DS542 Deep Learning Midterm Report

Hieu Nguyen

March 30 2025

1 AI Disclosure

ChatGPT was used in the process of doing this midterm assignment. The usage of ChatGPT is sparse, so to make things chronologically consistent, we will note where AI was used and what it was used for when we track our experiment.

2 Experiment Tracking

In this section, we will lay out in chronological order what was performed in this midterm project. This section will include a pre-study section where we study the theoretical aspects of convolutional neural networks to justify the design of our neural network. The following sections will analyze the design and performance of the CNN in the 3 parts of the exam.

2.1 Pre-Study

Even though this section is not an explicit requirement for the midterm, we will include it here briefly for completeness. The main text used for the pre-study was Prince's *Understanding Deep Learning* chapters 6 (gradients), 9 (regularization), and 10 (convolutional neural networks). In chapter 6, we studied stochastic gradient descent, momentum, and ADAM. The contents of this chapter will be used later to justify the usage of the optimizers that we implemented. In chapter 9, we studied double descent, and later tried to implement this into the neural network. Lastly, chapter 10 was used to understand the network architecture as a whole. The discussion here will be the foundation for a basic ablation study that is performed throughout this project. ChatGPT was used for clarification when studying the text. For example, when studying the effects of the ADAM optimizer, we asked ChatGPT to write a basic python script to implement and visualize the effects. When we studied convolutional neural networks, we asked chatGPT to give examples of a mathematical convolution.

Part 1: Establishing Baseline Performance

The first part of the midterm was to install all necessary dependencies, download all the training data, and then write the **starter_code.py** file to get a basic functioning model. In the class **SimpleCNN**, we asked ChatGPT to add a 2-layer CNN with 2x2 MaxPooling and a linear transformation to the incoming Cifar100 data. We asked ChatGPT to implement the **forward()** function in the **SimpleCNN** class. We manually chose to use the ReLU activation function on both convolutional layers since this is the most basic nonlinear activation function that we can implement (and have learned in class). The next section of this code builds the training and validation functions defined as **train()** and **validate()** respectively. We asked ChatGPT to complete this code in the interest of time. The only notable fine-tuning is the zeroing of the parameter gradients. Once we have all the prerequisite functions defined, we build the **main()** function. We kept all the pre-made parameters for this test, and manually filled in the code in all the TO-DO areas with the only exception being the scheduler, where ChatGPT recommended implementing the StepLR scheduler. The only notable modification that we made to the raw code was that we kept the data augmentation of the testing data the same as the training data. Lastly, we saved the result of our model on the OOD data as "submission_ood.csv". We did not change the name of this in the second part, so the results from this test are lost.

The purpose of this section was to get an understanding of the code and the function skeleton for the model that we will be building in the later parts of the midterm competition. The only major change moving forward will be the usage of a pretrained ResNet18 and ResNet50.

Part 2: Adding Sophistication

Despite the title of this section being "adding sophistication," the modifications that we made were, in fact, not very sophisticated. The only modifications from the **starter_code.py** are the deepening of the network and fine-tuning of the hyperparameters. We created a file called **Midterm_part2.py** duplicated the **starter_code.py** file and added 2 more convolutional layers to the network, making it a 4-layer convolutional network. Since we don't have control over the number of nodes in the CNN (as it corresponds to the number of pixels of the images we are feeding into the network), we can add more parameters into the network by increasing the depth. This time around, we changed the activation function across all layers from ReLU to leaky ReLU. Our justification for this is because ReLU will ultimately clip half of the function that is passed through it. We still want some information retained, especially since the network is not that deep, and so we chose to use leaky ReLU. We asked ChatGPT to do the dimensional analysis for us when changing from a 2-layer network to a 4-layer network. We also increased the batch size to 64, decreased the learning rate to 0.001, and increased the number of epochs to 10. We kept the optimizer, loss function, and scheduler the same as the last section. The one big modification that we made was augmenting the training data to include various distortions such as random cropping, rotations, and blurring. To do this, we asked ChatGPT to implement the augmentations using torchvision's **transforms** module.

The result of these modifications not only boosted the accuracy of the clean Cifar100 testing set from being 1%. This is the first run that we uploaded our results to Kaggle, and in doing so, got an accuracy score of approximately 35%. This is a 4% difference from the benchmark for this exam, which means that our hypothesis that making the network deeper and fine-tuning the hyperparameters would increase the accuracy is correct. This gives us hope that, like the gradients of our model, we are moving in the right direction.

