Deep Neural Networks Assignment 2
Duke University ECE 661
Fall 2023
Genesis Qu

Optimizing Convolutional Neural Networks

Part 1: True and False

1.1 True. Batch Normalization scales the output by the mean and variance.

1.2 True. Yes, you can use PyTorch's pre-defined optimizers to do back-propagation.

1.3 False. Not every data augmentation technique will be applicable to our image. Vertical flip, for example, will not be applicable to the CIFAR-10 dataset because a digit such as 6 will be recognized as 9 by the network.

1.4 False. Dropout and batch normalization do not make CNNs convert faster. They make convergence more stable and also limits overfitting to the training data.

1.5 False. Sometimes dropout layers do not perform well in cooperation with L-norm regularization.

1.6 True. Because the L1 regularization space has a diamond contour, it will often meet the solution loss function at its corner, thus obtaining sparser weights.

1.7 True. Although Leaky ReLU solves the dead neuron issue, the inconsistent slope of the activation makes some training processes unstable.

1.8 True. Using 3x3 depthwise convolution kernels in MobileNet as opposed to regular 3x3 convolutional kernels help reduces the MACs and thus boosts the computational speed by 9x.

1.9 True. SqueezeNet uses a combination of squeeze modules and expansion modules to preserve larger feature maps at the start of the network and keep computational costs lower until the end.

1.10 True. Shortcuts in ResNet address the vanishing gradient problem and makes the loss surface smoother.

Lab 1: Simple Neural Network and Learning Rate Decay

a.) To check the sanity of the network, we passed a batch of one dummy tensor with the same shape as the CIFAR-10 dataset (3 channels of 32x32 pixels) into the model to check if it was able to complete a forward pass. The output we received is a tensor of size (10) which is what we expected. We also checked to make sure that the parameter sizes of each layer conforms with our expectation.

```
conv1.weight: torch.Size([8, 3, 5, 5])
conv1.bias: torch.Size([8])
conv2.weight: torch.Size([16, 8, 3, 3])
conv2.bias: torch.Size([16])
fc1.weight: torch.Size([120, 576])
fc1.bias  : torch.Size([120])
fc2.weight: torch.Size([84, 120])
fc2.bias  : torch.Size([84])
fc3.weight: torch.Size([10, 84])
fc3.bias  : torch.Size([10])
```

The first convolutional layer goes from 3 channels to 8 channels and narrows the feature map to 28x28. Max pooling with a stride of 2 shrinks that feature map to 14x14. The second convolutional layer goes from 8 to 16 channels and narrows the feature map to 12x12. And another max pooling layers shrinks it even further to 6x6, with still 16 channels. The 16x6x6 = 576 pixels are then flattened and fed into a fully-connected layer, with 120 output nodes. After 2 more fully connected layers, the final output is linear with a size of 10, which makes sense for our digit classification algorithm.

b.) Assuming that the input images are in PIL format, I used `transform.toTensor` to convert the images to tensors and then normalized the input pixels using the mean and standard deviation given by the homework.

c.) To instantiate the data loaders, I passed in the transform functions defined in the previous part of the question. Also passing in the training and validation batch sizes, I set the shuffle in the train loader to be true and the shuffle in the validation loader to be false.

d.) I used the following code to check if my model is deployed on the GPU.

```
# check if gpu is available
device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device =='cuda':
    print("Run on GPU...")
else:
    print("Run on CPU...")

# Instantiate and deploy model
model = SimpleNN()
model = model.to(device)
```

To verify that the model is deployed on GPU, I checked what device the model parameters are on, and the output being `cuda` meant that they were indeed on GPU.

e.) In defining the loss function, we used `CrossEntropyLoss` to define the criterion for loss function and used the Stochastic Gradient Descent optimizer with pre-defined hyperparameters.

f.) I wrote a function to set up the training process and to make it reusable for later parts of this assignment.

I first switched the model to train and wrote a for loop to loop over the epochs. Within each epoch, I looped over the train loader to do forward propagation, back-propagation, and then gradient descent. Then I calculated the number of corrected examples and the loss. After each epoch, I append the number of correct examples and full loss to a tracker.

Switching over to evaluation mode in each epoch, I again looped over each batch in the validation set, disabling gradients, and did a forward propagation through the network using data in the validation batch. Then I calculated the number of correct examples and losses, which was compiled and tracked again after each epoch.

g.) The initial loss before I did any training steps is 2.304 in the training data, which is right about what we expected for cross-entropy loss on CIFAR-10 data. This is because we expect the cross-entropy loss $L_{CE} = -\sum_j y_i log(s_j)$. Where y_j is the probability of the j^{th} class in the target distribution, and s_j is the probability of the j^{th} class in the predicted distribution. Since s_j will be around $\frac{1}{10}$, the estimated CE is $log10$ which is around 2.31.

In Figure 1, we show the training and validation accuracy and loss plot for the simple neural network without implementing any learning rate decay.

The best validation accuracy we achieved was **0.6606**.

Looking at the training accuracy and validation accuracy, we notice training accuracy continuously improving throughout the 30 epochs. However, the validation accuracy starts hitting a ceiling at around 10 epochs and even begins declining after then. This suggests that the model is overfitting on training data and is not generalizing well to new data it has not seen before. The validation loss paints a similar picture: we see loss increasing at about the same number of epochs, even as training loss decreases.

The current training pipeline is not deep enough to detect the important features from the images. There is also no batch normalization that helps with stabilizing the training process and limiting over-fitting.

