Homework 3

November 19, 2021

Name - Parthasarathi Kumar
PID - A59003519
ECE 253 - Fundamentals of Digital Image Processing

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

Assignment3_Problem1

November 21, 2021

Import Libraries

[1]:
```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

Function performs Non-maximum supression

[2]:
```
def nms(I_grad_mag, I_grad_edge):
    #  Non-maximum supression
    I_grad_mag_nms = np.copy(I_grad_mag)
    a_temp = [];
    for i in range(1, I_grad_mag.shape[0]-1):
        for j in range (1, I_grad_mag.shape[1]-1):

            ang_val = np.rint(np.abs(I_grad_edge[i,j])/(np.pi/4))
            if (ang_val not in a_temp):
                a_temp.append(ang_val)

            if(ang_val == 0 or ang_val == 4):
                candidate = np.array([I_grad_mag[i,j+1], I_grad_mag[i,j-1]])
            elif (ang_val == 1):
                candidate = np.array([I_grad_mag[i-1,j+1], I_grad_mag[i+1,j-1]])
            elif (ang_val == 2):
                candidate = np.array([I_grad_mag[i+1,j], I_grad_mag[i-1,j]])
            elif (ang_val == 3):
                candidate = np.array([I_grad_mag[i+1,j+1], I_grad_mag[i-1,j-1]])

            if(np.all(I_grad_mag[i][j]>candidate)):
                I_grad_mag_nms[i,j] = I_grad_mag[i,j]
            else:
                I_grad_mag_nms[i,j] = 0

    #print(a_temp)
    return I_grad_mag_nms
```

[ ]:
```
```

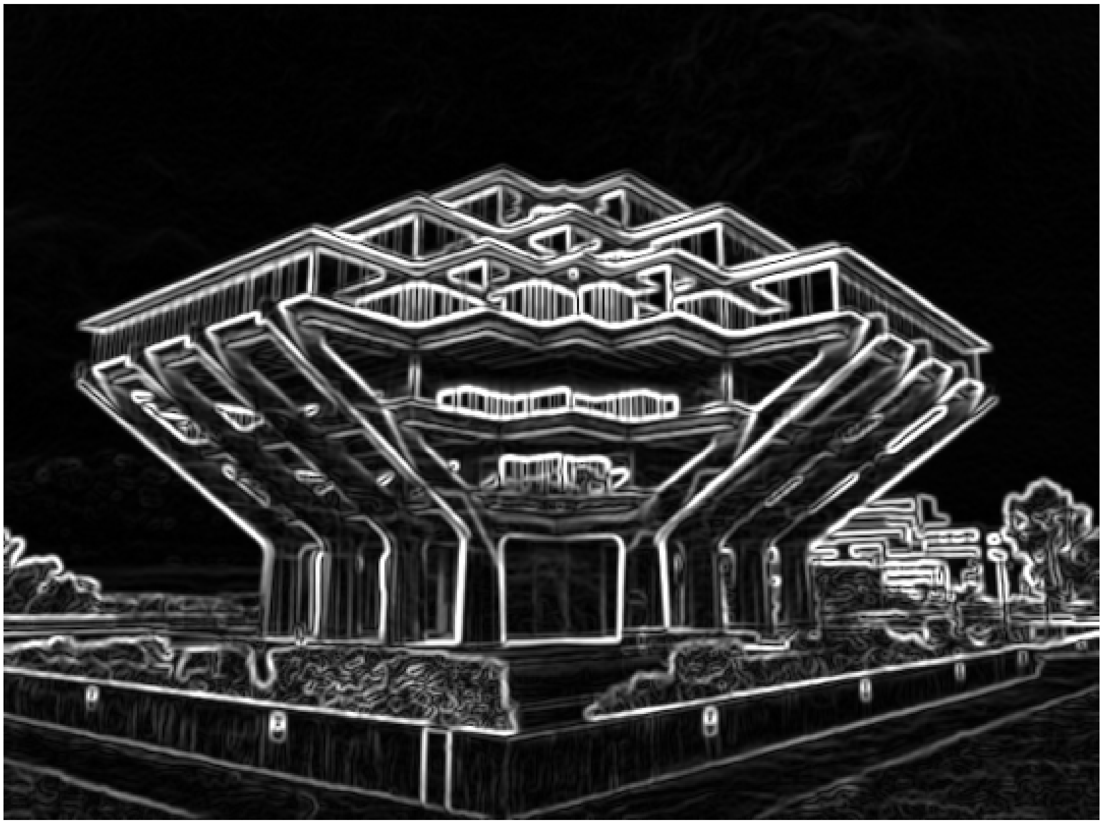
Main Canny Edge Detector

[3]:
```python
def cannyEdgeDetection(I, T):
    # Gaussian smoothing

