CSCE 636 Deep Learning Homework Code Report II

6. (90 points)(Coding Task) Deep Residual Networks for CIFAR-10 Image Classification: In this assignment, you will implement advanced convolutional neural networks on CIFAR-10 using PyTorch. In this classification task, models will take a 32 × 32 image with RGB channels as inputs and classify the image into one of ten pre-defined classes. The “ResNet” folder provides the starting code. You must implement the model using the starting code. In this assignment, you must use a GPU. Requirements: Python 3.8, PyTorch 1.7.0 (Make sure you use this particular version of installation and documentation!!!), tqdm, numpy. Please do not use torchvision.dataset and torchvision.transforms in the data pre-processing part for this homework. Please submit running report (briefly explain how you complete those functions, capture screen shot of your training loss, validation/test acc etc., anything required in the original assignment) for all coding tasks. And paste training and testing console record as an appendix in your report. You don’t have to submit the pre-trained model and the dataset which might be potentially large.

(a) (10 points) Download the CIFAR-10 dataset (https://www.cs.toronto.edu/~kriz/ cifar.html) and complete “DataReader.py”. For the dataset, you can download any version. But make sure you write corresponding code in “DataReader.py” to read it.
The following code is used in DataReader.py to read the training as well as testing data

```
import os
import pickle
import numpy as np

""" This script implements the functions for reading data.
"""

def load_data(data_dir):
    """ Load the CIFAR-10 dataset.

    Args:
        data_dir: A string. The directory where data batches are stored.

    Returns:
        x_train: An numpy array of shape [50000, 3072].
        (dtype=np.float32)
        y_train: An numpy array of shape [50000,].
        (dtype=np.int32)
        x_test: An numpy array of shape [10000, 3072].
        (dtype=np.float32)
        y_test: An numpy array of shape [10000,].
        (dtype=np.int32)
    """
    ### YOUR CODE HERE
    y_train = []
    x_train  = None
    for i in range(1,6):
        with open(os.path.join(data_dir,f'data_batch_{i}'), 'rb') as fo:
```

```
            data = pickle.load(fo, encoding='bytes')
            y_train.extend(data[b'labels'])
            if i!=1:
                x_train = np.vstack((x_train,data[b'data'].astype(np.float32)))
            else:
                x_train = data[b'data'].astype(np.float32)
    y_train = np.array(y_train)
    with open(os.path.join(data_dir,'test_batch'), 'rb') as fo:
        data = pickle.load(fo, encoding='bytes')
        x_test = data[b'data'].astype(np.float32)
        y_test = np.array(data[b'labels'])

    ### YOUR CODE HERE

    return x_train, y_train, x_test, y_test

def train_vaild_split(x_train, y_train, split_index=45000):
    """ Split the original training data into a new training dataset
        and a validation dataset.

    Args:
        x_train: An array of shape [50000, 3072].
        y_train: An array of shape [50000,].
        split_index: An integer.

    Returns:
        x_train_new: An array of shape [split_index, 3072].
        y_train_new: An array of shape [split_index,].
        x_valid: An array of shape [50000-split_index, 3072].
        y_valid: An array of shape [50000-split_index,].
    """
    x_train_new = x_train[:split_index]
    y_train_new = y_train[:split_index]
    x_valid = x_train[split_index:]
    y_valid = y_train[split_index:]

    return x_train_new, y_train_new, x_valid, y_valid
```

(b) (10 points) Implement data augmentation. To complete “ImageUtils.py”, you will implement the augmentation process for a single image using numpy.
For data augmentation, image is padded , 4 pixels on each side, and a random cropping is done such that the cropped image is same size as the original, and also random flipping(in horizontal direction) of the image is incorporated. In addition the image is normalized.

```
def preprocess_image(image, training):
    """ Preprocess a single image of shape [height, width, depth].

    Args:
        image: An array of shape [32, 32, 3].
```

```
        training: A boolean. Determine whether it is in training mode.

    Returns:
        image: An array of shape [32, 32, 3].
    """
    if training:
        ### YOUR CODE HERE
        # Resize the image to add four extra pixels on each side.

        image = np.pad(image, pad_width=((4, 4), (4, 4), (0, 0)),
mode='symmetric')

        ### YOUR CODE HERE
        # assuming the size of the image as [40,40,3], is should randomly
select one point at
        # upper left corner b/w[0-7,0-7]
        ### YOUR CODE HERE
        # Randomly crop a [32, 32] section of the image.
        # HINT: randomly generate the upper left point of the image
        a1 = np.random.randint(0, 8)
        a2 = np.random.randint(0, 8)
        image = image[a1:a1 + 32, a2:a2 + 32, :]

        ### YOUR CODE HERE

        ### YOUR CODE HERE
        # Randomly flip the image horizontally.
        horzontal = np.random.randint(0, 2)
        if horzontal == 0:
            image = np.flip(image, axis=1)

        ### YOUR CODE HERE

    ### YOUR CODE HERE
    # Subtract off the mean and divide by the standard deviation of the
pixels.
    _mean = np.mean(image)
    _std = np.std(image)

    #image = (image - _mean) / _std
    image = (image - np.mean(image, axis=(0, 1))) / np.std(image,
```

```
axis=(0, 1))
    ### YOUR CODE HERE

    return image
```

(c) (30 points) Complete “NetWork.py”. Read the required materials carefully before this step. You are asked to implement two versions of ResNet: version 1 uses original residual blocks (Figure4(a) in [2]) and version 2 uses full pre-activation residual blocks (Figure4(e) in [2]). In particular, for version 2, implement the bottleneck blocks instead of standard residual blocks. In this step, only basic PyTorch APIs in torch.nn and torch.optim are allowed to use.

