Barbara Flores

ECE 661: Homework #2 Construct, Train, and Optimize CNN Models

Hai Li

ECE Department, Duke University — Spring, 2025

Objectives

Homework #2 covers the contents of Lectures 05∼08. This assignment includes basic knowledge about CNNs, detailed instructions on how to setup a training pipeline for training image classifiers on the CIFAR- 10 dataset, how to improve the training pipeline, and how to use advanced CNN architectures to improve the performance of image classifiers. In this assignment, you will gain hands-on experience training a neural network model on a real computer-vision dataset (i.e., CIFAR-10), while also learning techniques for improving the performance of your CNN model.

We encourage you to complete the Homework #2 on the JupyterLab server or Google CoLab since the model training will require the computing power of GPUs. When conducting the lab projects, actively referring to the **NumPy/PyTorch tutorial** slides on Canvas for instructions on the environment setup and NumPy/PyTorch utilities can be very helpful.

Warning: You are asked to complete the assignment independently.

This lab has 100 points plus 10 bonus points, yet your final score cannot exceed 100 points. You must submit your report in PDF format and your original codes for the lab questions through Gradescope before **11:55pm, Monday, February 17**. We provide a template named simplenn-cifar10.ipynb to start with, and you are asked to develop your own code based on this template. You will need to submit five independent files including:

1. A self-contained PDF report, which provides answers to all the conceptual questions and clearly demonstrates all your lab results and observations. **This means including all relevant code snippets!**
2. Three jupyter notebooks for the three labs :
   simplenn-cifar10.ipynb,
   simplenn-cifar10-dev.ipynb,
   resnet-cifar10.ipynb.
3. predictions.csv, your predicted label for each image in the provided CIFAR-10 testing split. See detailed instructions in Lab (3).

Note that the grade will be deducted if the submissions doesn't follow the above guidance. Remember, do NOT generate PDF from your jupyter notebook to serve as the report, which can increase the TA's burden of grading.

Note that TAs hold the right to adjust grading based on the returned homeworks. We make sure that the grading rule is consistent among all students. Also, the results given for the Labs (for example the reported accuracies) are obtained from the specific runtime when TAs were working on the answers. We do not expect you to get exactly the same numbers; yet, it is necessary that your results show the same trends/patterns/observations in order to receive full credits.

1 True/False Questions (15 pts)

For each question, please provide a short 1-2 sentence explanation to support your judgment.

Problem 1.1 (3 pts) Data augmentation techniques are always beneficial for deep learning applications.

False. Data augmentation techniques are not always beneficial. For small models, they may lead to underfitting as the model may struggle to capture the essential patterns in the data. Additionally, if not applied carefully, data augmentation can introduce noise or distort the dataset, negatively impacting model performance.

Problem 1.2 (3 pts) Batch normalization and dropout are regularization techniques known to cause CNN training to converge much more quickly.

False. While both batch normalization and dropout are regularization techniques that prevent overfitting and improve generalization, they affect convergence differently. Batch normalization can accelerate convergence by stabilizing the learning process, while dropout generally slows down convergence because it randomly drops neurons during training, introducing noise.

Problem 1.3 (3 pts) Dropout is a common technique to combat overfitting. L-normalizations are another regularization technique used to combat overfitting. Using both will combat overfitting the best and overall help with training.

False. While both dropout and L-norm regularization help combat overfitting, using them together doesn't always yield the best results. Sometimes they don't cooperate well, as both introduce noise and constraints that can interfere with each other, potentially leading to underfitting or slowed convergence.

Problem 1.4 (3 pts) During training, the Lasso (L1) regularizer will cause the model to have a *lesser* degree of sparsity compared to the Ridge (L2) regularizer.

False. L1 regularization leads to sparse weights by pushing some of them to exactly zero, resulting in a more compact model. This happens because its diamond-shaped loss contour tends to intersect with the feasible region at the corners, where some dimensions are zero. In contrast, L2 regularization makes the weights small but not exactly zero due to its circular loss contour, leading to less sparsity.

Problem 1.5 (3 pts) The shortcut connections in ResNets improve training stability because they produce a *smoother* loss surface.

True. The shortcut connections in ResNets improve training stability because they produce a smoother loss surface, making the optimization process easier. By learning residual mappings instead of direct mappings, ResNets avoid noisy and curvy loss surfaces that can trap the optimization process in local minima. As a result, the smoother loss surface helps the model converge more easily and stably during training.

2 Computation Questions (15 pts)

For each question, provide the result and show the reasoning.
Hint: How many channels are in an RGB image?

Problem 2.1 (3pts) Consider a 100x100 RGB Image as an input. If the first hidden layer consists of 100 neurons, and each neuron is fully connected to the input, how many parameters does this hidden layer have including both the weights and the bias parameters?

RGB image → 3 channels (Red, Green, and Blue).

Total Parameters = $(C \times H \times W \times n_{out}) + Bias$

Total parameters = (3 x 100 x 100 x 100) + 100

Total parameters = 3,000,100

Problem 2.2 (3pts) Consider the same 100x100 RGB Image as an input. If you use a convolution layer with 100 filters, each with size 3x3, how many parameters does this hidden layer have (including both the weights and the bias parameters)?

Total Parameters = $(C \times H \times W \times n_{out}) + Bias$

Total parameters = (3 x 3 x 3 x 100) + 100

Total parameters = 2,800

Problem 2.3 (3pts) Consider an input volume with dimensions 100x100x16, how many parameters does **a single** 1x1 convolution filter have including the bias term?

Because is **a single** 1x1 convolution we have Total Parameters = $(C \times K_H \times K_W) + Bias$

Total parameters = 16 x 1 x 1 + 1

Total parameters = 17

Problem 2.4 (3pts) Consider the input volume of 100x100x16, and we apply convolution with 32 filters each 5x5 size with a stride of 1 and no padding. What is the output shape?

$$W_2 = \left|\frac{W_1 - K + 2P}{S}\right| + 1 \; ; \qquad H_2 = \left|\frac{H_1 - K + 2P}{S}\right| + 1$$

$$W_2 = H_2 = \left|\frac{100 - 5 + 2 * 0}{1}\right| + 1 = 96$$

The output shape of the convolutional layer is **96 x 96 x 32**

Problem 2.5 (3pts) MobileNets use depthwise separable convolution to improve the model efficiency. If we replace all the 3x3 convolution layers in a ResNet architecture with 3x3 depthwise separable convolution layers, what would be the likely speedup for these layers?

For a 3x3 regular convolution, we have

MACs = $3\times3\times M\times N\times D_F\times D_F$

Where:

- M = Number of input channels
- N = Number of output channels
- D_F = Spatial dimension of the feature ma

For a 3×3 depthwise separable convolution, we have:

MACs = $3\times3\times M\times N\times D_F\times D_F + D_F\times D_F\times M\times N$
Parameters = $3\times3\times M + M\times N$

When N, M is large, the theoretical speedup is calculated as:

$$\text{Speedup} = \frac{\text{MACs of regular convolution}}{\text{MACs of depthwise separable convolution}}$$

$$\text{Speedup} = \frac{3\times3\times M\times N\times D_F\times D_F}{3\times3\times M\times N\times D_F\times D_F + D_F\times D_F\times M\times N} \approx 9$$

If we replace all the 3×3convolution layers in a ResNet architecture with 3×3 depthwise separable convolution layers, the likely speedup would be approximately **9 times**. This is because depthwise separable convolutions significantly reduce the number of parameters and computations by breaking down the standard convolution into two simpler operations: depthwise convolution and pointwise convolution.

3 Lab (1): SimpleNN for CIFAR-10 classification (15+4 pts)

Just like in HW1, here we start with a simple CNN architecture which we term as SimpleNN. It is composed of 2 CONV layers, 2 POOL layers, and 3 FC layers. The detailed structure of this model is shown in Table 1.