    assert(np.shape(I.shape)[0] == 2)

    filter = np.array([[2, 4, 5, 4, 2],[4, 9, 12, 9, 4],[5, 12, 15, 12, 5],
                       [4, 9, 12, 9, 4],[2, 4, 5, 4, 2]])/159
    I_smooth = cv2.filter2D(I, -1,filter)


    # Apply Sobel operator
    filter = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    I_edge_x = cv2.filter2D(I_smooth, -1, filter)

    filter = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
    I_edge_y = cv2.filter2D(I_smooth, -1, filter)

    # Compute gradient images
    I_edge_mag = np.sqrt(np.square(I_edge_x) + np.square(I_edge_y))
    I_edge_dir = np.arctan2(I_edge_y,I_edge_x)
    I_edge_mag[I_edge_mag>=1] = 1.0

    # Apply NMS
    I_edge_NMS = nms(I_edge_mag, I_edge_dir)

    # Prevent clipping
    I_edge = np.copy(I_edge_NMS)

    # Threshold
    I_edge[I_edge<=T] = 0
    I_edge[I_edge>T] = 1.0

    return I_edge_mag , I_edge_NMS, I_edge
```

Read image and perform Canny edge detection

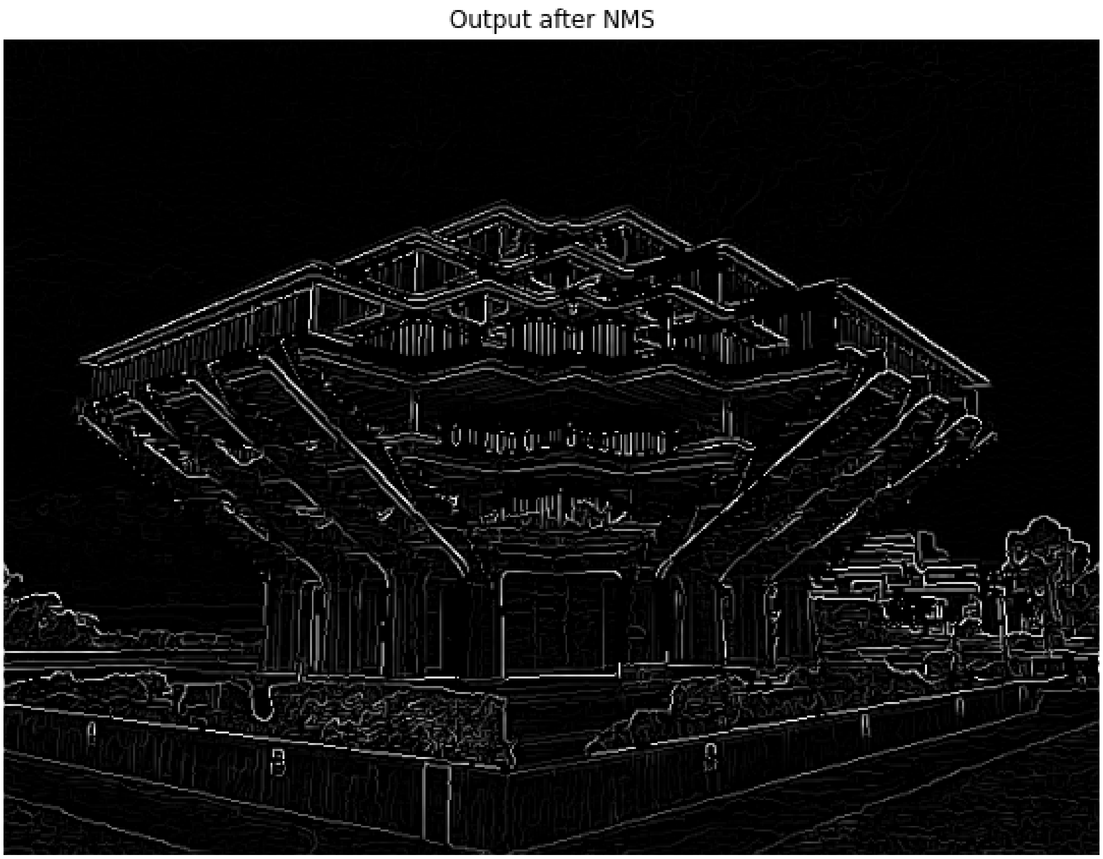
[4]:
```python
I = cv2.imread('/home/parth/work/UCSD/Fall 2021/ECE 253 Image Processing/
 ↪Assignment 3/HW3_geisel.jpg',0)
#I = cv2.resize(I, (0,0), fx=0.25, fy=0.25)
I = I/255

