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`$_d$*ecayintheAdamoptimizerIncreasedbatchsizeto*64*andloweredthelearnin*

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

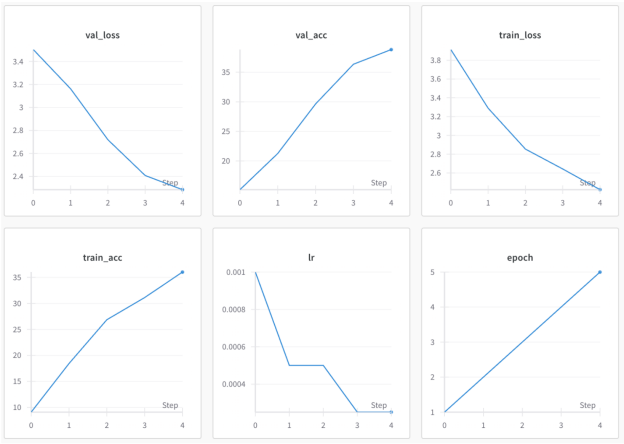

Figure 2: Evaluation Metrics for SimpleCNN (After Tunning)

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%.

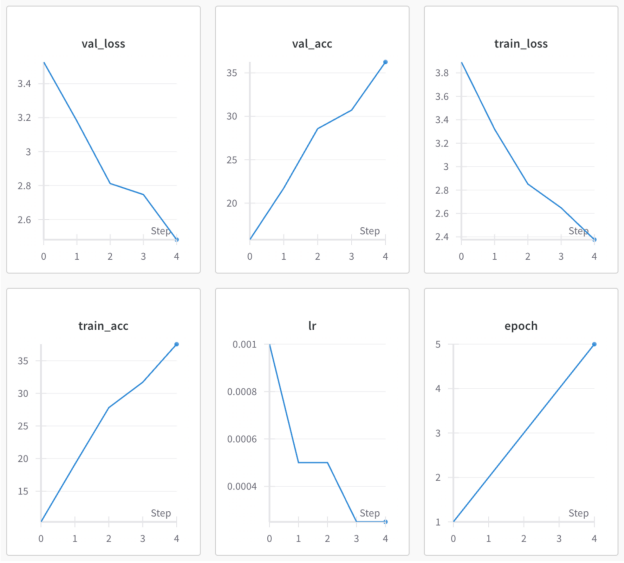

Figure 3: Evaluation Metrics for Part 2 (Without Further Tuning)

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.

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

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

```
criterion = nn.CrossEntropyLoss(label_smooting=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.

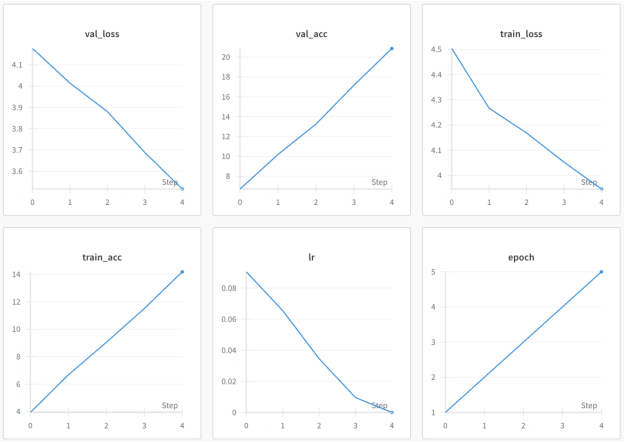

Figure 4: Evaluation Metrics for Part 2 (1st Try)

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:

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

```
        mixed_x = lam * x + (1 - lam) * x[index, :]
        y_a, y_b = y, y[index]
        return mixed_x, y_a, y_b, lam
```

This yielded:
Train accuracy: 25.91%, Validation accuracy: 25.06%.

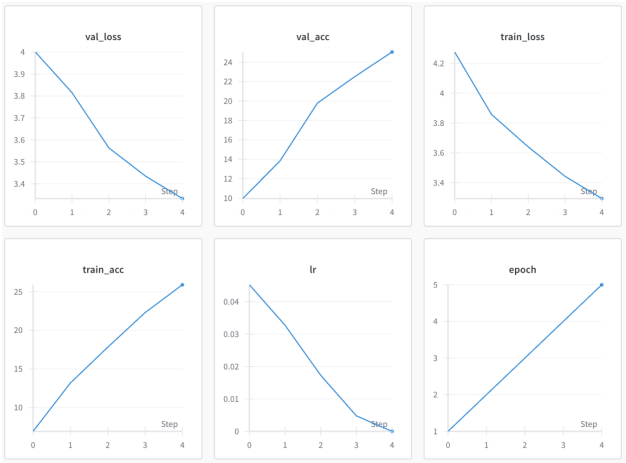

Figure 5: Evaluation Metrics for Part 2 (2nd Try)

Third Try:

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

```
    scheduler = optim.lr_scheduler.OneCycleLR(
        optimizer,
        max_lr=CONFIG["learning_rate"],
        steps_per_epoch=len(trainloader),
        epochs=CONFIG["epochs"],
        pct_start=0.3,
        anneal_strategy='cos',
        div_factor=25.0,
        final_div_factor=1e4
    )
```

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

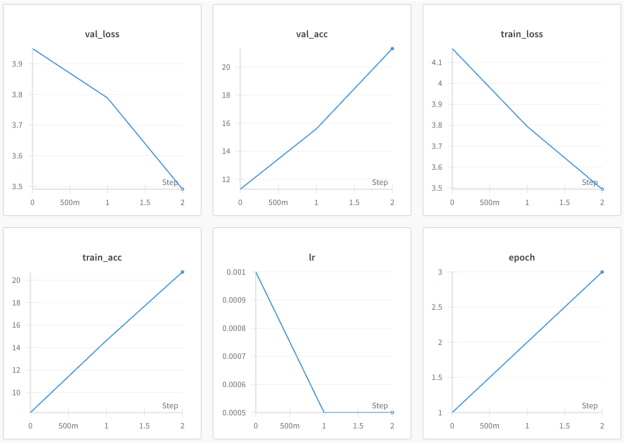

Figure 6: Evaluation Metrics for Part 2 (3rd Try)

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%

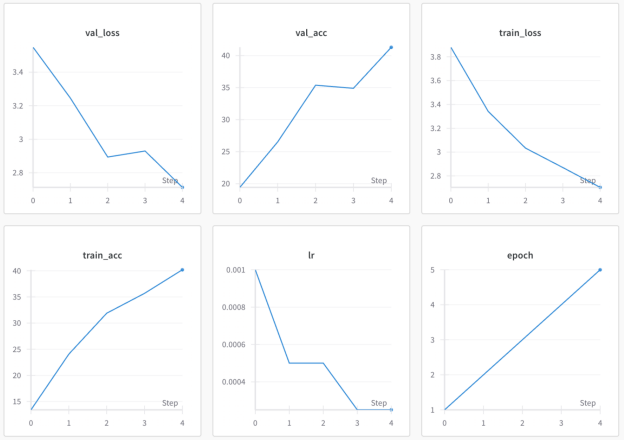

Figure 7: Evaluation Metrics for Part 2 (Final Version)

3 Part 3 – MobileNetV2

For the final part, I leveraged transfer learning by using `MobileNetV2` with `pretrained=True`, resizing CIFAR-100 images to 224×224 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%

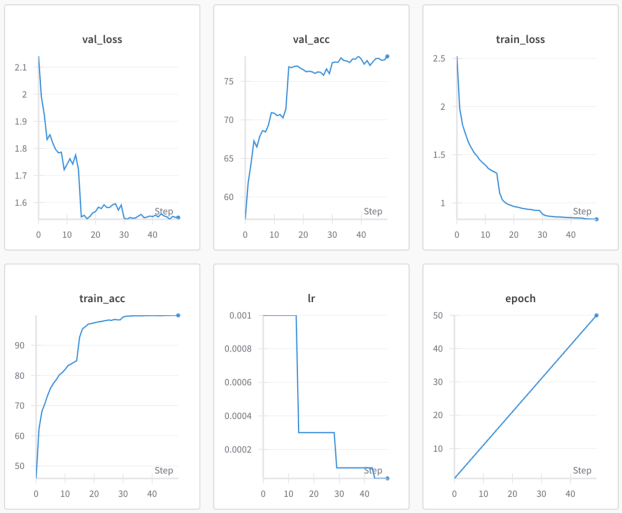

Figure 8: Evaluation Metrics for MobileNetV2

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:

```
    with torch.no_grad():
        for inputs in tqdm(dataloader, desc=f"Evaluating {distortion_name}
    (Severity {severity})", leave=False):
            inputs = inputs[0]
            inputs = normalize(inputs)
            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:

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

```
            _, predicted = outputs.max(1)
            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