Midterm Report

Michael Krah

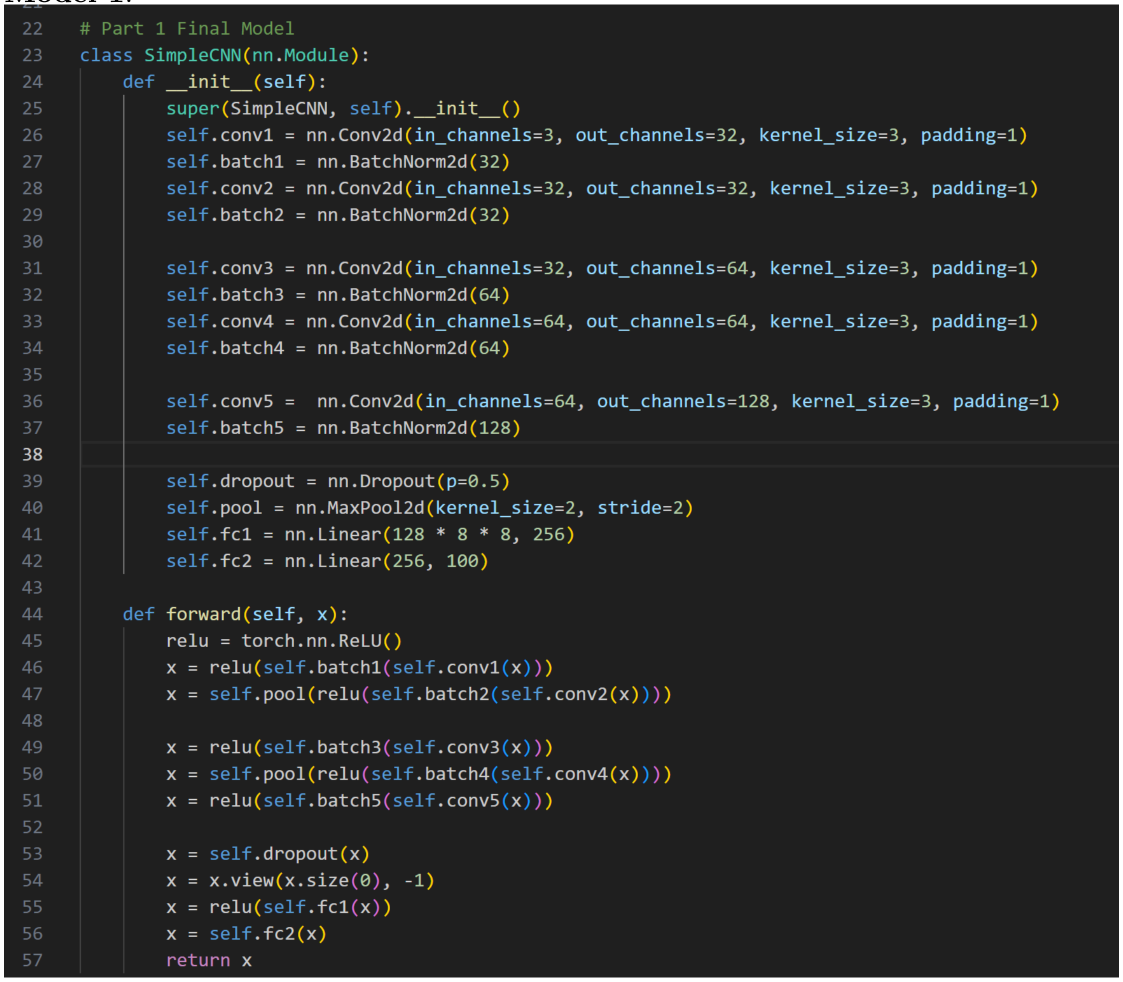
March 30, 2025

AI Disclosure

I used generative AI to suggest transformations I could implement in the training pipeline to address overfitting to the base CIFAR-100 database. While training model 3 I had issues with getting a relatively high accuracy on the training and testing set given, but a lower accuracy on the distorted set on Kaggle. I asked ChatGPT o3-mini for suggestions to address overfitting in this context, and implemented transformations is provided. A complete appendix is listed at the bottom.

