Tanmay Garg

CS20BTECH11063

Deep Learning Assignment 2

In [ ]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import math
import torch.functional as F
import torch.nn as nn
from sklearn.utils import shuffle
import PIL
from PIL import Image
```

Q1

Convolution Function

In [ ]:
```
# Convolution Function implementation
def my_conv2d(input, weight, bias=None, stride=1, padding=0, activation='relu'):
    # input is a 4D tensor of shape (N, C_in, H_in, W_in)
    # weight is 4D tensor of shape (D, C_in, H, W)
    # bias is a 1D tensor of shape (D)
    # stride is an integer
    # padding is an integer
    # activation is a string
    # output is a 4D tensor of shape (N, D, H_out, W_out)

    if padding != 0:
        image_pad = torch.zeros((input.shape[0], input.shape[1], input.shape[2] + 2*padding, input.shape[3] + 2*padding))
        print("Shape of padded image: ", image_pad.shape)
        # Applying padding
        image_pad[:, :, padding:-padding, padding:-padding] = input
        input = image_pad
```

```python
    # Calculating output shape
    H_out = (input.shape[2] - weight.shape[2])//stride + 1
    W_out = (input.shape[3] - weight.shape[3])//stride + 1

    # Initializing output tensor
    output = torch.zeros((input.shape[0], weight.shape[0], H_out, W_out))

    # Applying convolution
    for i in range(H_out):
        for j in range(W_out):
            for k in range(weight.shape[0]):
                output[:, k, i, j] = torch.sum(input[:, :, i*stride:i*stride+weight.shape[2],
                                                     j*stride:j*stride+weight.shape[3]] * weight[k, :, :, :],
                                               dim=(1, 2, 3))

    # Adding bias
    if bias is not None:
        output += bias.reshape(1, bias.shape[0], 1, 1)

    # Applying activation
    if activation == 'relu':
        output = torch.relu(output)
    elif activation == 'sigmoid':
        output = torch.sigmoid(output)
    elif activation == 'tanh':
        output = torch.tanh(output)
    elif activation == 'softmax':
        output = torch.softmax(output, dim=1)
    elif activation == 'prelu':
        output = torch.prelu(output)
    elif activation == 'leaky_relu':
        output = torch.leaky_relu(output)
    elif activation == 'none':
        pass
    else:
        print("Invalid activation function")

    return output
```

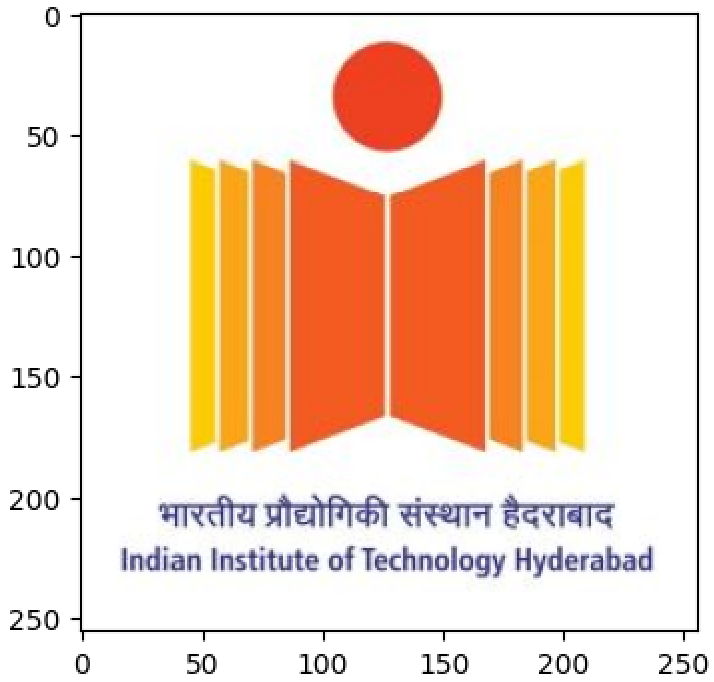
In [ ]:

```python
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
weight = torch.empty(5, 3, 5, 5).normal_()
bias = torch.empty(5).normal_()
```

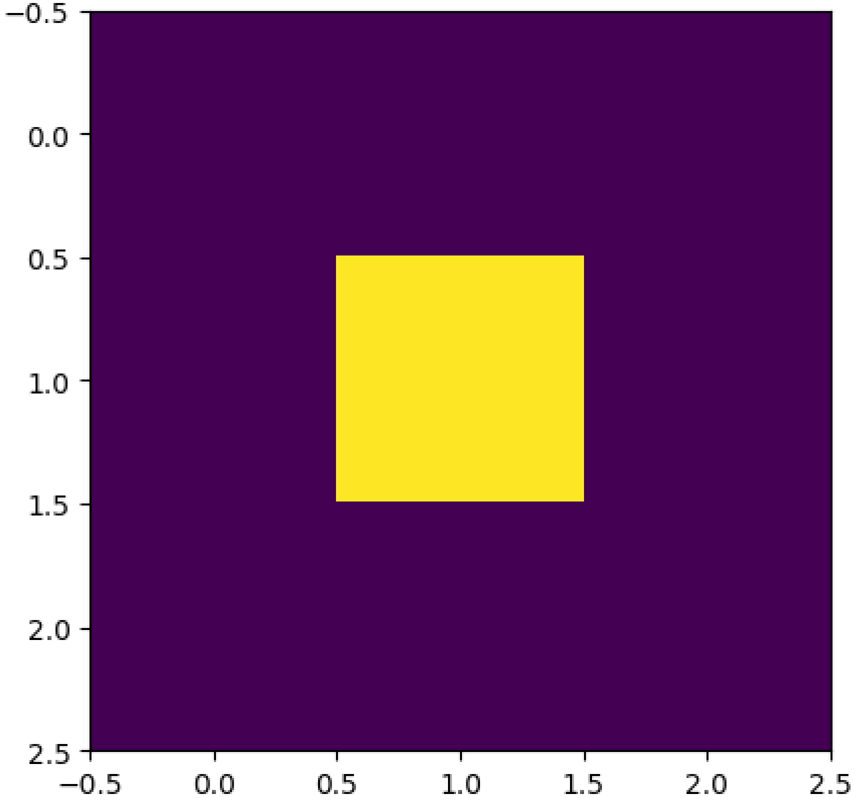
```python
output = my_conv2d(input, weight, bias)
output.size()
```

Out[ ]:
```
torch.Size([128, 5, 16, 16])
```

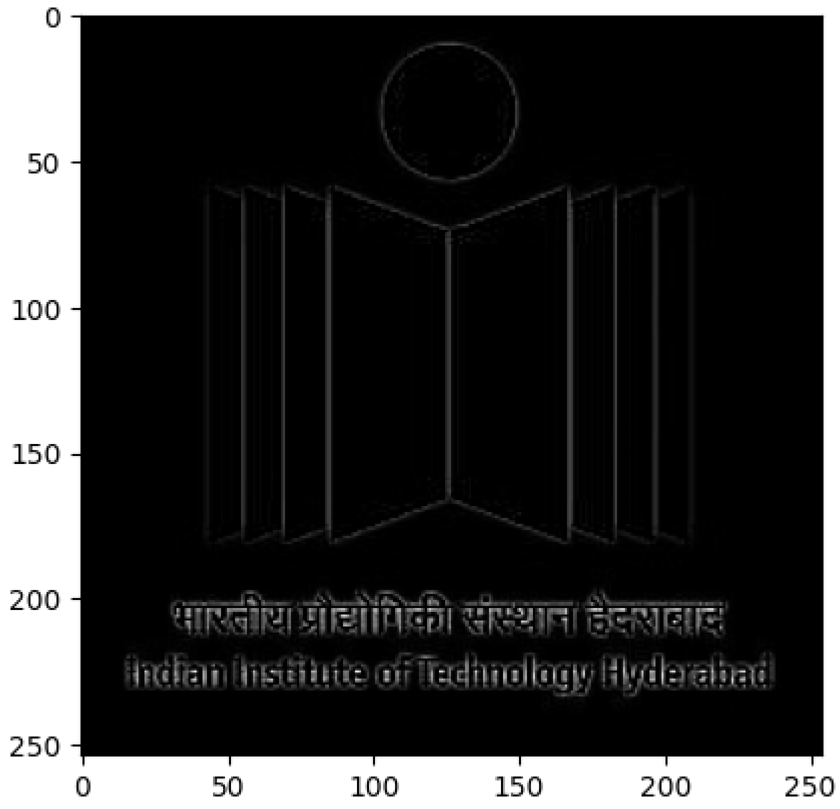
In [ ]:
```python
# Load the image and display it, display the filter kernel and display the output image
# Load the image
img = plt.imread('logo2.jpg')
plt.figure(figsize=(4, 4))
plt.imshow(img)
plt.show()
print("Shape of Image: ",img.shape)