Part 3: Increasing Confusion and Decreasing Hope

With the preliminary results, we reasoned (surprisingly incorrectly) that if we used a pre-trained CNN like ResNet18, we could get an increase in accuracy on the OOD data. As the title suggests, this is incorrect. The first major modification that we made was the removal of the **SimpleCNN** class and replaced it with the initialization of the ResNet18 model from PyTorch with the default weight settings. We chose ResNet because it has been stated to generalize well on a vast set of data. We chose ResNet18 specifically because it is the simplest ResNet model and we wanted to see how much of a boost we can get going from our humble **SimpleCNN** to a pretrained model in PyTorch. We asked ChatGPT to implement loading the ResNet18 model as well as label smoothing and the forward pass (defined as **LabelSmoothingLoss()** and **Forward()** respectively). Since we are working with a more ”sophisticated” model, we decided to add ”sophisticated” optimization techniques, such as switching to the ADAM optimizer as well as changing the scheduler to being OneCycleLR. We kept all other hyperparameters and the data transformations the same.

The increase in model complexity necessarily meant the increase in training time. It took the program approximately 7 hours to train. Upon finishing, we got a very high accuracy score on the clean Cifar100 data of approximately 75% but a very poor OOD accuracy of 3%, which is a decrease of almost 32%. The discrepancy between the high accuracy on the clean Cifar100 test and the very low OOD test indicated that perhaps we are overfitting the training data, and that is why we are not getting a good enough generalization to the distorted data. We now have 2 ways around this: either we go back to basics and fine-tune our simple model or we hope to see a double descent phenomenon by increasing the model complexity even more. We end up doing both.

Part 3 Continued: Even More Confusion And Tears

The deadline for the project is coming up in a few days (had to start late because other classes' work was time-consuming), and our models are taking many hours to compute. Although we have a PC at home that we can keep the programming running indefinitely, we are not home long enough for this to matter. We can combat this by running all the programs on the SCC. We also note that the SCC is a computing cluster where we can queue as many CPU units as we (theoretically) please. However, in order to take advantage of this, we need to implement some form parallelization. Being unfamiliar with how PyTorch does parallelization, we asked ChatGPT to implement parallelization using the cores allotted to us by the SCC. The only modification, as it turned out, was only 2 lines because PyTorch modules can be forced to parallelize their code executions. We add these 2 lines under the configuration section of the function **Main()**.

If there were to be an ablation study, this would be the time that we did it. We queued 3 separate jobs on the SCC and made small modifications to the code. These modifications include changing the batch size, changing the number of epochs, as well as changing how the training data is augmented. In most of these cases, we still observe that the evaluation on the clean Cifar100's test accuracy is very high and the OOD accuracy is very low (varying from 5% to 15%). We ran multiple tests on the SCC as well as using our PC at home to do some of the testing. Despite all this, we get very volatile accuracy scores on the OOD test.

2.2 Part 3 Continued: The VERY Dim Light At The End Of The Tunnel

Countless hours staring into the abyss (my computer screen) have passed, and the only thing I can do now is to go on the extremes. Either we use a super complex pretrained model or we use a very simple model; but in both cases, we send the number of epochs to orders of 30-100 and decrease the learning rate to .0001. With the existence of the SCC, why not do both? For the first case, we initialize a ResNet50 model using the exact same parameters as the ResNet18's. This time, we make the ResNet50 model run on 30 epochs with a learning rate of .0001. We keep all the data transformations the same except for the inclusion of Gaussian noise blurring. As for the simple model, we went back to using the 4-layer network on 50 epochs.