Model Descriptions

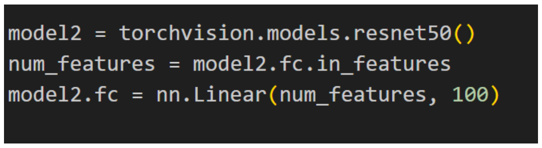
Model 1:



```python
# Part 1 Final Model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.batch1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1)
        self.batch2 = nn.BatchNorm2d(32)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.batch3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.batch4 = nn.BatchNorm2d(64)

        self.conv5 =  nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.batch5 = nn.BatchNorm2d(128)

        self.dropout = nn.Dropout(p=0.5)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(128 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 100)

    def forward(self, x):
        relu = torch.nn.ReLU()
        x = relu(self.batch1(self.conv1(x)))
        x = self.pool(relu(self.batch2(self.conv2(x))))

        x = relu(self.batch3(self.conv3(x)))
        x = self.pool(relu(self.batch4(self.conv4(x))))
        x = relu(self.batch5(self.conv5(x)))

        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

For model 1 I iteratively developed a relatively simple convolutional neural network. My final submission had a test accuracy on the Kaggle dataset of 0.31. The model includes 5 convolutional layers, 5 batch normalization layers, 2 max pooling layers, 1 dropout layer, and 2 linear layers. Convolutional layers are repeated prior to pooling to maximize the probability that necessary features are learned. Although I experimented with different numbers of max pooling layers, I found that more than two layers reduced the dimensionality too significantly for the model to recognize important features. A dropout layer was added prior to linear layers as I began to run into issues of overfitting, especially as the number of epochs increased. I found that two linear layers worked the best to limit overfitting. I chose to use Relu as that seems to be the most popular activation function for CNNs. These decisions and a lot of my architecture choices were inspired by CNNs we saw in class such as Alexnet.

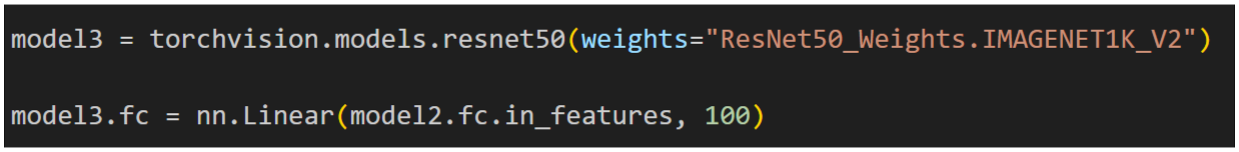
Model 2:



```
model2 = torchvision.models.resnet50()
num_features = model2.fc.in_features
model2.fc = nn.Linear(num_features, 100)
```

For model 2 I used the predefined resnet50 model from pytorch, modified to output a classification for the 100 classes in CIFAR-100. I spent the least amount of time working on this model, as I used a similar architecture for model 3 with pretrained weights. As a result, I used the same hyperparameters for the second and third model, and obtained a final accuracy on the Kaggle dataset of about 0.25. This was worse than my best result for model 1, which makes sense as I spent the most time iterating on the architecture of model 1 and the hyperparameters for model 3.

Model 3:



```
model3 = torchvision.models.resnet50(weights="ResNet50_Weights.IMAGENET1K_V2")

model3.fc = nn.Linear(model2.fc.in_features, 100)
```

For model 3 I used a predefined resnet50 architecture that was pretrained on imagenet. This was downloadable using pytorch. The model was modified to output on 100 classes for CIFAR-100 classification. In my final model, I left all layers unfrozen, though I experimented with unfreezing on the last last, and groupings of last final layers. Unfreezing the last four layers gave similar performance to unfreezing all layers, however, when unfreezing all layers my final accuracy on the Kaggle dataset was 0.421, which was about 0.005 higher. I used a resnet architecture trained on Imagenet as Imagenet has many classes that are similar to CIFAR.

Hyperparameter Tuning

General Strategy:

I did all hyperparameter tuning on the SCC, starting with hyperparameter values provided in the starter code or given as examples in class or in the textbook. I then trained a model, changed single hyperparameters, and compared results between runs on wandb to determine what changes had a positive affect on validation accuracy.