# kernel filter
filter_kernel = np.array([[[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]])
filter_kernel = torch.from_numpy(filter_kernel.copy()).float().unsqueeze(0)
print("Shape of Filter Kernel: ", filter_kernel.shape)

# plot the kernel filter
plt.imshow(filter_kernel.squeeze(0).permute(1, 2, 0).numpy())
plt.show()

# Convolution
output_conv = my_conv2d(torch.from_numpy(img.copy()).float().permute(2, 0, 1).unsqueeze(0), filter_kernel, activation='relu')
output_conv = output_conv.squeeze(0).permute(1, 2, 0).numpy()
print("Output of Convolution")
plt.imshow(output_conv, cmap='gray')
plt.show()
print("Shape of Output Image: ", output_conv.shape)
```




```
Shape of Image:  (256, 256, 3)
Shape of Filter Kernel:  torch.Size([1, 1, 3, 3])
```



Output of Convolution




```
Shape of Output Image:  (254, 254, 1)
```

Q2

Pooling Function

In [ ]:

```python
# Pooling Function implementation
def my_pool(input, stride=None, pool_type='max', kernel_size=2, ceil_mode=False):
    # input is a 4D tensor of shape (N, C_in, H_in, W_in)
    # stride is an integer
    # pool_type is a string
    # kernel_size is (H, W) or an integer

    # Calculating output shape
    if isinstance(kernel_size, int):
        kernel_size = (kernel_size, kernel_size)
    else:
        kernel_size = kernel_size
```

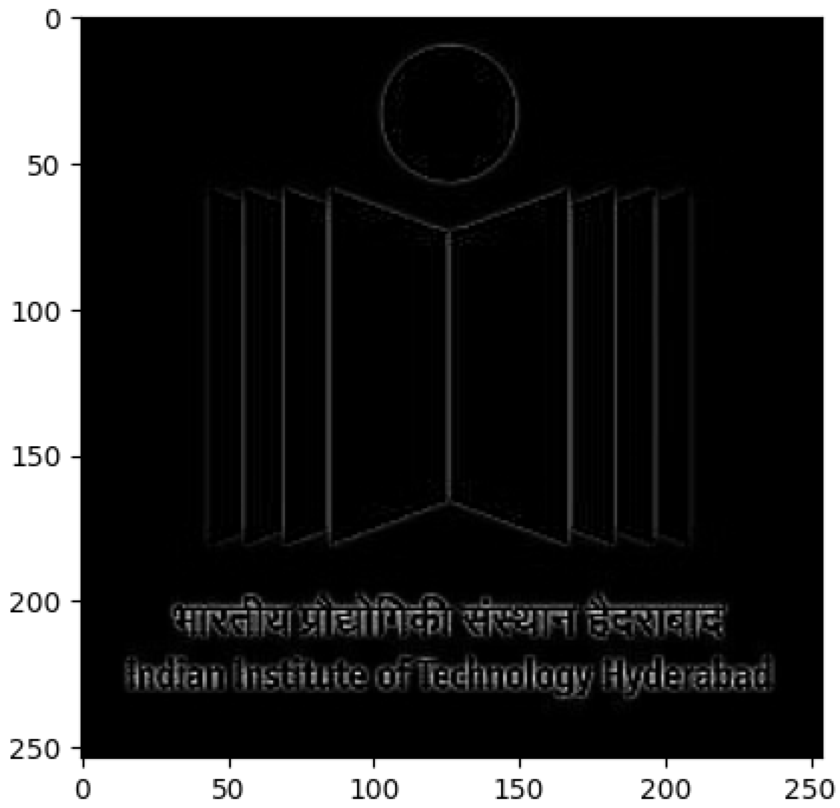
```python
    if stride is None:
        stride = kernel_size
    if isinstance(stride, int):
        stride = (stride, stride)

    if ceil_mode:
        H_out = math.ceil((input.shape[2] - kernel_size[0])/stride[0]) + 1
        W_out = math.ceil((input.shape[3] - kernel_size[1])/stride[1]) + 1
    else:
        H_out = (input.shape[2] - kernel_size[0])//stride[0] + 1
        W_out = (input.shape[3] - kernel_size[1])//stride[1] + 1

    # Initializing output tensor
    output = torch.zeros((input.shape[0], input.shape[1], H_out, W_out))

    # Applying pooling
    for i in range(H_out):
        for j in range(W_out):
            if pool_type == 'max':
                output[:, :, i, j] = torch.amax(input[:, :, i*stride[0]:i*stride[0]+kernel_size[0],
                                                        j*stride[1]:j*stride[1]+kernel_size[1]],
                                                  dim=(2, 3))[0]
            elif pool_type == 'avg':
                output[:, :, i, j] = torch.mean(input[:, :, i*stride[0]:i*stride[0]+kernel_size[0],
                                                        j*stride[1]:j*stride[1]+kernel_size[1]],
                                                  dim=(2, 3))
            else:
                print("Invalid pooling type")

    return output
```

In [ ]:
```python
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
output = my_pool(input)
print(output.size())

