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
```

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

    # 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
```

```python
    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 [ ]:

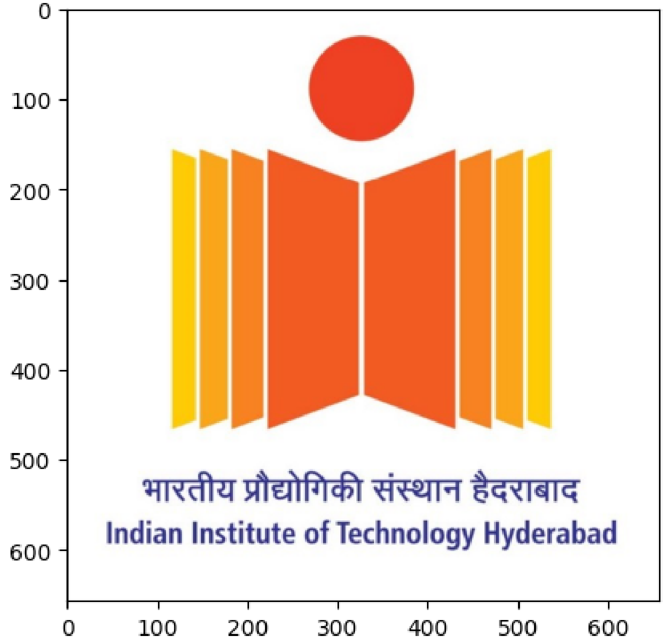
```python
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
weight = torch.empty(5, 3, 5, 5).normal_()
bias = torch.empty(5).normal_()
output = my_conv2d(input, weight, bias)
output.size()
```

Out[ ]:

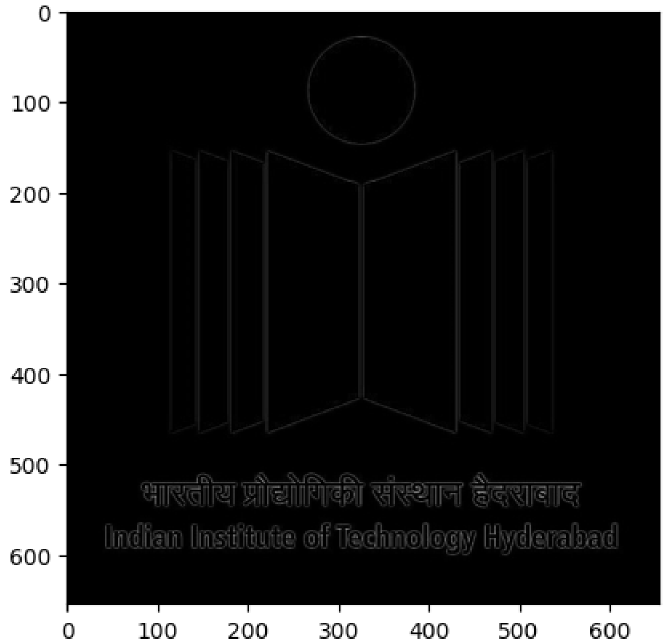
```
torch.Size([128, 5, 16, 16])
```

In [ ]:

```python
# Load the image and display it, display the filter kernel and display the output image
# Load the image
img = plt.imread('logo.jpg')
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).float().unsqueeze(0)
print("Shape of Filter Kernel: ", filter_kernel.shape)

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



Shape of Image: (657, 657, 3)
Shape of Filter Kernel: torch.Size([1, 1, 3, 3])



Shape of Output Image: (655, 655, 1)