Batch size:
This value was initially set to 8. A batch size this low slowed training down considerably. Although this was fine when using lower epoch numbers, as I increased the number of epochs I found that a batch size of 128 worked best. I experimented with batch sizes of 32, 64, 128, and 512. However, I found that batch sizes below 64 did not generalize well and a batch size as large as 512, although considerably faster, had a significantly lower accuracy.

Learning rate:
This value was initially set to 0.1. As I experimented with more epochs, I lowered this to 0.001 to ensure a model did not converge too quickly. This worked especially well with model 3, which is likely due to the fact that its pretrained weights provided a good point of initialization.

Number of epochs:
This was initially set to 5. Although this number allows simple networks to converge with a high enough learning rate, I quickly found that raising this to or 20 improve validation accuracy significantly. I primarily used wandb to optimize the number of epochs, as the visualizations provided made it easy to understand when models stopped converging and began to overfit to the test data. I found that 60 epochs worked best for model 1, using a StepLR optimizer that multiplied the learning rate by 0.1 every 20 epochs. For models 2 and 3 I found that about 40 epochs worked best with a StepLR optimizer that changed the learning rate by a factor of 0.1 after 20 epochs.

Loss function:
I used cross-entropy loss for my loss function. This was implemented using pytorch, and is the loss function used in the textbook for convolutional neural networks.

Optimizer: I initially used stochastic gradient descent as my optimizer, as we had the most experience with this from the textbook and class. This worked well for my initial smaller CNNs, but I found that simply changing the optimizer to the Adam optimizer had a significant impact on final validation accuracy, initialization, and time until convergence.

While working on model 3 I also included as a parameter to limit overfitting, finding that a value of 0.01 did not has a negative impact on validation accuracy.

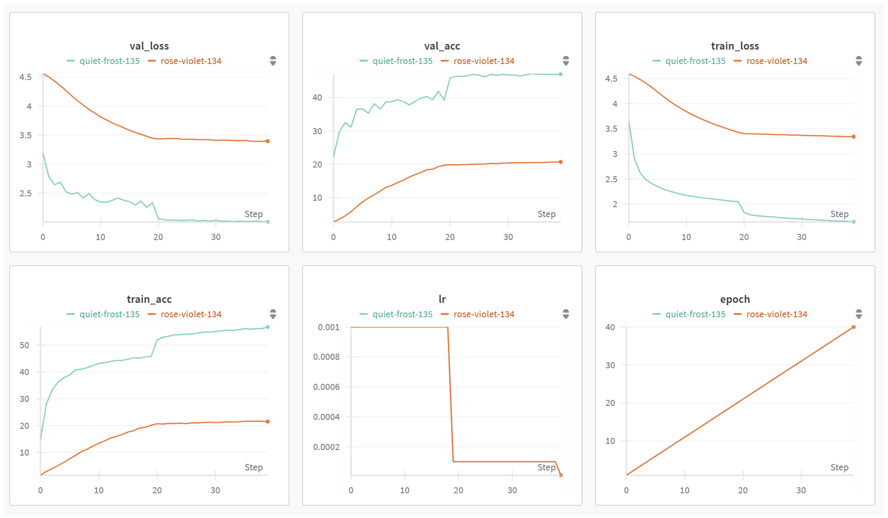


Figure 1: Wandb comparison of SGD (orange) and Adam (blue) optimizers on an early version of model 1. Using the Adam optimizer instantly improved validation accuracy by a significant margin, more than doubling it in some cases.

Scheduler:
I implemented a scheduler as I began to increase the number of epochs to 20. I found that a StepLR scheduler provided the best results and provided a significant increase in validation accuracy. It also helped to mitigate overfitting. I also experimented with using a cosine annealing scheduler, however, I was never able to obtain similar validation accuracy.

Regularization Techniques

Dropout:
I implemented a dropout layer with a probability value of 0.5 on model1. Implementing this layer significantly improved overfitting on the training data set, especially when running the model with 60 epochs.

L2 Regularization:
I implemented weight decay while using the Adam optimizer, a form of L2 regularization. Although this did not lead to a significant increase in validation accuracy, I chose to keep it as it did not hurt accuracy and likely makes the model more robust and adaptable to new results.

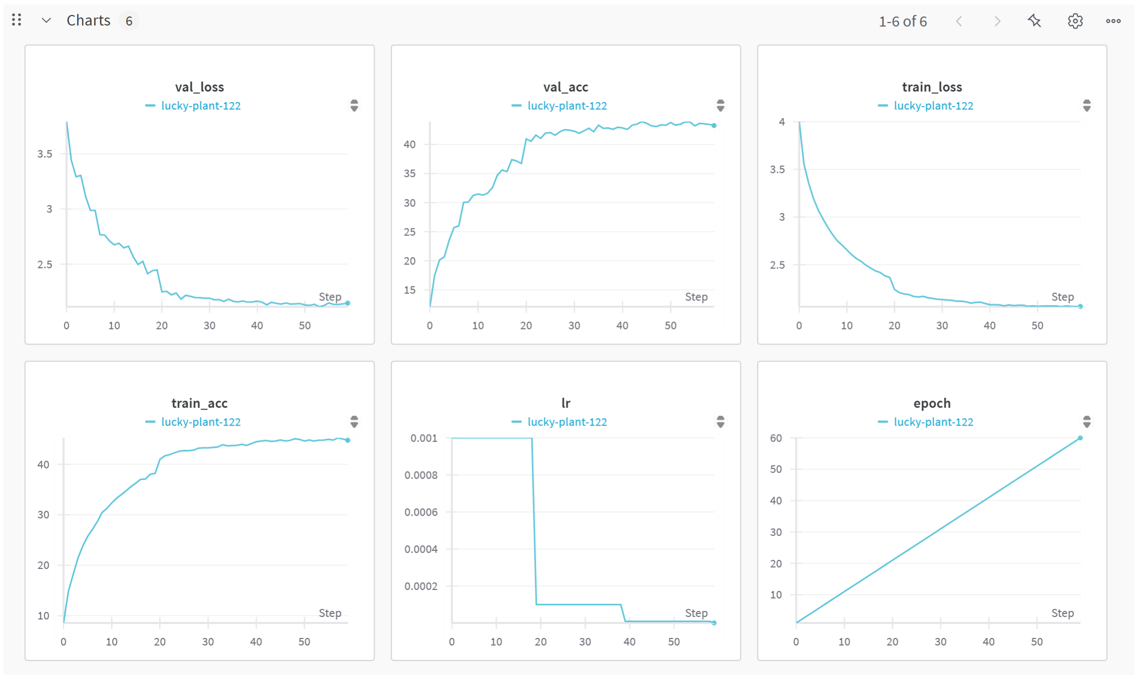
Data augmentation for regularization:
I implemented random erasing on tensors while working on model 3. This will delete a random rectangle of data with a probability of 0.5. I found that this implementation helped with generalization especially when performing inference on the distorted Kaggle dataset.

Data Augmentation Strategy

I initially implemented random rotation and random horizontal flipping when initially developing the code for the training loop. I was already familiar with these augmentations and assumed that they would help improve generalization without any significant drawbacks. While working on model 3, I implemented color jittering and random affines. I had issues with the model overfitting to the data and still obtaining a relatively low accuracy on the training dataset. Although implementing these did not have a significant impact on validation accuracy, they improved accuracy on the distorted data set by about 0.05. I added these incrementally to ensure they did not have any negative impact on accuracy. I also experimented with adding gaussian noise to the training data, as I assumed that this would reflect results in the distorted dataset. However, I found that this made training unstable and led to jumps in validation loss.

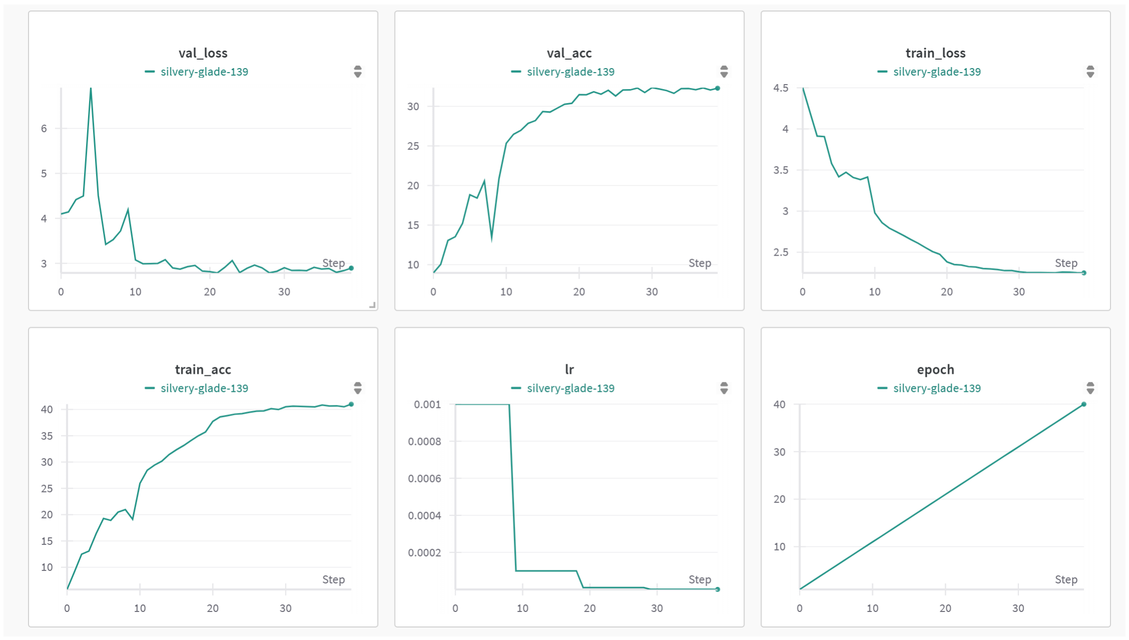
Results Analysis

Model 1 Results:



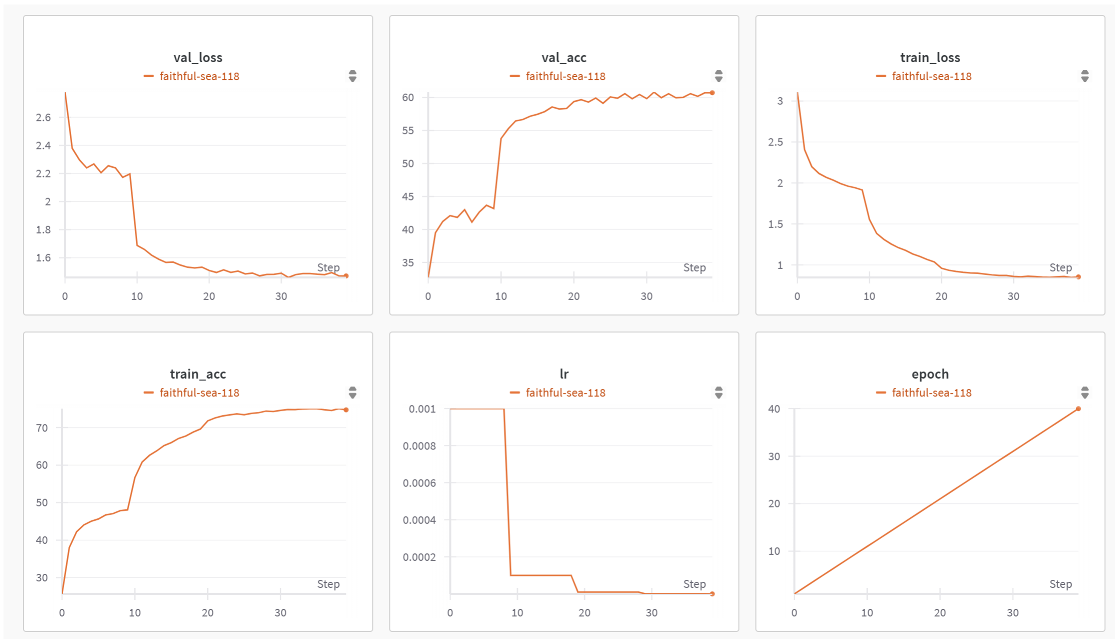
This model had an accuracy on the Kaggle dataset of 0.31. I had models with higher validation accuracies, however I found that no other models generalized better to the distorted dataset. Some potential improvements could be increasing the number of layers in the model, as this might help the model identify more features. I tried experimenting with more layers, however, I found it difficult to place more than two convolutional layers together without a reduction in performance. Furthermore, more regularization techniques may help the model better generalize to the distorted dataset, as there is still a 15 percentage point decrease in accuracy.