# cross verification with pytorch
input = torch.empty(128, 3, 20, 20).normal_()
output = nn.functional.max_pool2d(input, kernel_size=2, stride=None)
print(output.size())
```

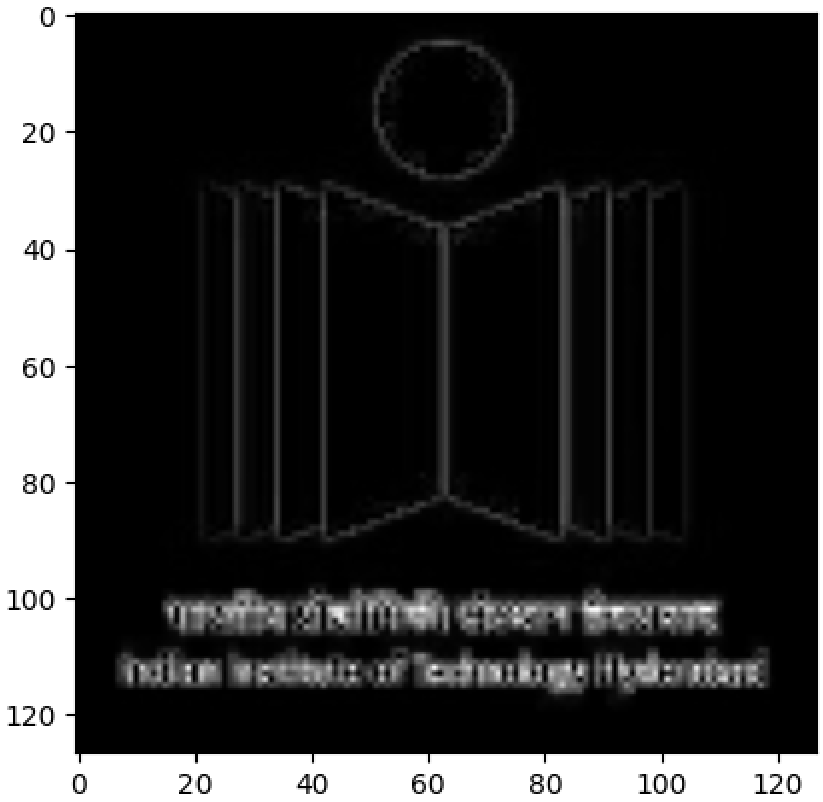
```
torch.Size([128, 3, 10, 10])
torch.Size([128, 3, 10, 10])
```

In [ ]:

```
# Display the image and the output image
# Load the image
plt.imshow(output_conv, cmap='gray')
plt.show()
print("Shape of Image: ", output_conv.shape)
# Pooling
# print(output_conv.dtype)
output = my_pool(torch.from_numpy(output_conv).float().permute(2, 0, 1).unsqueeze(0), pool_type='max')
output = output.squeeze(0).permute(1, 2, 0).numpy()
print("Output of Pooling")
plt.imshow(output, cmap='gray')
plt.show()
print("Shape of Output Image: ", output.shape)
```




```
Shape of Image:  (254, 254, 1)
Output of Pooling
```




```
Shape of Output Image:  (127, 127, 1)
```

Q3

Convolution Layer Class

In [ ]:

```python
# Convolution Layer implementation
class my_Conv2d(nn.Module):
    def __init__(self, in_channels, num_filters, kernel_size, stride=1, padding=0, bias=True, activation='relu'):
        super(my_Conv2d, self).__init__()
        self.in_channels = in_channels
        self.num_filters = num_filters
        if isinstance(kernel_size, int):
            self.kernel_size = (kernel_size, kernel_size)
        else:
            self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
```

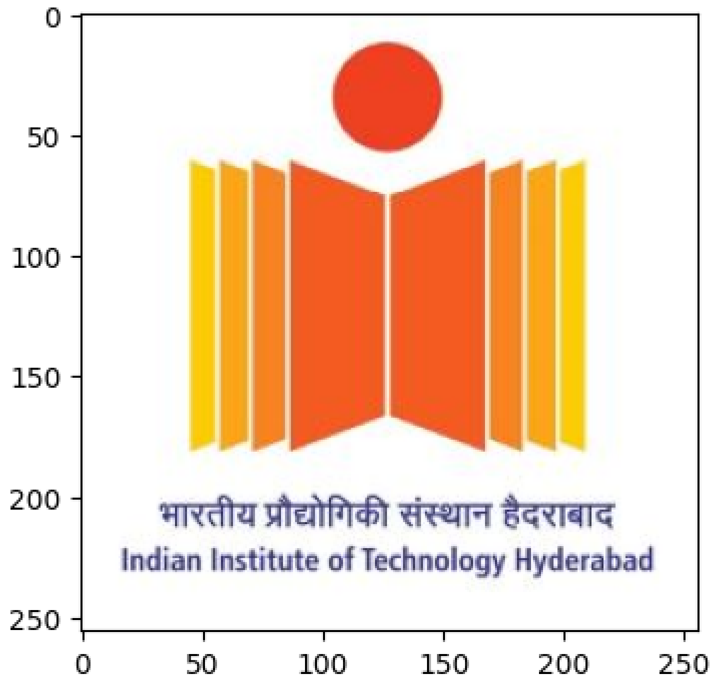
```python
        self.bias = bias
        self.activation = activation

        # self.weight = nn.Parameter(torch.empty(self.num_filters, self.in_channels,
        #                                         self.kernel_size[0], self.kernel_size[1]).normal_())
        self.weight = torch.empty(self.num_filters, self.in_channels,
                                          self.kernel_size[0], self.kernel_size[1]).normal_()
        if bias:
            # self.bias = nn.Parameter(torch.empty(num_filters).normal_())
            self.bias = torch.empty(num_filters).normal_()
        else:
            self.bias = None
        # print("Weight: ", self.weight)
    def forward(self, input):
        return my_conv2d(input=input, weight=self.weight, bias=self.bias, activation=self.activation)
```

In [ ]:
```python
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
conv = my_Conv2d(in_channels=3, num_filters=5, kernel_size=5)
output = conv(input)
print(output.size())

