# Midterm Challenge Report

## AI Disclosure

Throughout this assignment, I used ChatGPT (GPT-4) as a learning and support tool to help me complete the three parts of the deep learning midterm project. My usage of AI was limited to guidance, clarification, and troubleshooting rather than direct code generation. Specifically, I consulted ChatGPT when I encountered Python or PyTorch-related errors such as ModuleNotFoundError, installation and environment issues on BU's SCC system, and configuration of experiment tracking tools like Weights & Biases (W&B). I also asked for feedback on model design, including whether transfer learning was appropriate for Part 3, how to modify the final fully connected layer in ResNet architectures, and where to apply techniques like dropout.

In addition to technical help, I used ChatGPT to explain concepts such as data augmentation strategies (e.g., RandomRotation, ColorJitter, Cutout) and hyperparameter tuning, including trade-offs between learning rate, batch size, and training duration. For model evaluation, I referred to ChatGPT for help understanding the evaluation pipeline and for confirming how to implement functions that generated predictions across clean and OOD test sets. This included how to pass predictions into helper functions like create_ood_df() and properly save outputs to CSV files with to_csv() for Part 2 and Part 3.

I did not use Copilot, Cursor, or any other AI tools during this assignment. All code was manually written and tested by me, and the structure, design choices, and parameter selections were ultimately my own decisions based on experimentation and learning. I used AI assistance as a supplement to my understanding.

## Model Description

In Part 1, I implemented a simple CNN model from scratch using PyTorch. The model consisted of two convolutional layers followed by max-pooling and two fully connected layers. It was a lightweight architecture suitable for initial experiments on CIFAR-100 and achieved reasonable accuracy, helping to establish a performance baseline.

In Part 2, I switched to a ResNet-18 model from torchvision.models without pretrained weights. I replaced its final fully connected layer with a new nn.Linear layer with 100 outputs to match the CIFAR-100 classes. This model provided a deeper architecture than Part 1 and led to improved performance, but training from scratch made it harder to reach optimal accuracy.

Part 3 introduced the most significant improvement. I adopted a pretrained ResNet-50 model using ResNet50_Weights.DEFAULT. To adapt it for fine-tuning, I unfreeze all layers (param.requires_grad = True) so that the entire model could be trained on CIFAR-100. I modified the final classification head by adding a Dropout(0.5) layer followed by a Linear layer with 100 outputs, helping to reduce overfitting and improve generalization. The model's deeper architecture, combined with pretraining on ImageNet, allowed it to learn high-quality features relevant to the CIFAR-100 task. This setup produced the best results among all three parts.

## Hyperparameter Tuning

In Part 1, I used default learning rates and batch sizes common for small CNNs, such as a learning rate of 0.001 and batch size of 32. In Part 2, after switching to ResNet-18, I

experimented with batch size 64 and learning rates 0.001 and 0.0005, but training from scratch limited the benefits.

In Part 3, I significantly improved the model's performance through systematic hyperparameter tuning. Since I used a pretrained ResNet-50, I opted for a small learning rate (0.0001) to avoid overwriting the pretrained weights too aggressively. I chose the AdamW optimizer for its decoupled weight decay, which improves generalization and helps prevent overfitting. Additionally, I replaced StepLR with CosineAnnealingLR, which gradually decays the learning rate in a smooth, non-linear fashion, allowing the model to converge more reliably across 50 epochs. The batch size was increased from 32 to 64 to better utilize GPU memory and improve the stability of training, especially when using data augmentation. I also tuned the label smoothing parameter in the cross-entropy loss function (set to 0.1), which helped soften the targets and made the model more robust against noisy labels. To further improve training, I increased the number of threads using torch.set_num_threads(30) and torch.set_num_interop_threads(30) to parallelize data loading and processing on SCC, significantly reducing CPU bottlenecks. These collective adjustments in hyperparameters contributed to better generalization and helped me achieve the highest performance in Part 3.

Regularization Technique

In Part 1 and Part 2, regularization was minimal and mostly implicit. For Part 1's SimpleCNN model, I relied on its relatively shallow architecture and early stopping (by selecting the best model based on validation accuracy) to mitigate overfitting. In Part 2, I used ResNet18 without

adding explicit regularization methods, relying instead on its built-in BatchNorm layers and the data splits to provide enough generalization.

In contrast, Part 3 included several strong regularization techniques to improve generalization and robustness. I added a Dropout layer with p=0.5 before the final fully connected layer of ResNet50 to randomly deactivate neurons during training and prevent co-adaptation. I also used label smoothing (0.1) in the loss function, which encourages the model to avoid overconfidence and better handle noisy labels. Additionally, switching to AdamW optimizer provided built-in weight decay, which helps reduce overfitting by penalizing large weights. These methods together made Part 3 more robust, leading to a significantly improved accuracy and stronger performance on out-of-distribution data.

Data Augmentation Technique

In Part 1, I used no data augmentation beyond basic normalization, as the focus was on building a working CNN model from scratch. In Part 2 with ResNet18, I applied a standard augmentation pipeline including RandomHorizontalFlip and RandomCrop with padding, along with normalization, which slightly improved generalization and robustness.