The batchnormalisation_relu_layer, the residual block, the bottleneck blocks, the start layer, output layer are defined as different class objects in the code below and are used as the building blocks to construct the Resnet architecture. The class stack_layer is used to stack several blocks together and apply it on the incoming input under different conditions

```
import torch
from torch.functional import Tensor
import torch.nn as nn

""" This script defines the network.
"""

class ResNet(nn.Module):
    def __init__(self,
                 resnet_version,
                 resnet_size,
                 num_classes,
                 first_num_filters,
            ):
            """
            1. Define hyperparameters.
            Args:
                resnet_version: 1 or 2, If 2, use the bottleneck blocks.
                resnet_size: A positive integer (n).
                num_classes: A positive integer. Define the number of
classes.
                first_num_filters: An integer. The number of filters to use
for the
                    first block layer of the model. This number is then
doubled
```

```
                for each subsampling block layer.

        2. Classify a batch of input images.

        Architecture (first_num_filters = 16):
        layer_name      | start | stack1 | stack2 | stack3 | output
|
        output_map_size | 32x32 | 32X32  | 16x16  | 8x8    | 1x1
|
        #layers         | 1     | 2n/3n  | 2n/3n  | 2n/3n  | 1
|
        #filters        | 16    | 16(*4) | 32(*4) | 64(*4) | num_classes
|

        n = #residual_blocks in each stack layer = self.resnet_size
        The standard_block has 2 layers each.
        The bottleneck_block has 3 layers each.

        Example of replacing:
        standard_block      conv3-16 + conv3-16
        bottleneck_block    conv1-16 + conv3-16 + conv1-64

        Args:
            inputs: A Tensor representing a batch of input images.

        Returns:
            A logits Tensor of shape [<batch_size>, self.num_classes].
        """
        super(ResNet, self).__init__()
        self.resnet_version = resnet_version
        self.resnet_size = resnet_size
        self.num_classes = num_classes
        self.first_num_filters = first_num_filters

        ### YOUR CODE HERE
        # define conv1
        self.conv1 = nn.Conv2d(in_channels=3,
                               out_channels=self.first_num_filters, # first
num filters?
                               kernel_size=3,
                               stride=1,
```

```
                        padding=1,
                             bias=False,
                             padding_mode='zeros') # doubt about
padding.



        ### YOUR CODE HERE

        # We do not include batch normalization or activation functions
in V2
        # for the initial conv1 because the first block unit will perform
these
        # for both the shortcut and non-shortcut paths as part of the
first
        # block's projection.
        if self.resnet_version == 1:
            self.batch_norm_relu_start = batch_norm_relu_layer(
                num_features=self.first_num_filters,
                eps=1e-5,
                momentum=0.997,
            )
        if self.resnet_version == 1:
            block_fn = standard_block
        else:
            block_fn = bottleneck_block

        self.stack_layers = nn.ModuleList()
        for i in range(3):
            filters = self.first_num_filters * (2**i) # is there any
problem here?
            strides = 1 if i == 0 else 2
            self.stack_layers.append(stack_layer(filters, block_fn,
strides, self.resnet_size, self.first_num_filters))
        self.output_layer = output_layer(filters*4, self.resnet_version,
self.num_classes)

    def start_layer(self,inputs):
        return self.conv1(inputs)

    def forward(self, inputs):
```

```
        outputs = self.start_layer(inputs)
        if self.resnet_version == 1:
            outputs = self.batch_norm_relu_start(outputs)
        for i in range(3):
            outputs = self.stack_layers[i](outputs)
        outputs = self.output_layer(outputs)
        return outputs

################################################################################
#####
# Blocks building the network
################################################################################
#####

class batch_norm_relu_layer(nn.Module):
    """ Perform batch normalization then relu.
    """
    def __init__(self, num_features, eps=1e-5, momentum=0.997) -> None:
        super(batch_norm_relu_layer, self).__init__()
        ### YOUR CODE HERE
        self.batch_norm_layer = nn.BatchNorm2d(num_features=num_features,
                                               eps=eps,
                                               momentum=momentum)
        self.relu = nn.ReLU(inplace=True)

        ### YOUR CODE HERE
    def forward(self, inputs: Tensor) -> Tensor:
        ### YOUR CODE HERE
        out = self.batch_norm_layer(inputs)
        out = self.relu(out)
        return out

        ### YOUR CODE HERE

class standard_block(nn.Module):
    """ Creates a standard residual block for ResNet.

    Args:
        filters: A positive integer. The number of filters for the first
            convolution.
        projection_shortcut: The function to use for projection shortcuts
```

```
        (typically a 1x1 convolution when downsampling the input).
     strides: A positive integer. The stride to use for the block. If
        greater than 1, this block will ultimately downsample the input.
       first_num_filters: An integer. The number of filters to use for
the
           first block layer of the model.
           # first num filters, we will get when it is particularly
first block in given stack,
    """
   def __init__(self, filters, projection_shortcut, strides,
first_num_filters) -> None:
       super(standard_block, self).__init__()
       ### YOUR CODE HERE
       self.filters = filters
       self.projection_shortcut = projection_shortcut
       self.strides = strides
       self.first_num_filters = first_num_filters
       #standard_block      conv3-16 + conv3-16
       in_filters = self.filters
       if projection_shortcut:
           in_filters = self.filters//2
       #in_filters = self.filters if
self.filters==self.first_num_filters or self else self.filters//2
       self.conv3_blk1 = nn.Conv2d(in_channels=in_filters,
                           out_channels=self.filters, # first num filters?
