ASSIGNMENT 2 WRITEUP

Yelun Bao, Jiangang Chen
ybao35@wisc.edu, jiangang.chen@wisc.edu

In this assignment, we implemented convolutional and Transformer neural networks for image classification. We use PyTorch to write our classes and functions. Detailed division are

Yelun Bao	Section 3.1, 3.3 and part of 3.2
Jiangang Chen	Section 3.1 and 3.2

1 Understand Convolutions

The result of our custom convolution network has the same output compared with results from PyTorch-owned conv2D. As for the implementation, we first calculate output width and height based on the provided equations that use input size, stride size, padding size, and kernel. Then we unfold the input as well as the weight. We also reshape the weight and do careful matrix multiplication among input and weight. The unfolded output will be reshaped. This is for easier adding the bias term. In the final, the unfolded output will be folded based on the output size and the kernel size.
As for the backward propagation, we first get the unfolded output gradient and the unfolded weight. The unfolded weight will be reshaped by C_0 for careful matrix multiplication with unfolded output gradients. Then the output will be folded to folded input gradient based on the size of the input and kernel size. For the unfolded weights gradients, the unfolded one will be calculated through the matrix multiplication between unfolded gradient input and unfolded output gradient. In the final, the output unfolded gradient weights will be folded.

Examples of Adversarial Samples

2 Design and Train a Deep Neural Network

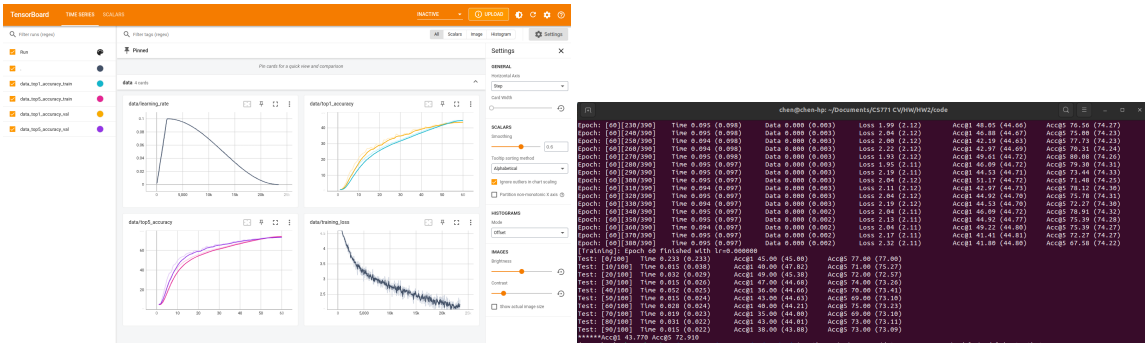
In this part, we will train a simple convoltional network and our custom convolution network from part I. Then a simpleViT is implemented as well as our self-designed network. The performance comparison between networks is processed.

2.1 A Simple Convolutional Network

We have trained a precoded simple convolution network. As shown in the following images, we get 43.77% top 1 accuracy and 72.91% top 5 accuracy after 60 epochs. The loss function gets steady after some epochs. The memory the network uses during the training is 3279MB. The training time is shorter and faster compared with our custom convolutions.

2.2 Train with Your Own Convolutions

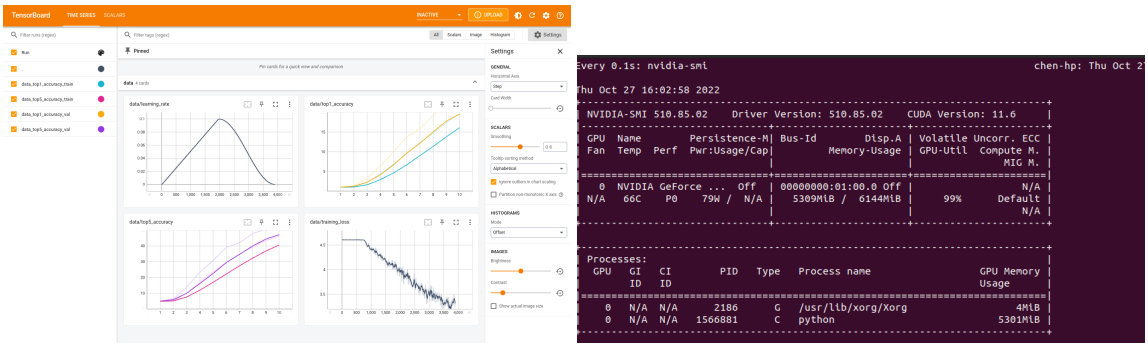
We've trained our custom convolution network in section 3.1 with 10 epochs, and the results are as follows. The accuracy at the top 1 is 22.22%, and 51.44% at the top 5 accuracy, which is only half of the accuracy compared with a simple network. Also, as the GPU monitoring figure indicates, the memory usage is 5301MB, which is