For Part 3, I designed a much richer data augmentation strategy to boost model performance and tackle overfitting. The training pipeline included RandomHorizontalFlip, RandomCrop (with padding = 4), RandomRotation (10 degrees), and ColorJitter with adjustments to brightness, contrast, saturation, and hue. These help simulate a wider variety of real-world scenarios. Additionally, I used RandomErasing, which randomly masks out image patches during training,

forcing the model to learn more robust features rather than relying on specific visual regions. This stronger augmentation helped the ResNet50 model generalize better and contributed significantly to the improved test and OOD performance.

Results Analysis

In Part 1, the simple CNN I built achieved a test accuracy around 23%, which is expected given its limited depth and lack of advanced regularization or pretrained features. Part 2 improved significantly with ResNet18, reaching around 48% accuracy on the CIFAR-100 test set and approximately 0.22 on the OOD challenge. This demonstrated the strength of using a deeper residual architecture.

Part 3 showed the biggest improvement. After switching to ResNet50 with pretrained weights, applying aggressive data augmentation, dropout, label smoothing, and switching to AdamW optimizer with CosineAnnealingLR, the model reached a clean test accuracy of 61.39% and OOD score of 0.44. This indicates that both in-distribution and out-of-distribution performance benefitted from these enhancements. The model showed strong generalization and robustness to distorted inputs. However, training time was longer and memory consumption higher due to the larger model and batch size. One limitation is that I did not do a full hyperparameter grid search, and the model still sometimes overfits slightly on the training set. For future improvement, I would experiment with MixUp, CutMix, longer training (e.g. 100 epochs), or even larger ViT or Swin architectures if resources allow. Exploring learning rate warmup and advanced schedulers like OneCycleLR could also help fine-tune training dynamics.

Experiment Tracking Summary

All experiments for this project were tracked using Weights & Biases (WandB), which allowed for comprehensive monitoring of training metrics, hyperparameters, and model performance across runs. Figure 1 shows the full dashboard view with comparisons of all 18 runs, enabling effective observation of how changes in configuration affected accuracy and loss over epochs. Each run is automatically logged with training/validation loss and accuracy, learning rate, and epoch progression.

For instance, Figure 2 illustrates the performance of an earlier run (gallant-snowflake-11) during hyperparameter tuning. This run achieved a validation accuracy of 65.58% by epoch 10 with a learning rate of 5e-5, showing strong potential but limited training duration.

The final submitted model, driven-water-18, shown in Figure 3, achieved the best performance with a validation accuracy of 60.84% and a training accuracy of 73.34% at epoch 50. This run benefited from extended training, cosine annealing learning rate scheduling, and well-tuned regularization. The detailed WandB logs allowed for clear performance comparisons and informed decisions when selecting the best checkpoint for final evaluation and submission.

These visualizations and logs provided crucial insights for iterative improvement and selecting the optimal model for Part 3 of the assignment. (see figures in the next page)