Name	Type	Kernel size	depth/units	Activation	Strides
Conv 1	Convolution	5	8	ReLU	1
MaxPool	MaxPool	2	N/A	N/A	2
Conv 2	Convolution	3	16	ReLU	1
MaxPool	MaxPool	2	N/A	N/A	2
FC1	Fully-connected	N/A	120	ReLU	N/A
FC2	Fully-connected	N/A	84	ReLU	N/A
FC3	Fully-connected	N/A	10	None	N/A

Table 1: SimpleNN structure. No padding is applied on both convolution layers. A flatten layer is required before FC1 to reshape the feature.

In this lab, beyond model implementation, you will learn to set up the whole training pipeline and ac- tually train a classifier to perform image classification on the CIFAR-10 dataset [1]. CIFAR-10 is one of the most famous/popular benchmarks for image recognition/classification. It consists of 10 categories (e.g., bird, dog, car, airplane) with 32x32 RGB images. You may go to the official website for more information https://www.cs.toronto.edu/~kriz/cifar.html.

In this assignment, please refer to Jupyter Notebook simplenn-cifar10.ipynb for detailed instruc- tions on how to construct a training pipeline for SimpleNN model. **Note, remember to unzip the provided** tools.zip **to your workspace before getting started.**

(a) (2 pts) As a sanity check, we should verify the implementation of the SimpleNN model at **Step 0**. Determine how you can check whether the model is implemented correctly, then check it.

Hints: 1) Consider creating dummy inputs that are of the same size as CIFAR-10 images, passing them through the model, and see if the model's outputs are of the correct shape. 2) Count the total number of parameters of all CONV/FC layers and see if it meets your expectation.

1) Consider creating dummy inputs that are of the same size as CIFAR-10 images, passing them through the model, and see if the model's outputs are of the correct shape.

I created a dummy dataset with 100 images, each of size 32x32 with 3 channels, which matches the input size of CIFAR-10 images. After passing these images through the model, the output shape was 100x10 (100 images, each classified into one of 10 categories). This confirms that the model is working as expected and the output dimensions are correct for the CIFAR-10 classification task

```
##########################################
# your code here
# sanity check for the correctness of SimpleNN


# Model Definition
net = SimpleNN()

# Test forward pass
data = torch.randn(100,3,32,32)

# Forward pass "data" through "net" to get output "out"
out =  net(data)

# Check output shape
assert(out.detach().cpu().numpy().shape == (100,10))
print("Forward pass successful")

##########################################
```

```
Forward pass successful
```

2) Count the total number of parameters of all CONV/FC layers and see if it meets your expectation.

As we saw in the example from Assignment 1, if we input a 3x32x32 image into our convolutional neural network, we obtain the following configuration:

Layer	Input shape	Output Shape	Weight shape	# Param
Conv 1	(1, 3, 32, 32)	(1, 8, 28, 28)	(8, 3, 5, 5)	600 + 8 = 608
Conv 2	(1, 8, 14, 14)	(1, 16, 12, 12)	(16, 8, 3, 3)	1,152 + 16 = 1,168
FC1	(1, 576)	(1, 120)	(120, 576)	60,120 + 120 = 60,240
FC2	(1, 120)	(1, 84)	(84,120)	10,080 + 84 = 10,164
FC3	(1, 84)	(1, 10)	(10,84)	840 + 10 = 850

Now, let's proceed to check if the number of parameters in our code match.

```
[4] # 2) Count the total number of parameters of all CONV/FC
    # layers and see if it meets your expectation.

    print(f"Conv1 weight: {net.conv1.weight.numel()}, Conv1 bias: {net.conv1.bias.numel()}, Total: {net.conv1.weight.numel() + net.conv1.bias.numel()}")
    print(f"Conv2 weight: {net.conv2.weight.numel()}, Conv2 bias: {net.conv2.bias.numel()}, Total: {net.conv2.weight.numel() + net.conv2.bias.numel()}")
    print(f"FC1 weight: {net.fc1.weight.numel()}, FC1 bias: {net.fc1.bias.numel()}, Total: {net.fc1.weight.numel() + net.fc1.bias.numel()}")
    print(f"FC2 weight: {net.fc2.weight.numel()}, FC2 bias: {net.fc2.bias.numel()}, Total: {net.fc2.weight.numel() + net.fc2.bias.numel()}")
    print(f"FC3 weight: {net.fc3.weight.numel()}, FC3 bias: {net.fc3.bias.numel()}, Total: {net.fc3.weight.numel() + net.fc3.bias.numel()}")

Conv1 weight: 600, Conv1 bias: 8, Total: 608
Conv2 weight: 1152, Conv2 bias: 16, Total: 1168
FC1 weight: 69120, FC1 bias: 120, Total: 69240
FC2 weight: 10080, FC2 bias: 84, Total: 10164
FC3 weight: 840, FC3 bias: 10, Total: 850
```

As we can see, the numbers match.

(b) (2 pts) Data preprocessing is crucial to enable successful training and inference of DNN models. Spec- ify the preprocessing functions at **Step 1** and briefly discuss what operations you use and why.

For preprocessing, I used the following operations:

- **ToTensor()** → In order to convert the input PIL images to PyTorch tensors and scales pixel values to the range [0,1].
- **Normalize()** → For Standardize the pixel values to have a mean of 0 and a standard deviation of 1, helping the model converge faster.

The update in the code was:

```
] # useful libraries
  import torchvision
  import torchvision.transforms as transforms

  ############################################
  # your code here
  # specify preprocessing function
  transform_train = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize(mean=(0.4914, 0.4822, 0.4465), std=(0.2023, 0.1994, 0.2010))])

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

(c) (2 pts) During the training, we need to feed data to the model, which requires an efficient data loading process. This is typically achieved by setting up a dataset and a dataloader. Please go to **Step 2** and build the actual training/validation datasets and dataloaders. Note, instead of using the CIFAR10 dataset class from torchvision.datasets, here you are asked to use our own CIFAR-10 dataset class, which is imported from tools.dataset. As for the dataloader, we encourage you to use torch.utils.data.DataLoader.

```
from google.colab import drive
drive.mount('/content/drive')

import sys
sys.path.append('/content/drive/MyDrive/2023 - Duke/2025-01/ECE661_ComputerEngineeringMachineLearning&DeepNeuralNets/Assigments/HW2')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader

# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

##########################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=transform_train
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,
    shuffle=True,
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,
    shuffle=False,
    num_workers=4
)
##########################################
```

```
Downloading https://www.dropbox.com/s/s8orza214q45b23/cifar10_trainval_F22.zip?dl=1 to ./data/cifar10_trainval_F22.zip
141746176it [00:08, 17114159.24it/s]
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

(d) (2 pts) Go to **Step 3** to deploy the SimpleNN model on GPUs for efficient training. How can you verify that your model is indeed deployed on GPU? *Hint: use* nvidia-smi *command in the terminal*

I verified that my model is deployed on the GPU by using the nvidia-smi command. The output shows the NVIDIA A100 GPU with CUDA version 12.4. The GPU memory usage and overall status confirm that the GPU is available for computation.

```
[8] !nvidia-smi
```

```
Sat Feb 15 01:04:42 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-SXM4-40GB          Off |   00000000:00:04.0 Off |                    0 |
| N/A   31C    P0             47W /  400W |       0MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

```
# specify the device for computation
##########################################
# your code here

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

# Model Definition
net = SimpleNN()
net = net.to(device)

##########################################
```

```
Run on GPU...
```

(e) (2 pts) Loss functions are used to encode the learning objective. Now, we need to define this problem's loss function as well as the optimizer which will update our model's parameters to minimize the loss. In **Step 4**, please fill out the loss function and optimizer part.

```
0s
import torch.nn as nn
import torch.optim as optim

# hyperparameters, do NOT change right now
# initial learning rate
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4

##########################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
optimizer = optim.SGD(
    net.parameters(),
    lr=INITIAL_LR,
    momentum=MOMENTUM,
    weight_decay=REG
)
##########################################
```

(f) (2 pts) Follow the instructions in **Step 5** to set up the training process of SimpleNN on the CIFAR-10 dataset.

