ECE 661 Homework-2

True/False Questions:

1. **False.** Batch Normalisation is when the batch mean is subtracted from the batch inputs and is divided by the batch standard deviation. The output of the batch normalisation module has approximately zero mean not exactly zero.
2. **True.** Pytorch provides an efficient way of tensor computation and many modularized implementations of layers. You do not need to write your own code for standard backpropagation algorithms like Adam when using PyTorch. Instead, you can simply instantiate an optimizer object with your model parameters.
3. **False.** Data augmentation introduces variations in the training data by applying transformations like cropping and flips, this allows for generalizability as it introduces the model to wider range or data patterns and orientations. However, for smaller models, it may cause the model to underfit, meaning it struggles to capture the underlying patterns in the data.
4. **False.** Without batch normalisation and dropout, it is not true that the CNN will coverage slowly. It depends on various factors like the architecture of the model, quality of the data and the weights. Batch normalization normalizes the intermediate outputs of each layer within a batch during training, making the optimization process more stable and faster. Dropout helps prevent overfitting, but if it is not used it is not necessary that the coverage will be slow.
5. **False.** Both are regularisation techniques so if you combine it may cause underfitting, if we need to use both dropout as well as L regularisation then we need to reduce the strength of either one.
6. **True.** L1 regularization has a penalty term that is proportional to the absolute values of the weights, this forces the weights to move towards zero. L2 regularization has a penalty term that is proportional to the of the squared values of the weights, this does not force the weights to move towards zero. Hence L1 has a higher sparsity.
7. **True.** While Leaky ReLU addresses the problem of dead neurons seen in the standard ReLU, it can introduce training instability. By introducing non-zero gradients for negative input values, it can complicate optimization and lead to slower convergence during the training process.
8. **True.** Depthwise separable convolution can help reduce the number of parameters and computation. Since ResNet has a higher number of channels, it does speed up.
9. **False.** SqueezeNet stacks more convolutional layers in the early stages of the CNN architecture. Down sampling happens at the end of the architecture.
10. **True.** The skip connections allow the gradient to flow easily during training as it skips one or more layers during the training.

LAB (1):

Question (a):

For output shape check:

A dummy input is generated with batch size 1, 3 channels and an image size of 32 x 32. The input is then passed to the SimpleNN model resulting in an output, the expected output shape is (1,10). The code checks if the expected output shape is the same as the output shape gotten from the model. If it's the same the output shape check passed.

For Parameter count check:

count_parameters is a function to count the parameters of the SimpleNN model. The parameters include the weights and the bias. The expected parameters are manually calculated based on the architecture specified on the homework.

Name	Input channels	Kernel size	depth	Bias	Parameter count
Conv1	3	5x5	8	8	3*5*5*8 + 8
Conv2	8	3x3	16	16	8*16*3*3 +16
FC1	16	NA	120	120	16*6*6*120 +120
FC2	120	NA	84	84	120*84 +84
FC3	84	NA	10	10	84 *10 +10

If the total parameters calculated by the model is the same as the parameter count calculated manually then the parameter count check will pass.

```
dummy_input = torch.randn(1, 3, 32, 32)
model = SimpleNN()
output = model(dummy_input)


expected_output_shape = (1, 10)
if output.shape == expected_output_shape:
    print("Output shape check passed!")
else:
    print("Output shape check failed.")

def count_parameters(model):
    return sum(p.numel() for p in model.parameters())

total_parameters = count_parameters(model)

expected_parameters = (3 * 8 * 5 * 5 + 8) + (8 * 16 * 3 * 3 + 16) + (16 * 6 * 6 * 120 + 120) + (120 * 84 + 84) + (84 * 10 + 10)

if total_parameters == expected_parameters:
    print("Parameter count check passed!")
else:
    print("Parameter count check failed.")

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

```
Output shape check passed!
Parameter count check passed!
```

Question (b):

```python
import torchvision
import torchvision.transforms as transforms

################################################
# your code here
mean = (0.4914, 0.4822, 0.4465)
std = (0.2023, 0.1994, 0.2010)
# specify preprocessing function

transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

When the data features are on different scales it becomes difficult to get to the optimal solution. We use normalisation to bring all the features to a similar scale for easier training. Normalisation is done by subtracting the mean from each feature which ensures that the data is around zero, this removes any bias that might be there, dividing by the standard deviation scales the data and ensures that the data has a unit variance.

Question (c):

This step is for building the actual training and validation datasets and dataloaders, for this homework we are using our own CIFAR-10 dataset class, which is imported from tools.dataset.

```python
# 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    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val    # your code
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=torch.triu_indices,     # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
################################################
```

```
Downloading https://www.dropbox.com/s/s8orza214q45b23/cifar10_trainval_F22.zip?dl=1 to ./data/cifar10_trainval_F22.zip
141746176it [00:03, 40203541.40it/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
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:560: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker
  warnings.warn(_create_warning_msg(
```

Question (d):

Instantiate and deploy the SimpleNN model on GPUs

```
!nvidia-smi
```

```
Tue Oct  3 21:53:06 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P8    10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

```
[11] # specify the device for computation
     ########################################
     # your code here
     if torch.cuda.is_available():
         device = torch.device("cuda")
         print("Using GPU")
     else:
         device = torch.device("cpu")
         print("Using CPU")

     model = SimpleNN()
     model.to(device)
     print(model)


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

```
Using GPU
SimpleNN(
  (conv1): Conv2d(3, 8, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Question (e):

Setting up cross entropy loss as the criterion, this calculates the cross-entropy loss between the predicted class probabilities and the true class labels. Here we are using the stochastic gradient decent as the optimization algorithm, the optimizer is used to adjust the model parameters to minimize the loss.

