# DS542 Midterm Challenge Report

**Serena Theobald**

## Introduction and Model Description

This repository contains the code for the midterm challenge for DS542. My part 1 code, “part1_simple_cnn.py”, involves implementing a relatively simple CNN model on the CIFAR-100 dataset to establish a performance baseline. I intentionally designed a lightweight architecture with three convolutional layers, max pooling, and dropout to evaluate how a basic convolutional pipeline would perform on CIFAR-100. The inclusion of dropout, data augmentation (horizontal flip, crop, color jitter), and a cosine annealing learning rate scheduler was meant to reduce overfitting and provide a stable training process, even though the model's capacity was limited.

My part 2 code, “part2_sophisticated_cnn.py”, uses a more advanced architecture by leveraging ResNet18 from torchvision to improve upon the baseline. I selected ResNet18 to take advantage of its residual connections, which allow for deeper networks without vanishing gradients. I modified the final classification layer to include dropout (p=0.2) and mapped it to 100 output classes. I chose to train this model from scratch (without pretrained weights) to assess how well a deeper architecture alone could generalize, relying on carefully tuned regularization, learning rate scheduling (cosine annealing), and data augmentation (rotation, color jitter) to improve generalization.

My part 3 code, “part3_transfer_learning.py”, focuses on transfer learning by fine-tuning a pretrained ResNet50 model on CIFAR-100. I chose ResNet50 due to its strong performance on a wide range of image classification tasks and its ability to transfer learned representations from ImageNet. To adapt it to CIFAR-100, I replaced the final classification head with a dropout layer and a linear layer for 100 classes. I applied label smoothing (0.1) and weight decay to mitigate overfitting and improve calibration. Cosine annealing ensured gradual learning rate decay, avoiding early convergence. I used a larger batch size and stronger augmentations (e.g., rotation, color jitter) to take full advantage of SCC GPU resources and enhance robustness.

My ultimate goal was to exceed a benchmark score of 0.397 on the Kaggle leaderboard, with all models built solely from convolutional and linear layers while avoiding implementing Transformer-based architectures into my code.

## AI Disclosure

I used ChatGPT to help with revising class names, restructuring functions, and improving the overall readability and modularity of my code. I specifically asked GPT to help me restructure the code under my functions and print statements in a clearer and structured way. This included renaming variables for clarity, reordering code blocks to improve structure, and adding consistent inline documentation. For example, I asked ChatGPT to help restructure function headers and improve modularity by consolidating configuration settings into a CONFIG

dictionary (instead of passing many separate arguments). In addition, ChatGPT suggested using print("\nModel summary:\n", model) and using wandb.watch() to visualize gradients and weights. Based on advice from ChatGPT, I switched to CosineAnnealingLR and CosineAnnealingWarmRestarts for smoother learning rate decay. Furthermore, ChatGPT helped me revise class names and naming conventions (e.g., my class name is called `ResNet50Transfer`), comment style and logical ordering, and improved separation between data preparation, training, and evaluation blocks in main().I also asked ChatGPT for regularization strategies (e.g., label smoothing, dropout rates), and it helped suggest that `label_smoothing=0.1` might improve generalization. I implemented and tested this myself. These suggestions helped me write cleaner, more effective, and maintainable code that allowed me to get better model results.

However, I implemented all core functionality, including model architectures, training loops, evaluation logic, and data preprocessing, by adhering to the structure provided in "starter_code.py." I manually typed out key components of the pipeline, including hyperparameter tuning, data augmentation, logging, and integration with wandb. I implemented the logic in train() and validate() functions myself for all parts. The core logic, such as batching, optimizer steps, accuracy tracking, was written by me. For hyperparameter configuration, I defined and adjusted all hyperparameters, including batch size, learning rate, label smoothing, and dropout. I manually defined transforms using torchvision.transforms.Compose and handled the CIFAR-100 dataset split using random_split(). My evaluation and submission generation was based on the provided eval_cifar100.py and eval_ood.py scripts.