                           kernel_size=(3,3),
                           stride=strides, # replace strides here.
                           padding=(1,1))
       self.conv3_blk2 = nn.Conv2d(in_channels=self.filters,
                                   out_channels=self.filters,  # first
num filters?
                                   kernel_size=(3, 3),
                                   stride=(1,1), # replace strides here
                                   padding=(1, 1))

       self.batch_norm_relu_layer_current =
batch_norm_relu_layer(num_features=self.filters)


       if self.projection_shortcut is not None:
           self.identity_conv = self.projection_shortcut
```

```
        self.relu = nn.ReLU(inplace=True)
        ### YOUR CODE HERE

        ### YOUR CODE HERE

        ### YOUR CODE HERE

    def forward(self, inputs: Tensor) -> Tensor:
        ### YOUR CODE HERE
        identity_mapping  = inputs
        if self.projection_shortcut:
            identity_mapping = self.projection_shortcut(identity_mapping)
            identity_mapping =
self.batch_norm_relu_layer_current(identity_mapping)

        out = self.conv3_blk1(inputs)
        out = self.batch_norm_relu_layer_current.forward(out)
        out = self.conv3_blk2(out)
        out = self.batch_norm_relu_layer_current.forward(out)

        out = out + identity_mapping

        out = self.relu(out)
        ### YOUR CODE HERE
        return out




class bottleneck_block(nn.Module):
    """ Creates a bottleneck block for ResNet.

    Args:
        filters: A positive integer. The number of filters for the first
            convolution. NOTE: filters_out will be 4xfilters.
        projection_shortcut: The function to use for projection shortcuts
          (typically a 1x1 convolution when downsampling the input).
      strides: A positive integer. The stride to use for the block. If
        greater than 1, this block will ultimately downsample the input.
        first_num_filters: An integer. The number of filters to use for
```

```
the
            first block layer of the model.
    """
    def __init__(self, filters, projection_shortcut, strides,
first_num_filters) -> None:
        super(bottleneck_block, self).__init__()

        ### YOUR CODE HERE
        self.filters = filters
        self.projection_shortcut = projection_shortcut
        self.strides = strides
        self.first_num_filters = first_num_filters
        #bottleneck_block    conv1-16 + conv3-16 + conv1-64

        in_filters = self.filters
        if self.projection_shortcut:
            in_filters = self.filters//2
            if self.filters==self.first_num_filters*4:
                in_filters = self.first_num_filters
            # update in_filters



        self.conv1_blk1 = nn.Conv2d(in_channels=in_filters,
                                    out_channels=self.filters//4,  #
first num filters?
                                    kernel_size=(1, 1),
                                    stride=self.strides
                                    )
        self.conv3_blk2 = nn.Conv2d(in_channels=self.filters//4,
                                    out_channels=self.filters//4,  #
first num filters?
                                    kernel_size=(3, 3),
                                    stride=(1,1),
                                    padding=(1, 1))
        self.conv1_blk3 = nn.Conv2d(in_channels=self.filters//4,
                                    out_channels=self.filters,  # first
num filters?
                                    kernel_size=(1, 1),
                                    stride=(1,1))
```

```
        self.batch_norm_relu_layer_first =
batch_norm_relu_layer(num_features=in_filters) # assumiing 4*filters
here
        self.batch_norm_relu_layer_second =
batch_norm_relu_layer(num_features=self.filters//4)
        self.relu = nn.ReLU(inplace=True)
        # check and calculate the count of filters once more.
        # Hint: Different from standard lib implementation, you need pay
attention to
        # how to define in_channel of the first bn and conv of each block
based on
        # Args given above.

        ### YOUR CODE HERE

    def forward(self, inputs: Tensor) -> Tensor:
        ### YOUR CODE HERE
        # The projection shortcut should come after the first batch norm
and ReLU
      # since it performs a 1x1 convolution.

        if self.projection_shortcut:
            inputs1 = self.batch_norm_relu_layer_first.forward(inputs)
            identity_mapping = self.projection_shortcut(inputs1)
        else:
            identity_mapping = inputs

        out = self.batch_norm_relu_layer_first.forward(inputs)
        out = self.conv1_blk1(out)

        out = self.batch_norm_relu_layer_second.forward(out)
        out = self.conv3_blk2(out)

        out = self.batch_norm_relu_layer_second.forward(out)
        out = self.conv1_blk3(out)

        out = out + identity_mapping
        return out # is relu required?
        ### YOUR CODE HERE

class stack_layer(nn.Module):
```

```
   """ Creates one stack of standard blocks or bottleneck blocks.

   Args:
       filters: A positive integer. The number of filters for the first
            convolution in a block.
     block_fn: 'standard_block' or 'bottleneck_block'.
     strides: A positive integer. The stride to use for the first block.
If
           greater than 1, this layer will ultimately downsample the
input.
       resnet_size: #residual_blocks in each stack layer
       first_num_filters: An integer. The number of filters to use for
the
           first block layer of the model.
