

# DS542 Midterm Report

Yuchen Li

Spring 2025

## 1 Part 1 – SimpleCNN

For the baseline, I manually defined a SimpleCNN model with:

- Three convolutional blocks with **MaxPooling** after each block
- No data augmentation
- Adam optimizer with **StepLR** scheduler
- Learning rate set to 0.1
- Batch size of 8

The result was suboptimal. Within 5 epochs, training accuracy was only 0.995, and validation accuracy was just 1.08%.

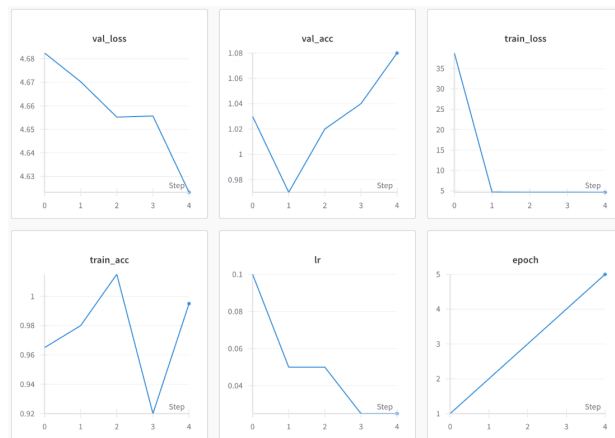


Figure 1: Evaluation Metrics for SimpleCNN (Before Tunning)

To improve model performance, I made several key changes:

- Built a deeper network, added Batch Normalization for faster convergence and generalization, Dropout layers to prevent co-adaptation, and used Global Average Pooling instead of flattening
- Applied data augmentation to introduce randomness (`RandomHorizontalFlip()` and `RandomCrop(32, padding=4)`)

- Incorporated L2 regularization via `weight_decay` in the `Adam` optimizer. Increased batch size to 64 and lowered the learning rate.

Under these modifications, the `SimpleCNN` reached 36.02% training accuracy and 38.83% validation accuracy within 5 epochs.

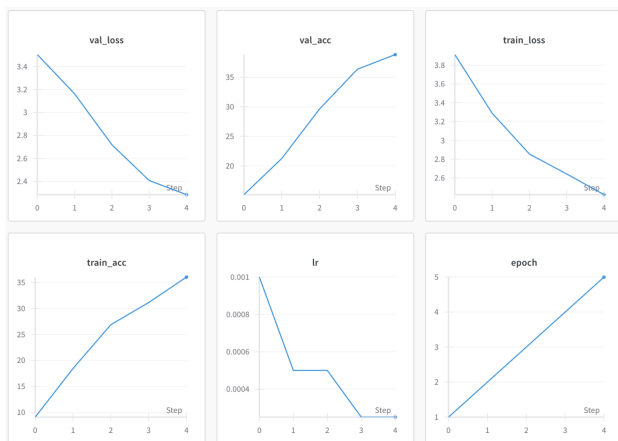


Figure 2: Evaluation Metrics for SimpleCNN (After Tuning)

## 2 Part 2 – ResNet18

In the second part, I transitioned to a more sophisticated architecture: `ResNet18` from `torchvision.models`, modifying the final fully connected layer to output 100 classes. This choice was motivated by its lightweight nature and residual connections that aid gradient flow.

Without further changes, switching from `SimpleCNN` to `ResNet18` resulted in training accuracy of 37.55% and validation accuracy of 36.26%.

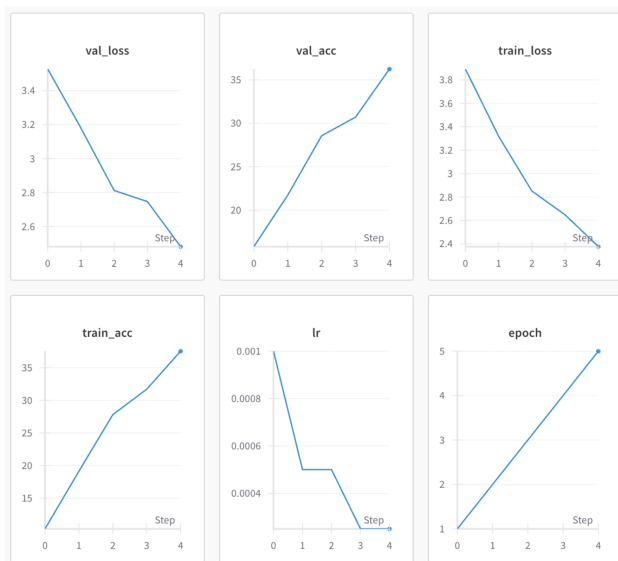


Figure 3: Evaluation Metrics for Part 2 (Without Further Tuning)

## Improvements Attempted

### First Try:

1. Replaced Adam optimizer with SGD + Momentum for better generalization and gradient smoothing.
2. Used CosineAnnealingLR instead of StepLR for smoother LR decay.

```
1 optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9,
2   weight_decay=1e-4)
3 scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=
4   CONFIG["epochs"])
```

3. Enhanced data augmentation with ColorJitter and RandomRotation.
4. Applied label smoothing in the loss function.

```
1 criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
```

However, these changes did not yield better results within 5 epochs. The model achieved only 14.20% training accuracy and 20.87% validation accuracy.