plt.figure(figsize=(10,10))
plt.imshow(I, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

Gradient magnitude image



[7]:

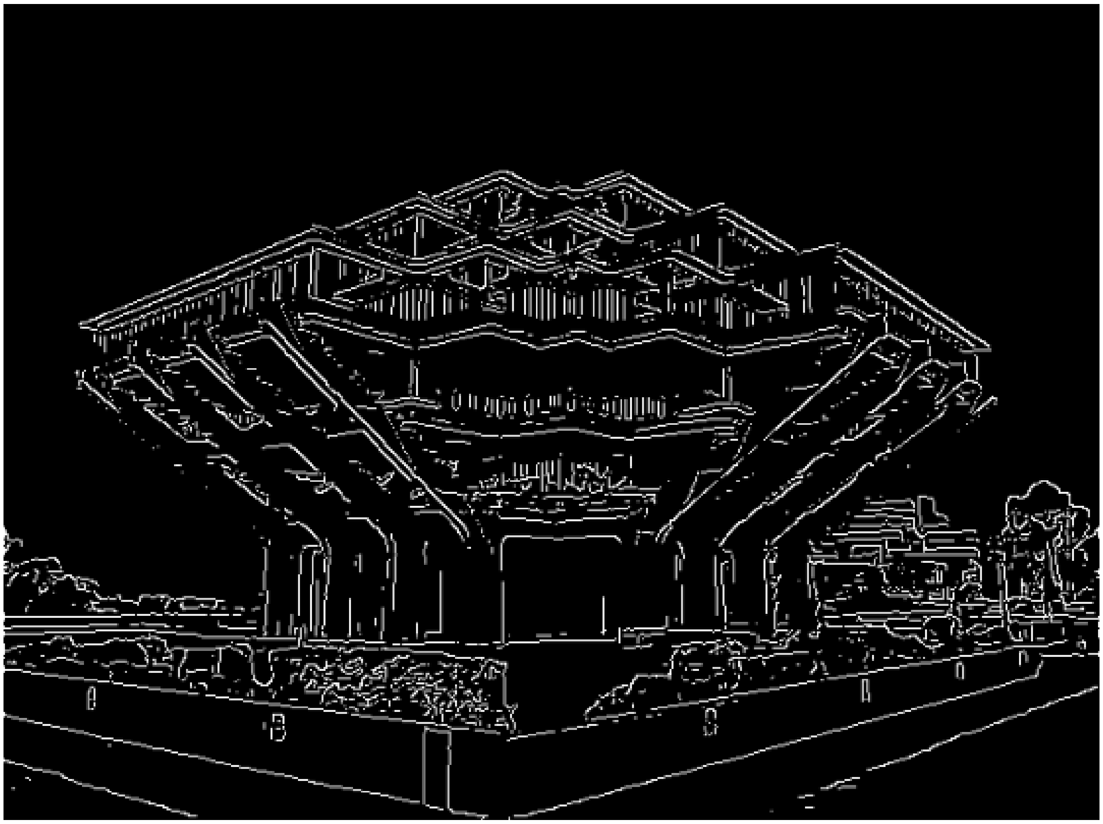
```
plt.figure(figsize=(10,10))
plt.imshow(I_edge_NMS ,cmap='gray')
plt.title('Output after NMS')
plt.axis('off')
plt.show()
```