# cross verification with pytorch
input = torch.empty(128, 3, 20, 20).normal_()
conv = nn.Conv2d(3, 5, 5)
output = conv(input)
print(output.size())
```

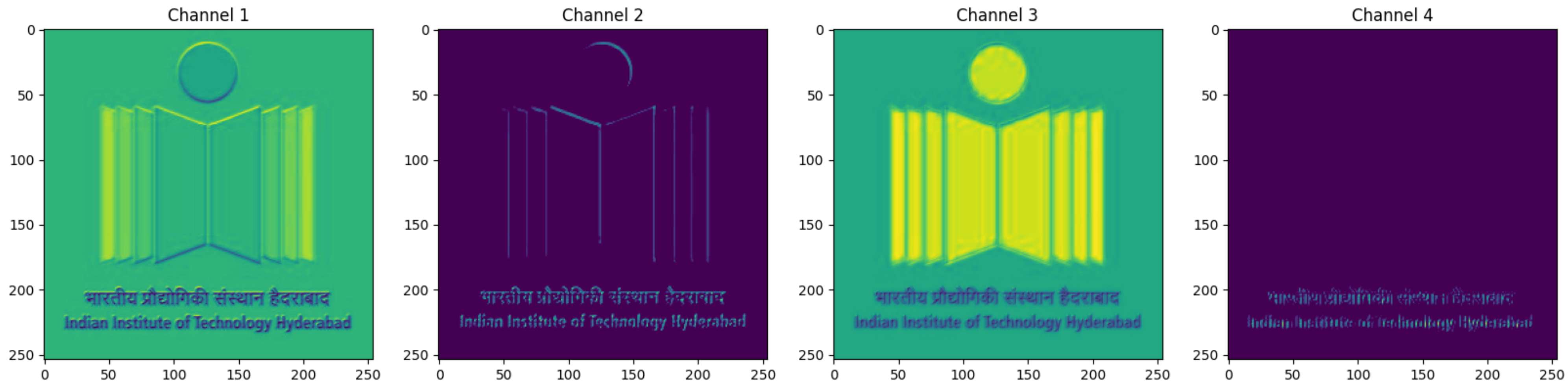
```
torch.Size([128, 5, 16, 16])
torch.Size([128, 5, 16, 16])
```

In [ ]:
```python
# Display the image and the output image
# Load the image
img = plt.imread('logo2.jpg')
plt.figure(figsize=(4, 4))
plt.imshow(img)
plt.show()
print("Shape of Image: ",img.shape)

# Convolution
num_filters = 4
Conv2D = my_Conv2d(in_channels=3, num_filters=num_filters, kernel_size=3)
output_conv = Conv2D(torch.from_numpy(img.copy()).float().permute(2, 0, 1).unsqueeze(0))
# print(type(output_conv))
output_conv = output_conv.squeeze(0).permute(1, 2, 0).detach().float().numpy()
output_conv = (output_conv/np.max(output_conv)).astype(np.float32)
```

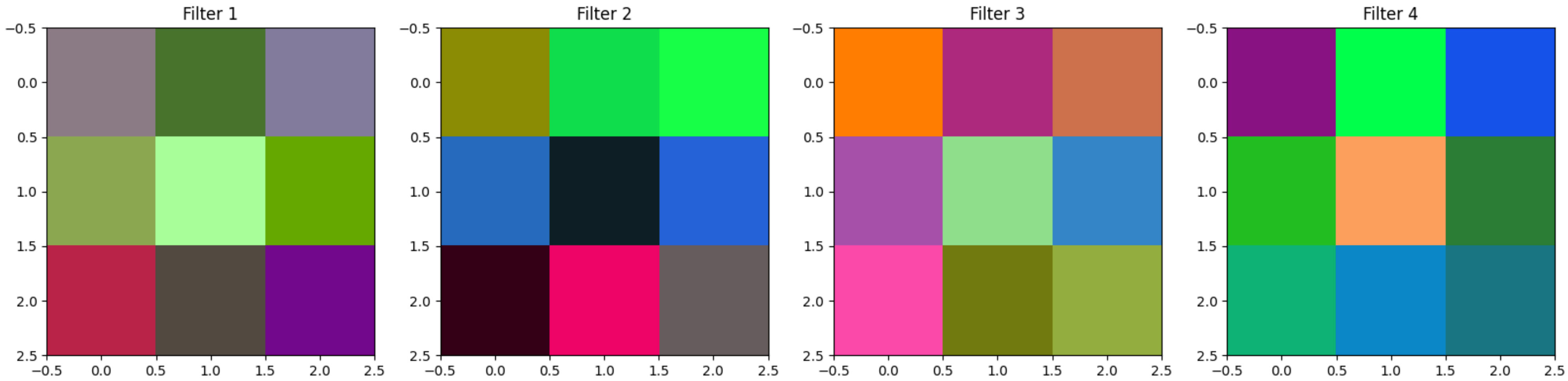
```
output_conv = (output_conv*255).astype(np.uint8)
print(np.max(output_conv), np.min(output_conv))

# create subfigure and plot output image and for each channel
fig, ax = plt.subplots(1, num_filters, figsize=(20, 5))
for i in range(num_filters):
    ax[i].imshow(output_conv[:, :, i])
    ax[i].set_title('Channel {}'.format(i+1))
plt.show()

# plt.imshow(output_conv)
# plt.show()
# print("Shape of Output Image: ", output_conv.shape)

# Filter Kernel
# print("Filter Kernel: ", Conv2D.weight)
print("Shape of Filter Kernel: ", Conv2D.weight.shape)
print("Bias: ", Conv2D.bias)

# plot the filter kernel
fig, ax = plt.subplots(1, num_filters, figsize=(20, 5))
for i in range(num_filters):
    # normalize the filter kernel
    filter_kernel = Conv2D.weight[i].detach().numpy()
    filter_kernel = (filter_kernel - np.min(filter_kernel))/(np.max(filter_kernel) - np.min(filter_kernel))
    ax[i].imshow(filter_kernel.transpose(1, 2, 0))
    ax[i].set_title('Filter {}'.format(i+1))
plt.show()
```




```
Shape of Image:  (256, 256, 3)
255 0
```




```
Shape of Filter Kernel:  torch.Size([4, 3, 3, 3])
Bias:  tensor([-0.4341, -0.6207,  0.3405, -0.2142])
```



Q4

Pooling Layer Class

In [ ]:

```
# Pooling
class my_Pool2d(nn.Module):
    def __init__(self, kernel_size, stride=None, pool_type='max', ceil_mode=False):
        super(my_Pool2d, self).__init__()
        if isinstance(kernel_size, int):
            self.kernel_size = (kernel_size, kernel_size)
        else:
            self.kernel_size = kernel_size
        if stride is None:
            self.stride = kernel_size
        else:
            self.stride = stride
        self.pool_type = pool_type
        self.ceil_mode = ceil_mode

    def forward(self, input):
        if self.pool_type == 'gap':
            return torch.mean(input, dim=(2, 3)).unsqueeze(2).unsqueeze(3)

        return my_pool(input=input, stride=self.stride, pool_type=self.pool_type,
                       kernel_size=self.kernel_size, ceil_mode=self.ceil_mode)
```

In [ ]:

```
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
print("Input Size: ", input.size())
pool = my_Pool2d(kernel_size=2, pool_type='max')
output = pool(input)
print("Output Size: ", output.size())

# # cross verification with pytorch
# input = torch.empty(128, 3, 20, 20).normal_()
# pool = nn.MaxPool2d(kernel_size=2, stride=None)
# output = pool(input)
# print(output.size())
```

```
Input Size:  torch.Size([128, 3, 20, 20])
Output Size:  torch.Size([128, 3, 10, 10])
```

Q5

Flattening Function

In [ ]:

```
# Flatten Function implementation
def my_flatten(input, weight=False, output_size=None):
    if weight:
        flatten_input = input.view(input.shape[0], -1)
        if output_size is None:
            output_size = flatten_input.shape[1]
        weight = torch.empty(flatten_input.shape[1], output_size).normal_()
        return torch.matmul(flatten_input, weight)
    else:
        return input.view(input.shape[0], -1)
```

In [ ]:

```
# Testing the function
input = torch.empty(2, 3, 20, 20).normal_()
output = my_flatten(input, weight=True)
print(output.size())

# cross verification with pytorch
input = torch.empty(2, 3, 20, 20).normal_()
output = nn.Flatten()(input)
print(output.size())
```

```
torch.Size([2, 1200])
torch.Size([2, 1200])
```

Q6

Multi Layer Perceptron Class

In [ ]:

```
# MLP Layer implementation

activation_dict = {
    'relu': nn.ReLU(),
    'sigmoid': nn.Sigmoid(),
    'tanh': nn.Tanh(),
    'softmax': nn.Softmax(),
    'none': nn.Identity(),
    'prelu': nn.PReLU(),
}

class MLP(nn.Module):
    def __init__(self, hidden_layers, output_size, input_size=8, softmax=False):
        super(MLP, self).__init__()
        self.hidden_layers = hidden_layers
        self.output_size = output_size
        self.input_size = input_size
        # print(self.hidden_layers)

        self.layers = nn.ModuleList()
        for i in range(len(hidden_layers)):
            if i == 0:
                self.layers.append(nn.Linear(self.input_size, hidden_layers[i][0]))
                # print("Hidden: ", hidden_layers[i][0], "Activation: ", hidden_layers[i][1])
                # initialize the weights and bias
                # nn.init.kaiming_normal_(self.layers[-1].weight)
                # if self.layers[-1].bias is not None:
                #     nn.init.normal_(self.layers[-1].bias)
                self.layers.append(activation_dict[hidden_layers[i][1]])
            else:
                self.layers.append(nn.Linear(hidden_layers[i-1][0], hidden_layers[i][0]))
                # print("Hidden: ", hidden_layers[i][0], "Activation: ", hidden_layers[i][1])
                # initialize the weights and bias
                # nn.init.kaiming_normal_(self.layers[-1].weight)
                # if self.layers[-1].bias is not None:
                #     nn.init.normal_(self.layers[-1].bias)
                # self.layers.append(activation_dict[hidden_layers[i][1]])
        self.layers.append(nn.Linear(hidden_layers[-1][0], output_size))
```

```
        # initialize the weights and bias
        # nn.init.kaiming_normal_(self.layers[-1].weight)
        # if self.layers[-1].bias is not None:
        #     nn.init.normal_(self.layers[-1].bias)

        if softmax:
            self.layers.append(nn.Softmax(dim=1))
        # Initialize the weights
        for layer in self.layers:
            if isinstance(layer, nn.Linear):
                nn.init.xavier_normal_(layer.weight)
                if layer.bias is not None:
                    nn.init.normal_(layer.bias)
        # print("Layers: ", self.layers)

    def forward(self, input):
        # may need to flatten the input
        for layer in self.layers:
            # print("MLP: ", input.shape)
            input = layer(input)

        return input
```

In [ ]:

```
# Testing MLP Layer
mlp_input = torch.empty(128, 3, 20, 20).normal_()
print("Input Size: ", mlp_input.size())
mlp = MLP(hidden_layers=[(100, 'relu'), (50, 'relu')], output_size=10, input_size=3*20*20, softmax=True)
output = mlp(my_flatten(mlp_input))
print("Output Size with Softmax: ", output.size())

mlp = MLP(hidden_layers=[(100, 'relu'), (50, 'relu')], output_size=10, input_size=3*20*20, softmax=False)
output = mlp(my_flatten(mlp_input))
print("Output Size without Softmax: ", output.size())
```

```
Input Size:  torch.Size([128, 3, 20, 20])
Output Size with Softmax:  torch.Size([128, 10])
Output Size without Softmax:  torch.Size([128, 10])
```

Q7

Feed Forward Function

In [ ]:

```
# CNN class implementation with the above classes of Conv2d, Pool2d and MLP
class my_CNN(nn.Module):
    def __init__(self, conv_layers, pool_layers, hidden_layers, output_size, softmax=False):
        super(my_CNN, self).__init__()
        self.conv_layers = conv_layers
        self.pool_layers = pool_layers
        self.hidden_layers = hidden_layers
        self.output_size = output_size
        self.softmax = softmax

        self.conv = nn.ModuleList()
        for i in range(len(conv_layers)):
            self.conv.append(my_Conv2d(in_channels=conv_layers[i][0], num_filters=conv_layers[i][1],
                                       kernel_size=conv_layers[i][2], stride=conv_layers[i][3],
                                       padding=conv_layers[i][4], bias=conv_layers[i][5],
                                       activation=conv_layers[i][6]))

        self.pool = nn.ModuleList()
        for i in range(len(pool_layers)):
            self.pool.append(my_Pool2d(kernel_size=pool_layers[i][0], stride=pool_layers[i][1],
                                       pool_type=pool_layers[i][2], ceil_mode=pool_layers[i][3]))

        self.pool_gap = my_Pool2d(kernel_size=1, pool_type='gap')
        self.mlp = MLP(hidden_layers=self.hidden_layers, output_size=self.output_size, softmax=self.softmax)

    def forward(self, input):
        for i in range(len(self.conv)):
            input = self.conv[i](input)
            # print(input.size())
            input = self.pool[i](input)
            # print(input.size())
        input = self.pool_gap(input)
        # print(input.size())
        input = my_flatten(input)
        # print("Size of MLP input: ", input.size())
        input = self.mlp(input)
        # print(input.size())
        return input
```

In [ ]:

```
# Test the CNN class
input = torch.empty(1, 3, 32, 32).normal_()
print("Input Size: ", input.size())