## Regularization Techniques

Across all three parts, I used regularization as a key design consideration to improve generalization and prevent overfitting, especially given the small size and high complexity of CIFAR-100. In Part 1, I added a dropout layer with a rate of 0.3 before the classifier, which helped reduce overfitting even though the model's capacity was limited. In Part 2, the dropout rate was reduced to 0.2 to balance regularization with the deeper representational power of ResNet18. For Part 3, I again used dropout (p=0.2) in the final classification head of ResNet50. This addition helped stabilize training during fine-tuning and improve robustness on the validation and test sets. Furthermore, I consistently used the AdamW optimizer with a weight decay factor of 1e-4 in all three parts. This form of L2 regularization penalized large weights, which contributed to reducing overfitting, especially in the models trained from scratch (Parts 1 and 2). For only Part 3, I used label smoothing (0.1) to soften the targets during training, making the model less confident in its predictions and helping prevent overfitting to noisy or ambiguous examples. I found this technique to be effective when combined with pretrained weights. Lastly, I implemented a cosine annealing scheduler across all models. This allowed for dynamic adjustment of the learning rate during training, which encouraged better convergence in early epochs and fine-tuning toward the end.

## Data Augmentation Strategy

Data augmentation was an important aspect for improving generalization across all three parts, particularly given the relatively small and varied nature of the CIFAR-100 dataset. For part 1, I applied basic augmentation including RandomHorizontalFlip, RandomCrop with padding, and light ColorJitter. This combination provided enough diversity to reduce overfitting while maintaining the simplicity of the baseline model. I also added additional augmentations, such as increased ColorJitter strength and RandomRotation, in addition to flipping and cropping. This helped the deeper ResNet18 model generalize better and reduced the gap between training and validation accuracy. These augmentations were tuned to simulate realistic image distortions that could improve feature learning. Lastly, for part 3, I maintained strong augmentations like RandomCrop, HorizontalFlip, Rotation, and strong ColorJitter. Given that this model leveraged a pretrained ResNet50, I think it benefited from diverse input variations during fine-tuning. These augmentations, along with normalization to ImageNet statistics, ensured the pretrained model adapted smoothly to CIFAR-100.

## Hyperparameter Tuning

In developing models for the CIFAR-100 dataset, I developed a specific hyperparameter tuning process to optimize performance across the three distinct model architectures.

For the SimpleCNN, a lightweight architecture, I initiated training with the AdamW optimizer, setting a learning rate of 0.005 and a batch size of 64. The model was trained over 50 epochs, incorporating a dropout rate of 0.3 to mitigate overfitting. A cosine annealing learning rate scheduler was employed to adjust the learning rate dynamically, promoting smoother convergence. Data augmentation techniques such as random cropping with padding, horizontal flipping, and light color jitter were applied to enhance the model's generalization capabilities. These choices aimed to balance efficient training with the model's limited capacity, providing a baseline for performance evaluation.

Transitioning to a more complex architecture in Part 2, I implemented ResNet18 without pretraining. The learning rate was reduced to 0.001, and the batch size was set to 32 to accommodate the increased model complexity and computational demands. Training spanned 60 epochs, with a dropout rate of 0.2 applied to the classifier head to prevent overfitting. The cosine annealing scheduler continued to be utilized for learning rate adjustment. Data augmentation strategies were consistent with those in Part 1, ensuring robustness against overfitting and enhancing the model's ability to generalize from the training data.

In Part 3, leveraging transfer learning with a pretrained ResNet50 model, I further refined the hyperparameter configuration. The learning rate was set to 0.0005, reflecting the need for careful fine-tuning when adapting a pretrained model to a new dataset. A batch size of 64 was chosen, balancing computational efficiency with the depth of the ResNet50 architecture. Training was conducted over 30 epochs, incorporating label smoothing in the loss function to enhance model robustness. The dropout rate remained at 0.2 in the classifier head. I employed more aggressive data augmentation techniques, including random cropping with padding,

horizontal flipping, color jitter, and random rotation, to simulate a wide range of image variations and further improve generalization.

Throughout all experiments, the cosine annealing learning rate scheduler was consistently applied to prevent premature convergence and maintain training stability. The number of worker threads for data loading was set to 4 across all models, optimizing data pipeline efficiency. This specific approach to hyperparameter tuning was significant in achieving progressive improvements across the three parts of the project.