Output of canny edge detector with a threshold 0.25

[8]:
```
plt.figure(figsize=(10,10))
plt.imshow(I_edge, cmap='gray')
plt.title('Output of canny edge detector')
plt.axis('off')
plt.show()
```

Output of canny edge detector



```
[ ]:
```

Assignment3_Problem2

November 18, 2021

[354]:
```
import numpy as np
import cv2
from matplotlib import pyplot as plt
```

Reading the original image

[355]:
```
I = cv2.imread('Car.tif', 0)
plt.imshow(I, cmap='gray')
plt.title('Original car.tif')
plt.axis('off')
plt.show()
```

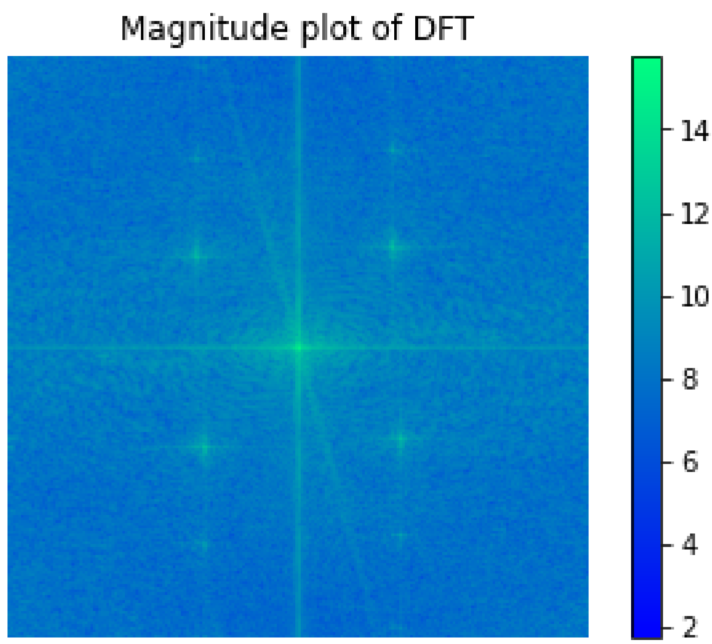
Padding the image uniformly to make it of size (512,512)

[356]:
```
pad_x = int((512-I.shape[0])/2)
pad_y = int((512-I.shape[1])/2)
I_pad = np.pad(I, ((pad_x, pad_x),(pad_y, pad_y)), constant_values = (0))
```

DFT of original image with the dc term shifted to the center

[357]:
```
I_DFT = np.fft.fftshift(np.fft.fft2(I_pad))
I_DFT_mag = np.abs(I_DFT)
```

[358]:
```
im = plt.imshow(np.log(I_DFT_mag), cmap='winter')
plt.title('Magnitude plot of DFT')
plt.colorbar(im)
plt.axis('off')
plt.show()
```



Notch reject filter with n = 4 and D_0 = 20. The four impulses are taken to be at (85,170), (85, 85), (85, -85) and (85, -170)

[359]:
```
# Notch Reject Butterworth filter
x_axis = np.linspace(-256,255,512)
y_axis = np.linspace(-256,255,512)
[u,v] = np.meshgrid(x_axis,y_axis)

H_NRBF = np.ones_like(u)

n = 4
D_0 = 20

U = np.array([1,1,1,1])*85
V = np.array([170,85,-85,-170])
```

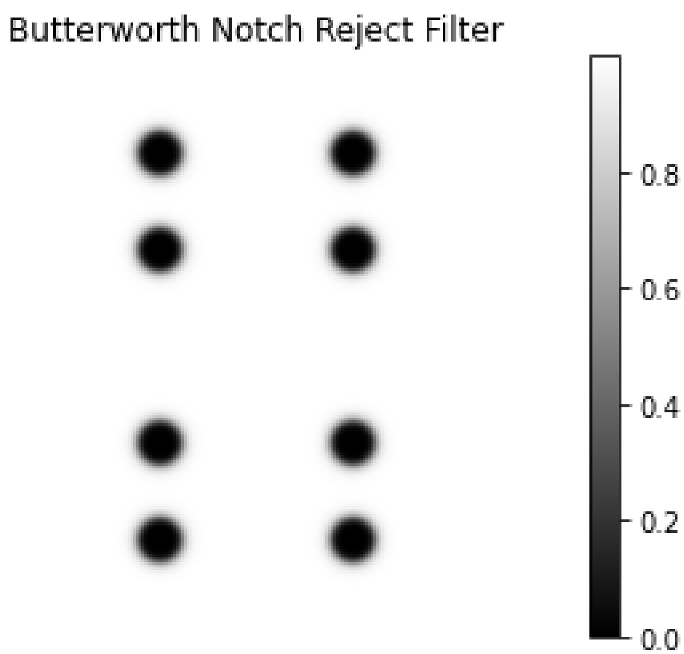
```
impulse_loc = np.stack((U,V))
```

Creating the BNRF.