(Left) Tensor board result for the training of a simple convolution network; (Right) Accuracy result for the simple convolution network

bigger than the counterpart. For the training speed, we didn't record the whole training time. But I can feel that for each epoch. The training time is shorter compared with our own custom network. This can be seen by comparing the training status on the command line. The time for each epoch in a simple convolution network is around 0.097, while the time for each epoch in our convolution network is 0.329. The loss curves for both networks seem similar, not too much difference.

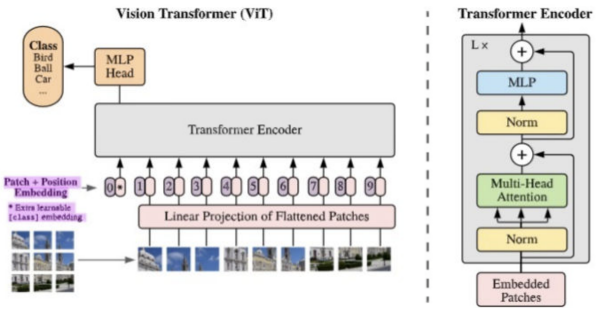
I think there are many reasons why our custom conv2D performance has lower performance. I can imagine two reasons. For example, there is a for loop in our code for adding bias to each unfolded output. This definitely will increase the running time. Another possible reason why memory occupation is large. One explanation is there are many matrix multiplications. The program needs places to store intermediate variables. Also, PyTorch is written in C/C++, so it is much more efficient compared with python.



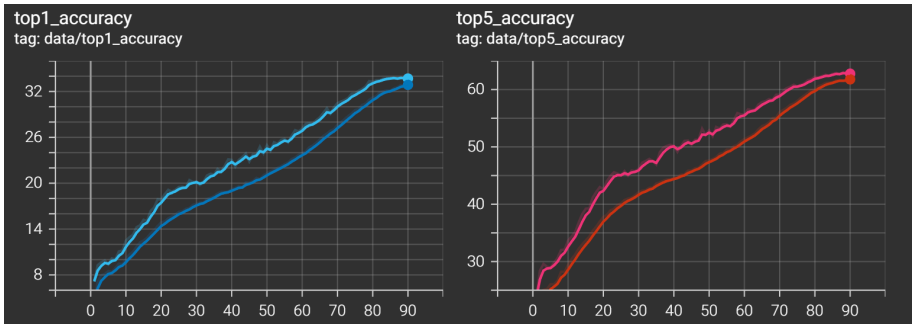
(Left) Tensor board result for the training of a our custom convolution network; (Right) GPU activities during training on our custom convolution network

2.3 Simple ViT

For the patch embedding and transformer block, we use the default parameters. And we've added a linear layer that maps from embedding dimensions to the number of classes. As for the forward pass, we basically follow the following flow chart. We first do the patching embedding, then add the position encoding to the patching embedding. After that, we pass the input to the transformer. After all the transformer, we mean the output and normalize it again. Lastly, the output is forward to the fully connected linear layer. As for improving the classification accuracy, so we first try the implement the classifier token, but since the patch embedding outputs four dimension tensors. It's hard to add a size-matched token directly to the inputs. Finally, we decide to abandon classifier token, which also gives us some decent result. The result is shown in the ViT model performance figure: In 90 epochs, we finally reach to Validation Accuracy @1 36.64% and 63.72% @5. But in the figure, we can observe that the Validation Accuracy is higher than training accuracy, which implies that our model have the potential to continue improving. We extend the training to 150 epochs and get Validation Accuracy 41.02% and 69.85%.



Flowchart for ViT implementation



ViT model performance

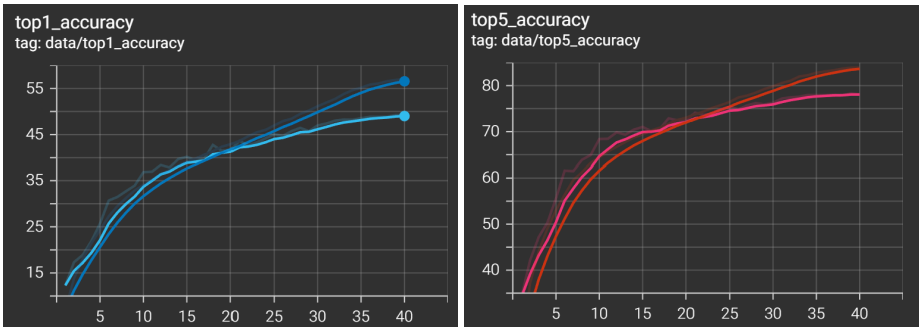
2.4 Design Your Own Network

We are going to improve the simple "SimpleNet" model in this part. We want to design a better network for this classification task in terms of accuracy and efficiency.