'''
```

```
• Input image of size 32 × 32 × 3. Use images from the CIFAR-10 dataset.
• Convolution layer with 16 kernels of size 3 × 3 spatial dimensions and sigmoid activation.
• Max pooling layer of size 2 × 2 with a stride of 2 along each dimension.
• Convolution layer with 8 kernels of spatial size 3 × 3 and sigmoid activation.
• Max pooling layer of size 2 × 2 with a stride of 2 along each dimension.
• A Global Average Pooling (GAP) layer.
• An MLP with one hidden layer (size same as input) that accepts as input the previous layer's
output and maps it to 10 output nodes. Use sigmoid activation for the MLP (softmax in the
o/p layer).
'''

cnn = my_CNN(conv_layers=[(3, 16, 3, 1, 0, True, 'sigmoid'), (16, 8, 3, 1, 0, True, 'sigmoid')],
             pool_layers=[(2, 2, 'max', False), (2, 2, 'max', False)],
             hidden_layers=[(8, 'sigmoid')],
             output_size=10,
             softmax=True)

output = cnn(input)
print("Output Size: ", output.size())
print(output)
```

```
Input Size:  torch.Size([1, 3, 32, 32])
Output Size:  torch.Size([1, 10])
tensor([[0.1738, 0.0264, 0.0149, 0.0487, 0.1022, 0.1210, 0.0697, 0.0227, 0.3435,
         0.0771]], grad_fn=<SoftmaxBackward0>)
```

In [ ]:

```
# Load the CIFAR-10 dataset
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose([transforms.ToTensor()])
# trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
# trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)

tmp = next(iter(testloader))
print("Input Size: ", tmp[0].size())
output = cnn(tmp[0])
print("Output Size: ", output.size())
```

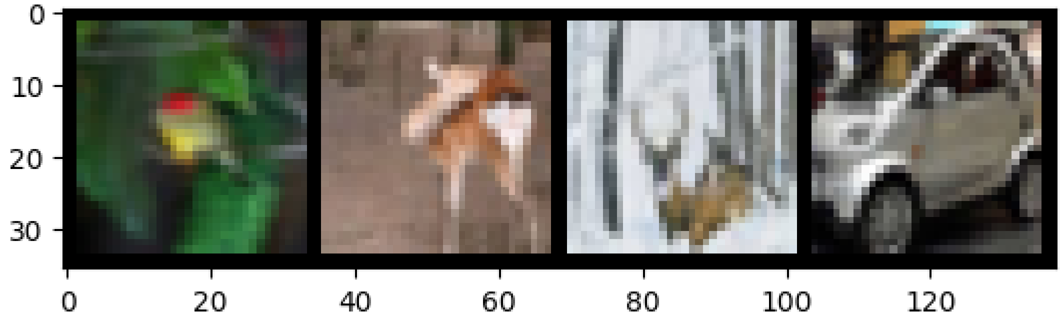
```
Files already downloaded and verified
Input Size:  torch.Size([4, 3, 32, 32])
Output Size:  torch.Size([4, 10])
```

Q8

In [ ]:

```
# Choose an image from each class and print the predicted class label for each image.
import numpy as np
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose([transforms.ToTensor()])
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=True, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# get some random training images
dataiter = iter(testloader)
images, labels = dataiter.next()

# show images
print("Images Input: ", images.size())
plt.imshow(torchvision.utils.make_grid(images).permute(1, 2, 0))
plt.show()
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))

# print images
for i in range(images.size()[0]):
    print("Image: ", i)
    # plt.imshow(images[i].permute(1, 2, 0))
    # plt.show()
    output = cnn(images[i].unsqueeze(0))
    print("Output: ", output)
    # print("Predicted: ", classes[torch.argmax(output)])
```

```
Files already downloaded and verified
Images Input:  torch.Size([4, 3, 32, 32])
```



```
GroundTruth:   bird  deer  deer   car
Image:  0
Output:  tensor([[0.1742, 0.0273, 0.0148, 0.0488, 0.1013, 0.1278, 0.0684, 0.0235, 0.3326,
         0.0814]], grad_fn=<SoftmaxBackward0>)
Image:  1
Output:  tensor([[0.1762, 0.0275, 0.0150, 0.0501, 0.1029, 0.1269, 0.0673, 0.0235, 0.3295,
         0.0811]], grad_fn=<SoftmaxBackward0>)
Image:  2
Output:  tensor([[0.1762, 0.0276, 0.0149, 0.0505, 0.1038, 0.1265, 0.0674, 0.0235, 0.3280,
         0.0814]], grad_fn=<SoftmaxBackward0>)
Image:  3
Output:  tensor([[0.1750, 0.0272, 0.0150, 0.0488, 0.1006, 0.1280, 0.0677, 0.0234, 0.3333,
         0.0809]], grad_fn=<SoftmaxBackward0>)
```

All of the output values are almost same for individual classes.

Randomly initialized weights do not show any pattern and do not correspond to any class.

In [ ]:

```
# Choose 3 images from each class and print the predicted class label for each image.
import numpy as np
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose([transforms.ToTensor()])
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

num_img = 3
selected_indices = []
for i in range(10):
    selected_indices.append(np.where(np.array(testset.targets) == i)[0][:num_img])
selected_indices = np.array(selected_indices).flatten()
print("Selected Indices: ", selected_indices)
```

```
Files already downloaded and verified
Selected Indices:  [ 3 10 21  6  9 37 25 35 65  0  8 46 22 26 32 12 16 24  4  5  7 13 17 20
  1  2 15 11 14 23]
```

In [ ]:

```
# display images from selected indices
images = []
labels = []
for i in selected_indices:
    images.append(testset[i][0])
    labels.append(testset[i][1])
```

```
images = torch.stack(images)
labels = torch.tensor(labels)

# show images
print("Image Size: ", images.size())
plt.imshow(torchvision.utils.make_grid(images, nrow=6).permute(1, 2, 0))
plt.show()
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(len(labels))))

output_img = torch.tensor([])
# Finding Outut for each image after passing through CNN and Pooling layers including GAP and then Flatten
for img_num in range(images.size()[0]):
    tmp = images[img_num].unsqueeze(0)
    for i in range(len(cnn.conv)):
        tmp = cnn.conv[i](tmp)
        tmp = cnn.pool[i](tmp)
    tmp = cnn.pool_gap(tmp)
    tmp = my_flatten(tmp)
    output_img = torch.cat((output_img, tmp), 0)
    # print("Output Size: ", tmp.size())
    print("Output for image {}: {}".format(img_num, tmp))
    print("Predicted: ", classes[torch.argmax(tmp)])
print("Output Size: ", output_img.size())


# for i in range(len(cnn.conv)):
#     images = cnn.conv[i](images)
#     images = cnn.pool[i](images)
# images = cnn.pool_gap(images)
# output = my_flatten(images)
# print("Output Size: ", output.size())