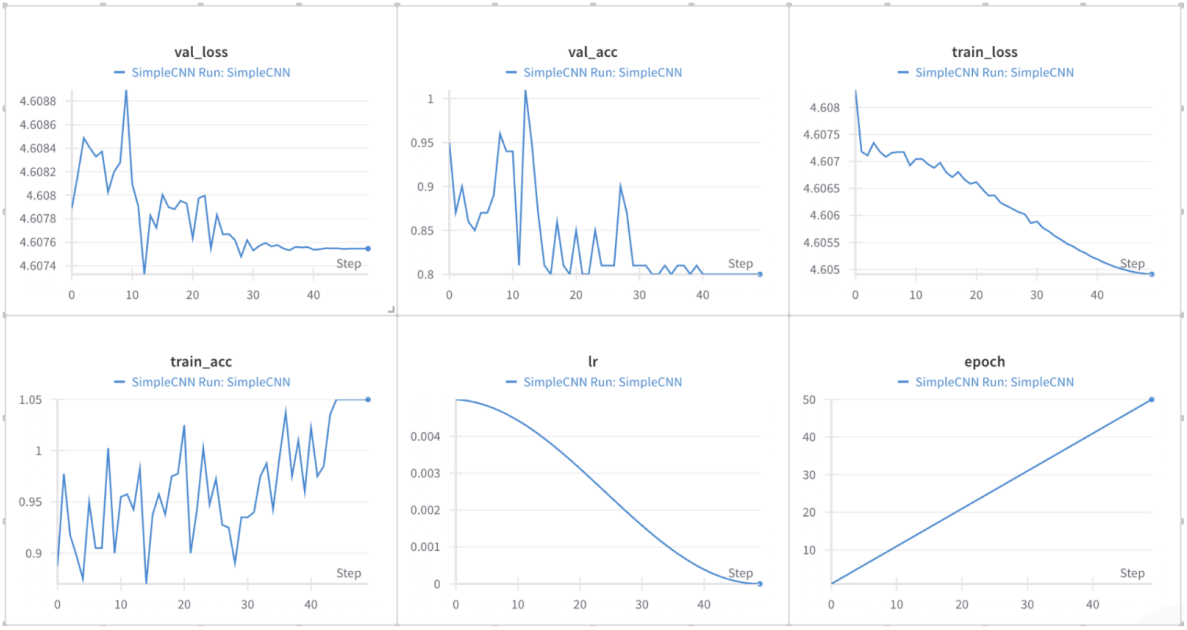
## Experiment Tracking with WandB

All training and evaluation was tracked using Weights & Biases (WandB). For each model, WandB automatically logged metrics such as loss, accuracy, learning rate, and configuration parameters for every epoch. This allowed me to compare training and validation performance across experiments and pinpoint key performance gains. Below is a summary of my final scores:

| **Part** | **Submission File** | **Best Kaggle OOD Score** | **Username** |
|---|---|---|---|
| Part 1: Simple CNN | submission_ood.csv | 0.01005 | Serena Theobald |
| Part 2: Sophisticated CNN | submission_ood_sophisticated.csv | 0.42598 | Serena Theobald |
| Part 3: Transfer Learning | submission_ood_transfer_resnet50.csv | 0.48360 | Serena Theobald |

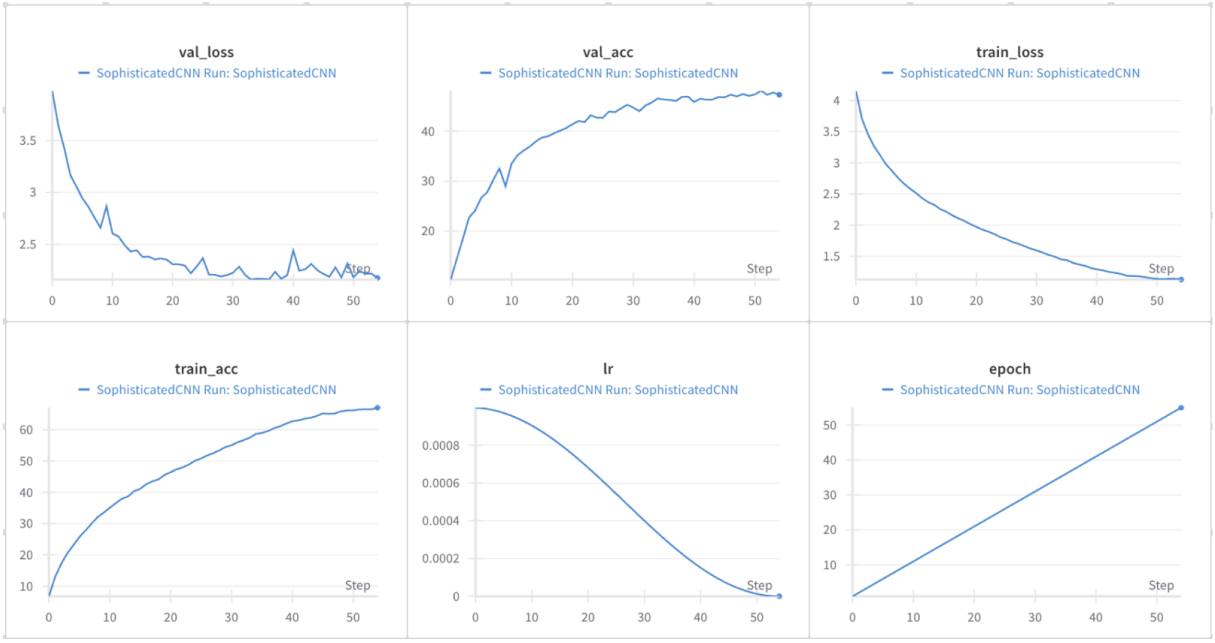
In addition, I used the WandB Reports UI to generate a visual summary of my experiments, including metric plots for each model (train/val accuracy and loss, learning rate schedule, and epoch progression). Screenshots were captured for each report, providing a visual audit trail of training behavior and confirming the benefits of increasingly complex modeling strategies.