```
# some hyperparameters
# total number of training epochs
EPOCHS = 30

# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR

print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    ########################
    # your code here
    # switch to train mode
    net.train()
    ########################

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        ###################################
        # your code here
        # copy inputs to device
        inputs = inputs.to(device)
        targets = targets.to(device)

        # compute the output and loss
        outputs = net(inputs)
        loss = criterion(outputs, targets)

        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()

        train_loss += loss.item()

        # count the number of correctly predicted samples in the current batch
        max_values, predicted = outputs.max(1)
        total_examples += targets.size(0)
        correct_examples += (predicted == targets).sum().item()
        ###################################
```

```
    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))

    # Validate on the validation dataset
    ########################
    # your code here
    # switch to eval mode
    net.eval()

    ########################

    # this help you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ###################################
            # your code here
            # copy inputs to device
            inputs = inputs.to(device)
            targets = targets.to(device)

            # compute the output and loss
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            val_loss += loss.item()

            # count the number of correctly predicted samples in the current batch
            max_values, predicted = outputs.max(1)
            total_examples += targets.size(0)
            correct_examples += (predicted == targets).sum().item()
            ###################################

    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss, avg_acc))

    # save the model checkpoint
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
        #if not os.path.exists(CHECKPOINT_FOLDER):
        #    os.makedirs(CHECKPOINT_FOLDER)
        #print("Saving ...")
        #state = {'state_dict': net.state_dict(),
        #         'epoch': i,
        #         'lr': current_learning_rate}
        #torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'simplenn.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.4f}")
```

(g) (3 pts) Start training with the provided hyperparameter setting. What is the initial loss value before you conduct *any* training step? How is it related to the number of classes in CIFAR-10? What can you observe from **training accuracy** and **validation accuracy**? Do you notice any problems with the current training pipeline?

As we saw in Lecture 5, slide 39, the cross-entropy loss function for multiclass classification is defined as:

$$l_{CE} = \sum_{j=1}^{N} y_i \log(s_j)$$

where

y_j → probability of the j-th class in the true label distribution.

s_j → probability of the j-th class in the predicted distribution.

For the case of CIFAR-10, which has $N = 10$ classes, the initial loss value before starting the training should approximate $\log(N)$. This is because if the model weights are initialized randomly, the network's output will not have any preference for any particular class, and the predicted probabilities sj will be approximately equal for each class. Therefore, the initial loss will be:

$$\log(10) \approx 2.31$$

This value is the expected initial loss value before the model has learned anything.

We can verify this by running the following code before starting the training

```
net.eval()
with torch.no_grad():
    inputs, targets = next(iter(train_loader))
    inputs, targets = inputs.to(device), targets.to(device)
    outputs = net(inputs)
    loss = criterion(outputs, targets)

print(f"Initial loss: {loss.item()}")
```

```
Initial loss: 2.3032326698303223
```

As observed from the printed training and validation accuracy values, both accuracies initially increase as the model starts to learn. However, around epochs 10 to 15, we can notice that the training accuracy continues to increase, but with smaller improvements, and begins to show some oscillations toward the end of the training. This could indicate that the model is nearing its capacity for learning from the training data, but is not making significant progress beyond this point.

On the other hand, the validation accuracy starts to plateau and shows a more pronounced lack of improvement, which is a typical sign of **overfitting**. The model seems to be fitting better to the training data without generalizing well to the validation set.

As noted in Lecture 6, slide 39, a possible reason for the validation accuracy stalling could be the learning rate being too high, causing the model to overshoot optimal solutions. A solution to this could be implementing a learning rate schedule, where the learning rate decays after a certain number of epochs, especially when the validation accuracy plateaus. This could help the model make finer adjustments and potentially improve generalization.

Besides learning rate decay, other methods to address overfitting and improve performance on unseen data include using dropout regularization, data augmentation, and early stopping.

(h) (**Bonus**, 4 pts) Currently, we do not decay the learning rate during the training. Try to decay the learning rate (you may play with the DECAY_EPOCHS and DECAY hyperparameters in Step 5). What can you observe compared with no learning rate decay?

At the end of Lab 1, we expect at least 65% validation accuracy if all the steps are completed properly. You are required to submit the completed version of simplenn-cifar10.ipynb for Lab (1).

```
EPOCHS = 30
DECAY_EPOCHS = 10
DECAY = 0.1
net = SimpleNN()
net = net.to(device)

# Add optimizer
optimizer = optim.SGD(
    net.parameters(),
    lr=INITIAL_LR,
    momentum=MOMENTUM,
    weight_decay=REG
)
# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR

print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    ######################
    # your code here
    # switch to train mode
    net.train()
    ######################

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        ###################################
        # your code here
        # copy inputs to device
        inputs = inputs.to(device)
        targets = targets.to(device)

        # compute the output and loss
        outputs = net(inputs)
        loss = criterion(outputs, targets)

        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()
        train_loss += loss.item()

        # count the number of correctly predicted samples in the current batch
        max_values, predicted = outputs.max(1)
        total_examples += targets.size(0)
        correct_examples += (predicted == targets).sum().item()
        ###################################

    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))

    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" % current_learning_rate)
```

```
    # Validate on the validation dataset
    ######################
    # your code here
    # switch to eval mode
    net.eval()

    ######################

    # this help you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ###################################
            # your code here
            # copy inputs to device
            inputs = inputs.to(device)
            targets = targets.to(device)

            # compute the output and loss
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            val_loss += loss.item()

            # count the number of correctly predicted samples in the current batch
            max_values, predicted = outputs.max(1)
            total_examples += targets.size(0)
            correct_examples += (predicted == targets).sum().item()
            ###################################

    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss, avg_acc))

    # save the model checkpoint
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
        #if not os.path.exists(CHECKPOINT_FOLDER):
        #    os.makedirs(CHECKPOINT_FOLDER)
        #print("Saving ...")
        #state = {'state_dict': net.state_dict(),
        #         'epoch': i,
        #         'lr': current_learning_rate}
        #torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'simplenn.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.4f}")
```

After applying learning rate decay with DECAY_EPOCHS = 10 and DECAY = 0.1, the model's performance has improved. Previously, the best validation accuracy was 0.6564, but after applying the decay, the validation accuracy increased to 0.6978. This shows that learning rate decay helped the model continue improving, allowing for finer adjustments during training and better generalization.

4 Lab (2): Improving the training pipeline (35+6 pts)

In Lab (1), we develop a simplified training pipeline. To obtain better training result, we will improve the training pipeline by employing data augmentation, improving the model design, and tuning the hyperparameters.

Before start, please duplicate the notebook in Lab (1) and name it as simplenn-cifar10-dev.ipynb**, and work on the new notebook**. You goal is to reach at least 70% validation accuracy on the CIFAR-10 dataset.

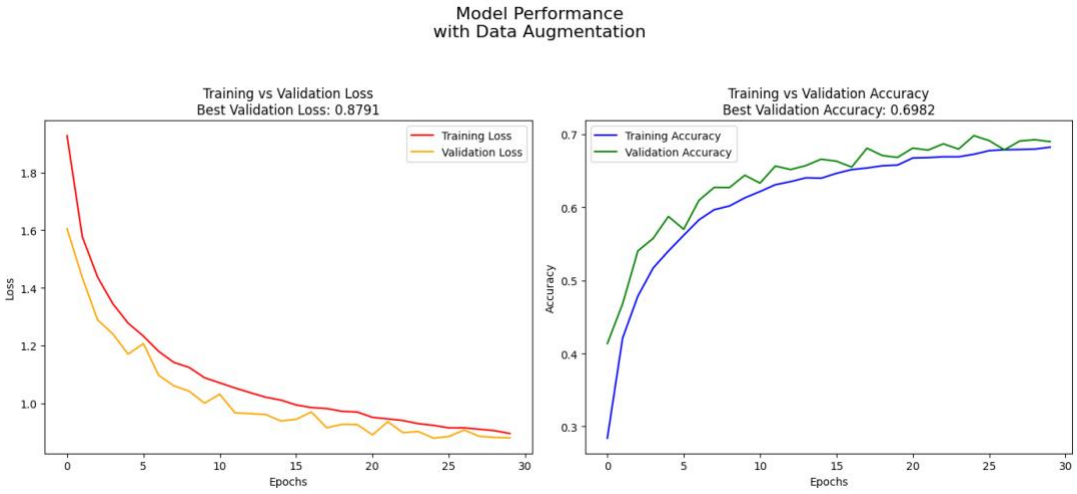
Note that in the following implementations during Lab 2 of my model, I am not using learning rate decay. Instead, I am considering **DECAY = 1** as a baseline model without decay.

(a) (6 pts) Data augmentation techniques help combat overfitting. A typical strategy for CIFAR classifica- tion is to combine 1) *random cropping* with a *padding* of 4 and 2) *random flipping*. Train a model with such augmentation. How is the validation accuracy compared with the one without augmentation? **Note that in the following questions we all use augmentation. Also remember to reinitialize the model whenever you start a new training!**

The code used to perform the data augmentation is as shown in the image below, adding the following lines:
transforms.RandomCrop(32, padding=4)
transforms.RandomHorizontalFlip()