   """
   def __init__(self, filters, block_fn, strides, resnet_size,
first_num_filters) -> None:
       super(stack_layer, self).__init__()

       self.filters = filters
       self.filters_out = filters * 4 if block_fn is bottleneck_block
else filters # for projection shortcut code.
       self.first_num_filters = first_num_filters
       self.resnet_size = resnet_size
       self.strides = strides




       if block_fn is bottleneck_block:
           #bb_infliters
           projection_def = nn.Conv2d(in_channels=self.filters*2,
                                      out_channels=self.filters_out,
                                      kernel_size=(1,1),
                                      stride=strides
                                      )

       else:
           #_in_filters = self.filters if
self.filters==self.first_num_filters else self.filters//2
           projection_def = nn.Conv2d(in_channels=filters//2,
```

```
                                    out_channels=self.filters,
                                    kernel_size=(1,1),
                                    stride=strides
                                    )
            #self.std_params = [_in_filters,self.filters_out]


        if filters==first_num_filters:
            if block_fn is standard_block:
                projection_def = None
            else:
                projection_def =
nn.Conv2d(in_channels=self.first_num_filters,
                                          out_channels=self.filters_out,
                                          kernel_size=(1, 1),
                                          stride=strides
                                          )

        self.block_fn_blk0 = block_fn(filters=self.filters_out,
                                      projection_shortcut=projection_def,
                                      strides=self.strides,

first_num_filters=self.first_num_filters)
        self.block_fn = block_fn(filters=self.filters_out,
                                      projection_shortcut=None,
                                      strides=(1,1),

first_num_filters=self.first_num_filters)

        ### END CODE HERE
        # projection_shortcut = ?
        # Only the first block per stack_layer uses projection_shortcut
and strides

        ### END CODE HERE

    def forward(self, inputs: Tensor) -> Tensor:
        ### END CODE HERE
        out = inputs

        for i in range(self.resnet_size):
```

```
            if i==0:
                #print('first of stack')
                #print(self.filters,self.filters_out)
                out = self.block_fn_blk0.forward(out)
            else:
                #print('remaining stack')
                out = self.block_fn.forward(out)

        return out


        ### END CODE HERE

class output_layer(nn.Module):
    """ Implement the output layer.

    Args:
        filters: A positive integer. The number of filters.
        resnet_version: 1 or 2, If 2, use the bottleneck blocks.
        num_classes: A positive integer. Define the number of classes.
    """
    def __init__(self, filters, resnet_version, num_classes) -> None:
        super(output_layer, self).__init__()
        # Only apply the BN and ReLU for model that does pre_activation
in each
      # bottleneck block, e.g. resnet V2.
        #self.filters = filters
        self.resnet_version = resnet_version
        self.num_classes = num_classes
        if (resnet_version == 2):
            #self.filters=filters//4
            self.bn_relu = batch_norm_relu_layer(filters, eps=1e-5,
momentum=0.997)

        self.global_pool
=nn.AdaptiveAvgPool2d((1,1))#nn.AvgPool2d(kernel_size=8)
        if resnet_version==1:
            self.fc = nn.Linear(filters//4,10,bias=True)
        else:
            self.fc = nn.Linear(filters , 10, bias=True)
```

```
        self.softmax = nn.Softmax(dim=-1)

        # recheck this code, seems fishy

        ### END CODE HERE

        ### END CODE HERE

    def forward(self, inputs: Tensor) -> Tensor:
        ### END CODE HERE
        if self.resnet_version==2:
            out = self.bn_relu(inputs)
        else:
            out = inputs

        out = self.global_pool(out)
        out = out.view(out.size(0), -1)
        #print(out.shape,.shape)
        out = self.fc(out)
        out = self.softmax(out)
        return out

        ### END CODE HERE
```

(d) (20 points) Complete “Model.py”. Note: For this step and last step, pay attention to how to use batch normalization

Cross entropy loss function, optimiser, training and testing code is below.

```
import os
import time
import torch
import torch.nn as nn
import numpy as np
from tqdm import tqdm

from NetWork import ResNet
from ImageUtils import parse_record

""" This script defines the training, validation and testing process.
"""
```

```
class Cifar(nn.Module):
    def __init__(self, config):
        super(Cifar, self).__init__()
        self.config = config
        self.network = ResNet(
            self.config.resnet_version,
            self.config.resnet_size,
            self.config.num_classes,
            self.config.first_num_filters,
        )
        self.lr = 0.1
        ### YOUR CODE HERE
        # define cross entropy loss and optimizer
        self.cross_entropy_loss = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.SGD(self.network.parameters(),
momentum=0.9,

weight_decay=self.config.weight_decay,
                                         lr=self.lr# should i take it in
input?
                                         )
        #if torch.cuda.is_available():
        torch.cuda.set_device('cuda:0')
        self.network = self.network.cuda()
        self.cross_entropy_loss = self.cross_entropy_loss.cuda()
     self.learning_rate = 0.1

        ### YOUR CODE HERE

    def train(self, x_train, y_train, max_epoch):
        self.network.train()
        # Determine how many batches in an epoch
        num_samples = x_train.shape[0]
        num_batches = num_samples // self.config.batch_size

        print('### Training... ###')
        for epoch in range(1, max_epoch+1):
            start_time = time.time()
            # Shuffle
            shuffle_index = np.random.permutation(num_samples)
            curr_x_train = x_train[shuffle_index]
```

```
            curr_y_train = y_train[shuffle_index]

            ### YOUR CODE HERE
            # Set the learning rate for this epoch
            # Usage example: divide the initial learning rate by 10 after
several epochs
            if (epoch % 20 == 0):
                self.learning_rate = self.learning_rate * (0.1 ** (epoch
// 20))

            ### YOUR CODE HERE

            for i in range(num_batches):
                ### YOUR CODE HERE
                # Construct the current batch.
                # Don't forget to use "parse_record" to perform data
preprocessing.