Q2

Pooling Function

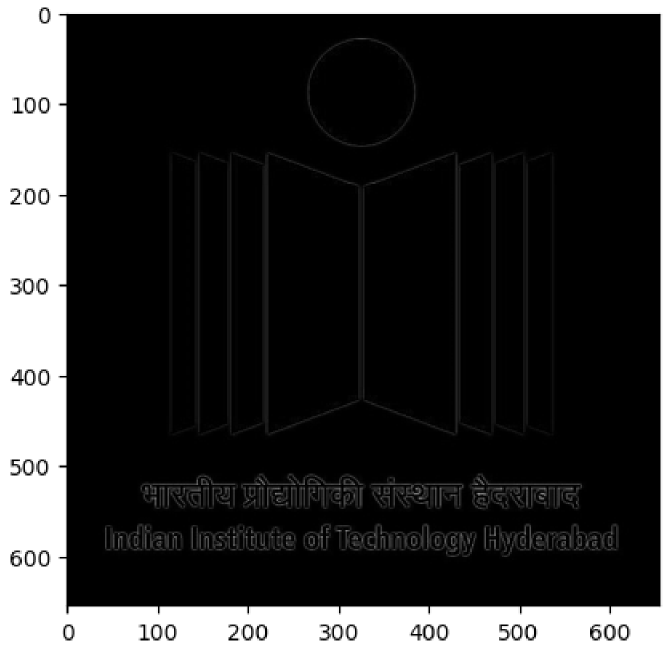
In [ ]:
```
# Pooling Function implementation
def my_pool(input, stride=None, pool_type='max', kernel_size=2, ceil_mode=False):
    # input is a 4D tensor of shape (N, C_in, H_in, W_in)
    # stride is an integer
    # pool_type is a string
    # kernel_size is (H, W) or an integer

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

    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
```

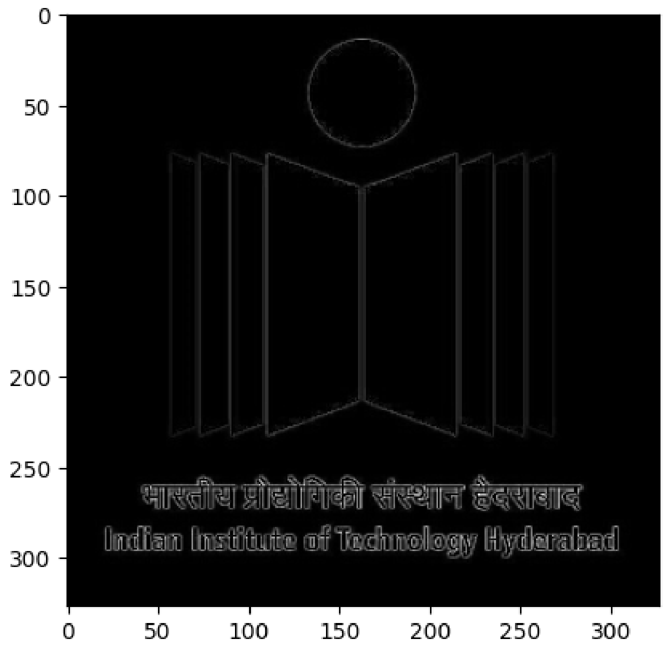
In [ ]:
```
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
output = my_pool(input)
print(output.size())

# 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()
# 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()
plt.imshow(output, cmap='gray')
plt.show()
print("Shape of Output Image: ", output.shape)
```






```
Shape of Output Image:  (327, 327, 1)
```

Q3

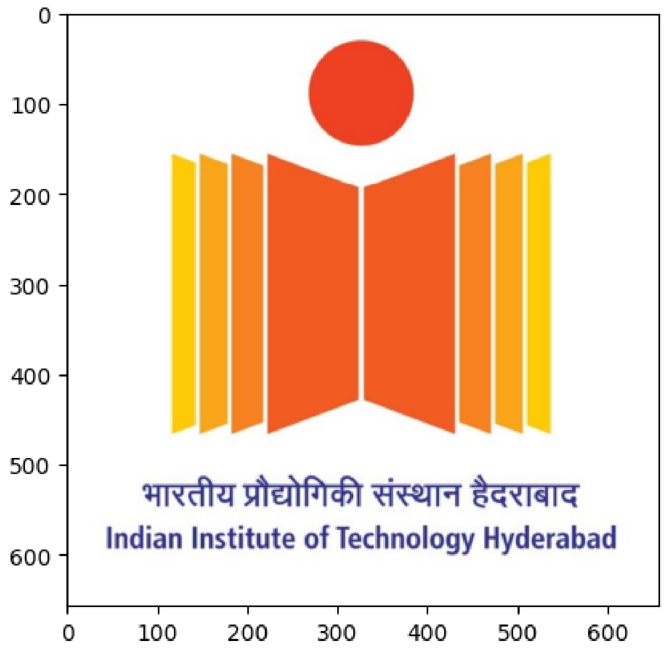
Convolution Layer Class

```
In [ ]:
# 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
        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_())
        if bias:
            self.bias = nn.Parameter(torch.empty(num_filters).normal_())
        else:
            self.bias = None

    def forward(self, input):
        return my_conv2d(input=input, weight=self.weight, bias=self.bias, activation=self.activation)
```