We consider adding more convolutional layers. However, simply adding convolutional did not improve our model's accuracy. We have to add residual connections to avoid divergence during training process.

We define a ResBlock class in the student-code file. We have 4 resblocks in the customized model. For each block, there are two convolutional layers. We normalize and activate the output after each convolutional layers. Then, add an identity shortcut from input to output. Use ReLU to activate the final output. You can run this model by command:
python ./main.py ../data –epochs=40 –use-custom-model

Our training scheme hyperparameters are all default except initial learning rate and weight decay. We set learning rate as 0.05 and weight decay as 0.01. It works well and convergence is really fast. Here is the result:



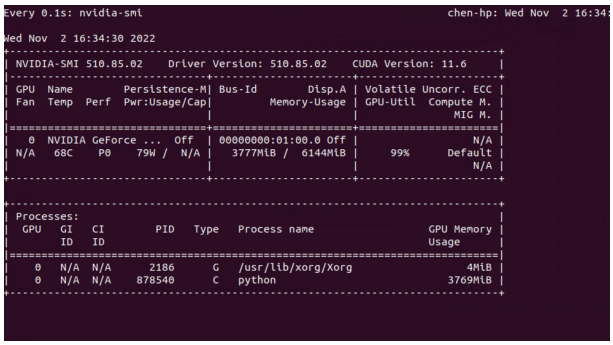
Custom model performance

From the figure, we can see that training and validation accuracy curves for the our custom resnet shows both good convergence and decent accuracy. Its Validation Accuracy @1 reaches 49.16% and reaches 78.3% @5. It is approximately 6% better than SimpleNet after 60 epochs, in both acc@1 and acc@5. This model is so light that

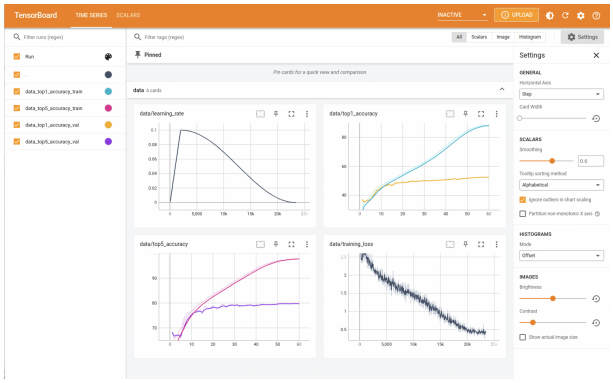
it only takes 25 minutes on single RTX 2070 super for 40 epochs. It achieves higher accuracy than the SimpleNet model while only costing around 80% training time.

2.5 Fine-Tune a Pre-trained Model

We will fine-tune a pre-trained ResNet18 model and compare it with our custom model. The memory usage for the reset-18 is 3769MB and the training time is about 1.219h. Compared to our MyResNet 25 minutes training time, it is really a long period.



GPU activities during training ResNet18



ResNet18 performance

The top1 validation accuracy for the 52.48%; the top5 validation accuracy for the resnet-18 is 79.54%. We can compare it with our custom model in the following table:

Model	Top-1 Accuracy	Top-5 Accuracy
MyResNet	49.16	78.30
ResNet18	52.48	79.54

We can observe that even though the our custom model's validation accuracy is slightly lower than the fine-tuned ResNet18, the training epochs and training time is much shorter than ResNet18. What's more, ResNet18's validation accuracy tends to converge around 30 fine-tune epochs, while our custom model's continue to increase during the 40 training epochs. Thus, our model is still a good choice when compared to ResNet18.

3 Attention and Adversarial Samples

In this section, we are going to talk about the implementation of Saliency Maps and Adversarial Samples.