                # Don't forget L2 weight decay
                input_raw = curr_x_train[i * self.config.batch_size:(i +
1) * self.config.batch_size]
                input = np.apply_along_axis(lambda x: parse_record(x,
True), 1, input_raw)
                self.optimizer.zero_grad()

                #if torch.cuda.is_available():
                    #print('cuda is available.')
                output = self.network(torch.tensor(input).to('cuda'))
                target = curr_y_train[i * self.config.batch_size:(i + 1)
* self.config.batch_size]
                loss = self.cross_entropy_loss(output,
torch.tensor(target).long().to('cuda'))
                #else:
                #    output = self.network(torch.tensor(input))
                #    target = curr_y_train[i * self.config.batch_size:(i
+ 1) * self.config.batch_size]
                #    loss = self.cross_entropy_loss(output,
torch.tensor(target).long())


                # accuracy, acc5 = self.accuracy(output, target, topk=(1,
5))
```

```
                ### YOUR CODE HERE

                loss.backward()
                self.optimizer.step()

                print('Batch {:d}/{:d} Loss {:.6f}'.format(i,
num_batches, loss), end='\r', flush=False)

            duration = time.time() - start_time
            print('Epoch {:d} Loss {:.6f} Duration {:.3f}
seconds.'.format(epoch, loss, duration))

            if epoch % self.config.save_interval == 0:
                self.save(epoch)


    def test_or_validate(self, x, y, checkpoint_num_list):
        self.network.eval()
        print('### Test or Validation ###')
        for checkpoint_num in checkpoint_num_list:
            checkpointfile = os.path.join(self.config.modeldir,
'model-%d.ckpt'%(checkpoint_num))
            self.load(checkpointfile)

            preds = []
            for i in tqdm(range(x.shape[0])):
                ### YOUR CODE HERE
                current_input=parse_record(x[i], False)
                output=
self.network(torch.tensor(current_input).float().view(1, 3, 32,
32).to('cuda'))
                _, preds  = torch.max(output, dim=1)
                ### END CODE HERE

            y = torch.tensor(y)
            preds = torch.tensor(preds)
            print('Test accuracy:
{:.4f}'.format(torch.sum(preds==y)/y.shape[0]))

    def save(self, epoch):
```

```
        checkpoint_path = os.path.join(self.config.modeldir,
'model-%d.ckpt'%(epoch))
        os.makedirs(self.config.modeldir, exist_ok=True)
        torch.save(self.network.state_dict(), checkpoint_path)
        print("Checkpoint has been created.")

    def load(self, checkpoint_name):
        ckpt = torch.load(checkpoint_name, map_location="cpu")
        self.network.load_state_dict(ckpt, strict=True)
        print("Restored model parameters from
{}".format(checkpoint_name))
```

e) (20 points) Tune all the hyperparameters in “main.py” and report your final testing accuracy.

Accuracy obtained for standard residual block with the following parameters (resnet_version=1,resnet_size=3,weight_decay=2e-4,batch_size=128) is 85.03percent

Accuracy obtained for bottleneck block with the following parameters (resnet_version=2,resnet_size=3,weight_decay=2e-4,batch_size=128) is 70.7percent

Data and Logs:
Resnet version=1: standardblock:

```
...
--- Preparing Data ---
Cifar(
  (network): ResNet(
    (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (batch_norm_relu_start): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (stack_layers): ModuleList(
      (0): stack_layer(
        (block_fn_blk0): standard_block(
          (conv3_blk1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (conv3_blk2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (batch_norm_relu_layer_current): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (relu): ReLU(inplace=True)
        )
        (block_fn): standard_block(
          (conv3_blk1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (conv3_blk2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (batch_norm_relu_layer_current): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (relu): ReLU(inplace=True)
        )
      )
      (1): stack_layer(
        (block_fn_blk0): standard_block(
          (projection_shortcut): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2))
          (conv3_blk1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
          (conv3_blk2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (batch_norm_relu_layer_current): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(32, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (identity_conv): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2))
          (relu): ReLU(inplace=True)
        )
        (block_fn): standard_block(
          (conv3_blk1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (conv3_blk2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (batch_norm_relu_layer_current): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(32, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
```

```
    )
    (2): stack_layer(
      (block_fn_blk0): standard_block(
        (projection_shortcut): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2))
        (conv3_blk1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (conv3_blk2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (batch_norm_relu_layer_current): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (identity_conv): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2))
        (relu): ReLU(inplace=True)
      )
      (block_fn): standard_block(
        (conv3_blk1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv3_blk2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (batch_norm_relu_layer_current): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (relu): ReLU(inplace=True)
      )
    )
  )
  (output_layer): output_layer(
    (global_pool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=64, out_features=10, bias=True)
    (softmax): Softmax(dim=-1)
  )
)
(cross_entropy_loss): CrossEntropyLoss()
```

```
Epoch 1 Loss 2.063962 Duration 62.312 seconds.
Epoch 2 Loss 1.959726 Duration 62.129 seconds.
Epoch 3 Loss 2.002911 Duration 62.188 seconds.