[360]:
```
for i in range(H_NRBF.shape[0]):
    for j in range(H_NRBF.shape[1]):
        p = np.array([[u[i,j]],[v[i,j]]])
        d1 = np.linalg.norm(impulse_loc-p, axis=0) + 1e-9
        d2 = np.linalg.norm(impulse_loc+p, axis=0) + 1e-9
        f1 = 1/(1+pow(D_0/d1,2*n))
        f2 = 1/(1+pow(D_0/d2,2*n))
        H_NRBF[i][j] = np.prod(f1*f2)
```

BNRF in the frequency domain

[361]:
```
im  = plt.imshow(H_NRBF, cmap='gray')
plt.colorbar(im)
plt.title('Butterworth Notch Reject Filter')
plt.axis('off')
plt.show()
```



Filtering the given image with the constructed filter in the frequency domain

[362]:
```
I_filered = I_DFT* H_NRBF
I_filered_mag = np.abs(I_filered)
```

Obtaining the filter image by taking the inverse fourirer transform

[363]:
```
I_final = np.abs(np.fft.ifft2(I_filered))
I_final_crop = I_final[pad_x+1:pad_x+246, pad_y+1:pad_y+168]
plt.imshow(I_final_crop, cmap='gray')
plt.title('Filtered Image')
plt.axis('off')
plt.show()
```

Filtered Image

Readin the original image Street.png

[364]:
```
I = cv2.imread('Street.png',0)

plt.imshow(I, cmap='gray')
plt.title('Original Image street.png')
plt.axis('off')
plt.show()
```

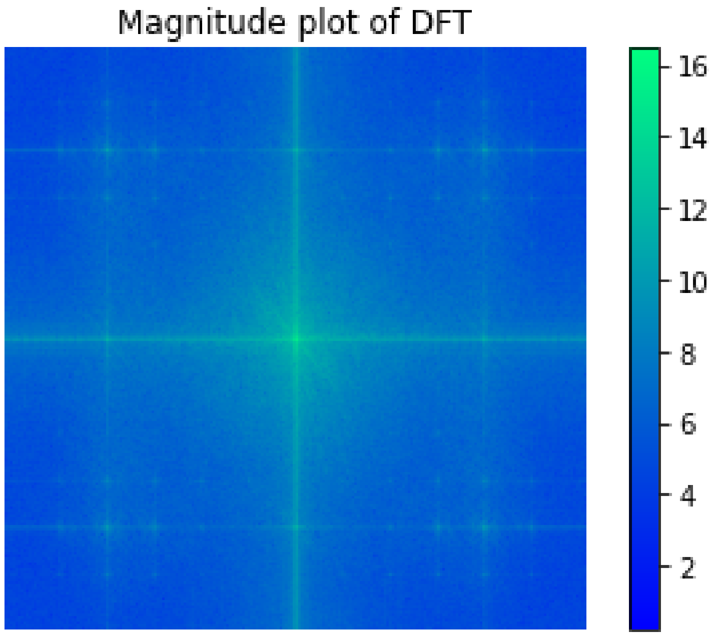
Padding the image

[365]:
```
pad_x = int((512-I.shape[0])/2)
pad_y = int((512-I.shape[1])/2)
I_pad = np.pad(I, ((pad_x, pad_x),(pad_y, pad_y+1)), constant_values = (0))
```

Magnitude plot of the DFT of the image shifted to the center

[366]:
```
I_DFT = np.fft.fftshift(np.fft.fft2(I_pad))
I_DFT_mag = np.abs(I_DFT)
```