3.1 Saliency Maps

We tried to minimize the loss of the predicted label and compute the gradient of the loss w.r.t. the input image. Firstly, forward and find the predict scores, pick up the label with the highest score and backward to compute its gradient w.r.t input. In this step, we note that we have to use gradient parameter when calling backward(), since

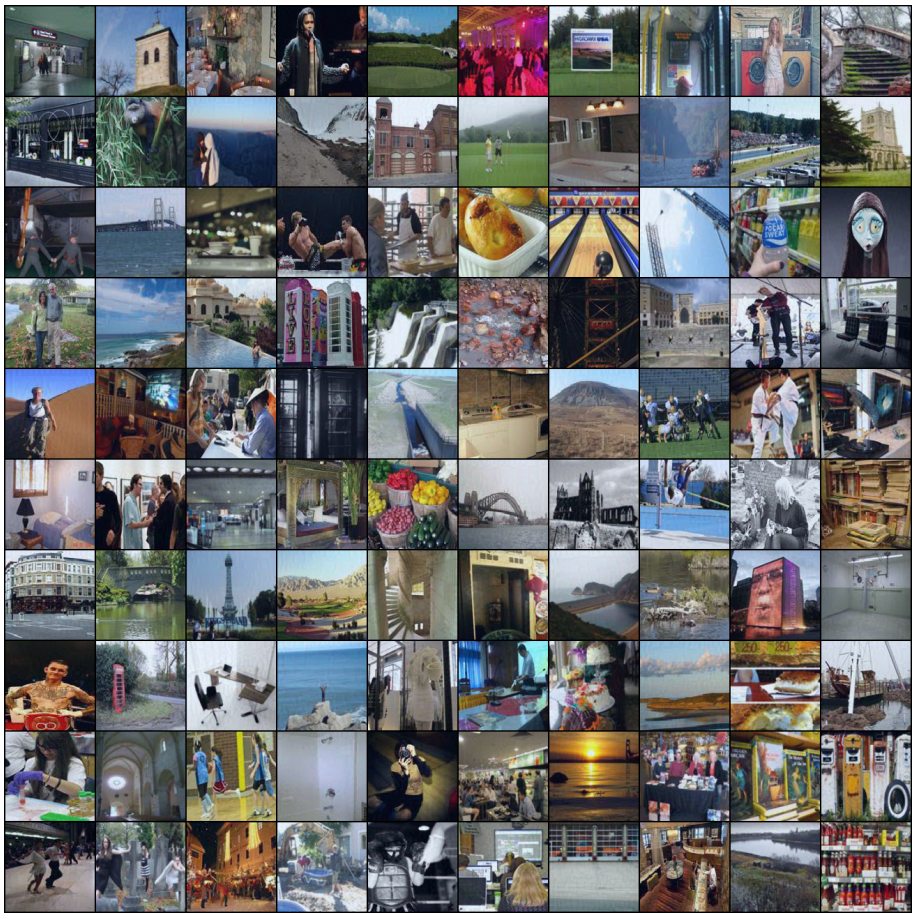
we have to identify the shape of maximum (loss function). Then, we take the absolute values of the gradients and find the maximum values across three channels. We output the max-gradient as the saliency map.
Here is the the visualization of Saliency Maps generated. The red dots indicate how these pixels affect the model's prediction. We can observe that most pictures are very intuitive. For example, in river water image, the red pixels are mainly overlaying the river surface.

Visualization of Saliency Maps

3.2 Adversarial Samples

Another thing we tried to trick with the input image is generating Adversarial Samples. In the implementation, we forward and find the predict scores, pick up the label with the lowest score (least confident prediction) and backward to compute its gradient w.r.t input. Same as last subsection, we note that we have to use gradient parameter when calling backward(), since we have to identify the shape of minimum (loss function). Then, we apply Projected Gradient Descent to the input with fixed steps. We take the sign of the gradient tensor, and directly increment the image with stepsize times sign of the gradient. At the end of each iteration, we clip the image within the ϵ-neighborhood of the original image. Also, empty the gradient to avoid a computational graph that grows indefinitely over time. After all steps, the image is the Adversarial Samples we need.
Here are the examples of Adversarial Samples generated. We can observe that the Adversarial Samples are very similar to the original images.
It is very hard for human eyes to tell the real one from the fake. But it will dramatically affect the accuracy of our model. We now attack our SimpleNet model with these adversarial examples. As the adversarial samples added to validation set, the accuracy of prediction decreases a bit. The accuracy are as shown as below:

Model	Top-1 Accuracy	Top-5 Accuracy
SimpleNet	43.930	73.110
SimpleNet(with attack)	31.990	57.720



Examples of Adversarial Samples

We did not observe significant change when increasing the number of iterations and reducing the error bound. It is likely that our PGD had already reached the bound and all changes were clipped.