Epoch 4 Loss 1.948865 Duration 62.258 seconds.
Epoch 5 Loss 1.863454 Duration 62.274 seconds.
Epoch 6 Loss 1.853503 Duration 62.252 seconds.
Epoch 7 Loss 1.860531 Duration 62.298 seconds.
Epoch 8 Loss 1.814735 Duration 62.255 seconds.
Epoch 9 Loss 1.753281 Duration 62.278 seconds.
Epoch 10 Loss 1.754852 Duration 62.261 seconds.
Checkpoint has been created.
Epoch 11 Loss 1.805236 Duration 62.239 seconds.
Epoch 12 Loss 1.787339 Duration 62.380 seconds.
Epoch 13 Loss 1.717457 Duration 62.380 seconds.
Epoch 14 Loss 1.678932 Duration 62.376 seconds.
Epoch 15 Loss 1.676123 Duration 62.336 seconds.
Epoch 16 Loss 1.743854 Duration 62.394 seconds.
Epoch 17 Loss 1.712539 Duration 62.375 seconds.
Epoch 18 Loss 1.717602 Duration 62.350 seconds.
Epoch 19 Loss 1.676736 Duration 62.365 seconds.
Epoch 20 Loss 1.708898 Duration 62.335 seconds.
Checkpoint has been created.
Epoch 21 Loss 1.636313 Duration 62.347 seconds.
Epoch 22 Loss 1.760962 Duration 62.395 seconds.
Epoch 23 Loss 1.717784 Duration 62.447 seconds.
Epoch 24 Loss 1.647758 Duration 62.428 seconds.
Epoch 25 Loss 1.668062 Duration 62.386 seconds.
Epoch 26 Loss 1.686247 Duration 62.426 seconds.
Epoch 27 Loss 1.617589 Duration 62.398 seconds.
Epoch 28 Loss 1.639228 Duration 62.395 seconds.
Epoch 29 Loss 1.686574 Duration 62.432 seconds.
Epoch 30 Loss 1.684226 Duration 62.406 seconds.
Checkpoint has been created.
Epoch 31 Loss 1.685233 Duration 62.426 seconds.
Epoch 32 Loss 1.740310 Duration 62.397 seconds.
Epoch 33 Loss 1.703576 Duration 62.411 seconds.
Epoch 34 Loss 1.723105 Duration 62.437 seconds.
Epoch 35 Loss 1.683765 Duration 62.434 seconds.
Epoch 36 Loss 1.629665 Duration 62.437 seconds.
Epoch 37 Loss 1.642728 Duration 62.431 seconds.
Epoch 38 Loss 1.728939 Duration 62.413 seconds.
Epoch 39 Loss 1.654000 Duration 62.435 seconds.
Epoch 40 Loss 1.709497 Duration 62.461 seconds.
Checkpoint has been created.
Epoch 41 Loss 1.619983 Duration 62.475 seconds.
Epoch 42 Loss 1.678408 Duration 62.472 seconds.
Epoch 43 Loss 1.680824 Duration 62.456 seconds.
Epoch 44 Loss 1.699121 Duration 62.425 seconds.
Epoch 45 Loss 1.691303 Duration 62.422 seconds.
Epoch 46 Loss 1.632417 Duration 62.457 seconds.
Epoch 47 Loss 1.637103 Duration 62.484 seconds.
Epoch 48 Loss 1.694183 Duration 62.481 seconds.
```

```
Epoch 49 Loss 1.667701 Duration 62.476 seconds.
Epoch 50 Loss 1.669260 Duration 62.499 seconds.
Checkpoint has been created.
Epoch 51 Loss 1.648994 Duration 62.376 seconds.
Epoch 52 Loss 1.737322 Duration 62.267 seconds.
Epoch 53 Loss 1.703362 Duration 62.357 seconds.
Epoch 54 Loss 1.589242 Duration 62.396 seconds.
Epoch 55 Loss 1.639687 Duration 62.427 seconds.
Epoch 56 Loss 1.670009 Duration 62.389 seconds.
Epoch 57 Loss 1.718067 Duration 62.364 seconds.
Epoch 58 Loss 1.638454 Duration 62.419 seconds.
Epoch 59 Loss 1.625735 Duration 62.406 seconds.
Epoch 60 Loss 1.644672 Duration 62.380 seconds.
Checkpoint has been created.
Epoch 61 Loss 1.661167 Duration 62.350 seconds.
Epoch 62 Loss 1.603198 Duration 62.311 seconds.
Epoch 63 Loss 1.665791 Duration 62.299 seconds.
Epoch 64 Loss 1.660122 Duration 62.329 seconds.
Epoch 65 Loss 1.686378 Duration 62.361 seconds.
Epoch 66 Loss 1.671311 Duration 62.358 seconds.
Epoch 67 Loss 1.627235 Duration 62.401 seconds.
Epoch 68 Loss 1.592627 Duration 62.355 seconds.
Epoch 69 Loss 1.609810 Duration 62.370 seconds.
Epoch 70 Loss 1.674299 Duration 62.390 seconds.
Checkpoint has been created.
Epoch 71 Loss 1.657786 Duration 62.395 seconds.
Epoch 72 Loss 1.664300 Duration 62.392 seconds.
Epoch 73 Loss 1.636161 Duration 62.426 seconds.
Epoch 74 Loss 1.665088 Duration 62.367 seconds.
Epoch 75 Loss 1.671768 Duration 62.346 seconds.
Epoch 76 Loss 1.688044 Duration 62.385 seconds.
Epoch 77 Loss 1.622798 Duration 62.396 seconds.