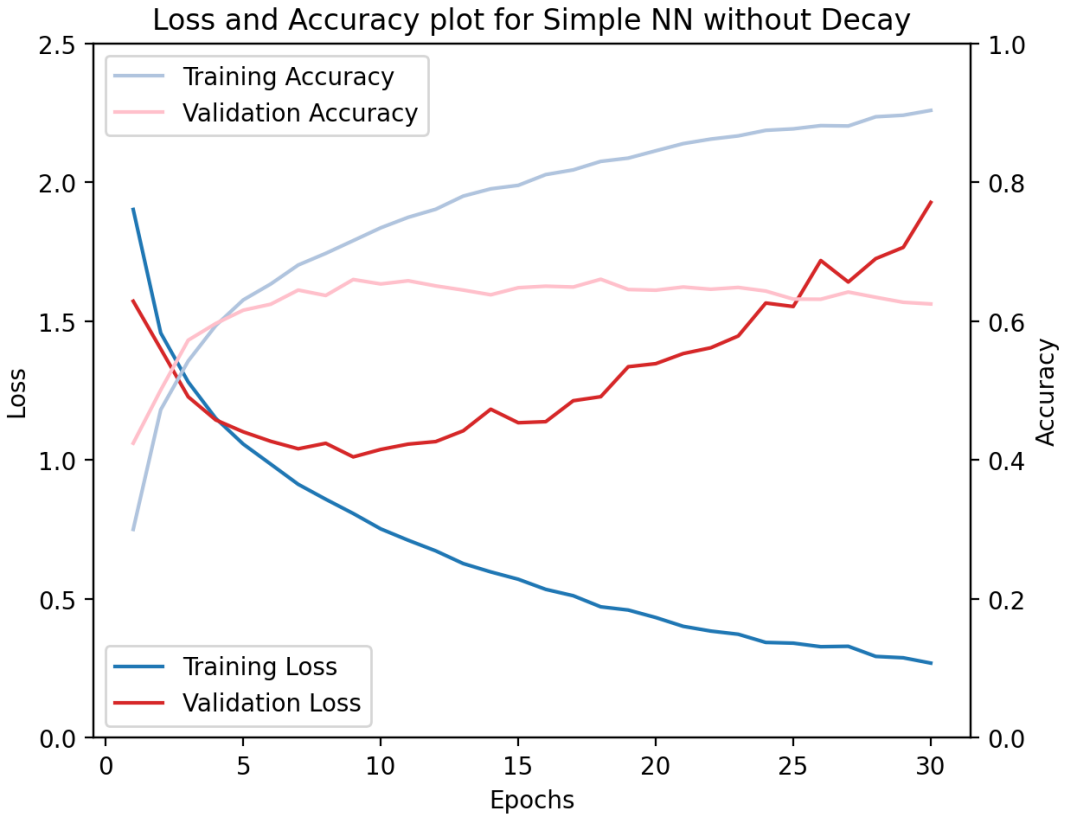

Figure 1: Accuracy and Loss Plot for SimpleNN Model

h.) We used a `DECAY_EPOCHS` of 10 and a `DECAY` strength of 0.9, so the learning rate decays by 0.9 every 10 epochs.

We ended up achieving the best validation accuracy of **0.6576**, which is not as high as the validation accuracy in the model without decay.

Figure 2 shows the loss and accuracy plot for the simple neural network with decay.

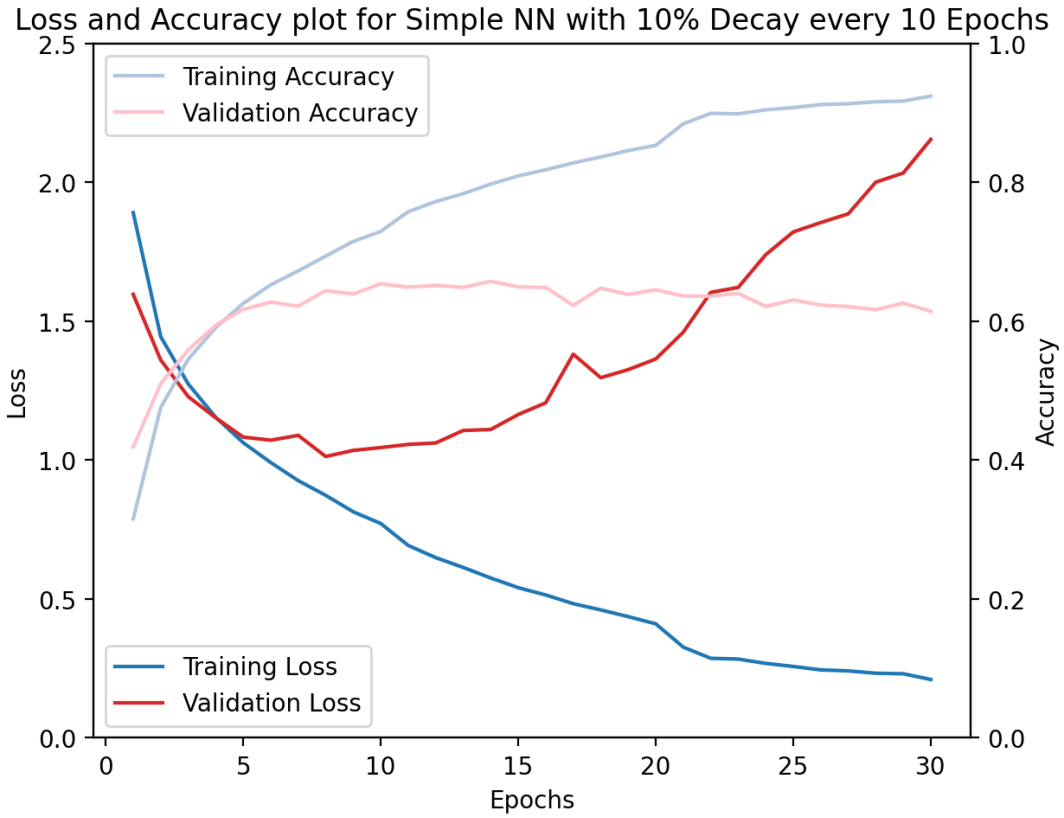

Figure 2: Accuracy and Loss Plot for SimpleNN Model with decay

We see that the amount of decay and the number of epochs did not help much with limiting the problem of over-fitting. And the validation loss was still increasing as the training loss was decreasing.