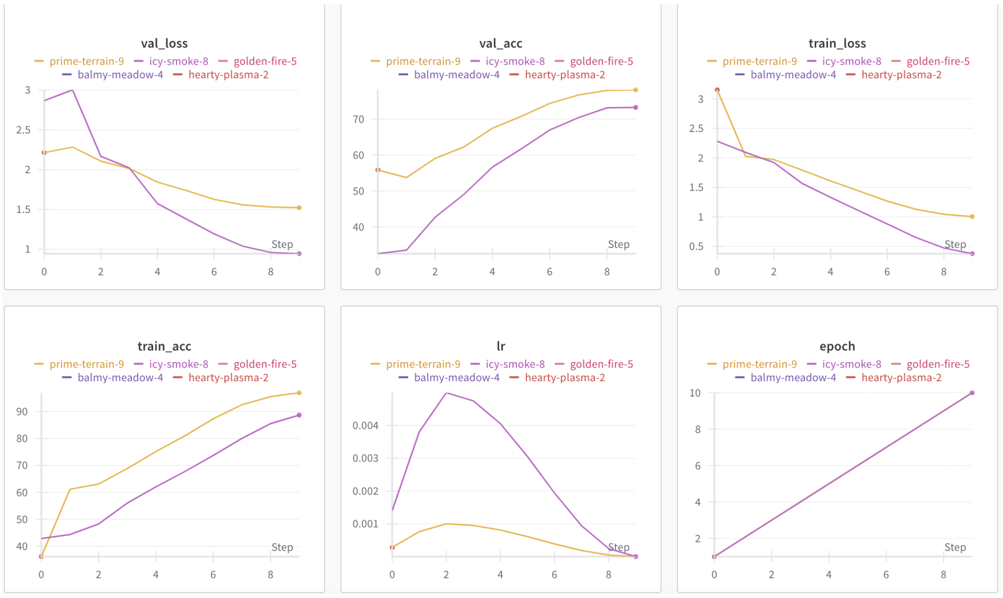

Figure 1: The figures of some the model performance as recorded on WandB.

When we run this model, we get an accuracy score of .44 and .41 on the OOD test respectively. Perhaps for the former, this is due to the double descent phenomenon. At this point, we are done with our work, but it is Saturday night so maybe we can attempt to go higher. We initialize 3 models on the SCC, each with variations on increasing network depth and number of epochs trained. Unfortunately, on Sunday morning, we woke up to the SCC telling us that there was an error in the code but only after training the 100 epochs. We might never know because the submissions on Kaggle are almost closed. The names of the files are called **Midterm_part3ResNet50.py** and **Midterm_part3Simple.py**

3 Analysis

In this section, we will analyze the (few) successes of our models as well as their (many) failures. To talk about the failures, we realized that increasing the model complexity (using a pretrained ResNet) sometimes leads to a significant decrease in performance. This could be due to a lot of different variables, such as not enough time for the models to find a minimum. When the complexity of the model increases, the terrain of the loss function can admit many local minimums, and it is possible (we saw it happening) that the gradients are getting stuck. This problem is only exacerbated by the fact that we decreased our learning rate, making it potentially harder and longer to escape from local traps. This also serves as an explanation as to why when we increased the training time on the ResNet50 model, we got a higher accuracy on the OOD. This also explains why when we used our simple model from part 2 and increased the training time, we also saw an improvement in test performance.

If we had more time, we would like to explore the more complex ResNet models such as ResNet101 or ResNet152 and test our hypothesis that the low test performance was due to the model not training for long enough. If we take advantage of the SCC, perhaps we can have models that run for days or weeks and this could potentially lead to a better generalization. Alternatively, we could do the same thing with the simple model where we gradually increase the depth and allow it to

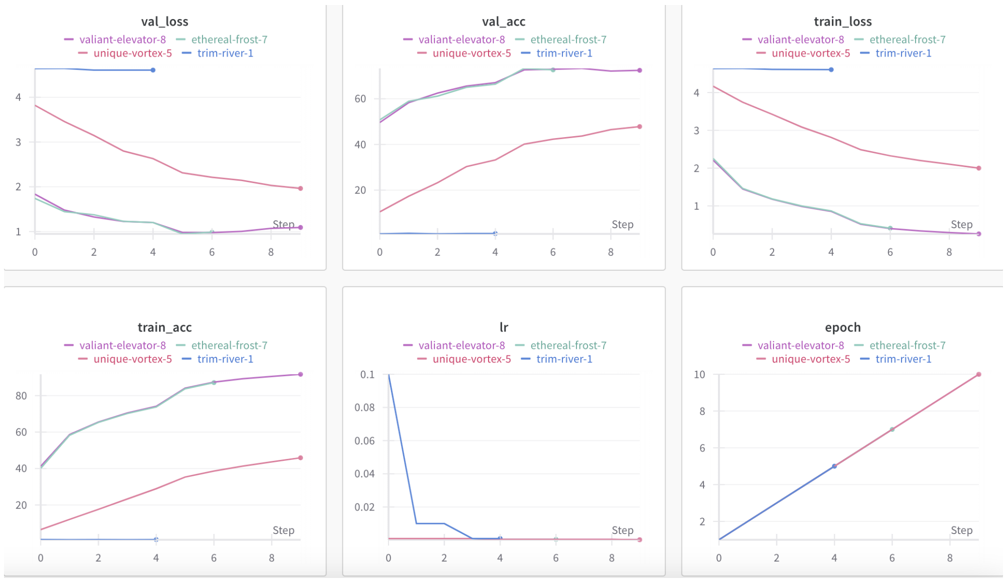

Figure 2: The rest of the model performances as recorded on WandB

train for longer and then show the performance of the model as a function of the model complexity.

In figures 1.1 and 1.2, we plot some of the results from our testing. We note that since most of the testing is done on the SCC, we were not able to connect it to WandB.