Model 2 Results:



This model had an accuracy on the Kaggle dataset of 0.25. This model would have likely benefited significantly from further hyperparameter tuning, as I used the same hyperparameters as I did for model 3 when running this model. My reasoning was that a model with pretrained weights would likely have a higher performance regardless of what I did, and spent more effort on model 3 than on model 2. Examining the validation loss graph, it's clear that the results are volatile. More regularization techniques could have helped ensure a stable convergence. However, I still found that it obtained a relatively high accuracy without any significant effort, which was promising to see.

Model 3 Results:



This model had an accuracy of 0.42150 on Kaggle. This model benefited most from hyper-parameter training, and pretraining seemed to provide a significant advantage on the final distorted test set. Likely this model would benefit from more finetuned training, unlocking only certain layers or having different learning rates for different layers may help improve performance. However, I tried to unfreeze only the last layer, the last 4 layers, and the last 10 layers and I saw no improvement in validation accuracy or accuracy on the distorted dataset. More data augmentations may help

Experiment Tracking Summary

In total, I tracked 140 training runs using wandb. Below are the results from all three models compared. 5 runs crashed, 20 runs failed, 26 were killed, and 89 finished running.

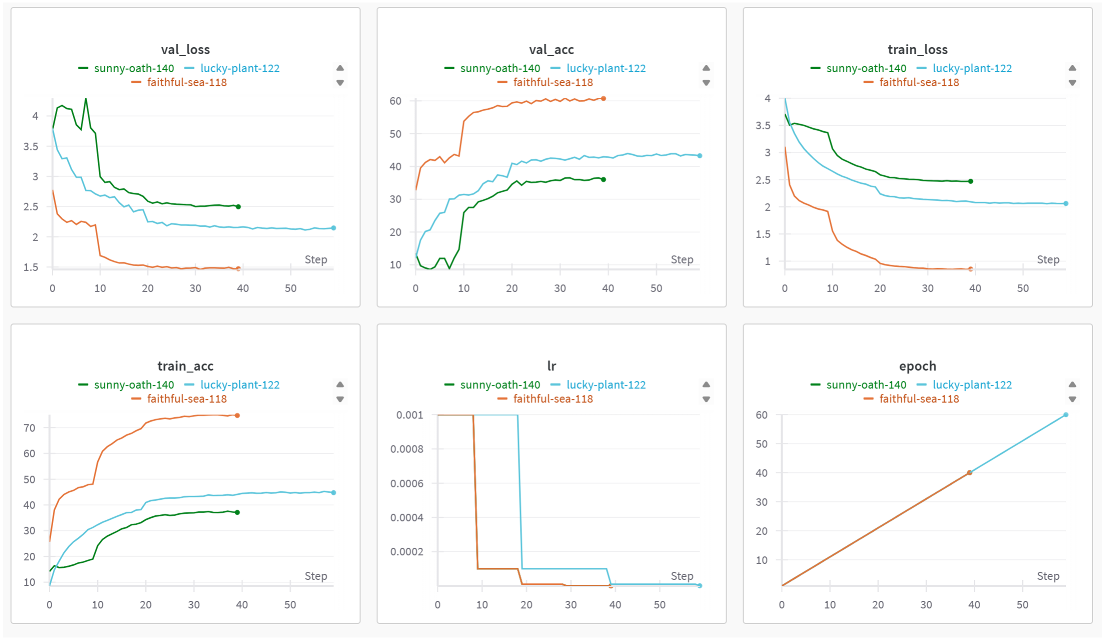


Figure 2: Model 1 is in blue, model 2 in green, and model 3 in orange

AI Appendix

Results from the conversation with ChatGPT are linked below. ChatGPT 03-mini was used on an account with ChatGPT plus. The chat is relatively short, and asks about techniques to adress the distorted dataset as well as a few clarifying questions about pytorch code.

https://chatgpt.com/share/67e9b89e-8d14-8004-8bc5-c308cc587820