Lab 2: Improving Training Pipeline

a.) We instantiated the SimpleNN model from the first lab and changed our data transform pipeline to include data augmentation. We included random cropping with padding and random horizontal flips for the training batch. Below is the code we used to do this:

```
transform_train = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
    (0.2023, 0.1994, 0.2010)),
    transforms.RandomCrop(size = 32, padding = 4),
    transforms.RandomHorizontalFlip()]
    )

transform_val = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
    (0.2023, 0.1994, 0.2010))]
    )
```

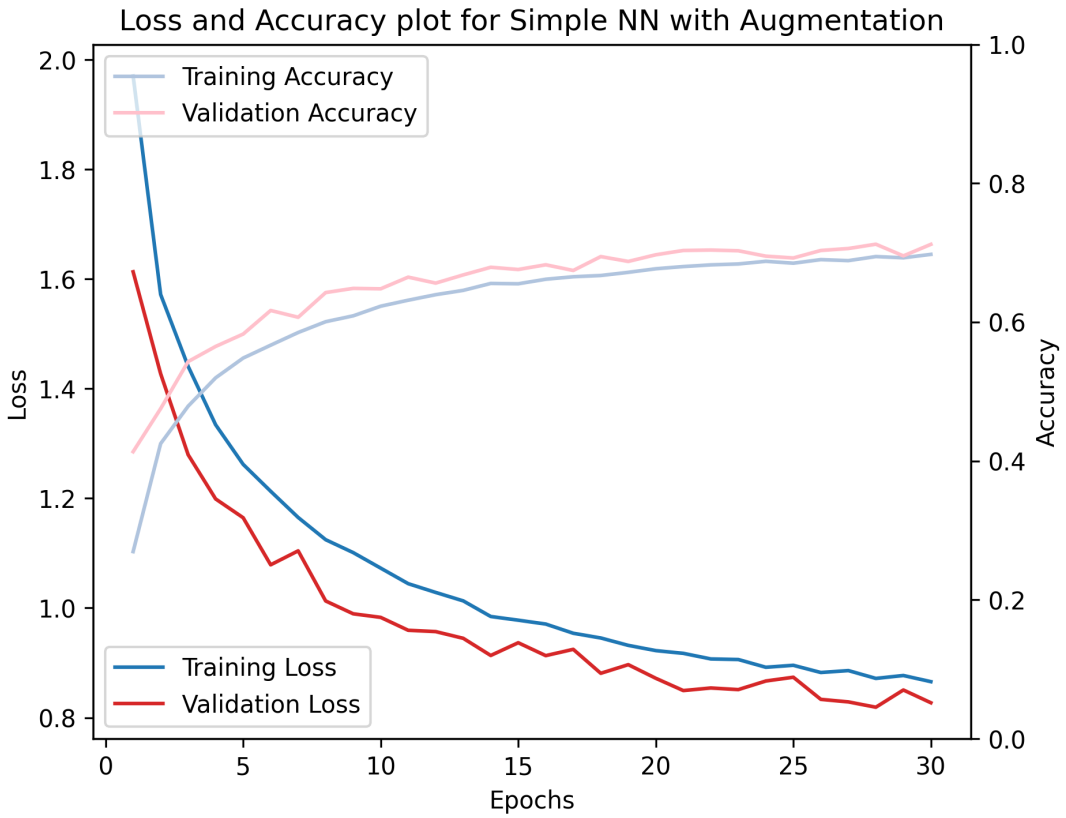

Figure 3: Accuracy and Loss Plot for SimpleNN Model with Data Augmentation

We show the model performance of the model with data augmentation in Figure 3. The model performance improved significantly from the model without aug-

mentation (without learning rate decays). The model achieved the best validation accuracy of **0.7122**, which is a significant improvement from the best validation accuracy in the SimpleNN model without augmentation. And the model did not show signs of significant over-fitting.

b.) 1. We created a new training pipeline with Batch Normalization after each convolutional layer. Figure 4 shows the loss and accuracy plots of the model.

The best validation accuracy of **0.7144** is a small improvement upon the SimpleNN model in part (a) of this question.

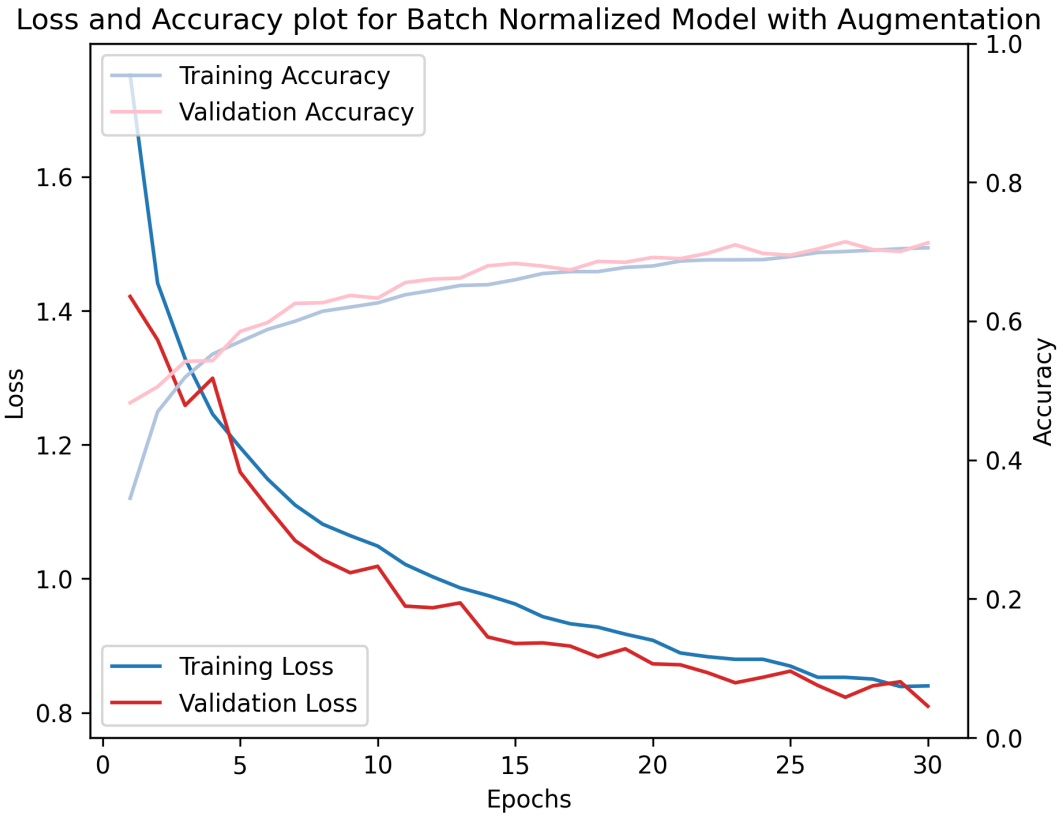

Figure 4: Accuracy and Loss Plot for Batch Normalized Model

2. To show that batch normalization allows for a higher learning rate, we train two models with a moderately high learning rate of 0.1 – one with batch normalization, and one without – both with the same set of hyperparameters. In Figure 4, we compare the training and validation loss and accuracies in both models to determine whether batch normalization improved model performance with a high learning rate.

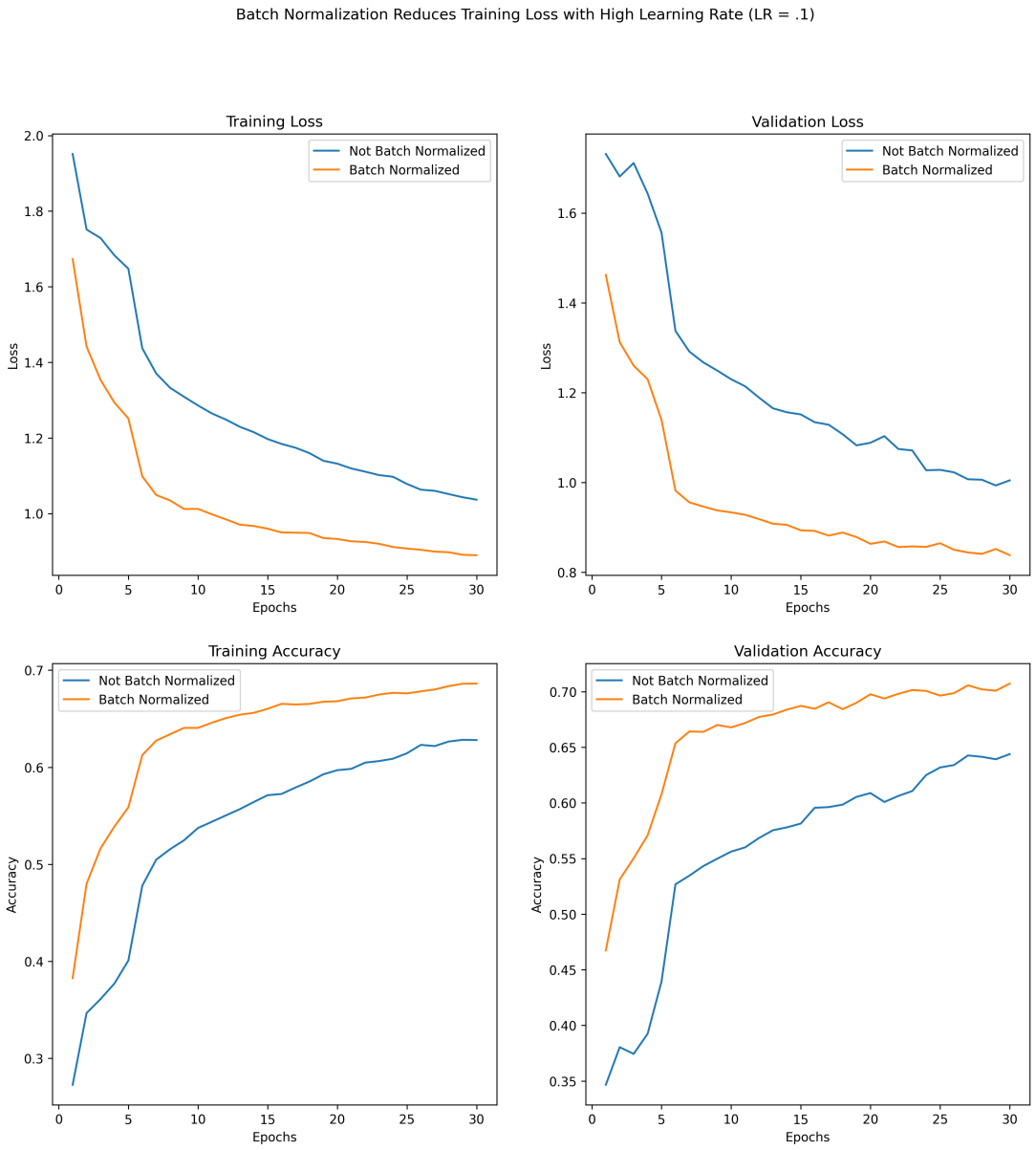

Figure 5: Comparison of loss and accuracies for a batch-normalized model and a non-batch-normalized model, trained with Learning Rate = 0.1

We see that batch normalization was able to significantly reduce the loss and increase accuracy in both the training and validation set when the learning rate is set high.

3. To implement the Swish activation function, I defined a PyTorch module that takes a tensor as input and outputs the activated tensor $f(x) = x * \sigma(x)$. Here is how that function was defined:

```
class Swish(nn.Module):
def __init__(self):
    super(Swish, self).__init__()

def forward(self, x):
    return torch.mul(x, torch.sigmoid(x))
```

We replaced the ReLU modules in our previous model with Swish activation functions and then trained the model on the dataset with BN and a learning rate of 0.1. This model was able to achieve the best validation accuracy of **0.7258**, which was an improvement upon the best validation accuracy of the ReLU model. Figure 6 shows the accuracy and loss graphs of training the Swish model. We see that the loss descent is really stable and the model performs well relatively.

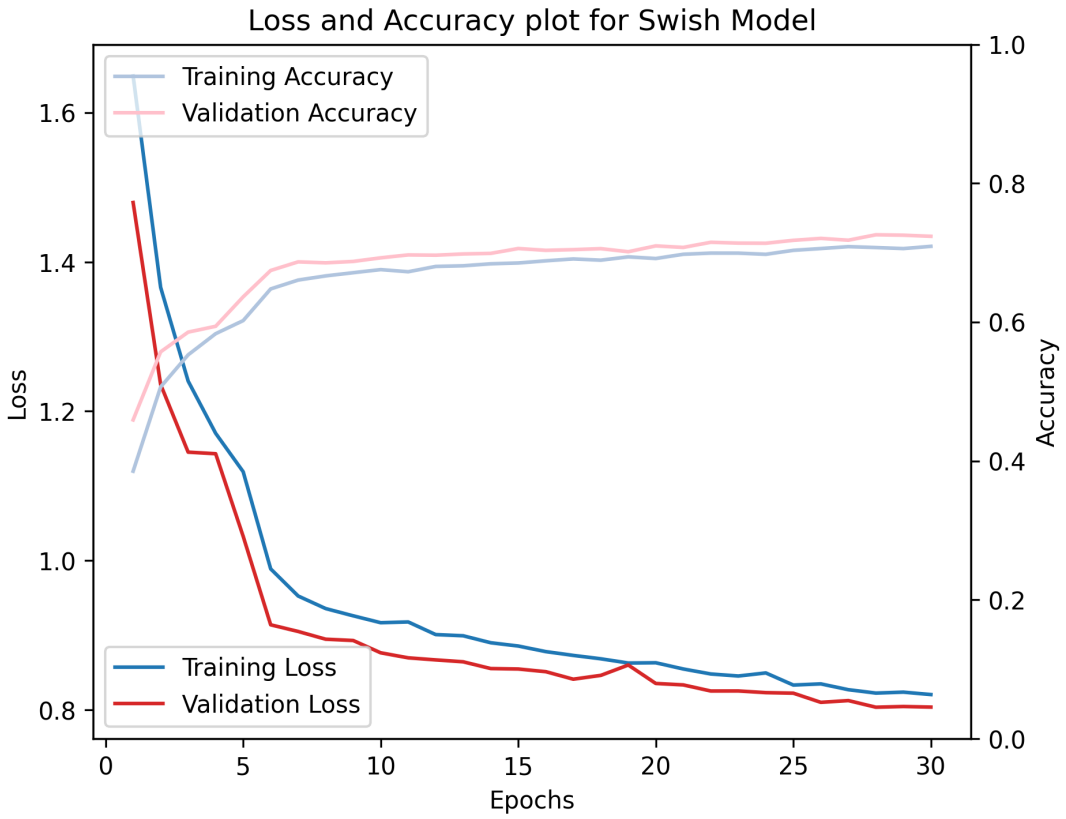

Figure 6: Accuracy and Loss Plot of Swish Model

c.) In this section, we will tune the hyperparameters such as learning rates and regularization strengths.

1. To tune the learning rates, we instantiated 6 models with varied learning rates: [1, 0.1, 0.05, 0.01, 0.005, 0.001] and tracked the loss and accuracy plots during the training of each of the models. Figure 7 shows the training loss, validation loss, training accuracy, and validation accuracy of all six models during training.

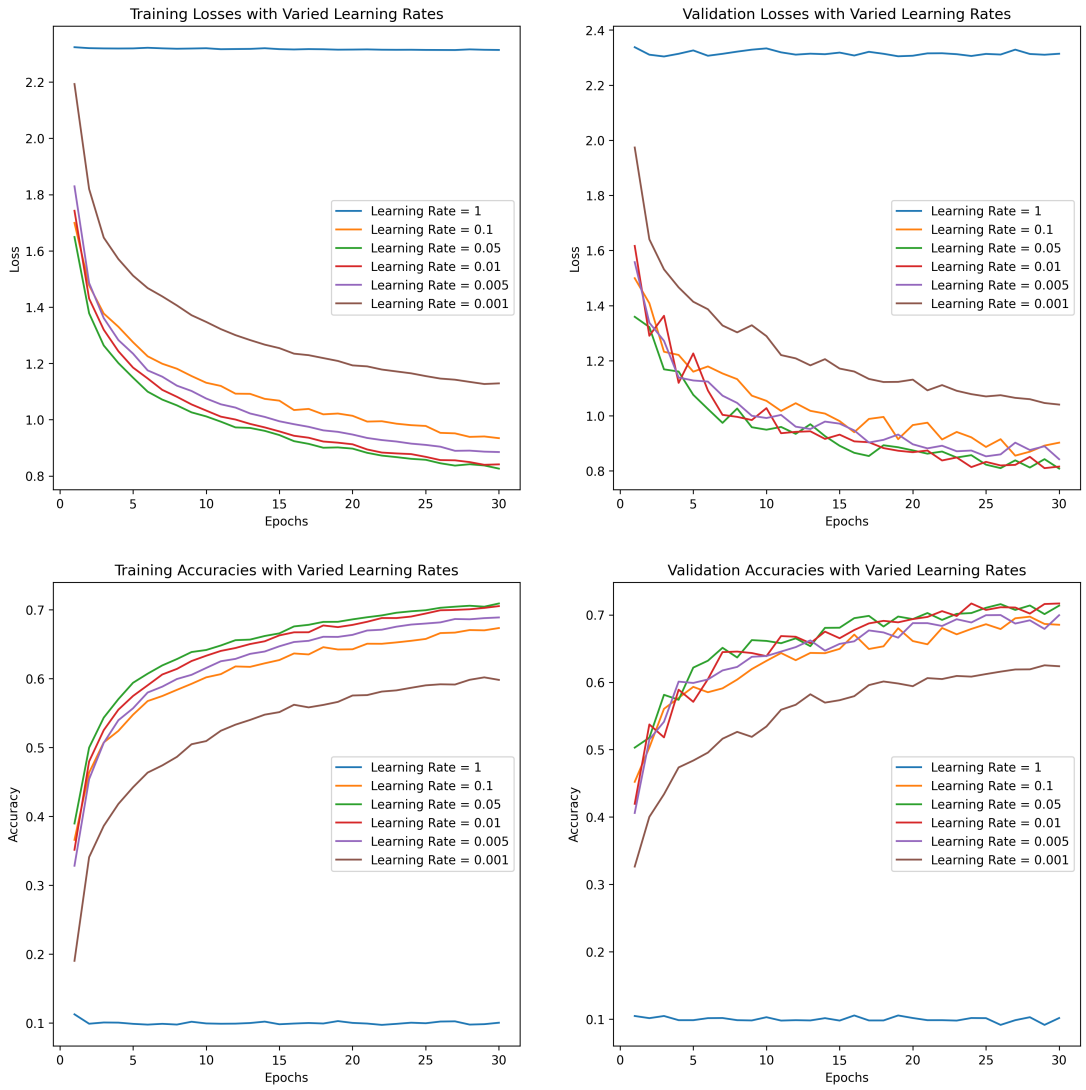

Figure 7: Accuracy and Loss Plots of Models with Varied Learning Rates

From the plot, we can determine that larger learning rates are not necessarily beneficial for model training. In fact, the model with a learning rate of 1 did not converge through the training at all. Instead, it's learning rates that are middle of the pack that stand the best chance of getting the lowest losses and highest accuracy. We see that a learning rate of 0.05 had the best performance in this instance. Having too small a learning rate also costs the model in terms of loss and accuracy. Therefore, we can conclude that it's optimal to choose a mid-ranged learning rate.

2. To test the performance of different L2 regularization strengths, we also instantiated different models with varied L2 regularization parameters: [1e-2, 1e-3, 1e-4, 1e-5, 0.0]. Figure 8 shows the results of these models.

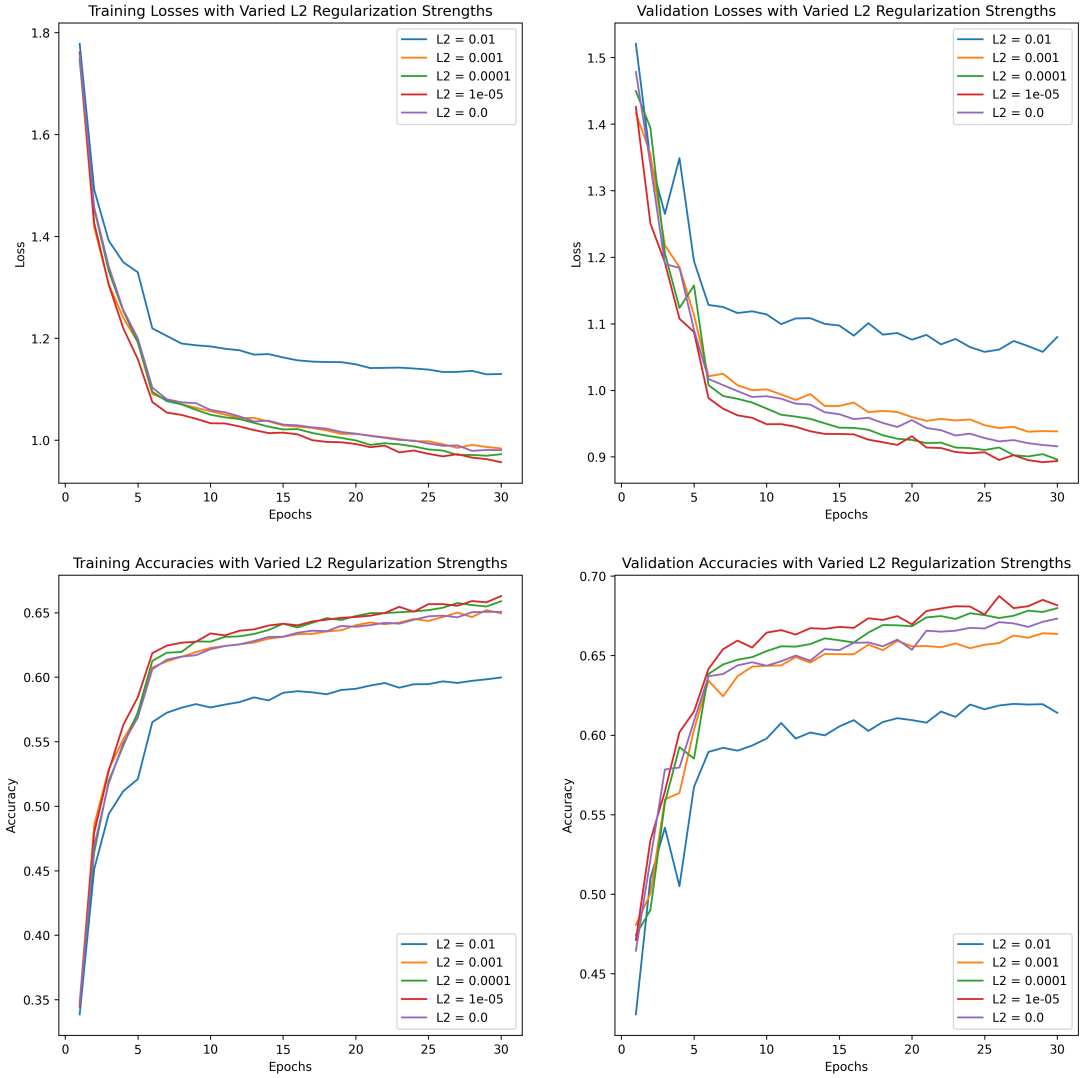

Figure 8: Accuracy and Loss Plots of Models with Varied L2 Regularization Strengths

From the plot and the model performances, we learn that having an L2 strength of 1e-5 was the best for loss functions and accuracies. Penalizing the L2 norm by too much resulted in suboptimal model performance, and having no regularization also resulted in a small reduction in accuracy. Therefore, the strength parameter is important and should be set after trial and error.

3. To implement L1 regularization, I made a custom loss function that took in the outputs, targets, model, and strength parameter to calculate an adjusted version of the cross entropy loss by adding on the L1-norm sum of each model parameter. Here is how I defined the `L1CrossEntropyLoss` function:

```
def L1CrossEntropyLoss(outputs, targets, model, reg):
# Instantiate L1 loss at 0
l1_loss = 0.0
# Iterate over model parameters to get model L1 weights
for param in model.parameters():
    l1_loss += torch.norm(param.data, p = 1)
    pass
```

```
# Get the Cross Entropy Loss
cel = nn.CrossEntropyLoss()
# Ad the L1 regularization loss
loss = cel(outputs, targets) + l1_loss * reg
return loss
```

After training a model with this implementation of L1 regularization. The model of L1 and L2 regularized model performance in accuracy and loss curve are shown in Figure 9 and 10 respectively. I then plotted the histogram of model parameters as seen in figure 11.

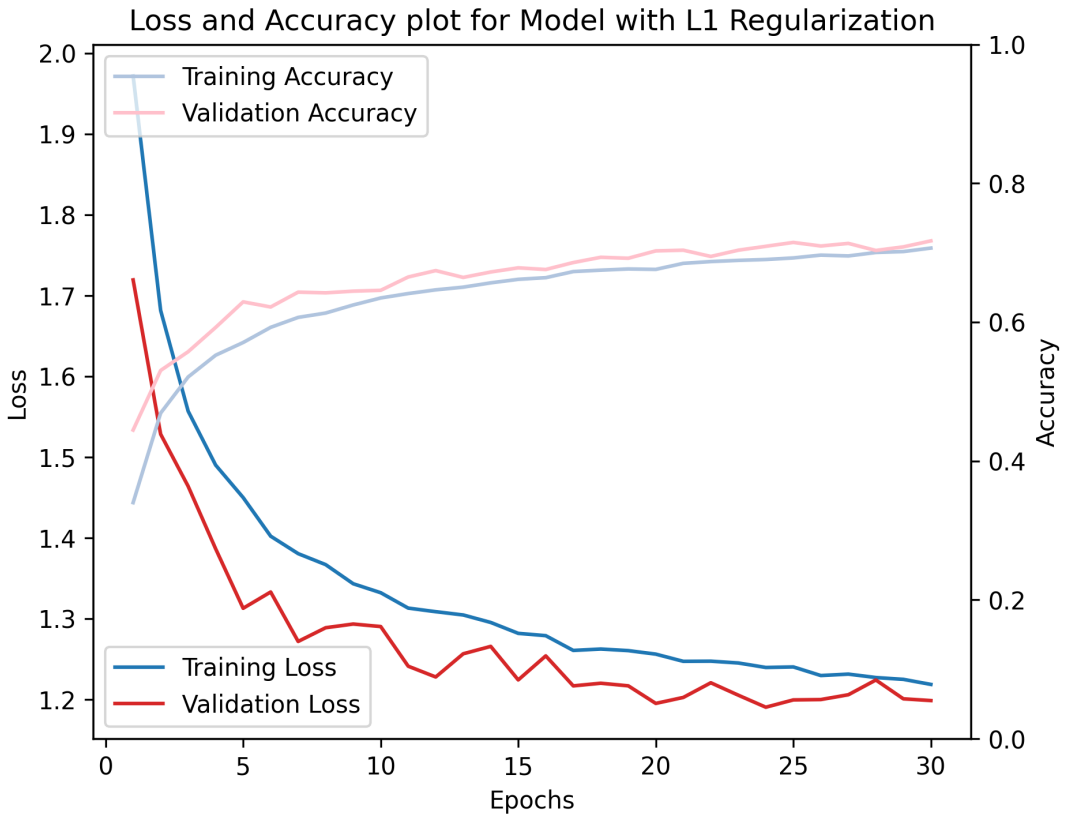

Figure 9: Accuracy and Loss Plots of Model with L1 Regularization

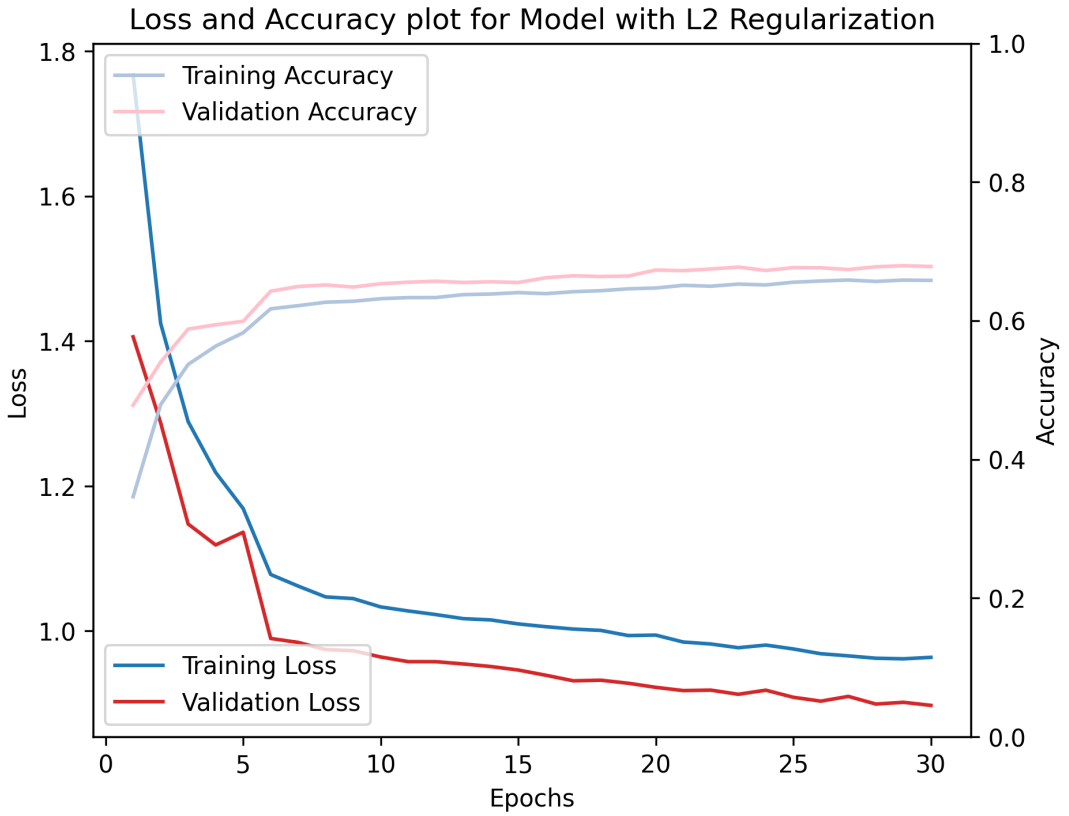

Figure 10: Accuracy and Loss Plots of Model with L2 Regularization

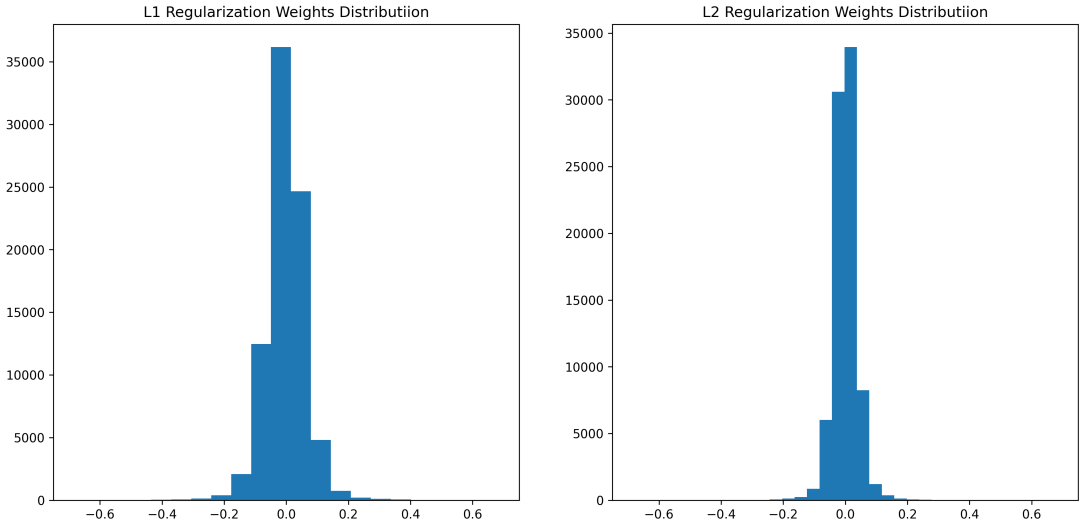

Figure 11: Weight Distribution of Models with L1 and L2 Regularization

We see that there are slightly more weights around 0 in the model with L1 regularization compared to the model with L2 regularization, even though the weight distributions are not distinctly different.

Lab 3: Advanced CNN Architectures

a.) In implementing the ResNet Model, I followed the Deep residual learning for image recognition paper by He et.al. I built a 20-layer model with 19 convolutional layers with shorcut connections after each pair of convolutional layers, with 9 shorcut connections in total. We implemented residual learning through these shortcuts to enable deeper networks, smooth learning curves, and limit gradient vanishing. I used the paper's suggested version of implementing shortcuts (option A) while switching between different channel sizes, using zero padding for extra channels. *Additional implementation details can be found in the resnet-cifar10.ipynb notebook.*