# Apply PCA to output and plot it
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca.fit(output_img.detach().numpy())
output_pca = pca.transform(output_img.detach().numpy())
print("PCA Output Shape: ", output_pca.shape)

plt.figure(figsize=(10, 10))
plt.scatter(output_pca[:, 0], output_pca[:, 1], c=labels, cmap='tab10')
plt.colorbar()
plt.title("PCA of CNN Bottleneck Layer Output")
plt.xlabel("PC1")
plt.ylabel("PC2")
```

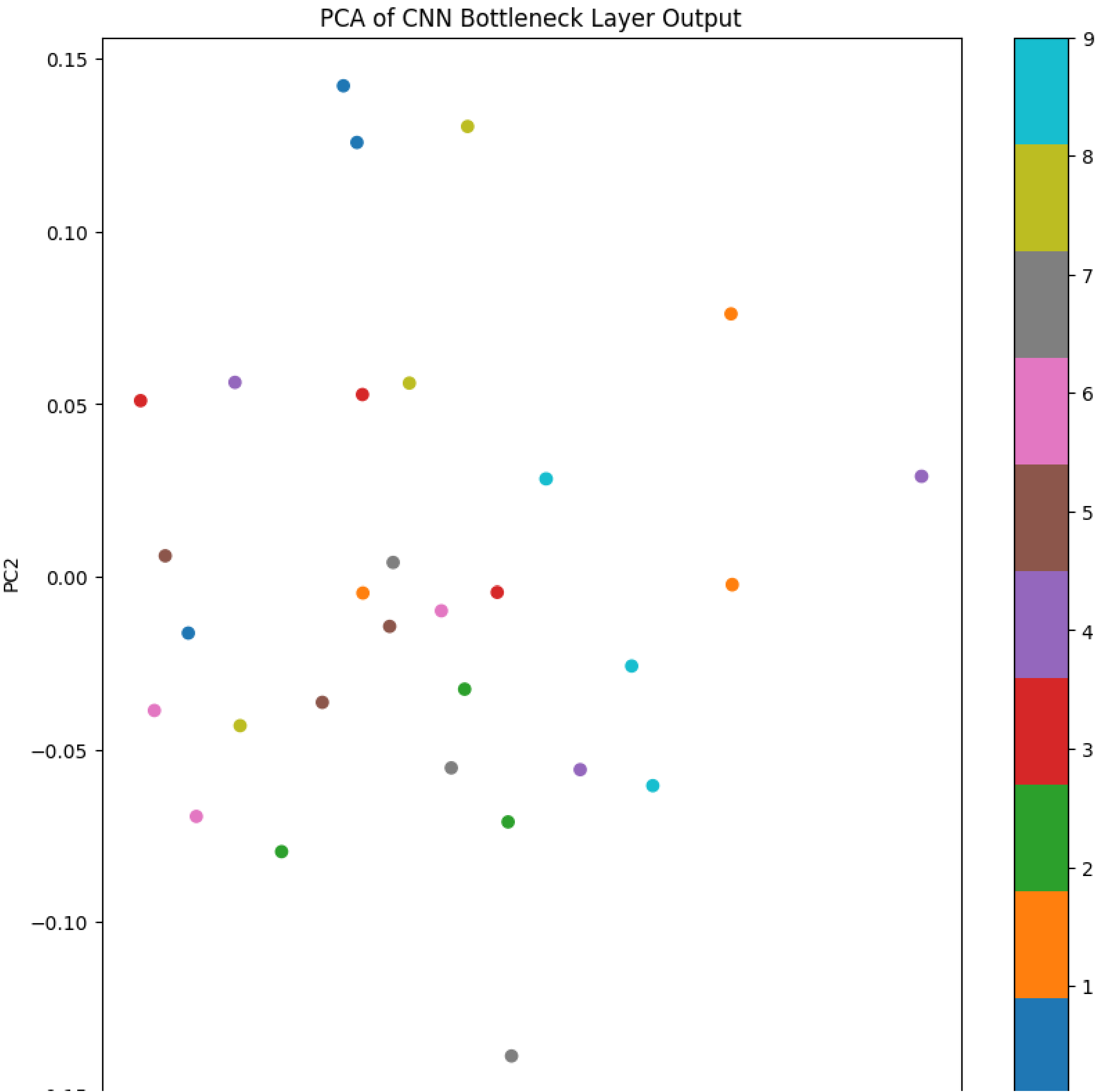
```
plt.show()
print("Output PCA: \n", output_pca)
```

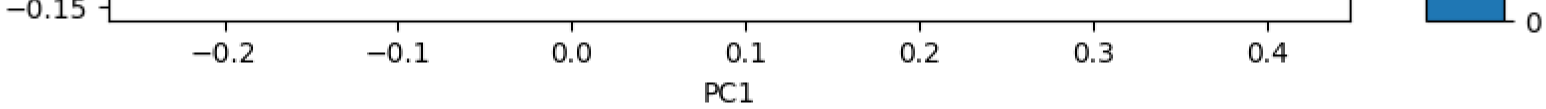
```
Image Size:  torch.Size([30, 3, 32, 32])
```

```
GroundTruth:  plane plane plane   car   car   car  bird  bird  bird   cat   cat   cat  deer  deer  deer   dog   dog   dog  frog  frog  frog
horse horse horse  ship  ship  ship truck truck truck
Output for image 0: tensor([[4.3414e-01, 5.0364e-02, 9.9685e-01, 1.0267e-02, 1.1989e-07, 9.9723e-01,
         4.2871e-01, 1.0000e+00]])
Predicted:  horse
Output for image 1: tensor([[4.5221e-01, 3.7022e-02, 9.9259e-01, 2.2612e-03, 1.0517e-07, 9.9854e-01,
         4.3707e-01, 1.0000e+00]])
Predicted:  horse
Output for image 2: tensor([[5.6683e-01, 5.2266e-02, 9.9758e-01, 2.6284e-02, 1.4503e-06, 9.9594e-01,
         2.7314e-01, 1.0000e+00]])
Predicted:  horse
Output for image 3: tensor([[6.3396e-01, 5.1083e-02, 9.9557e-01, 2.9655e-02, 5.7203e-07, 9.9603e-01,
         7.2071e-01, 1.0000e+00]])
Predicted:  horse
Output for image 4: tensor([[5.5673e-01, 5.8294e-02, 9.9497e-01, 1.4717e-02, 4.7422e-07, 9.9530e-01,
         7.3384e-01, 9.9999e-01]])
Predicted:  horse
Output for image 5: tensor([[5.8161e-01, 6.7017e-02, 9.9759e-01, 1.2670e-02, 1.7073e-07, 9.9874e-01,
         4.1826e-01, 1.0000e+00]])
Predicted:  horse
Output for image 6: tensor([[6.2373e-01, 2.1828e-02, 9.8829e-01, 2.6419e-03, 3.5826e-07, 9.9586e-01,
         4.9702e-01, 1.0000e+00]])
Predicted:  horse
Output for image 7: tensor([[6.6810e-01, 3.1323e-02, 9.8966e-01, 5.0672e-03, 4.9562e-07, 9.9317e-01,
         5.2557e-01, 1.0000e+00]])
Predicted:  horse
Output for image 8: tensor([[6.4293e-01, 3.0012e-02, 9.9128e-01, 1.4166e-03, 1.2893e-07, 9.9821e-01,
         3.3873e-01, 1.0000e+00]])
Predicted:  horse
Output for image 9: tensor([[6.0123e-01, 4.5445e-02, 9.9393e-01, 4.4024e-03, 2.0968e-07, 9.9788e-01,
         5.2857e-01, 1.0000e+00]])
Predicted:  horse
Output for image 10: tensor([[4.9363e-01, 3.7452e-02, 9.9723e-01, 1.8446e-03, 5.4732e-08, 9.9965e-01,
         2.4692e-01, 1.0000e+00]])
Predicted:  horse
Output for image 11: tensor([[5.2486e-01, 4.6530e-02, 9.9488e-01, 2.5965e-03, 3.5058e-07, 9.9414e-01,
         4.2840e-01, 1.0000e+00]])
Predicted:  horse
Output for image 12: tensor([[6.3068e-01, 2.0296e-02, 9.6441e-01, 2.0893e-03, 5.5651e-07, 9.7960e-01,
         8.8115e-01, 9.9999e-01]])
Predicted:  horse
Output for image 13: tensor([[6.6390e-01, 2.7380e-02, 9.9052e-01, 1.7396e-03, 3.1023e-07, 9.9644e-01,
         5.8721e-01, 1.0000e+00]])
Predicted:  horse
Output for image 14: tensor([[5.0230e-01, 2.7282e-02, 9.9416e-01, 1.0354e-03, 5.4245e-08, 9.9951e-01,
         3.2495e-01, 1.0000e+00]])
```

