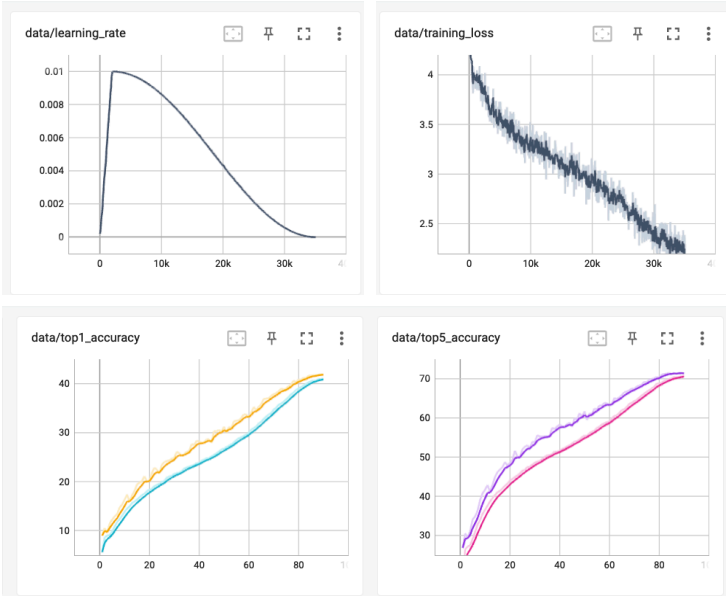

Figure 3: Training Curves for a simple convolutional network with default convolutions.

4) Design Your Own Network: Method: Our custom network consisted of a series of Bottleneck blocks with residual connections and Batch Normalization, as described in He *et. al.* The model, MyNet in student_code.py, used two sets of three Bottleneck Blocks that expand feature dimension from 64, to 128, to 256. The final Bottleneck Block is kept the same as in Simplenet, following a MaxPool and expanding to 512 channels. Bottleneck blocks are defined as three sets of convolution-batchnorm-relu, with the original input added back after the final batchnorm. **Results:** Our small residual model performed slightly better than the SimpleNet with the default training configuration (60 epochs, lr=0.01 etc.), with a top1 accuracy of 51.95 (~7% higher), but required about 3x longer to train (~6.5 hours). Tensorboard visualization for our custom architecture are shown in **Figure 4.**

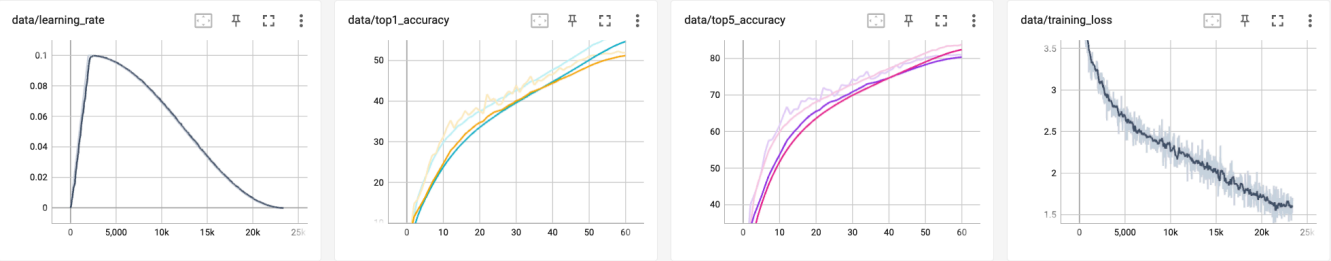

Figure 4: Training Curves for custom Bottleneck Architecture

5) Fine-Tune a Pre-Trained Model: Our custom model performs similarly to the pre-trained model in terms of validation accuracy (both ~52%), but the fine-tuning process only took 2.5 hours. However, since the fine-tuned model started at 39% accuracy, and ours started at 7%, the pre-training time should also be factored in for a fair comparison. Interestingly, the fine-tuned model had much higher *training* accuracy (87.5% vs 53% for our model), suggesting that it could be overfitting. Training curves for the fine-tuned model are shown in **Figure 5.**

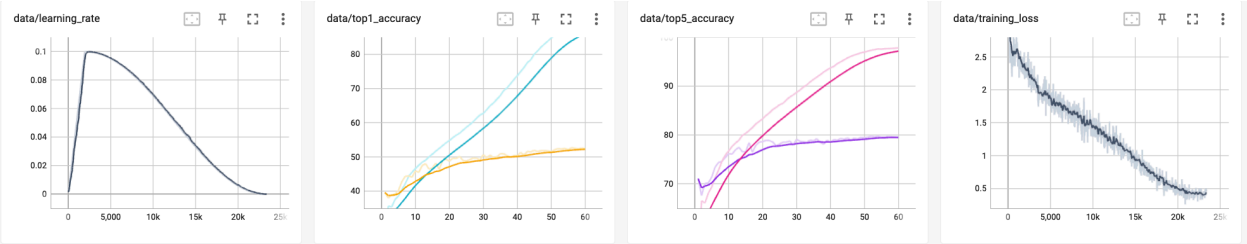

Figure 5: Training Curves for Pre-Trained ResNet

3.3 Attention and Adversarial Samples

1) Saliency maps: Saliency maps for the SimpleNet are shown in **Figure 6.** Qualitatively, they do not seem to focus on the "subject" of each image as often as expected. For example, the bridge image in the lower left corner has must attention on the water. However some images do focus more on the image, e.g. the boxers in the bottom center, or the image of a balloon in the

bottom right (All attention is localized exactly on the balloon). Some of this result could be explained by the relative simplicity of the model or under-training. In general, details of the background seem to be more important than details of the subject.

Figure 6: Saliency maps for SimpleNet

2) Adversarial Samples: Adversarial samples were generated for SimpleNet as described, and with half e (0.15) and double num_steps (20). The default settings produced images with subtle changes, most noticeable in landscapes, e.g. around the clouds in the bottom-center image. For more steps and smaller e, these changes became much more pronounced. Validation top-1 accuracy on the adversarial images dropped from 44.12 to 28.620 (36%), indicating that these images are very confusing for the model

Figure 7: Adversarial Images for SimpleNet: Left: Original image. **Center**: Perturbation with e = 0.3, num_steps = 10. **Right**: e = 0.15, num_steps = 20.

3) Adversarial Training: Robust SimpleNet was trained following Goodfellow *et. al.;* adversarial samples were generated for half (alpha=0.5 in their formulation) of each training batch and replaced their original counterparts. To speed up training, we only trained this model for 30 epochs, and used 5 steps of PGD. Training this model took longer (~2.5 for 30 epochs, so

double as long), but reached the same accuracy for the training period (~38% top-1 after 30 epochs). Adversarial samples generated from this model are shown in **Figure 8**; qualitatively they appear much less distorted. In terms of resistance to adversarial samples, this model's validation accuracy dropped from 38.79 to 30.49 (21%) when evaluated on these samples; even with the stunted PGD parameters this model is 15% more resistant to adversarial attack than a standard model.

Figure 8: Adversarial Images for SimpleNet: Left: Adversarial Image generated for SimpleNet. **Right**: Adversarial Image generated fro robust SimpleNet trained with half adversarial images.

***All team members contributed equally to this assignment**