```
[4] # useful libraries
    import torchvision
    import torchvision.transforms as transforms

    ##########################################
    # your code here
    # specify preprocessing function
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.4914, 0.4822, 0.4465), std=(0.2023, 0.1994, 0.2010))])

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

We can see below in the graph the results obtained from the model after applying data augmentation. Our new best validation accuracy is **0.6982, which represents an improvement** over our previous model in Lab 1 (before applying learning rate decay), where the best validation accuracy was 0.6564. Data augmentation helps improve performance by increasing the diversity of the training data, which reduces overfitting and allows the model to generalize better to new data.



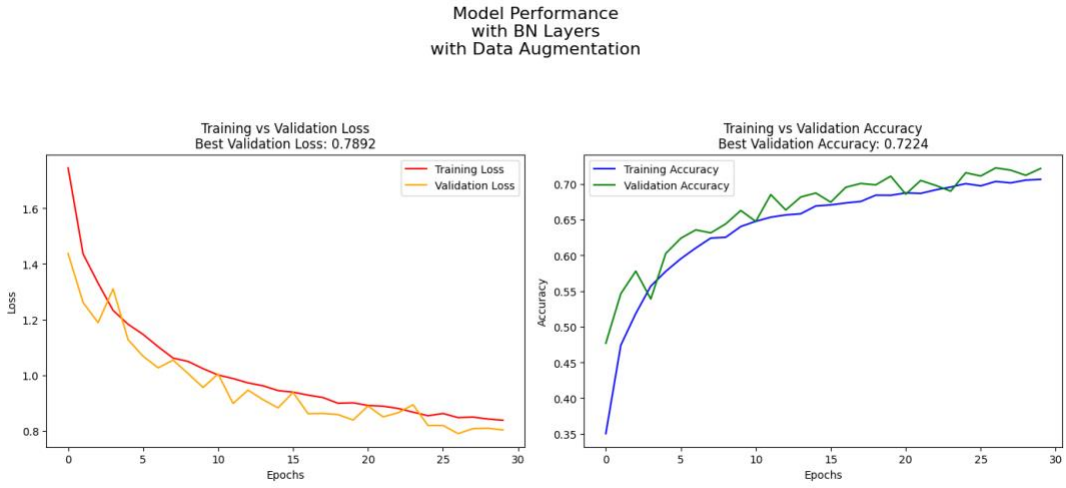
(b) (15 pts) Model design is another important factor in determining performance on a given task. Now, modify the design of SimpleNN as instructed below:

i. (5 pts) Add a batch normalization (BN) layer after each convolution layer. Compared with no BN layers, how does the best validation accuracy change?

I updated the following code inside the SimpleNN class to add a Batch Normalization (BN) layer after each convolution layer

```
# SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.fc1   = nn.Linear(16*6*6, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

After implementing this change, I obtained the following results:



We can see that our best validation accuracy increased from 0.6982 to 0.7224 with the incorporation of Batch Normalization.

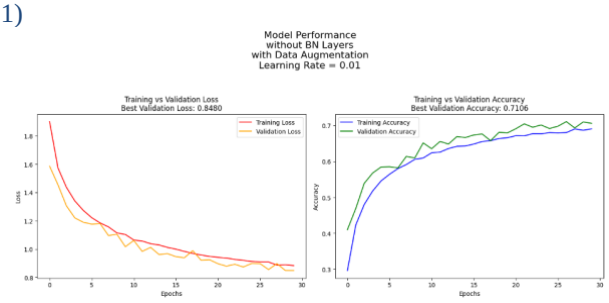
Batch Normalization improves performance by normalizing activations, which accelerates convergence and stabilizes training. It also reduces internal covariate shift, making the model less sensitive to weight initialization and learning rate, resulting in better generalization and higher validation accuracy.

ii. (5 pts) Use empirical results to show that batch normalization allows a larger learning rate.

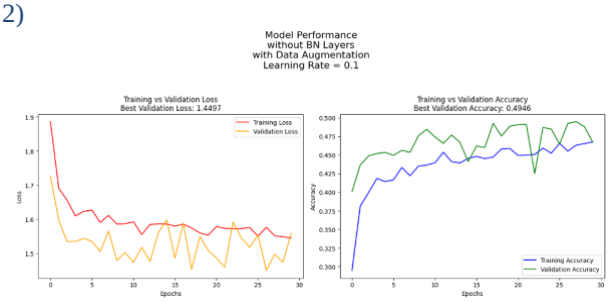
I compared how Batch Normalization (BN) affects the performance of higher learning rates by training four versions of our CNN:

1. CNN without Batch Normalization, learning rate = 0.01
2. CNN without Batch Normalization, learning rate = 0.1
3. CNN with Batch Normalization, learning rate = 0.01
4. CNN with Batch Normalization, learning rate = 0.1

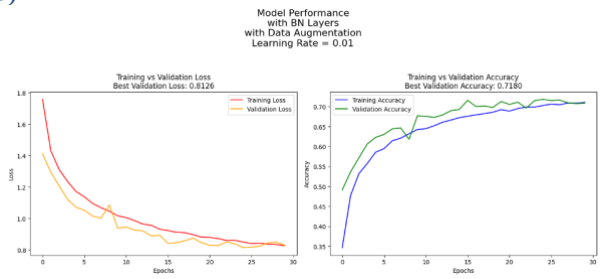
1)



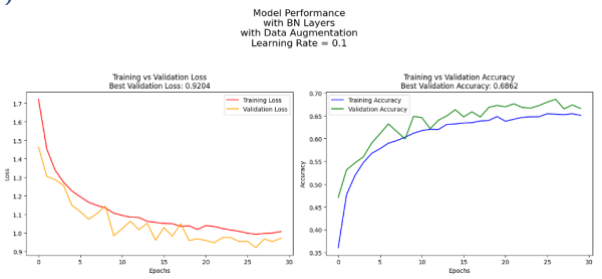
2)



3)



4)



- Model 1 (No BN, LR=0.01) - Best validation accuracy: 0.7106
- Model 2 (No BN, LR=0.1) - Best validation accuracy: 0.4946
- Model 3 (BN, LR=0.01) - Best validation accuracy: 0.7180
- Model 4 (BN, LR=0.1) - Best validation accuracy: 0.6862

We can see that when not using Batch Normalization, increasing the learning rate from 0.01 to 0.1 significantly decreases performance. This is because the model becomes unstable and oscillates more, as observed in the graphs. Specifically, the best validation accuracy drops from 0.7106 in Model 1 (No BN, LR=0.01) to 0.4946 in Model 2 (No BN, LR=0.1).

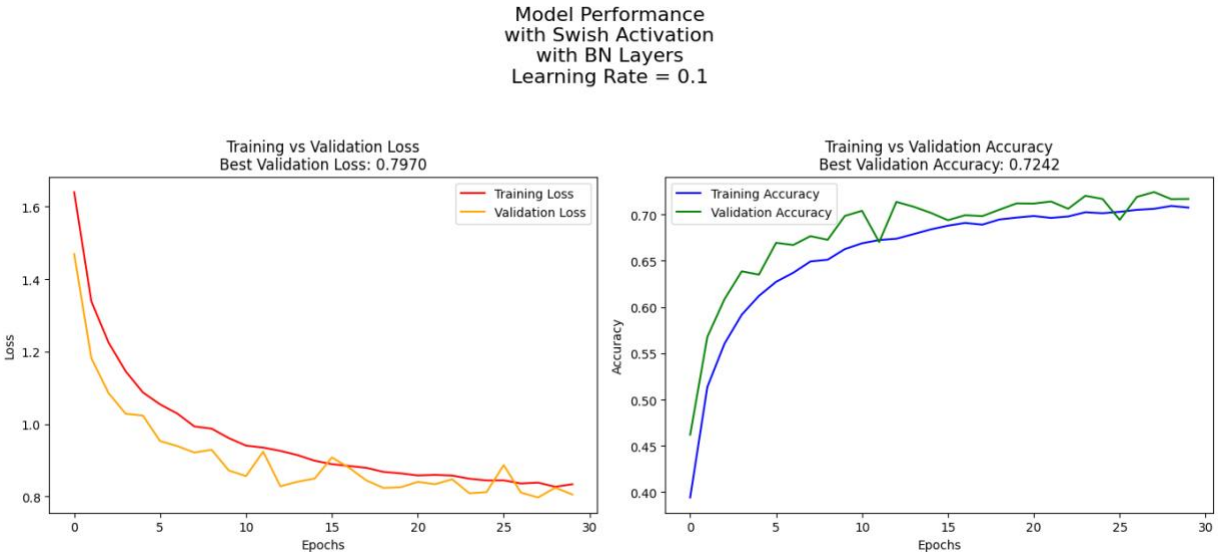
In contrast, when using Batch Normalization, increasing the learning rate from 0.01 to 0.1 does not lead to such a significant drop in performance. The decrease is more moderate, with the best validation accuracy going from 0.7180 in Model 3 (BN, LR=0.01) to 0.6862 in Model 4 (BN, LR=0.1). Additionally, the loss graph appears much more stable compared to the models without Batch Normalization.

iii. (5 pts) Implement Swish [2] activation on you own, and replace all of the ReLU activations in SimpleNN to Swish. Train the model with BN layers and a learning rate of 0.1. Does Swish outperform ReLU?

To implement the Swish activation, I made the following modification to the code:

```
def swish(x):
    return x * torch.sigmoid(x)

# SimpleNN mode;
class SimpleNN_Swish(nn.Module):
    def __init__(self):
        super(SimpleNN_Swish, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.bn1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1   = nn.Linear(16*6*6, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        out = swish(self.bn1(self.conv1(x)))
        out = F.max_pool2d(out, 2)
        out = swish(self.bn2(self.conv2(out)))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = swish(self.fc1(out))
        out = swish(self.fc2(out))
        out = self.fc3(out)
        return out

# Model Definition
net = SimpleNN_Swish()
net = net.to(device)
```

I obtained the following results:



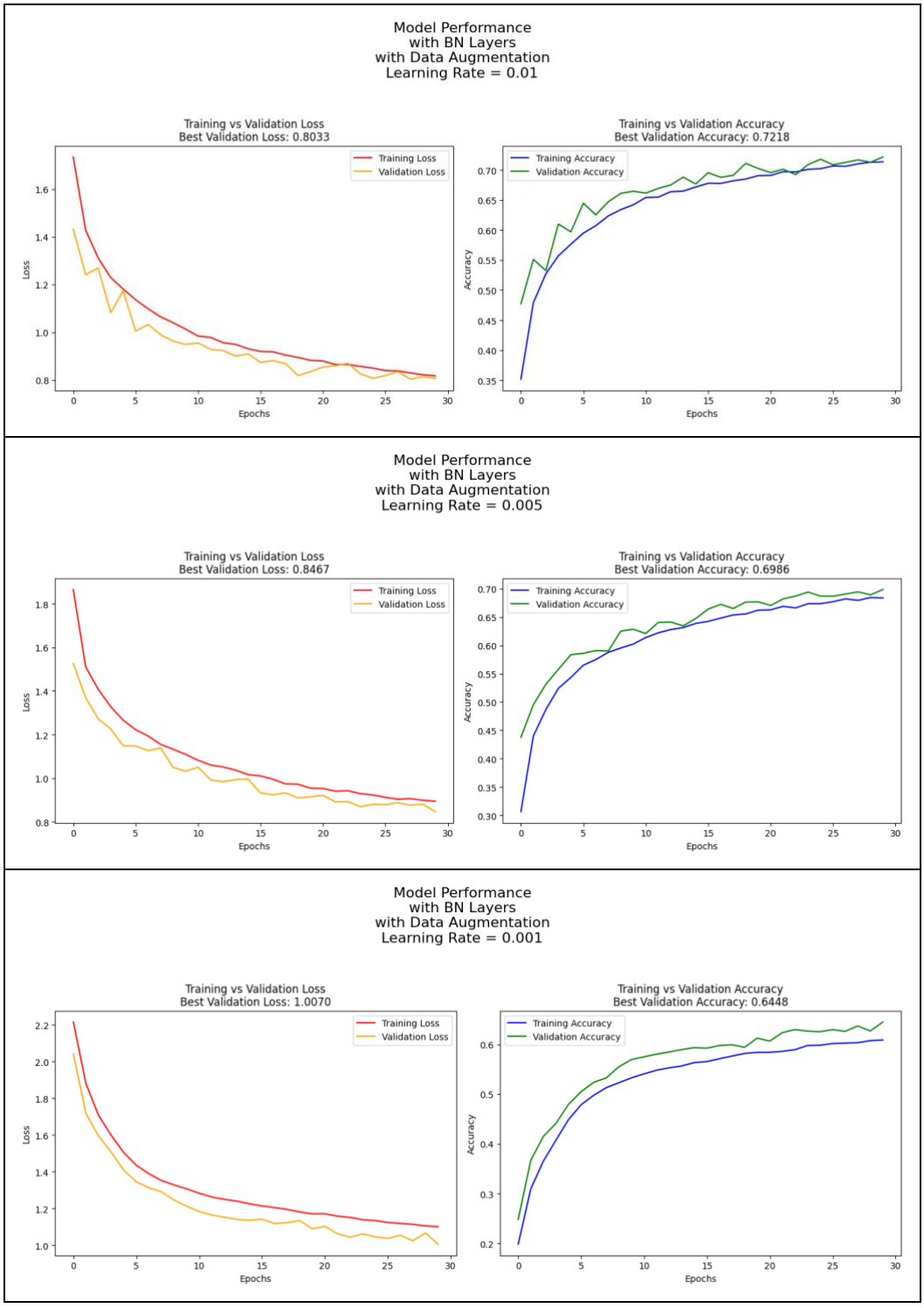
With Swish activation, Batch Normalization, and a learning rate of 0.1, **the best validation accuracy was 0.7242. This represents an improvement** compared to the previous Model 4 (BN, LR=0.1, with ReLU), which achieved a best validation accuracy of **0.6862.**
This improvement can be attributed to the Swish activation function's ability to provide smoother gradients and more effective feature learning compared to ReLU. Swish's self-gated nature helps the network perform better by preventing issues like vanishing gradients and enabling faster and more stable training, especially when combined with Batch Normalization.

(c) (14 pts) Hyperparameter settings are very important and can have a large impact on the final model performance. Based on the improvements that you have made to the training pipeline thus far (with data augmentation and BN layers), tune some of the hyperparameters as instructed below:

Note: Although we achieved better performance with a learning rate of 0.1 and Swish activation, for the next tuning, I will be using a learning rate of 0.01 and ReLU as the base case (without learning rate decay). The focus here is to evaluate how tuning certain hyperparameters affects performance, rather than necessarily aiming for the best accuracy at this stage. I will focus on achieving the best accuracy in Lab 3.