```
Predicted:  horse
Output for image 15: tensor([[6.0658e-01, 4.8518e-02, 9.9349e-01, 2.5704e-03, 1.1698e-07, 9.9760e-01,
         3.7963e-01, 1.0000e+00]])
Predicted:  horse
Output for image 16: tensor([[5.9479e-01, 2.9396e-02, 9.9652e-01, 1.2131e-02, 1.5298e-07, 9.9900e-01,
         4.3883e-01, 1.0000e+00]])
Predicted:  horse
Output for image 17: tensor([[5.4155e-01, 4.0337e-02, 9.9563e-01, 1.8727e-03, 4.6107e-08, 9.9955e-01,
         2.5902e-01, 1.0000e+00]])
Predicted:  horse
Output for image 18: tensor([[5.8376e-01, 2.4802e-02, 9.9165e-01, 6.0195e-04, 7.2633e-08, 9.9910e-01,
         2.4223e-01, 1.0000e+00]])
Predicted:  horse
Output for image 19: tensor([[5.9803e-01, 2.4451e-02, 9.9608e-01, 3.0636e-03, 2.9769e-07, 9.9829e-01,
         4.8216e-01, 1.0000e+00]])
Predicted:  horse
Output for image 20: tensor([[6.2004e-01, 1.4634e-02, 9.8767e-01, 4.1819e-04, 5.5396e-08, 9.9914e-01,
         2.7087e-01, 1.0000e+00]])
Predicted:  horse
Output for image 21: tensor([[7.3552e-01, 8.5047e-02, 9.9346e-01, 1.8233e-02, 8.1474e-07, 9.7799e-01,
         5.1534e-01, 1.0000e+00]])
Predicted:  horse
Output for image 22: tensor([[5.7724e-01, 3.4012e-02, 9.9496e-01, 3.4929e-03, 1.3781e-07, 9.9914e-01,
         4.4503e-01, 1.0000e+00]])
Predicted:  horse
Output for image 23: tensor([[6.4438e-01, 4.2707e-02, 9.9318e-01, 9.5869e-03, 3.6571e-07, 9.9739e-01,
         4.8180e-01, 1.0000e+00]])
Predicted:  horse
Output for image 24: tensor([[5.2850e-01, 7.8311e-02, 9.9484e-01, 3.0829e-02, 5.2844e-07, 9.7094e-01,
         4.6630e-01, 1.0000e+00]])
Predicted:  horse
Output for image 25: tensor([[4.6417e-01, 4.6265e-02, 9.9140e-01, 4.2241e-03, 1.2803e-07, 9.9755e-01,
         5.2834e-01, 1.0000e+00]])
Predicted:  horse
Output for image 26: tensor([[6.0115e-01, 6.4318e-02, 9.9351e-01, 2.8078e-03, 7.2568e-08, 9.9953e-01,
         3.1097e-01, 1.0000e+00]])
Predicted:  horse
Output for image 27: tensor([[5.7620e-01, 4.5768e-02, 9.9509e-01, 9.3292e-03, 2.5681e-07, 9.9632e-01,
         5.7424e-01, 1.0000e+00]])
Predicted:  horse
Output for image 28: tensor([[6.4221e-01, 5.1702e-02, 9.9099e-01, 6.8759e-03, 3.4800e-07, 9.9726e-01,
         6.3440e-01, 1.0000e+00]])
Predicted:  horse
Output for image 29: tensor([[6.7940e-01, 4.3626e-02, 9.9555e-01, 1.7938e-02, 3.8159e-07, 9.9736e-01,
         6.4552e-01, 1.0000e+00]])
Predicted:  horse
```

```
Output Size:  torch.Size([30, 8])
PCA Output Shape:  (30, 2)
```


PCA of CNN Bottleneck Layer Output
PC2
0.15
0.10
0.05
0.00
−0.05
−0.10
9
8
7
6
5
4
3
2
1




```
Output PCA:
 [[-0.06587367  0.14213507]
 [-0.0546109   0.12571694]
 [-0.19488707 -0.01623771]
 [ 0.25733185 -0.00225963]
 [ 0.25631362  0.07615073]
 [-0.04965954 -0.00463657]
 [ 0.03497866 -0.03249705]
 [ 0.07112905 -0.07096186]
 [-0.11729828 -0.07957168]
 [ 0.06208555 -0.00443973]
 [-0.23431931  0.05100163]
 [-0.05003345  0.0527735 ]
 [ 0.41467574  0.02908562]
 [ 0.13083877 -0.05581112]
 [-0.15607749  0.05630541]
 [-0.08341673 -0.03631153]
 [-0.02740643 -0.01429895]
 [-0.21380387  0.00604226]
 [-0.22285073 -0.03866892]
 [ 0.01562344 -0.00978049]
 [-0.18822758 -0.06936353]
 [ 0.07397202 -0.13867377]
 [-0.02451119  0.00412613]
 [ 0.02387436 -0.05529872]
 [-0.01099721  0.05608876]
 [ 0.03744108  0.13035062]
 [-0.1517974  -0.0430757 ]
 [ 0.10262038  0.02834837]
 [ 0.17364493 -0.02578699]
 [ 0.19124007 -0.06045297]]
```