[367]:
```
im = plt.imshow(np.log(I_DFT_mag), cmap='winter')
plt.colorbar(im)
plt.axis('off')
plt.title('Magnitude plot of DFT')
plt.show()
```



Notch reject filter with n = 5 and $D_0 = 10$. The four impulses are taken to be at (0,200) and (170,0)

[368]:
```
# Notch Reject Butterworth filter
x_axis = np.linspace(-256,255,512)
y_axis = np.linspace(-256,255,512)
[u,v] = np.meshgrid(x_axis,y_axis)

H_NRBF = np.ones_like(u)

n = 5
D_0 = 10

U = np.array([0,1])*170
V = np.array([1,0])*200
impulse_loc = np.stack((U,V))
```

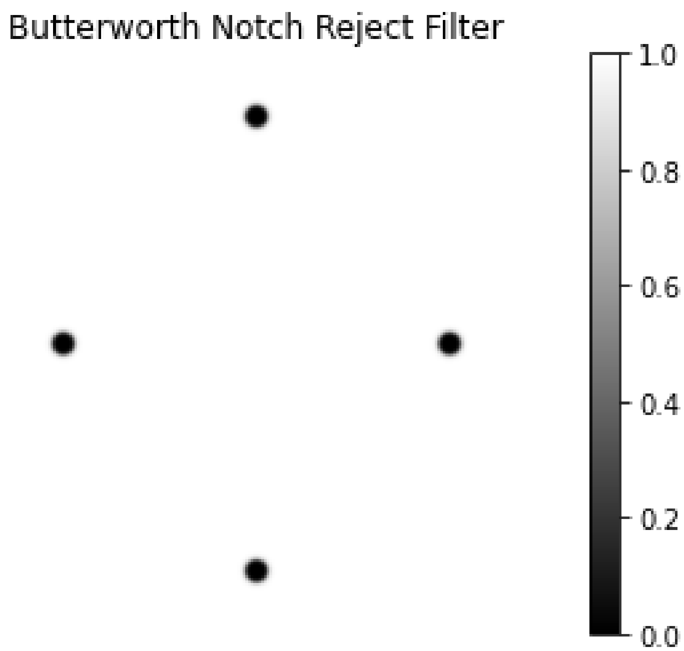
Constructing the Butterworth notch reject filter in the frequency domain

[369]:
```
for i in range(H_NRBF.shape[0]):
    for j in range(H_NRBF.shape[1]):
        p = np.array([[u[i,j]],[v[i,j]]])
        d1 = np.linalg.norm(impulse_loc-p, axis=0) + 1e-9
        d2 = np.linalg.norm(impulse_loc+p, axis=0) + 1e-9
        f1 = 1/(1+pow(D_0/d1,2*n))
        f2 = 1/(1+pow(D_0/d2,2*n))
```

```
        H_NRBF[i][j] = np.prod(f1*f2)
```

Filter plotted in the frequency domain

[370]:
```
im  = plt.imshow(H_NRBF, cmap='gray')
plt.colorbar(im)
plt.axis('off')
plt.title('Butterworth Notch Reject Filter')
plt.show()
```



Filtering the given image with the constructed filter in the frequency domain

[371]:
```
I_filered = I_DFT* H_NRBF
I_filered_mag = np.abs(I_filered)
```

Obtaining the filter image by taking the inverse fourirer transform

[372]:
```
I_final = np.abs(np.fft.ifft2(I_filered))
I_final_crop = I_final[pad_x+1:pad_x+I.shape[0], pad_y+1:pad_y+I.shape[1]]
plt.imshow(I_final_crop, cmap='gray')
plt.title('Filtered Image')
plt.axis('off')
plt.show()
```

Filtered Image

Assignment3_Problem3

November 19, 2021

[27]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

[28]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          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=batch_size,
                                         shuffle=False, num_workers=2)

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

```
Files already downloaded and verified
Files already downloaded and verified
```

(i)
The batch size is 4 and the number of images used for training is 50,000.

(ii) Yes, we do normalize the images in this example.