i. (7 pts) Apply different learning rate values: 1.0, 0.1, 0.05, 0.01, 0.005, 0.001, to see how the learning rate affects the model performance, and report results for each. Is a large learning rate beneficial for model training? If not, what can you conclude from the choice of learning rate?

I have the following results:



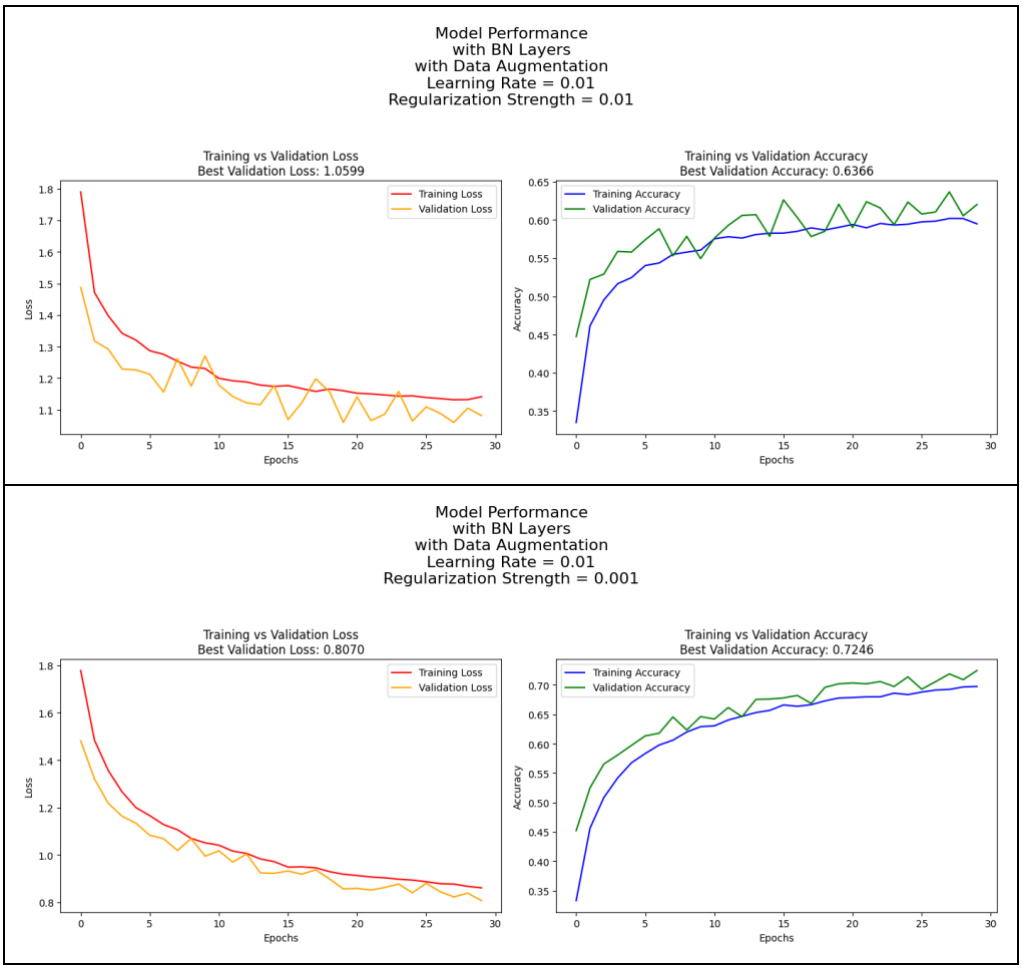
- Model 1 (LR=1.0) - Best validation accuracy: 0.1054
- Model 2 (LR=0.1) - Best validation accuracy: 0.6780
- Model 3 (LR=0.05) - Best validation accuracy: 0.7120
- Model 4 (LR=0.01) - Best validation accuracy: 0.7218
- Model 5 (LR=0.005) - Best validation accuracy: 0.6986
- Model 6 (LR=0.001) - Best validation accuracy: 0.6448

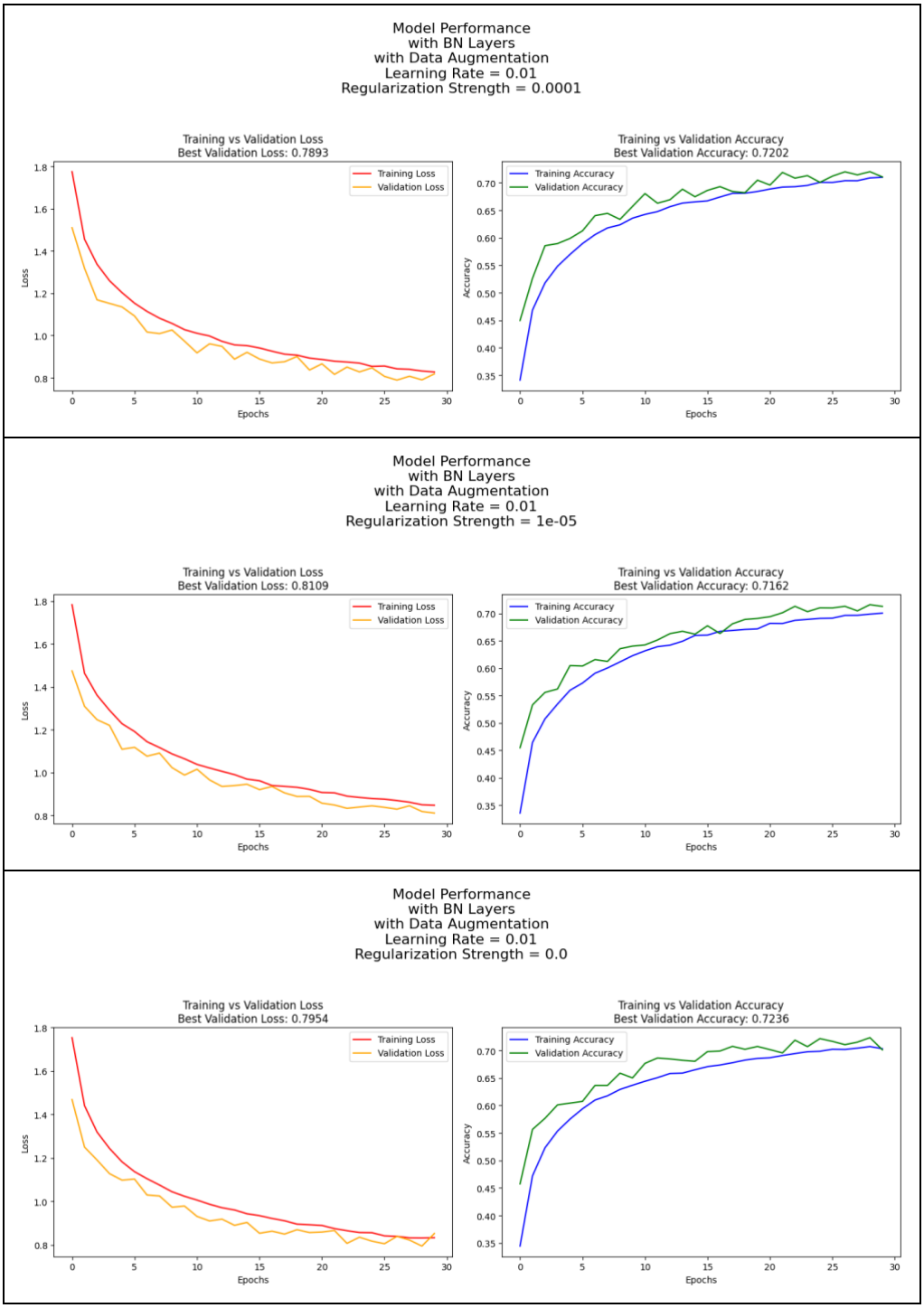
We can see that for very high learning rates, the loss oscillates significantly and the validation accuracy is very low, as seen with LR=1.0, which achieved a best validation accuracy of 0.1054. This happens because with a high learning rate, the model makes large weight updates that cause it to jump around the optimal solution, making it difficult to converge properly during training.

On the other hand, with a very low learning rate, the training process is stable but very slow. As seen in the graphs with lower learning rates, the model hasn't fully converged, and the best validation accuracy decreases. For example, with LR=0.001, the best validation accuracy was only 0.6448. This highlights the importance of finding a balance, as we observed that the best results were achieved with a learning rate of 0.01, which gave a best validation accuracy of 0.7218.