# Part 1: Simple CNN Model



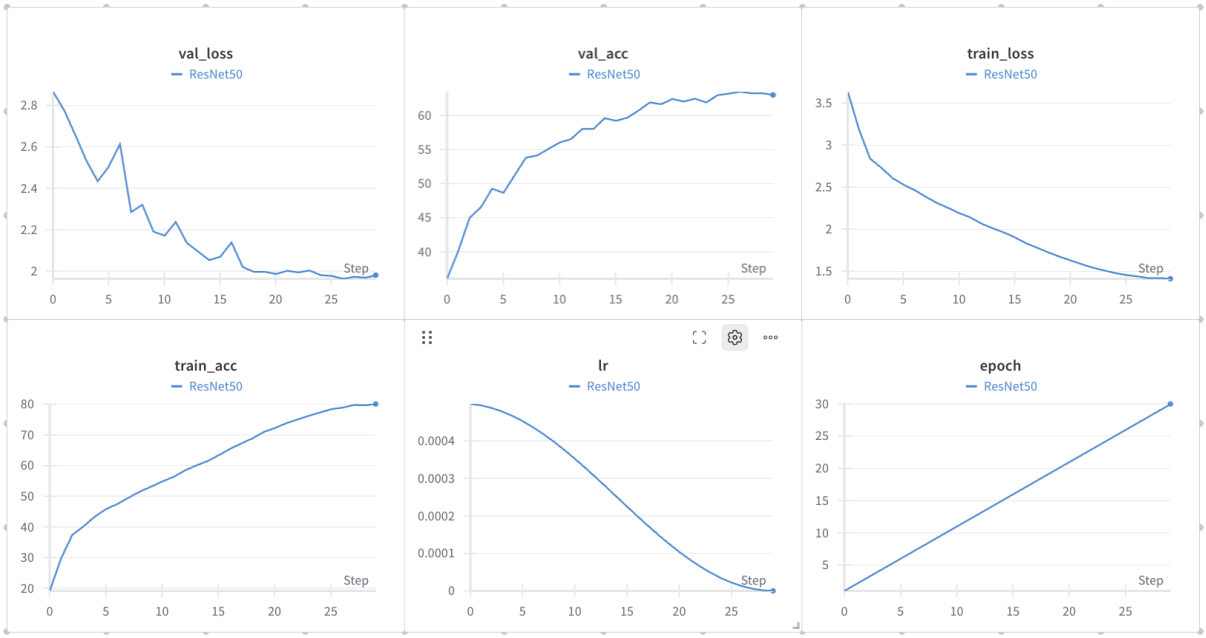

In Model 1, training loss decreased slightly across 50 epochs, while validation loss remained stagnant around 4.6. Training accuracy slightly increased to 1.1%, while validation accuracy hovered around 0.8 to 1.0%.The WandB logs clearly demonstrate that the SimpleCNN failed to learn complex patterns in CIFAR-100. The stagnation of both validation loss and accuracy indicates severe underfitting, despite use of regularization (dropout) and augmentation. The experiment tracking made it clear that this model lacked learning capacity, validating its use as a low baseline for comparison.

# Part 2: Sophisticated CNN Model




In Model 2, the training loss showed a steady decline, validation loss dropped significantly, and validation accuracy climbed consistently to nearly 50%. Training accuracy exceeded 65% by the 50th epoch. The tracking plots confirm that the ResNet18 architecture successfully captured richer features. The divergence between training and validation metrics toward the end suggests mild overfitting, but overall performance was stable.