[29]:
```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
```

```
    img = img / 2 + 0.5      # unnormalize
    npimg = img.numpy()
    plt.figure(figsize=(12,7))
    plt.axis('off')
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


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

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(batch_size)))
```

```
  cat truck horse  bird
```

[30]:
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
```

```
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def visualizeFirstLayer(self, x):
        return self.pool(F.relu(self.conv1(x)))

net = Net()
```

[31]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assuming that we are on a CUDA machine, this should print a CUDA device:

print(device)
net.to(device)
```

```
cuda:0
```

[31]:
```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

[32]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

[33]:
```
loss_list = []
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()
```

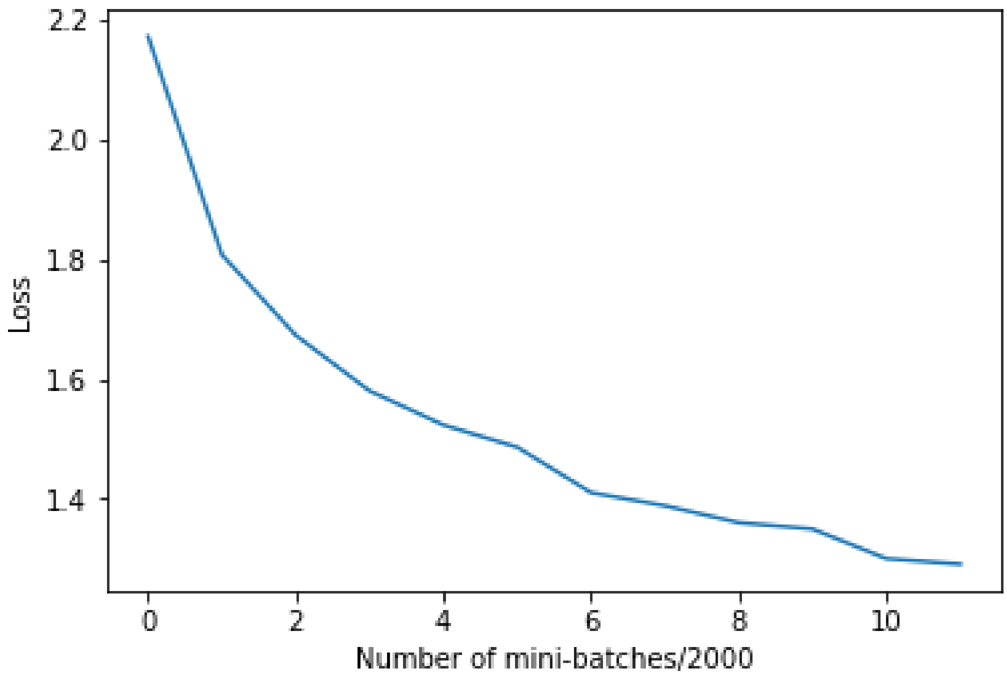
```
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            loss_list.append(running_loss/2000)
            running_loss = 0.0
print('Finished Training')
```

```
[1,  2000] loss: 2.172
[1,  4000] loss: 1.808
[1,  6000] loss: 1.673
[1,  8000] loss: 1.580
[1, 10000] loss: 1.523
[1, 12000] loss: 1.486
[2,  2000] loss: 1.410
[2,  4000] loss: 1.388
[2,  6000] loss: 1.360
[2,  8000] loss: 1.350
[2, 10000] loss: 1.300
[2, 12000] loss: 1.291
Finished Training
```

Plot of loss function with each 2000 mini-batch across the 2 epochs

[47]:
```
plt.plot(loss_list)
plt.ylabel('Loss')
plt.xlabel('Number of mini-batches/2000')
```

[47]:
```
Text(0.5, 0, 'Number of mini-batches/2000')
```



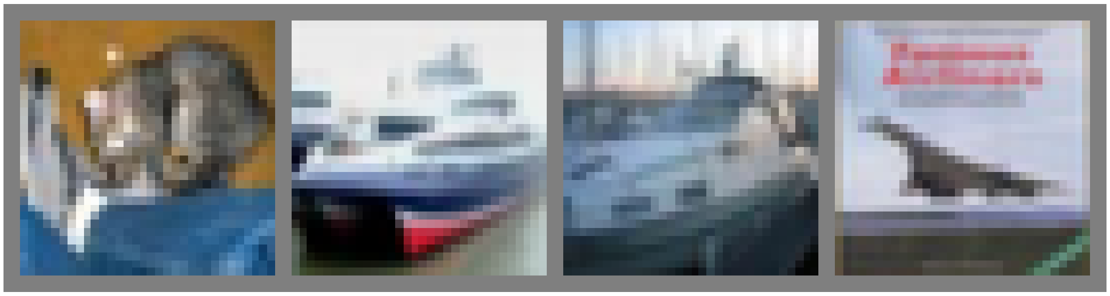
[35]:
```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

[37]:
```
net = Net()
net.load_state_dict(torch.load(PATH))
```

[37]: <All keys matched successfully>

[40]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

outputs = net(images)
```

[41]:
```
_, predicted = torch.max(outputs, 1)

# print images
imshow(torchvision.utils.make_grid(images))

print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```



```
GroundTruth:    cat  ship  ship plane
Predicted:    cat  ship  ship  ship
```

[43]:
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our␣
↪outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data #data[0].to(device), data[1].to(device)
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 55 %
```

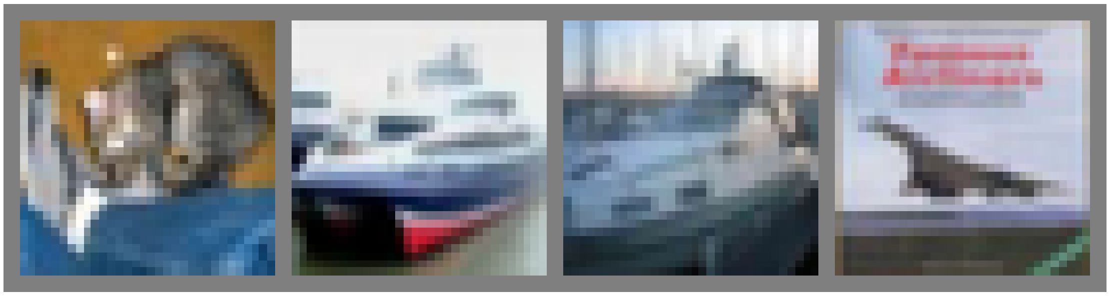
[44]:
```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
```

```
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                       accuracy))
```

```
Accuracy for class plane is: 61.0 %
Accuracy for class car   is: 76.1 %
Accuracy for class bird  is: 20.0 %
Accuracy for class cat   is: 43.8 %
Accuracy for class deer  is: 65.2 %
Accuracy for class dog   is: 41.8 %
Accuracy for class frog  is: 62.0 %
Accuracy for class horse is: 51.8 %
Accuracy for class ship  is: 73.3 %
Accuracy for class truck is: 57.1 %
```

(v) Predicting the class labels of 4 sample images.

[45]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()
layer = net.visualizeFirstLayer(images)
imshow(torchvision.utils.make_grid(images))

outputs = net(images)
_, predicted = torch.max(outputs, 1)
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```



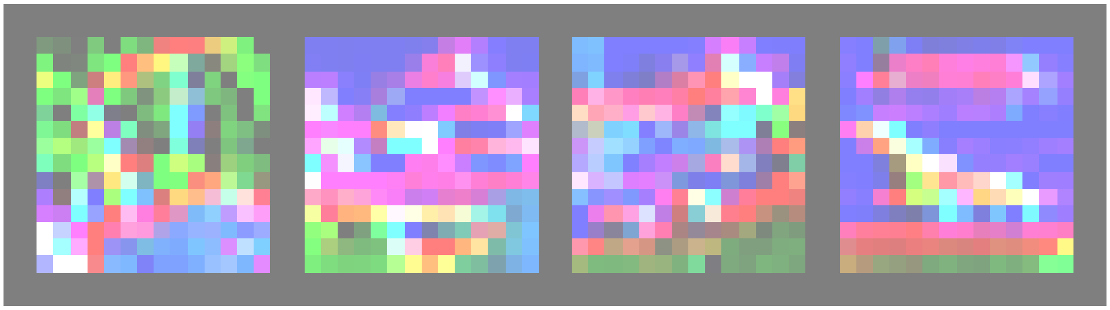
```
GroundTruth:    cat  ship  ship plane
Predicted:    cat  ship  ship  ship
```

(vi)

Visualization of the output of the first layer of the network

[46]:
```
imshow(torchvision.utils.make_grid(layer[:,0:3,:,:]))
imshow(torchvision.utils.make_grid(layer[:,3:6,:,:]))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```