DS 542 Midterm Challenge Report

Part 1 - Simple CNN (part1.py)

Goal: Define a relatively simple CNN model and train it on the CIFAR-100 dataset to get a complete pipeline and establish baseline performance.

Model Description:

For our first approach, we built a straightforward CNN to establish a baseline on CIFAR-100. The network follows a pretty basic design pattern: three convolutional blocks that gradually increase in depth while reducing spatial dimensions. We started with a wider 5x5 kernel in the first layer to capture larger image features, then switched to standard 3x3 kernels as the network deepened. Each block follows a combination of convolution, batch normalization, ReLU activation, and max pooling. After the convolutional layers extract features, we flatten the output and pass it through two fully connected layers. We added dropout between these layers with a 0.4 probability to help prevent overfitting. This seems to be a pretty a common issue when training on a dataset like CIFAR-100, and is something I had to deal with heavily in part 3 later on. For training, we stuck to basic methods, using an Adam optimizer with a learning rate of 0.001 and a bit of weight decay (1e-4) to keep the weights in check. We also implemented a learning rate scheduler that reduces the rate when validation accuracy plateaus, helping us fine-tune the model's performance. The training data had some basic augmentation by adding random crops and flips to help the model generalize better. The model scored a validation accuracy of 36.56%, and on Kaggle we achieved a score of 25.206%

Part 2 - More Sophisticated CNN Models (part2.py)

Goal: Use a more sophisticated model, including predefined models from torchvision to train and evaluate on CIFAR-100.

Model Description:

Having established a baseline, we moved to the more sophisticated ResNet18. While we used the model structure from torchvision, we trained it from scratch instead of using pretrained weights. I chose ResNet18 because it introduces residual connections to help with the vanishing gradient problem, making it easier to train deeper networks effectively.Since ResNet18 was originally designed for ImageNet's larger images, we had to make some adjustments for CIFAR-100's 32x32 images. The main changes were in the first layer - we used a smaller 3x3 kernel instead of the original 7x7, and we removed the initial max pooling layer that would have been too aggressive for our smaller images. These modifications helped preserve important features in the early stages of the network, which is important when working with smaller images.

The training approach here differed significantly from part 1. ResNets typically train better with SGD and momentum, so we switched from Adam to SGD with a momentum of 0.9. We also found that a higher learning rate of 0.01 worked well with this setup since ResNets can handle larger learning rates due to skip connections. To help with regularization, we increased up the weight decay to 5e-4, which proved important for preventing overfitting on our relatively small dataset. For the learning rate schedule, we implemented a cosine annealing scheduler that smoothly decreases the learning rate over our 15 training epochs. This approach seemed to work better than step-based decay, mainly because it provided a more gradual exploration of the loss landscape. We kept our batch size at 128, mainly because I wanted to keep the training time short for this model.For dataset handling we split our training data 80/20 into training and validation sets using a random split with a fixed seed for reproducibility. For the validation set, we made sure to use the test transforms (without augmentation) to get a clean evaluation of the model's performance. The data augmentation strategy remained simple, using random crops with padding and horizontal flips.

This more advanced model reached a validation accuracy of 46.67% and scored 34.323% on Kaggle. The notable improvement over our simple CNN baseline showed how important it is to have a well-designed architecture, even when trained from scratch on a relatively small dataset.

Part 3 - Transfer Learning from a Pretrained Model (part3.py)

Goal: Pretrain a model, or use one of the pretrained models from torchvision, and fine-tune it on CIFAR-100. Try to beat the best benchmark performance on the leaderboard.

Model Description:

Building on our experience from part 2, we decided to stick with ResNet18 for our transfer learning approach. While we could have chosen a larger architecture like ResNet50 or ResNet101, ResNet18's lighter architecture made more sense for CIFAR-100 the dataset isn't as complex as ImageNet, and we were worried about a larger model being more prone to overfitting. Plus, our results from part 2 showed that ResNet18's architecture was capable of learning good features for CIFAR-100.

Both versions of our model used a two-phase training strategy, but we made important refinements in the final version based on our observations of overfitting in the initial. In both versions, we started with:

- Phase 1: Freeze the pretrained backbone and train only the classifier
- Phase 2: Carefully fine-tune the entire network

However, in the initial version, we noticed some issues. While using 5 epochs for Phase 1 worked well, our 35 epochs in Phase 2 proved to be too many, and the model started to overfit significantly around epoch 20. The training accuracy would climb close to 100%, but validation accuracy would plateau or even decrease. This suggested we were giving the model too much

time to adapt to our training data, causing it to lose the beneficial features it had learned from ImageNet.

This led to several key improvements in the final version

1. Refined Two-Phase Strategy:
    - Phase 1: Kept at 5 epochs, as this worked well for initial classifier adaptation
    - Phase 2: Reduced from 35 to 20 epochs to minimize overfitting

2. Batch Size and Learning Rate Dynamics:
    - Reduced batch size from 128 to 32 to provide more frequent updates and better generalization.
        - The reduction in batch size from 128 to 32 proved to be our most effective change against overfitting, as smaller batches introduce more noise in gradient updates, naturally providing regularization. This noise prevents the model from finding overly perfect solutions for the training data, encouraging more robust feature learning.
    - Implemented different optimizers for each phase:
        - Adam for Phase 1 due to its adaptive nature when training just the classifier
        - SGD with momentum for Phase 2, as it typically works better for fine-tuning CNNs and provides more stable updates

3. Enhanced Regularization:
    - Added Dropout (p=0.5) before the final classifier
    - Reduced weight decay to 1e-4 (from 5e-4 in Part 2) as we found the higher value was too aggressive for transfer learning
    - Added label smoothing (0.1) to prevent the model from becoming overconfident in its predictions

4. Augmentation Strategy:
   We significantly expanded our data augmentation pipeline:

```
transforms.RandomCrop(32, padding=4),
transforms.RandomHorizontalFlip(),
transforms.RandomRotation(15),
transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
transforms.RandomErasing(p=0.5)
```

Each of these augmentations helps prevent overfitting in different ways. RandomRotation and ColorJitter help with orientation and color invariance, while RandomErasing helps the model focus on multiple discriminative parts of the images.

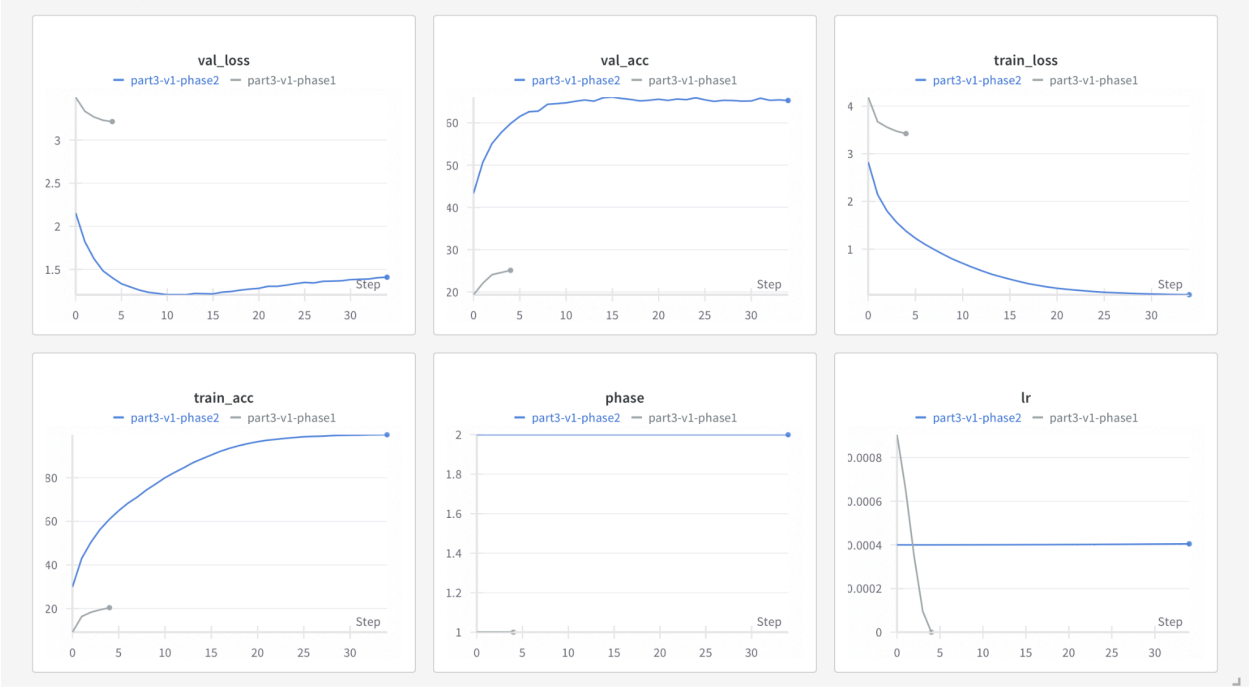
Interestingly, we experimented with Mixup augmentation, which theoretically should help with overfitting by creating virtual training examples. However, we found it actually hurt our

performance. We also made sure to use the correct normalization values for our pretrained model - switching from CIFAR-100's mean/std to ImageNet's (mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)). This small but crucial detail ensures the input distribution matches what the pretrained model expects.

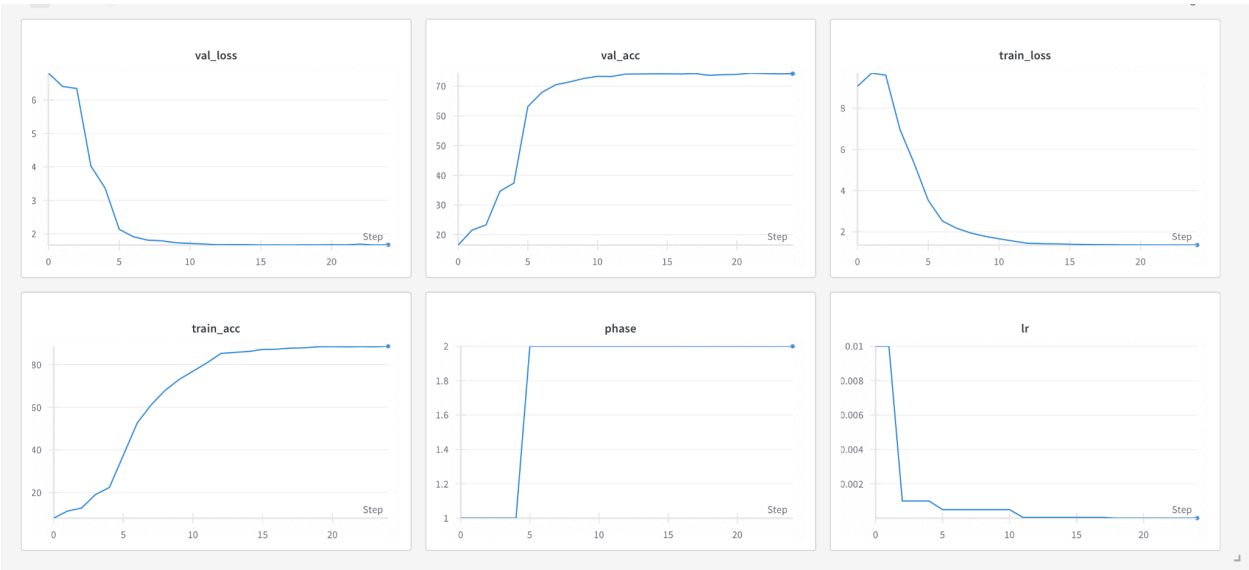
Our final model achieved a validation accuracy of 74.25% and a Kaggle score of 52.964%, successfully beating the benchmark score of 39.797 %

Experiment Tracking Summary (WandB)

We used Weights & Biases (WandB) to monitor training and compare model versions, which was crucial for refining our Part 3 model.



WandB charts for Part 3 v1, showing overfitting via diverging train/val accuracy.



WandB charts for the final Part 3 model, showing reduced overfitting and higher peak validation accuracy.

Comparing the WandB charts clearly shows the progress. In our initial attempt, the validation accuracy plateaued around 64% while training accuracy kept rising, indicating significant overfitting. Our final model achieved a higher peak validation accuracy and demonstrated much better control over overfitting, validating our adjustments like reduced batch size, shorter fine-tuning, and enhanced regularization. WandB was really useful for visualizing these improvements and confirming the effectiveness of our changes.

AI Statement

The foundational code structure was based on the provided starter_code.py. Throughout development, I used AI tools (chat gpt and claude) for idea validation, code implementation, and generating suggestions.

For Part 1 (part1.py), I used AI to validate my CNN architecture choices, particularly the progression of kernel sizes and channel depths. AI helped implement the PyTorch code for the model and training loop, and suggested adding dropout and learning rate scheduling, which I hadn't initially considered.

In Part 2 (part2.py), AI helped validate my approach to modifying ResNet18 for 32x32 images and assisted in implementing these modifications. The suggestions to use cosine learning rate scheduling and increased weight decay came from AI and improved training stability.

For Part 3 (part3.py), AI helped validate and implement the two-phase training approach. When version 1 showed overfitting, AI suggested the key improvements implemented in version 2:

reduced batch size and more aggressive data augmentation. These changes significantly improved the model's performance.

For all 3 parts, AI was used in actually writing/implementing the code.