ii. (7 pts) Use different L2 regularization strengths of 1e-2, 1e-3, 1e-4, 1e-5, and 0.0 to see how the L2 regularization strength affects the model performance. In this problem use a learning rate of 0.01. Report the results for each regularization strength value. What did you expect? Are the results what you expected?

I have the following results:





- Model 1 (L2=1e-2) - Best validation accuracy: 0.6366
- Model 2 (L2=1e-3) - Best validation accuracy: 0.7246
- Model 3 (L2=1e-4) - Best validation accuracy: 0.7202
- Model 4 (L2=1e-5) - Best validation accuracy: 0.7162
- Model 5 (L2=0.0) - Best validation accuracy: 0.7236

Before running the experiments, I expected that a moderate value of L2 regularization would lead to the best model performance by preventing overfitting without excessively penalizing the weights. A very high regularization strength might lead to underfitting, while no regularization could result in overfitting.
The results generally aligned with these expectations. The model with the highest regularization strength (L2=1e-2) showed the lowest best validation accuracy (0.6366), indicating underfitting. The best performance was achieved with L2=1e-3, which resulted in a validation accuracy of 0.7246, supporting the idea that moderate regularization helps with generalization.

iii. (Bonus, 6 pts) Switch the regularization penalty from L2 penalty to L1 penalty and train with the default hyperparameters. *Hint: This means you may not use the* weight_decay *parameter in PyTorch builtin optimizers, as it does not support L1 regularization. Instead, you need to add L1 penalty as a part of the loss function.* Compare the distribution of weight parameters after L1/L2 regularization. Describe your observations, are they what you expected? Why or why not?

Up to now, you shall have an improved training pipeline for CIFAR-10. Remember, you are required to submit simplenn-cifar10-dev.ipynb for Lab (2).

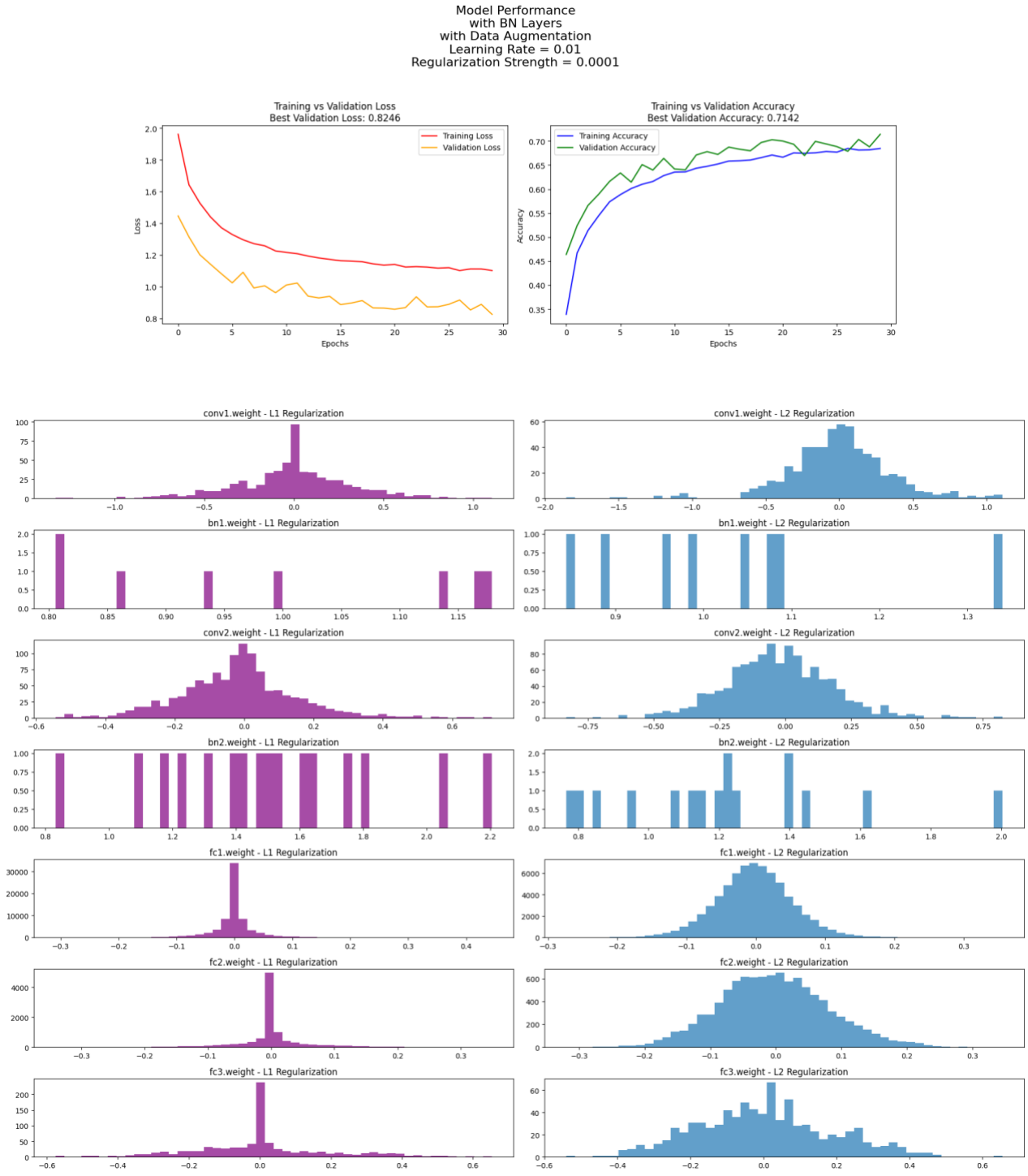
I made this changes to the code

```
# Model Definition
net = SimpleNN()
net = net.to(device)

# initial learning rate
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4
L1_lambda = REG  # we are using same number

# loss function
criterion = nn.CrossEntropyLoss()

# optimizer
optimizer = optim.SGD(
    net.parameters(),
    lr=INITIAL_LR,
    momentum=MOMENTUM,
    #weight_decay=REG. --> USING L1 instead of L2
)
```

…..

```
# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR

print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    ######################
    # your code here
    # switch to train mode
    net.train()
    ######################

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        ###################################
        # your code here
        # copy inputs to device
        inputs = inputs.to(device)
        targets = targets.to(device)

        # compute the output and loss
        outputs = net(inputs)
        loss = criterion(outputs, targets)

        # L1 Regularization

        L1_penalty = sum(torch.sum(torch.abs(param)) for param in net.parameters())

        loss += L1_lambda * L1_penalty
```

I have the following results:



Comparing the weight distributions after applying L1 and L2 regularization, we observe that L1 leads to many weights being exactly zero or close to zero, indicating sparsity, which aligns with the theory that L1 encourages a more compact model by forcing some weights to zero and selecting relevant features. In contrast, L2 results in weights following a more Gaussian distribution around zero without being exactly zero, reflecting the expected behavior of L2 to reduce weight magnitudes without inducing sparsity.

5 Lab (3): Advanced CNN architectures (20 pts)

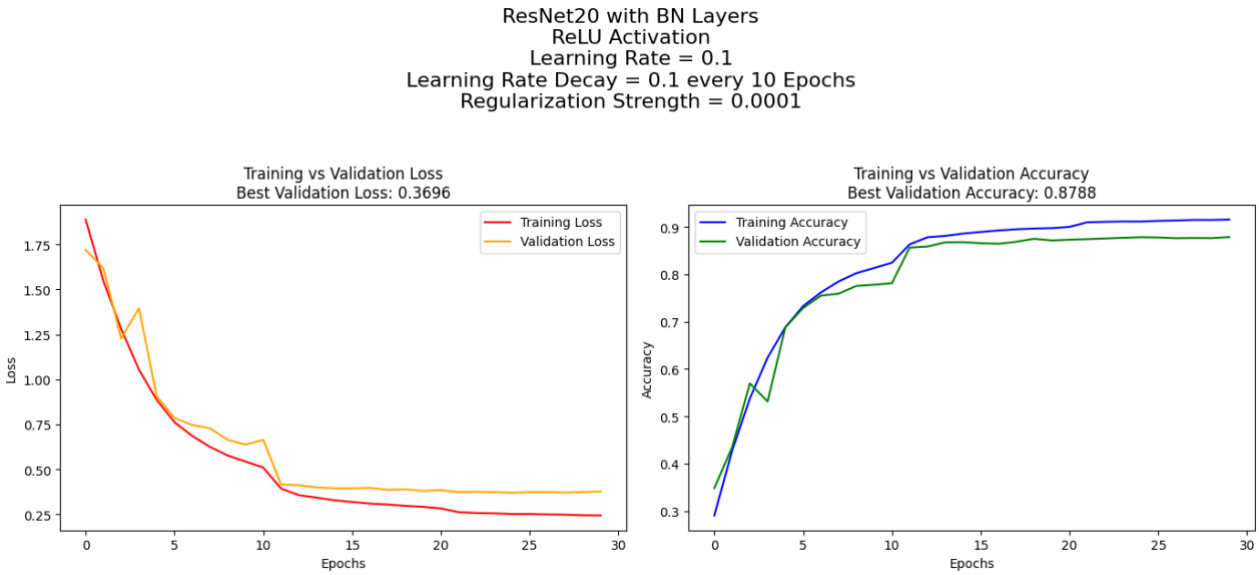
The improved training pipeline for SimpleNN developed in Lab (2) still has limited performance. This is mainly because the SimpleNN has a rather small capacity (learning capability) for the CIFAR-10 task. Thus, in this lab, we replace the SimpleNN model with a more advanced ResNet [3] architecture. We expect to see much higher accuracy on CIFAR-10 when using ResNets. **Here, you may duplicate your jupyter notebook for Lab (2) as** resnet-cifar10.ipynb **to serve as a starting point.**

(a) (8 pts) Implement the ResNet-20 architecture by following Section 4.2 of the ResNet paper [3]. This lab is designed to have you learn how to implement a DNN model yourself, **so do NOT borrow any code from online resource.**

I made these changes to the code

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out

class ResNet20(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet20, self).__init__()
        self.in_channels = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)

        self.layer1 = self._make_layer(ResidualBlock, 16, 6, stride=1)
        self.layer2 = self._make_layer(ResidualBlock, 32, 6, stride=2)
        self.layer3 = self._make_layer(ResidualBlock, 64, 6, stride=2)

        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1)) # "The network endswith a global average pooling"
        self.fc = nn.Linear(64, num_classes)

    def _make_layer(self, block, out_channels, blocks, stride):
        layers = []
        layers.append(block(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

I have the following results:



In the ResNet-20 implementation, I achieved a best accuracy of 0.8788, which was better than 0.7112, the best accuracy I had obtained previously with SimpleNN using the same hyperparameters.

(b) (12 pts) Tune your ResNet-20 model to reach an accuracy of higher than 90% on the validation dataset. You may use all of the previous techniques that you have learned so far, including data augmentations, hyperparameter tuning, learning rate decay, etc. Training the model longer is also essential to obtaining good performance. You should be able to achieve >90% validation accuracy with a maximum of 200 epochs. **Remember to save your trained model during the training!!!** Check out this tutorial https://pytorch.org/tutorials/beginner/saving_loading_models.html on model saving/loading.

Note: We will grade this task by evaluating your trained model on the holdout testing dataset (which you do not have any labels). **After your ResNet-20 model is trained, you need to make predictions on test data, and save the predictions into the** predictions.csv **file**. Please utilize the given notebook, save_test_predictions.ipynb, to save your predictions in required format. The saved file should look like the provided example sample_predictions.csv. Upon submission, we will directly compare your predicted labels with the ground-truth labels to compute your score.

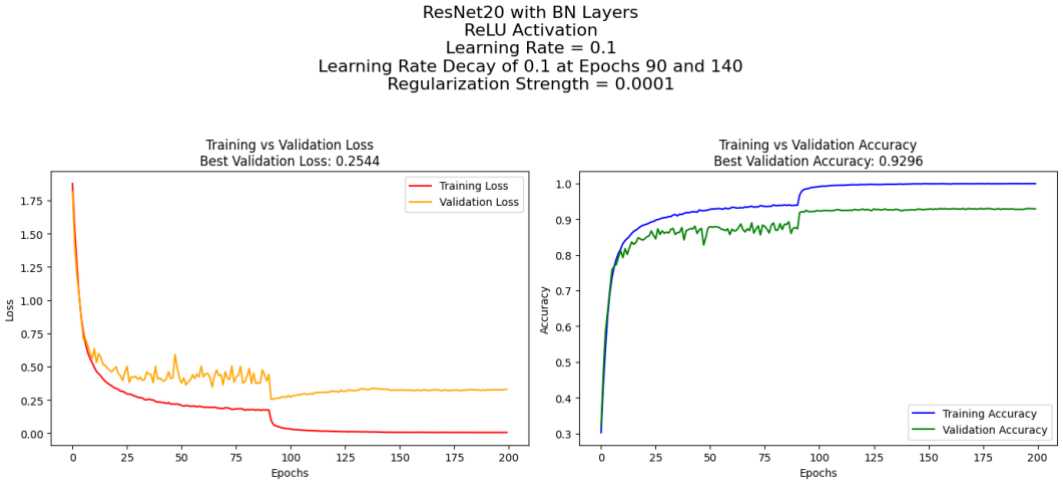
After completing Lab (3), you are required to submit resnet-cifar10.ipynb and your prediction results predictions.csv.

In this task, I've used a combination of techniques to fine-tune the ResNet-20 model, drawing on those that have proven effective in previous iterations. I've also taken into account the design and hyperparameters from the ResNet paper, as their results demonstrated strong performance, which made it logical to apply them here. The configuration that ultimately achieved the best validation accuracy was:

- **Data Augmentation**: Random cropping with a padding of 4 and random flipping to increase the diversity of the training data and improve generalization.
- **Activation Function**: ReLU activations after each convolutional layer, which are commonly used to improve training efficiency and model performance.
- **Learning Rate Schedule**: Starting with an initial learning rate of 0.1, followed by decaying it by a factor of 10 at epochs 90 and 140. This schedule is based on the ResNet paper, which divides the learning rate at iterations 32k and 48k, corresponding roughly to epochs 90 (32k*128/45k) and 140 (48k*128/45k) in this case, considering the training batch size and the number of iterations per epoch. This approach helps the model converge faster during the initial stages of training and refine its performance in the later stages.
- **Training Duration**: I also extended the training duration to 200 epochs, which allowed the model more time to improve its performance and achieve a higher best validation accuracy. •
- **Batch Normalization**: Adding batch normalization layers after each convolutional layer to stabilize training and allow for higher learning rates.

- **L2 Regularization**: Implementing L2 regularization with a strength of 1e-4 to prevent overfitting by penalizing large weights.

In the accuracy plot we observe a steady increase in both training (blue) and validation (green) accuracy during the initial epochs. At epoch 90, when the learning rate decreases from 0.1 to 0.01, there is a noticeable jump in accuracy, followed by a period of stabilization. A similar pattern is observed at epoch 140, when the learning rate further decreases. After these adjustments, the validation accuracy remains relatively stable, ultimately reaching a best value of **92.96%**, surpassing the target of 90%.



Info: Additional requirements:

- **DO NOT** train on the test set or use pretrained models to get unfair advantage. We have con- ducted a special preprocessing on the original CIFAR-10 dataset. As we have tested, "cheating" on the full dataset will give only 6% accuracy on our final test set, which means being unsuc- cessful in this assignment.
- **DO NOT** copy code directly online or from other classmates. We will check it! The result can be severe if your codes fail to pass our check.

Info: As this assignment requires significant computing resources (GPUs), we suggest:

- Plan your work in advance and start early. We will **NOT** extend the deadline because of the unavailability of computing resources.
- Be considerate and kill Jupyter Notebook instances when you do not need them.
- **DO NOT** run your program forever. Please follow the recommended/maximum training budget in each lab.

References

[1] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[2] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.