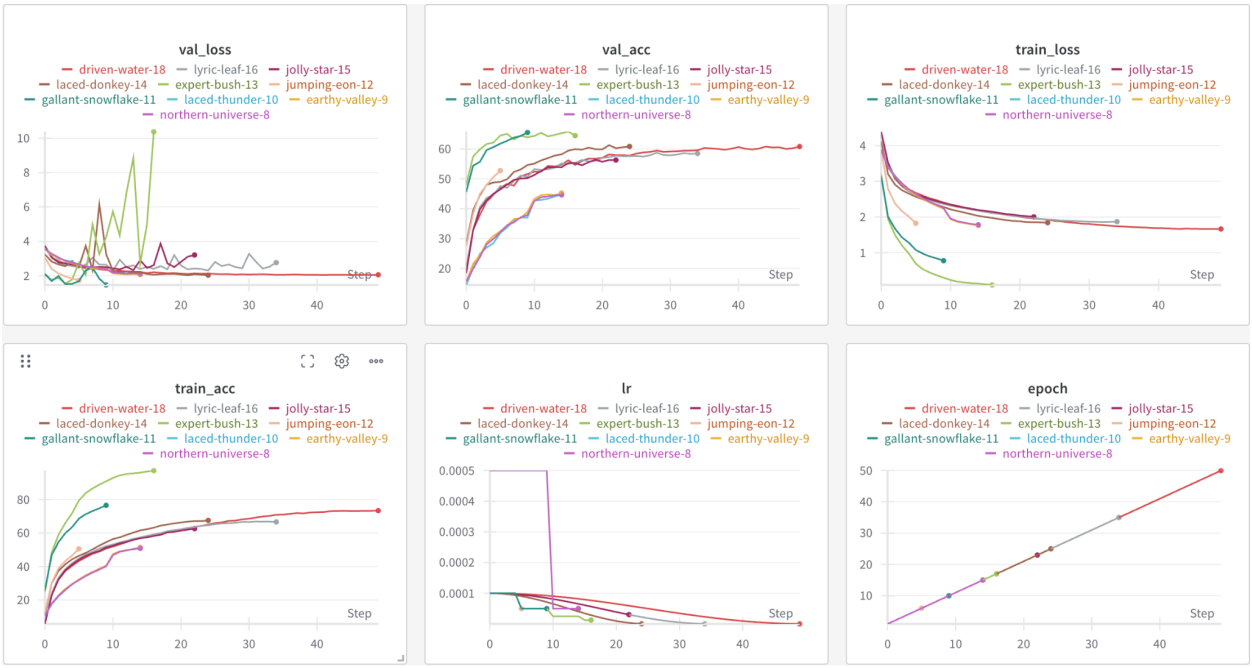


Fig.1: dashboard

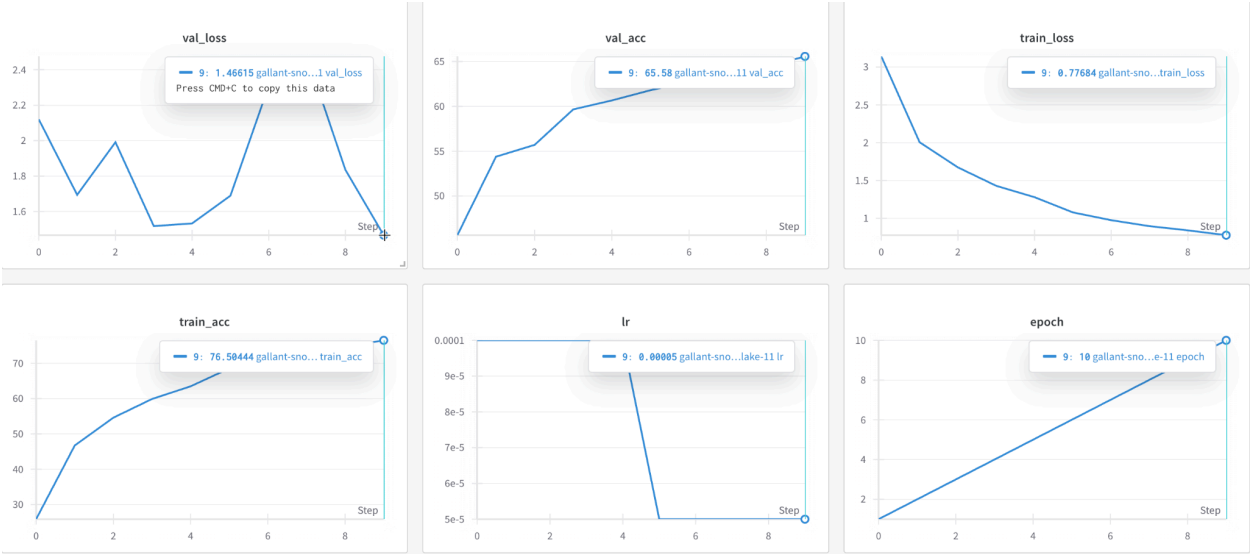


Fig.2: gallant-snowflake-11

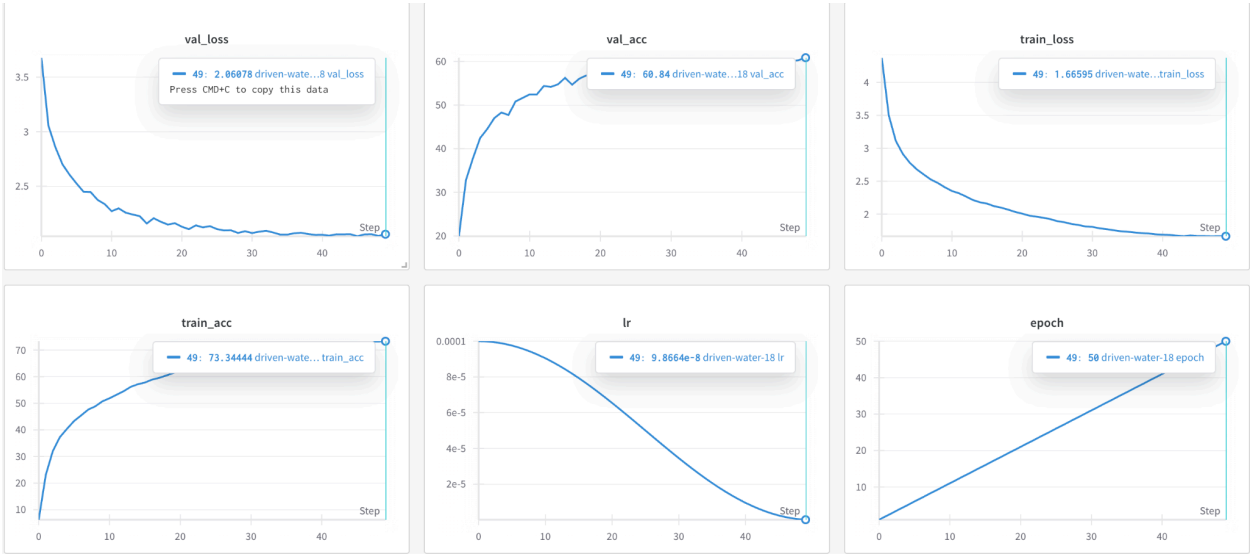


Fig.3: driven-water-18