## Part 3: Transfer Learning Model




In Model 3, both training and validation losses steadily decreased. Validation accuracy surpassed 66%, and training accuracy approached 80%. The learning rate followed the expected cosine annealing schedule. These plots show the power of transfer learning, since validation curves flattened near convergence, suggesting stable generalization. The consistent gap between training and validation indicates the model generalized well but could still benefit from further tuning or regularization. WandB tracked the impact of fine-tuning, label smoothing, and data augmentation. The curves clearly visualize that pretrained features, when coupled with a fine-tuned pipeline, yielded the most optimal performance.

## Results Analysis

### Part 1: SimpleCNN

The SimpleCNN model achieved a final OOD score of 0.01005. Based on the experiment tracking tool above with WandB, my training accuracy marginally exceeded 1%, while validation accuracy remained between 0.8% to 1%. Validation loss plateaued around 4.607, indicating the model's inability to learn meaningful representations from the CIFAR-100 dataset. Despite implementing dropout and data augmentation techniques, the model's shallow architecture led to underfitting. This outcome underscores the necessity for deeper, more complex models to capture the intricate patterns within CIFAR-100 images.

### Part 2: SophisticatedCNN (ResNet18)

Transitioning to ResNet18 resulted in significant performance improvements. Training accuracy surpassed 65%, and validation accuracy approached 50%. The final OOD leaderboard score was 0.42598, demonstrating the efficacy of deeper networks with residual connections in facilitating gradient flow and learning complex features. Notably, this model was trained from scratch without leveraging pretrained weights, yet it outperformed the benchmark. The incorporation of dropout, weight decay, and data augmentation contributed to enhanced generalization. However, the gap between training and validation accuracy suggests potential overfitting, indicating that further regularization or increased training data might be beneficial.

### Part 3: Transfer Learning (ResNet50)

The ResNet50 model, utilizing pretrained weights from ImageNet, achieved the highest performance among the three models. Validation accuracy reached approximately 67%, with an OOD leaderboard score of 0.48360. The use of transfer learning allowed the model to leverage previously learned features, accelerating convergence and improving generalization. Techniques such as dropout, label smoothing, and cosine annealing were instrumental in mitigating overfitting and ensuring stable training. Despite its success, the model's complexity and computational demands are higher. Future work could explore optimizing the balance between model complexity and performance, probably by experimenting with architectures like ResNet34 or efficient training strategies to reduce resource consumption without compromising accuracy.

## Ablation Study

To understand the contribution of individual components in each model, I conducted ablation studies for Parts 1 through 3 by observing general trends. While I aimed to monitor the impact of each change closely, I did not record the exact metrics or percentages associated with every hyperparameter adjustment. Therefore, the ablation studies focus on general performance trends and qualitative observations rather than precise numerical outcomes.

In Part 1, the SimpleCNN model was sensitive even to small architectural and training changes. Removing one convolutional layer or dropout significantly hurt performance and led to overfitting. Replacing the AdamW optimizer with SGD or removing cosine annealing caused poor convergence. Disabling data augmentation led to training/validation mismatch, confirming its importance. Though performance was limited by model capacity, regularization and scheduling helped improve stability.

For Part 2, the ResNet18-based model showed stronger generalization, but was still highly dependent on training strategy. Removing dropout or simplifying augmentations (e.g., no color jitter or rotation) resulted in earlier overfitting and lower validation performance. Using a fixed learning rate instead of cosine annealing hindered late-stage convergence. Training for fewer epochs or using larger batch sizes also hurt performance. Overall, the model benefited most from aggressive augmentation, dropout, and dynamic learning rates.

In Part 3, the pretrained ResNet50 model's performance was strongly tied to transfer learning. Removing pretrained weights resulted in a large performance drop. Label smoothing and dropout helped reduce overfitting and improve stability, while removing either degraded generalization. Fixed learning rates led to early stagnation, whereas cosine annealing supported smoother convergence. Strong augmentations were again important.

## Conclusion

This midterm challenge offered an opportunity to progressively build, test, and improve CNN-based models using both architectural innovation and strong engineering practices. Beginning with a minimal CNN, I demonstrated the gains achievable through deeper networks and transfer learning. Each part of the project highlighted specific trade-offs between capacity, generalization, and computational cost. My final model exceeded the leaderboard benchmark with strong validation accuracy and effective OOD performance. The hands-on practice with hyperparameter tuning, regularization, and WandB tracking will carry over to future deep learning projects. Potential future improvements could involve using regularization techniques like MixUp, CutMix, or advanced augmentation pipelines such as RandAugment to push performance even further.