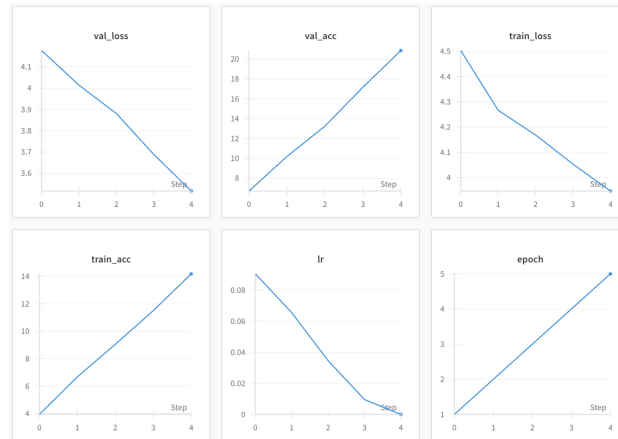


Figure 4: Evaluation Metrics for Part 2 (1st Try)

### Second Try:

Introduced **Mixup** for advanced data augmentation. Since ResNet18 is deep and prone to overfitting, Mixup helped reduce overfitting by smoothing decision boundaries and improving generalization. The technique was implemented as:

```
1 def mixup_data(x, y, alpha=1.0):
2     if alpha > 0:
3         lam = np.random.beta(alpha, alpha)
4     else:
5         lam = 1.0
6     batch_size = x.size(0)
7     index = torch.randperm(batch_size).to(x.device)
```

```

8      mixed_x = lam * x + (1 - lam) * x[index, :]
9      y_a, y_b = y, y[index]
10     return mixed_x, y_a, y_b, lam
11

```

This yielded:

*Train accuracy: 25.91%, Validation accuracy: 25.06%.*

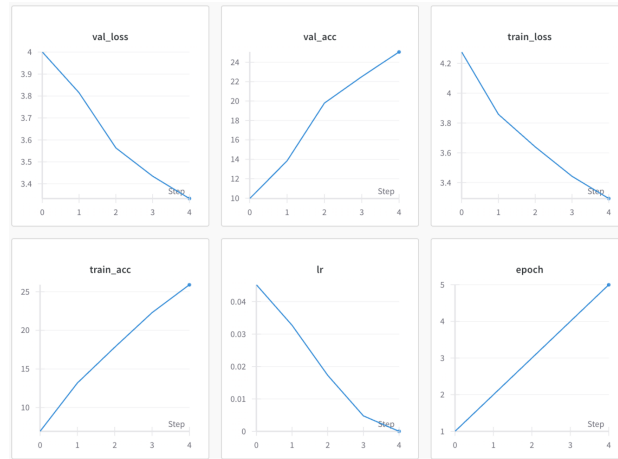


Figure 5: Evaluation Metrics for Part 2 (2nd Try)

### Third Try:

Replaced `CosineAnnealingLR` with `OneCycleLR`, which is often effective with SGD, for it helps models converge faster and generalize better.

```

1      scheduler = optim.lr_scheduler.OneCycleLR(
2          optimizer,
3          max_lr=CONFIG["learning_rate"],
4          steps_per_epoch=len(trainloader),
5          epochs=CONFIG["epochs"],
6          pct_start=0.3,
7          anneal_strategy='cos',
8          div_factor=25.0,
9          final_div_factor=1e4
10     )

```

However, the result was worse than before, achieving only 20.73% training accuracy and 21.32% validation accuracy.

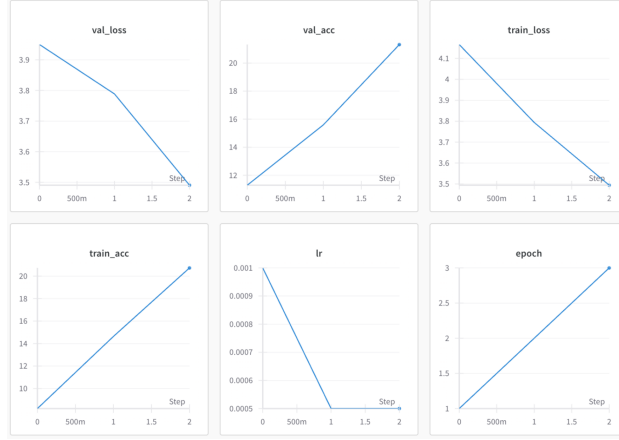


Figure 6: Evaluation Metrics for Part 2 (3rd Try)

### Final Decision:

- Reverted to Adam optimizer with StepLR
- Removed Mixup

This configuration achieved the best result in 5 epochs:

*Train accuracy: 40.21%, Validation accuracy: 41.3%*

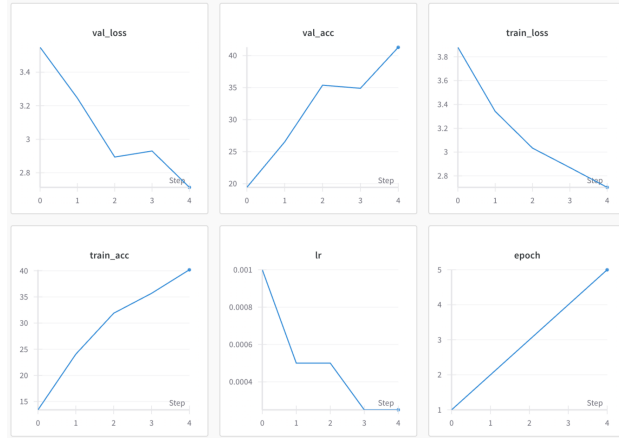


Figure 7: Evaluation Metrics for Part 2 (Final Version)

## 3 Part 3 – MobileNetV2

For the final part, I leveraged transfer learning by using MobileNetV2 with `pretrained=True`, resizing CIFAR-100 images to  $224 \times 224$  and increasing batch size to 128.

Reasons for choosing MobileNetV2:

- Efficient, fast, and memory-friendly
- Uses depthwise separable convolutions to reduce computational cost

- Well-suited for training on SCC with limited GPU resources

With 50 epochs, the final result was:

*Train accuracy: 99.95%, Validation accuracy: 78.25%*

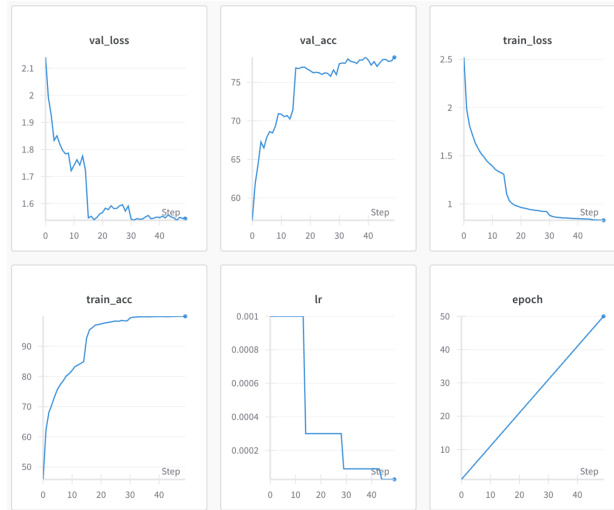


Figure 8: Evaluation Metrics for MobileNetV2

## Challenges and Debugging

Initially, the predictions in the OOD submission CSV were all the same. After debugging, I discovered the problem stemmed from how inputs were unpacked in the TensorDataset and normalized in batch. With the original code:

```
1 with torch.no_grad():
2     for inputs in tqdm(dataloader, desc=f"Evaluating {distortion_name}
3         (Severity {severity})", leave=False):
4         inputs = inputs[0]
5         inputs = normalize(inputs)
6         inputs = inputs.to(device)
```

Since each inputs from TensorDataset is a tuple, if it repeats the process in a loop, it might return the same tensor reference repeatedly. Also, the code normalizes the whole batch tensor without looping. So I fix that into:

```
1 normalize = transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675,
2     0.2565, 0.2761))
3 resize = transforms.Resize((224, 224))
4 images = torch.stack([normalize(resize(img)) for img in images])
5 ...
6 with torch.no_grad():
7     for batch in tqdm(dataloader, desc=f"Evaluating {distortion_name}
8         (Severity {severity})", leave=False):
9         inputs = batch[0].to(device)
10        outputs = model(inputs)
```

```
9         _, predicted = outputs.max(1)
10        predictions.extend(predicted.cpu().numpy())
```

Once corrected, the submission results became reasonable.

## 4 Future Improvements

If I had more time, I would explore:

- EfficientNet B0 with transfer learning
- CutMix or Stochastic Depth
- Learning rate finder for auto-tuning
- Extended training schedules (100+ epochs) with checkpointing
- Early stopping and ensemble modeling

## AI Disclosure

For this assignment, I used AI assistance from ChatGPT to help guide my research and decision-making throughout the modeling process. Specifically, ChatGPT was instrumental in providing insights on advanced model tuning strategies, including which pretrained models from `torchvision.models` are suitable for CIFAR-100, and alternative optimizers and learning rate schedulers beyond the standard Adam and `StepLR` combination (e.g., SGD with momentum and `CosineAnnealingLR`). ChatGPT also recommended trying the Mixup technique for advanced data augmentation and provided initial code snippets for its implementation. While Mixup was explored during experimentation, it was not included in the final version of the model. All architectural decisions, model configurations, and final code integration were done by me, with AI serving as a supplementary guide for experimentation and exploration.