After some trials and errors, I ended up using the paper's suggestions for hyperparameters. With an initial learning rate of 0.1, I implemented a 90% learning rate decay every 50 epochs, meaning that every 50 epochs, the learning rate shrinks from 0.1 to 0.01 or from 0.01 to 0.001. I ran the model for 200 epochs, with 1e-4 L2 regularization strengths, and using stochastic gradient descent optimizer.

b.) The model achieved the best validation accuracy of **0.924**. The loss curve and the accuracy curve of the model on the training and validation sets can be found in Figure 12 below.

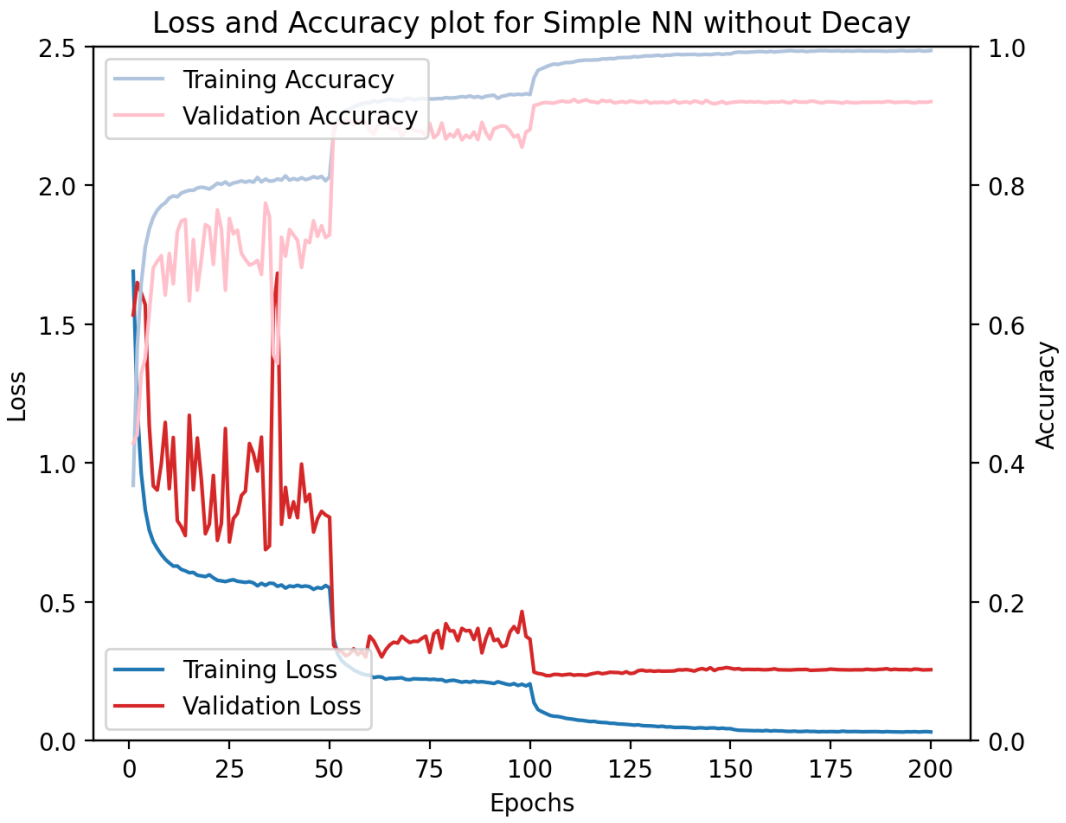

Figure 12: Accuracy and Loss Curves in ResNet Model

Interestingly, we see that at the epochs where learning rates decay, the model is able to achieve dramatic reductions in training and validation losses. This boost starts to depreciate at around 150 epochs, where the loss is already low enough that there are no additional performance benefits from decay.