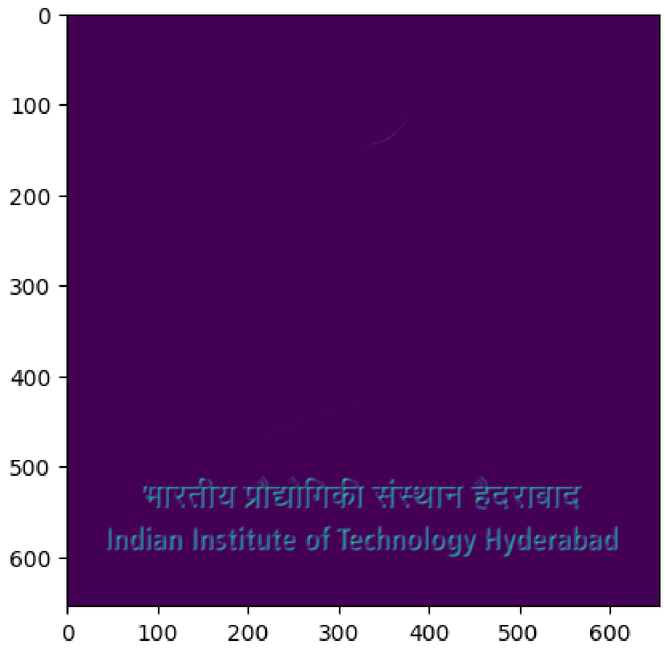
In [ ]:
```
# Testing the function
input = torch.empty(128, 3, 20, 20).normal_()
conv = my_Conv2d(in_channels=3, num_filters=5, kernel_size=5)
output = conv(input)
print(output.size())

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

# Convolution
Conv2D = my_Conv2d(in_channels=3, num_filters=1, kernel_size=3)
output_conv = Conv2D(torch.from_numpy(img).float().permute(2, 0, 1).unsqueeze(0))
output_conv = output_conv.squeeze(0).permute(1, 2, 0).detach().numpy()
plt.imshow(output_conv)
plt.show()
print("Shape of Output Image: ", output_conv.shape)
```




```
Shape of Image:  (657, 657, 3)
```




```
Shape of Output Image:  (655, 655, 1)
```

Q4

Pooling Layer Class

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

```python
pool = my_Pool2d(kernel_size=2, pool_type='max')
output = pool(input)
print(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())
```

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

Q5

Flattening Function

In [ ]:
```python
# Flatten Function implementation
def my_flatten(input, weight=False):
    if weight:
        weight = torch.empty(input.shape[0], input.shape[1]*input.shape[2]*input.shape[3]).normal_()
        return torch.matmul(input, weight.t())
    else:
        return input.view(input.shape[0], -1)
```

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

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

```
torch.Size([128, 1200])
torch.Size([128, 1200])
```

Q6

Multi Layer Perceptron Class

In [ ]:
```python
# MLP Layer implementation

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

```python
class MLP(nn.Module):
    def __init__(self, hidden_layers, output_size, softmax=False):
        super(MLP, self).__init__()
        self.hidden_layers = hidden_layers
        self.output_size = output_size

        self.layers = nn.ModuleList()
        for i in range(len(hidden_layers)):
            if i == 0:
                self.layers.append(nn.Linear(8, hidden_layers[i][0]))
                self.layers.append(activation_dict[hidden_layers[i][1]])
            else:
                self.layers.append(nn.Linear(hidden_layers[i-1][0], hidden_layers[i][0]))
                self.layers.append(activation_dict[hidden_layers[i][1]])
        self.layers.append(nn.Linear(hidden_layers[-1][0], output_size))
        if softmax:
            self.layers.append(nn.Softmax(dim=1))
        # Initialize the weights
        for layer in self.layers:
            if isinstance(layer, nn.Linear):
                nn.init.kaiming_normal_(layer.weight)
                if layer.bias is not None:
                    nn.init.zeros_(layer.bias)

    def forward(self, input):
        # may need to flatten the input
        for layer in self.layers:
            input = layer(input)
        return input
```

Q7

Feed Forward Function

In [ ]:

```python
# 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.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[

        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, output_size, softmax)
```

```python
    def forward(self, input):
        for i in range(len(self.conv)):
            input = self.conv[i](input)
            input = self.pool[i](input)
        input = self.pool_gap(input)
        input = my_flatten(input)
        input = self.mlp(input)
        return input
```

In [ ]:
```python
# Test the CNN class
input = torch.empty(1, 3, 32, 32).normal_()

'''
• 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, 1, True, 'sigmoid'), (16, 8, 3, 1, 1, True, 'sigmoid')],
             pool_layers=[(2, 2, 'max', False), (2, 2, 'max', False)],
             hidden_layers=[(8*8*8, 'sigmoid')],
             output_size=10,
             softmax=True)

output = cnn(input)
print(output.size())
print(output)
```

```
torch.Size([1, 10])
tensor([[0.0577, 0.0249, 0.1007, 0.0796, 0.0488, 0.0722, 0.0867, 0.3892, 0.0750,
         0.0651]], grad_fn=<SoftmaxBackward0>)
```

In [ ]:
```python
# 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(tmp[0].size())
output = cnn(tmp[0])
print(output.size())
```

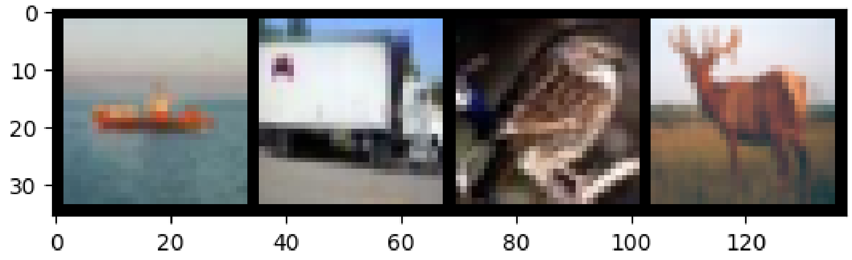
```
Files already downloaded and verified
torch.Size([4, 3, 32, 32])
torch.Size([4, 10])
```

Q8

In [ ]:

```python
# 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.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
outputs = cnn(images)
print(outputs.size())
print(outputs)
```

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