Epoch 78 Loss 1.679244 Duration 62.321 seconds.
Epoch 79 Loss 1.639078 Duration 62.353 seconds.
Epoch 80 Loss 1.681686 Duration 62.324 seconds.
Checkpoint has been created.
Epoch 81 Loss 1.663954 Duration 62.324 seconds.
Epoch 82 Loss 1.678027 Duration 62.361 seconds.
Epoch 83 Loss 1.647133 Duration 62.350 seconds.
Epoch 84 Loss 1.678431 Duration 62.387 seconds.
Epoch 85 Loss 1.620991 Duration 62.338 seconds.
Epoch 86 Loss 1.670054 Duration 62.383 seconds.
Epoch 87 Loss 1.650006 Duration 62.364 seconds.
Epoch 88 Loss 1.665723 Duration 62.373 seconds.
Epoch 89 Loss 1.596577 Duration 62.311 seconds.
```

Test or Validation
Restored model parameters from model_v1/model-190.ckpt
100% 10000/10000 [06:52<00:00, 24.25it/s]
Test accuracy: 0.8503

Resnet version-2: Bottleneck block:

```
--- Preparing Data ---
Cifar(
  (network): ResNet(
    (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (stack_layers): ModuleList(
      (0): stack_layer(
        (block_fn_blk0): bottleneck_block(
          (projection_shortcut): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
          (conv1_blk1): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1))
          (conv3_blk2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (conv1_blk3): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
          (batch_norm_relu_layer_first): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (batch_norm_relu_layer_second): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (relu): ReLU(inplace=True)
        )
        (block_fn): bottleneck_block(
          (conv1_blk1): Conv2d(64, 16, kernel_size=(1, 1), stride=(1, 1))
          (conv3_blk2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (conv1_blk3): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
          (batch_norm_relu_layer_first): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (batch_norm_relu_layer_second): batch_norm_relu_layer(
            (batch_norm_layer): BatchNorm2d(16, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
          )
          (relu): ReLU(inplace=True)
```

```
      )
    )
    (1): stack_layer(
      (block_fn_blk0): bottleneck_block(
        (projection_shortcut): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2))
        (conv1_blk1): Conv2d(64, 32, kernel_size=(1, 1), stride=(2, 2))
        (conv3_blk2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv1_blk3): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
        (batch_norm_relu_layer_first): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (batch_norm_relu_layer_second): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(32, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (relu): ReLU(inplace=True)
      )
      (block_fn): bottleneck_block(
        (conv1_blk1): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
        (conv3_blk2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv1_blk3): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
        (batch_norm_relu_layer_first): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(128, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (batch_norm_relu_layer_second): batch_norm_relu_layer(
          (batch_norm_layer): BatchNorm2d(32, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
        )
        (relu): ReLU(inplace=True)
      )
    )
```

```
(2): stack_layer(
  (block_fn_blk0): bottleneck_block(
    (projection_shortcut): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2))
    (conv1_blk1): Conv2d(128, 64, kernel_size=(1, 1), stride=(2, 2))
    (conv3_blk2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv1_blk3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
    (batch_norm_relu_layer_first): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(128, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (batch_norm_relu_layer_second): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (relu): ReLU(inplace=True)
  )
  (block_fn): bottleneck_block(
    (conv1_blk1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
    (conv3_blk2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv1_blk3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
    (batch_norm_relu_layer_first): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(256, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (batch_norm_relu_layer_second): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(64, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (relu): ReLU(inplace=True)
  )
)
```

```
output_layer): output_layer(
!python3 main.py --resnet_version=2 --model_dir=ckpt_v2 --resnet_size=5
    )
  )
  (output_layer): output_layer(
    (bn_relu): batch_norm_relu_layer(
      (batch_norm_layer): BatchNorm2d(256, eps=1e-05, momentum=0.997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (global_pool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=256, out_features=10, bias=True)
    (softmax): Softmax(dim=-1)
  )
)
(cross_entropy_loss): CrossEntropyLoss()
)
```

```
### Training... ###
Epoch 1 Loss 2.094461 Duration 35.690 seconds.
Epoch 2 Loss 1.981249 Duration 27.714 seconds.
Epoch 3 Loss 1.958333 Duration 27.900 seconds.
Epoch 4 Loss 1.880860 Duration 27.736 seconds.
Epoch 5 Loss 1.898471 Duration 27.958 seconds.
Epoch 6 Loss 1.813861 Duration 27.685 seconds.
Epoch 7 Loss 1.802229 Duration 27.928 seconds.
Epoch 8 Loss 1.848494 Duration 27.504 seconds.
Epoch 9 Loss 1.786436 Duration 27.813 seconds.
Epoch 10 Loss 1.785991 Duration 27.661 seconds.
Checkpoint has been created.
Epoch 11 Loss 1.698144 Duration 27.823 seconds.
Epoch 12 Loss 1.773492 Duration 27.801 seconds.
Epoch 13 Loss 1.723158 Duration 28.084 seconds.
Epoch 14 Loss 1.725232 Duration 27.958 seconds.
Epoch 15 Loss 1.707285 Duration 27.905 seconds.
Epoch 16 Loss 1.749344 Duration 27.858 seconds.
Epoch 17 Loss 1.681610 Duration 27.817 seconds.
```

```
Epoch 18 Loss 1.740912 Duration 27.895 seconds.
Epoch 19 Loss 1.729946 Duration 28.011 seconds.
Epoch 20 Loss 1.675037 Duration 27.953 seconds.
Checkpoint has been created.
Epoch 21 Loss 1.692883 Duration 27.889 seconds.
Epoch 22 Loss 1.669573 Duration 27.905 seconds.
Epoch 23 Loss 1.719617 Duration 27.834 seconds.
Epoch 24 Loss 1.714611 Duration 27.952 seconds.
Epoch 25 Loss 1.718670 Duration 28.109 seconds.
Epoch 26 Loss 1.727162 Duration 27.944 seconds.
Epoch 27 Loss 1.733073 Duration 27.940 seconds.
Epoch 28 Loss 1.698324 Duration 27.915 seconds.
Epoch 29 Loss 1.760591 Duration 28.013 seconds.
Epoch 30 Loss 1.738113 Duration 28.035 seconds.
Checkpoint has been created.
Epoch 31 Loss 1.724104 Duration 28.304 seconds.
Epoch 32 Loss 1.678053 Duration 28.076 seconds.
Epoch 33 Loss 1.709585 Duration 28.068 seconds.
Epoch 34 Loss 1.658436 Duration 28.025 seconds.
Epoch 35 Loss 1.727518 Duration 27.938 seconds.
Epoch 36 Loss 1.688387 Duration 28.079 seconds.
Epoch 37 Loss 1.672081 Duration 28.233 seconds.
Epoch 38 Loss 1.719710 Duration 28.094 seconds.
Epoch 39 Loss 1.645733 Duration 28.062 seconds.
Epoch 40 Loss 1.697063 Duration 28.014 seconds.
Checkpoint has been created.
Epoch 41 Loss 1.707797 Duration 28.062 seconds.
Epoch 42 Loss 1.636274 Duration 27.969 seconds.
Epoch 43 Loss 1.618022 Duration 28.061 seconds.
Epoch 44 Loss 1.627311 Duration 28.063 seconds.
Epoch 45 Loss 1.660175 Duration 27.974 seconds.
Epoch 46 Loss 1.681223 Duration 27.961 seconds.
Epoch 47 Loss 1.658405 Duration 28.017 seconds.
Epoch 48 Loss 1.633694 Duration 27.940 seconds.
Epoch 49 Loss 1.662622 Duration 27.991 seconds.
Epoch 50 Loss 1.691146 Duration 27.923 seconds.
Checkpoint has been created.
Epoch 51 Loss 1.692591 Duration 27.987 seconds.
Epoch 52 Loss 1.668391 Duration 27.965 seconds.
Epoch 53 Loss 1.651972 Duration 28.015 seconds.
Epoch 54 Loss 1.672360 Duration 27.954 seconds.
Epoch 55 Loss 1.632628 Duration 28.260 seconds.
Epoch 56 Loss 1.657507 Duration 27.981 seconds.
Epoch 57 Loss 1.657879 Duration 27.962 seconds.
Epoch 58 Loss 1.689448 Duration 27.977 seconds.
Epoch 59 Loss 1.665843 Duration 27.996 seconds.
Epoch 60 Loss 1.625153 Duration 28.104 seconds.
```

```
Checkpoint has been created.
Epoch 61 Loss 1.622722 Duration 28.095 seconds.
Epoch 62 Loss 1.614306 Duration 27.966 seconds.
Epoch 63 Loss 1.707844 Duration 27.936 seconds.
Epoch 64 Loss 1.668001 Duration 27.985 seconds.
Epoch 65 Loss 1.620628 Duration 27.986 seconds.
Epoch 66 Loss 1.643934 Duration 27.775 seconds.
Epoch 67 Loss 1.628760 Duration 28.060 seconds.
Epoch 68 Loss 1.661892 Duration 28.016 seconds.
Epoch 69 Loss 1.689790 Duration 28.048 seconds.
Epoch 70 Loss 1.661760 Duration 27.980 seconds.
Checkpoint has been created.
Epoch 71 Loss 1.655019 Duration 27.833 seconds.
Epoch 72 Loss 1.602953 Duration 27.633 seconds.
Epoch 73 Loss 1.632984 Duration 28.030 seconds.
Epoch 74 Loss 1.642811 Duration 27.955 seconds.
Epoch 75 Loss 1.676110 Duration 27.852 seconds.
Epoch 76 Loss 1.614668 Duration 27.973 seconds.
Epoch 77 Loss 1.686430 Duration 27.830 seconds.
Epoch 78 Loss 1.630258 Duration 27.883 seconds.
Epoch 79 Loss 1.654759 Duration 27.736 seconds.
Epoch 80 Loss 1.659786 Duration 27.730 seconds.
Checkpoint has been created.
Epoch 81 Loss 1.601070 Duration 27.800 seconds.
Epoch 82 Loss 1.641936 Duration 27.797 seconds.
Epoch 83 Loss 1.659665 Duration 27.737 seconds.
Epoch 84 Loss 1.654438 Duration 27.612 seconds.
Epoch 85 Loss 1.643312 Duration 27.796 seconds.
Epoch 86 Loss 1.651522 Duration 27.878 seconds.
Epoch 87 Loss 1.693769 Duration 27.845 seconds.
Epoch 88 Loss 1.680024 Duration 27.864 seconds.
Epoch 89 Loss 1.630552 Duration 27.764 seconds.
Epoch 90 Loss 1.652674 Duration 27.764 seconds.
Checkpoint has been created.
```

Test or Validation
Restored model parameters from model_v2/model-190.ckpt
100% 10000/10000 [06:52<00:00, 24.25it/s]
Test accuracy: 0.707