```
# 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(model.parameters(), lr=INITIAL_LR, momentum=MOMENTUM, weight_decay=REG)
############################################
```

Question (f):

Setting up the training process of SimpleNN on the CIFAR-10 dataset

(i) Switching the model to train mode.

```
for i in range(0, EPOCHS):
    ######################
    # your code here
    # switch to train mode
    model.train()


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

    print("Epoch %d:" %i)
```

(ii) Train the model

Copying the inputs and the targets to the device to make sure computations are performed on the selected device. The inputs are passed through the neural network model to obtain predictions. The cross-entropy loss is calculated between the predicted values and the actual target labels.

The gradient is zeroed out from previous iterations. The gradients of the loss are calculated using backpropagation this is with respect to the model's parameters. The optimizer adjusts the weights and biases of the model to reduce loss. Finally, the number of correctly predicted samples in the current batch is calculated.

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

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

     # zero the gradient

     optimizer.zero_grad()
     # backpropagation
     loss.backward()


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


    # count the number of correctly predicted samples in the current batch
     _, predicted = output.max(1)
     total_examples += targets.size(0)
     correct_examples += predicted.eq(targets).sum().item()
     val_loss += loss.item()
```

(iii) Validate on the validation dataset.

Switching to eval mode, and doing the steps as mentioned above while training.

```
# your code here
# switch to eval mode
model.eval()


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

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

Question (g):

(i) Initial loss

The initial loss is calculated on the first batch of data, it represents how well or poorly the model is performing on this initial batch before any optimization or parameter updates have occurred.

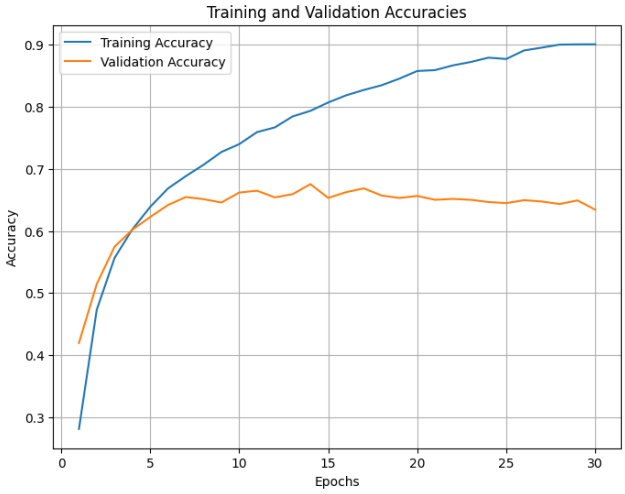
The initial loss calculated: 2.998

```
model1 = SimpleNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer =optim.SGD(model1.parameters(), lr=INITIAL_LR, momentum=MOMENTUM, weight_decay=REG)
model1.train()

batch_idx, (inputs, targets) = next(enumerate(train_loader))
inputs, targets = inputs.to(device), targets.to(device)

output = model1(inputs)
loss = criterion(output, targets)

print(f"Initial Loss for First Batch: {loss.item():.4f}")
```

```
Initial Loss for First Batch: 2.2998
```

(ii) Training and validation accuracies

From the below graph we can observe that the training and validation accuracies initially are increasing, the training accuracies continue to increase but the validation accuracies are around 0.60. The initial increase in both training and validation accuracies indicates that the model is learning the patterns in the data very well. This is a positive indication.



LAB 2:

Question(a):

Adding data augmentation techniques like random cropping with a padding of 4 and random flipping. Data augmentation techniques are used to help combat overfitting.

```
import torchvision
import torchvision.transforms as transforms

########################################
# your code here
mean = (0.4914, 0.4822, 0.4465)
std = (0.2023, 0.1994, 0.2010)
# specify preprocessing function

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
########################################
```

Validation Accuracy without Data Augmentation:

```
==================================================
==> Optimization finished! Best validation accuracy: 0.6754
```

Validation Accuracy with Data Augmentation:

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7030
```

By comparing the results of validation accuracy, we can see that the model performs slightly better with data augmentation as compared to when the model without data augmentation. When the data features are on different scales it becomes difficult to get to the optimal solution. We use normalisation to bring all the features to a similar scale for easier training.

Data augmentation introduces variations in the training data by applying transformations like cropping and flips, this allows for generalizability as it introduces the model to wider range or data patterns and orientations.

Question(b):

(i) Adding Batch Normalisation:

```
# define the SimpleNN mode with Batch Normalization
class SimpleNNWithBN(nn.Module):
    def __init__(self):
        super(SimpleNNWithBN, 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 = F.relu(self.bn1(self.conv1(x)))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.bn2(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
```

Validation Accuracy without Batch Normalisation:

```
    ================================================
    ==> Optimization finished! Best validation accuracy: 0.7030
```

Validation Accuracy with Batch Normalisation:

```
================================================
==> Optimization finished! Best validation accuracy: 0.7170
```

By comparing the results of validation accuracy without batch normalisation after the convolutional layers and with batch normalisation we can observe that the validation frequency is slightly greater when batch normalisation is added to the model.

(ii) Empirical results to show that batch normalization allows a larger learning rate

Validation Accuracy without Batch Normalisation and a higher learning rate:

```
================================================
==> Optimization finished! Best validation accuracy: 0.4996
```

Validation Accuracy with Batch Normalisation and a higher learning rate:

```
================================================
==> Optimization finished! Best validation accuracy: 0.6628
```

The learning rate in this scenario is set to 0.1, which is higher than the learning rate used in the previous models. When we compare the validation accuracies of two models, one without batch normalization and a learning rate of 0.1, and the other with batch normalization and the same learning rate of 0.1, we observe that the latter achieves a significantly higher validation accuracy, specifically a 16.32% improvement.

(iii) Implementing Swish Activation

```
class Swish(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)


class SimpleNNWithBNSwish(nn.Module):
    def __init__(self):
        super(SimpleNNWithBNSwish, 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)
        self.swish = Swish()

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

Validation Accuracy with ReLu Activation Function:

```
================================================
==> Optimization finished! Best validation accuracy: 0.6628
```

Validation Accuracy with Swish Activation Function:

```
================================================
==> Optimization finished! Best validation accuracy: 0.7268
```

Swish introduces a non-linearity that is more flexible than ReLu, this flexibility allows the neural network to learn complex functions and capture patterns in the data. Swish tends to have nonzero gradients for a wide range of inputs. We can observe that the Validation accuracy is greater when using the Swish Activation Function as compared to the ReLu Activation Function.

Question(c):

(i) Apply different learning rate values:

Validation Accuracy with Learning Rate =1.0

```
==================================================
==> Optimization finished! Best validation accuracy: 0.1028
```

Validation Accuracy with Learning Rate= 0.1

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7076
```

Validation Accuracy with Learning Rate= 0.05

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7322
```

Validation Accuracy with Learning Rate= 0.01

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7322
```

Validation Accuracy with Learning Rate= 0.005

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7322
```

Validation Accuracy with Learning Rate= 0.001

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7322
```

When assessing the performance of a model trained with various learning rates, a notable finding is that using an excessively high learning rate, such as a value of 1.0, tends to result in a substantial drop in validation accuracy. When the learning rate is too high, the model's parameter updates can overshoot the optimal values, causing it to oscillate or diverge rather than converge. This instability can prevent the model from effectively learning and generalizing from the data, resulting in poor performance. A higher learning rate can make the data more sensitive to noise, leading to overfitting.

(ii) Applying different L2 regularisation strengths

Validation Accuracy with L2 regularisation strength=0.01

```
==================================================
==> Optimization finished! Best validation accuracy: 0.5998
```

Validation Accuracy with L2 regularisation strength=0.001

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7262
```

Validation Accuracy with L2 regularisation strength=0.0001

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7334
```

Validation Accuracy with L2 regularisation strength=1e-05

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7334
```

Validation Accuracy with L2 regularisation strength=0.0

```
==================================================
==> Optimization finished! Best validation accuracy: 0.7334
```

Higher regularisation values can lead to underfitting where the model cannot learn the complex data patterns resulting in a poor performance on both the training and validation datasets. On the other hand, very low or no regularisation can allow the model to be overly complex. It also makes the model prone to fitting noise. This can lead to overfitting.

LAB 3:

Question(a):

Resnet Architecture

```
ResNet(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer1): Sequential(
    (0): ResidulBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (1): ResidulBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): ResidulBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
```

```
(layer2): Sequential(
  (0): ResidulBlock(
    (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential(
      (0): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): ResidulBlock(
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
  (2): ResidulBlock(
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
```

```
)
(layer3): Sequential(
  (0): ResidulBlock(
    (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential(
      (0): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): ResidulBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
  (2): ResidulBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
)
(linear): Linear(in_features=64, out_features=10, bias=True)
```

Question(b):

Validation Accuracy: 0.9070

```
Epoch 197:
Training loss: 0.0442, Training accuracy: 0.9846
Validation loss: 0.4939, Validation accuracy: 0.8956

Epoch 198:
Training loss: 0.0460, Training accuracy: 0.9834
Validation loss: 0.4938, Validation accuracy: 0.8890

Epoch 199:
Training loss: 0.0425, Training accuracy: 0.9854
Validation loss: 0.4717, Validation accuracy: 0.9006

================================================
==> Optimization finished! Best validation accuracy: 0.9070
```