```
GroundTruth:   ship truck  bird  deer
torch.Size([4, 10])
tensor([[0.0690, 0.0283, 0.1117, 0.0906, 0.0431, 0.0638, 0.0601, 0.3925, 0.0648,
         0.0758],
        [0.0690, 0.0283, 0.1117, 0.0906, 0.0431, 0.0638, 0.0601, 0.3925, 0.0648,
         0.0758],
        [0.0690, 0.0283, 0.1117, 0.0906, 0.0431, 0.0638, 0.0601, 0.3925, 0.0648,
         0.0758],
        [0.0690, 0.0283, 0.1117, 0.0906, 0.0431, 0.0638, 0.0601, 0.3925, 0.0648,
         0.0758]], grad_fn=<SoftmaxBackward0>)
```

All of the output values are same.

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

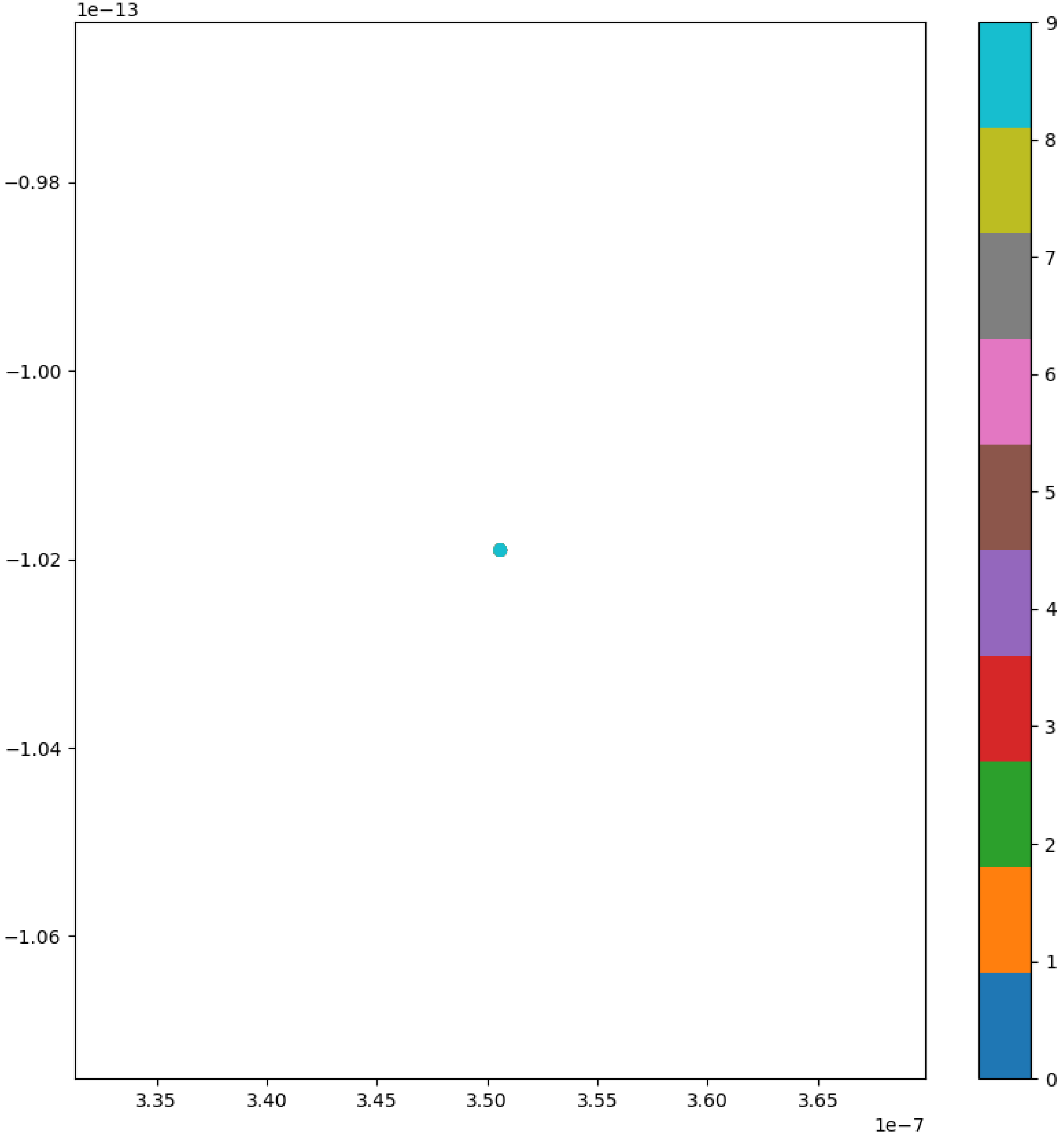
In [ ]:
```python
# Choose 3 images from each class and print the predicted class label for each image.
import numpy as np
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose([transforms.ToTensor()])
testset = torchvision.datasets.CIFAR10(root='./data', 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)
```

```
Files already downloaded and verified
[ 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 [ ]:
```python
# 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(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(len(labels))))

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())
# Apply PCA to output and plot it
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(output.detach().numpy())
output_pca = pca.transform(output.detach().numpy())
print(output_pca.shape)

plt.figure(figsize=(10, 10))
plt.scatter(output_pca[:, 0], output_pca[:, 1], c=labels, cmap='tab10')
plt.colorbar()
plt.show()
print(output_pca)
```

```
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 tr
uck truck truck
torch.Size([30, 8])
(30, 2)
```



1e−13
−0.98
−1.00
−1.02
−1.04
−1.06
3.35
3.40
3.45
3.50
3.55
3.60
3.65
1e−7
9
8
7
6
5
4
3
2
1
0

```
[[ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]
 [ 3.5059333e-07 -1.0190438e-13]]
```

In [ ]: