CS 7150: Deep Learning — Spring 2021— Paul Hand

HW 3
Due: Friday March 26, 2021 at 5:00 PM Eastern time via Gradescope

Names: Gourang Patel, Sonal Jain, Sanjan Vijayakumar

You will submit this homework in groups of 2. You may consult any and all resources. Note that some of these questions are somewhat vague by design. Part of your task is to make reasonable decisions in interpreting the questions. Your responses should convey understanding, be written with an appropriate amount of precision, and be succinct. Where possible, you should make precise statements. For questions that require coding, you may either type your results with figures into this tex file, or you may append a pdf of output of a Jupyter notebook that is organized similarly. You may use code available on the internet as a starting point.

Question 1. *Image denoising by ResNets*

Consider the following denoising problem. Let x be a color image whose values are scaled to be within [0,1]. Let y be a noisy version of x where each color channel of each pixel is subject to additive Gaussian noise with mean 0 and variance σ^2. You will need to clip the values of y in order to ensure it is a valid image. The denoising problem is to estimate x given y.

(a) Look up the definition of Peak Signal-to-Noise Ratio (PSNR). Determine what value of σ corresponds to an expected PSNR between x and y of approximately 20 dB.

Response:

Using the equation above for PSNR and MSE,

PSNR = $20log_{10}(MAX_I) - 10log_{10}(MSE)$

PSNR = $20log_{10}(1) - 10log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(x[i,j]-y[i,j])^2}{mn}$

PSNR = $0 - 10log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(x[i,j]-(x[i,j]-\eta[i,j]))^2}{mn}$

PSNR = $-10log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}$

Given, expected PSNR between x and y is of approximately 20db, we get E[PSNR] = 20

$E[-10log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}] = 20$

$E[-log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}] = 2$

Using Jensen's inequality:

$E[-log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}] \geq -log_{10}E[\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}]$

$2 \geq -log_{10}E[\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{mn}]$

$2 \geq -log_{10}\frac{1}{mn}E[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(v)^2]$

$2 \geq -log_{10}\frac{1}{mn}m.nE[(v)^2]$

$2 \geq -log_{10}E[(v)^2]$

Now we know that,

a) $E[(v)^2] = Var[v] - E[(v)]^2$

b) $E[(v)]^2$ is 0 as noise is Gaussian with mean 0

Substituting back in the equation,

$2 \geq -log_{10}Var[v]$

$Var[v] = 10^{-2}$

The value of σ corresponds to an expected PSNR between x and y of approximately 20 dB is 0.1

(b) Create a noisy version of the CIFAR-10 training and test dataset, such that it has additive Gaussian white noise with PSNR approximately 20 dB . Show several pairs of images and their noisy version.

Response:
Here, we have added Gaussian noise to the complete CIFAR10 dataset, i.e as derived in the previous part we have used Standard Deviation as 0.1 refer Fig(1).
We have then plotted the CIFAR10 images with noise refer Fig(3) and without noise refer Fig(2).
We have also computed the PSNR ratio between for the complete set of train data with and without noise so as to verify our findings and we observed that PSNR came out to be around $20dB$ refer Fig(4)

```python
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=.1):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean
        ## torch.randn produces a tensor with elements drawn from a Gaussian distribution
        # of zero mean and unit variance and we multiply Multiply by 0.1 to have the desired std.

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={.1})'.format(self.mean, self.std)
```

Figure 1: Code for adding gaussian noise

Figure 2: Images from CIFAR-10 without noise

Figure 3: Images from CIFAR-10 with noise

```
for noise, normal in zip(trainloader_noise, trainloader_no_noise):
    noise_image, _ = noise
    normal_image, _ = normal
    noise_image = noise_image.numpy()
    normal_image = normal_image.numpy()
    mean_squared_error = []
    mse_1b = []
    for i, j in zip(noise_image, normal_image):
      mse_cal = np.mean(np.square(np.subtract(j,i)))
      mse_1b.append(mse_cal)
      mean_squared_error.extend(mse_1b)

PSNR = -10*np.log10(mean_squared_error)
print("Mean PSNR for the CIFAR10 data : {}".format(np.mean(PSNR)))
```

```
Mean PSNR for the CIFAR10 data : 19.996047973632812
```

Figure 4: Mean PSNR obtained

(c) Train a ResNet to denoise noisy CIFAR-10 images. Your net should take a noisy 32x32 px image as an input, and it should output a denoised 32x32 px image. Specify the architecture and training details of your network. Determine the mean and standard deviation of the recovery PSNRs over the noisy test set. Visually show the performance on three noisy test images.

Response:
Model Architecture : Refer Model Parameters for the detailed Model Architecture.
First we have used Conv2d layer Conv2d$(3, 64, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$

Then, we have used BatchNormalization 2D layer: BatchNorm2d$(64, eps = 1e{-}05, momentum = 0.1, affine = True, track_running_stats = True)$

After, that we have used a Relu activation (relu): ReLU(inplace=True)

After Relu we have implemented two layers with two Residual Blocks each,both the layers are similar in architecture and they look like -
(layer1): Sequential((0): ResidualBlock((conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)

We have similar layer2 as well. And in each Residual block we are adding the input to the output of last batchNormalization i.e (bn2) and then applying RELU on the ouput. Lastly, we have one more convolution layer: Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

This last layer gives 3 channels as the output and also the image size will remain consistent all through the net.

Training Details

Optimizer - Adam Optimizer
criterion = MSELoss
Learning Rate - 0.001
Epochs - 2
BATCH_SIZE = 3

Mean PSNR = Mean PSNR for the model with SKIP Connections for test set is : 27.252046585083008 refer Fig(5) **Standard Deviation PSNR** = Standard Deviation of PSNR for model with SKIP Connections for the test set is : 1.3458813428878784 refer Fig(5)

We can observe the 3 original images from test set in fig(6), the same 3 noisy images after adding noise to CIFAR10 in fig(7) and the denoised version of these test images given by our model in fig(8)

```
print(mean_squared)
PSNR = -10*np.log10(mean_squared)
print("Mean PSNR for the model with SKIP Connections for test set is : {}".format(np.mean(PSNR)))
print("Standard Deviation of PSNR for model with SKIP Connections for the test set is : {}".format(np.std(PSNR)))
```

```
[0.0024136526, 0.0020498026, 0.0019262683, 0.0015292036, 0.0022150783, 0.0018906262, 0.0020331796, 0.0014242209, 0.0023035968
Mean PSNR for the model with SKIP Connections for test set is : 27.252046585083008
Standard Deviation of PSNR for model with SKIP Connections for the test set is : 1.3458813428878784
```

Figure 5: Mean and STD PSNR

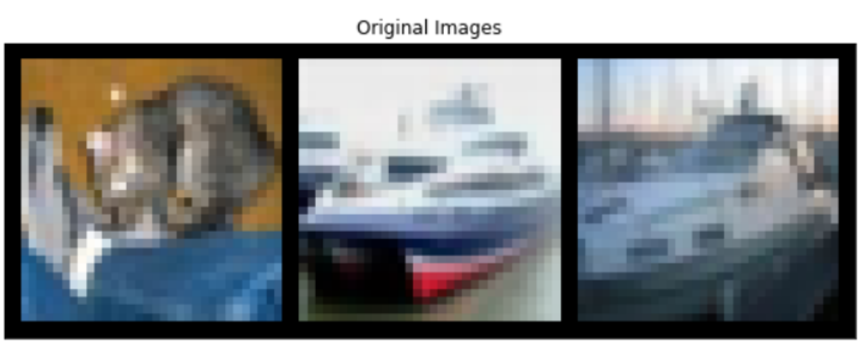


Figure 6: Original Images

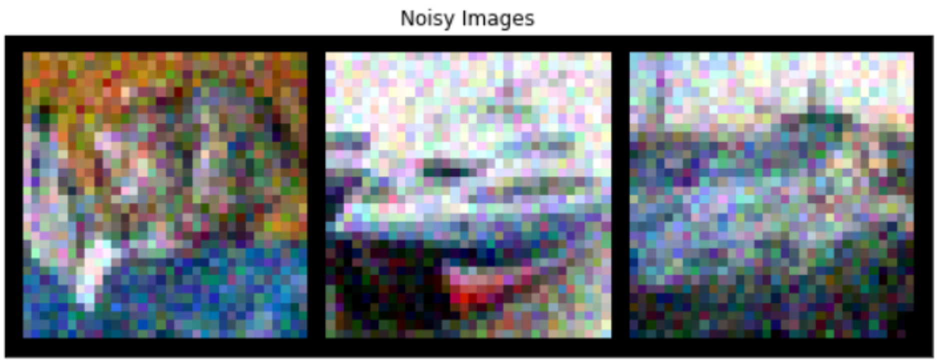


Figure 7: Noisy Images

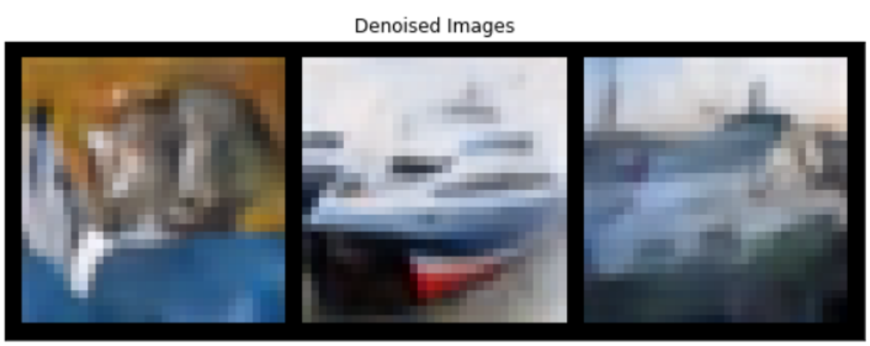


Figure 8: Denoised Images

(d) Repeat the previous task but without the skip connections in your model.
Model Architecture - The Model Architecture is same as that of the previous part 1(c), we are just removing the skip connection layer i.e the part from the ResidualBlock where we are adding the input to the output of (bn2) layer, and then observing our results.
Training Details

Optimizer - Adam Optimizer
criterion = MSELoss
Learning Rate - 0.001
Epochs - 2
BATCH_SIZE = 3

Mean PSNR = Mean PSNR for the model with No SKIP Connections for test set is : 26.09 refer Fig(9) **Standard Deviation PSNR** = Standard Deviation of PSNR for model with No SKIP Connections for the test set is : 1.403 refer Fig(9)

We can observe the 3 original images from test set in fig(10), the same 3 noisy images after adding noise to CIFAR10 in fig(11) and the denoised version of these test images with no Skip Connections model in fig(12)

```
PSNR = -10*np.log10(mean_squared)
print("Mean PSNR for the model with no SKIP Connections for test set is : {}".format(np.mean(PSNR)))
print("Standard Deviation of PSNR for model with no SKIP Connections for the test set is : {}".format(np.std(PSNR)))

Mean PSNR for the model with no SKIP Connections for test set is : 26.090673446655273
Standard Deviation of PSNR for model with no SKIP Connections for the test set is : 1.403568148612976
```

Figure 9: Mean PSNR and SD of PSNR

Figure 10: Original Images

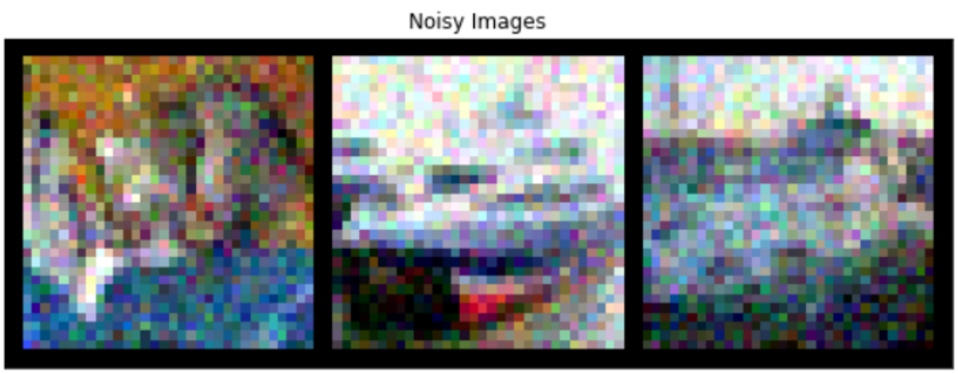


Figure 11: Noisy Images

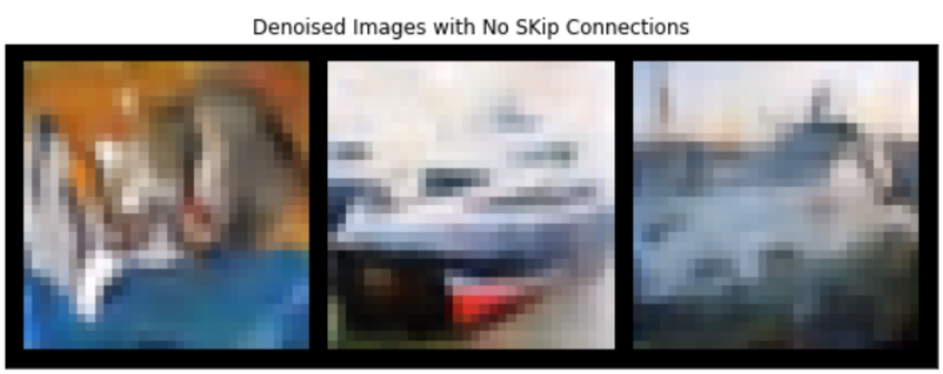


Figure 12: Denoised Images with no skip connections

Question 2. *Adversarial examples*

Obtain a pretrained classifier for ImageNet, such as AlexNet or ResNet101 from TorchVision. Using a camera, take a picture of an object that belongs to one of the ImageNet classes. Resize it as appropriate. Select a target class that is different from the image's true class. Compute an adversarial perturbation that is barely perceptible to the human eye and that results in the image being misclassified as the target class. Clearly state the method that you used to generate the perturbation. Show the underlying image, the perturbed image, the perturbation, the classifier's confidence for the underlying image, and the classifier's confidence for the perturbed image.

Response:

For this Question, we are using a dog image which is one of the imagenet class captured by a camera which was resized appropriately. The target class is is chosen to be the least likely class selected. After this perturbation was added to the original image. We chose "Iterative Least likely class method". In this method we need to remove perturbation from original image which we do for multiple iteration. For desired class we chose the least-likely class according to the prediction of the trained network on image X:

$$yLL = \underset{y}{\operatorname{argmin}}\, p(y|X)$$

The least likely class will be the highly disimilar class as compared to the true class.

$$X_0^{adv} = X, X_{N+1}^{adv} = Clip_{X,\epsilon} X_{adv}^{N} - \alpha * sign(\nabla_X J(X_N^{adv}, y_{LL}))$$

For this iterative procedure we used the same α and same number of iterations as for the basic iterative method.

This last expression equals sign for neural networks with cross-entropy loss:

$$sign(-\nabla_X J(X, y_{LL}))$$

Fig shows the result that we observed:

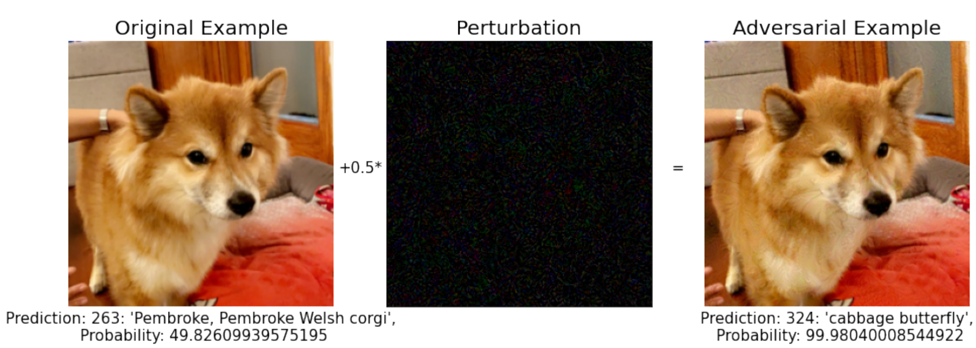


Figure 13: Original vs perturbed image

Here "263: 'Pembroke, Pembroke Welsh corgi' is the original image class and the least likely class comes to be "324: cabbage butterfly". We are using ResNet model for predicting the adversarial example with the probability is approximately 49.82 % for original image and 99.99 % for the perturbed image.

$epsilon = 0.5, numsteps = 10, alpha = 0.025$

Note: The perturbations are visible in the notebook but not very much visible in this pdf after taking the photo from the notebook. You can refer the notebook to see the perturbation.

```
import torchvision
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torchvision import transforms
import torch.utils.data as Data
import pandas as pd
import matplotlib.pyplot as plt
from pandas import DataFrame
import cv2
from google.colab.patches import cv2_imshow
import math
import PIL
import matplotlib.pyplot as plt
import numpy as np
from torch.autograd.gradcheck import zero_gradients
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Question 1.b

```
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=.1):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean
        ## torch.randn produces a tensor with elements drawn from a Gaussian distribution of zero mean and unit variance and we multiply

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={.1})'.format(self.mean, self.std)
```

```
transform_noise = transforms.Compose(
    [transforms.ToTensor(),
     AddGaussianNoise(0., 0.1)])

transform_no_noise = transforms.Compose(
    [transforms.ToTensor(),
])

trainset_noise = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_noise)
trainloader_noise = torch.utils.data.DataLoader(trainset_noise, batch_size=32,
                                          shuffle=False, num_workers=2)

testset_noise = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_noise)
testloader_noise = torch.utils.data.DataLoader(testset_noise, batch_size=32,
                                         shuffle=False, num_workers=2)



trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_no_noise)
trainloader_no_noise = torch.utils.data.DataLoader(trainset, batch_size=32,
                                          shuffle=False, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_no_noise)
testloader_no_noise = torch.utils.data.DataLoader(testset, batch_size=32,
                                         shuffle=False, num_workers=2)
```

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

```
def imshow(img):

    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


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

# show images
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.title("Images from CIFAR10 with Noise ")
imshow(torchvision.utils.make_grid(images))
```

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


```
dataiter = iter(trainloader_no_noise)
images, labels = dataiter.next()

# show images
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.title("Images from CIFAR10 without Noise ")
imshow(torchvision.utils.make_grid(images))
```


```
for noise, normal in zip(trainloader_noise, trainloader_no_noise):
    noise_image, _ = noise
    normal_image, _ = normal
    noise_image = noise_image.numpy()
    normal_image = normal_image.numpy()
    mean_squared_error = []
    mse_1b = []
    for i, j in zip(noise_image, normal_image):
      mse_cal = np.mean(np.square(np.subtract(j,i)))
      mse_1b.append(mse_cal)
      mean_squared_error.extend(mse_1b)

PSNR = -10*np.log10(mean_squared_error)
```

```
print("Mean PSNR for the CIFAR10 data : {}".format(np.mean(PSNR)))
```

```
Mean PSNR for the CIFAR10 data : 19.996047973632812
```

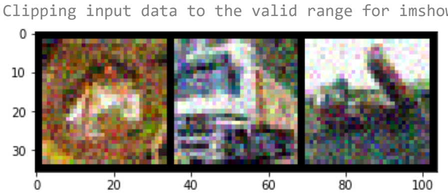
Question 1 part c

```
transform_noise = transforms.Compose(
    [transforms.ToTensor(),
     AddGaussianNoise(0., 0.1)])

transform_no_noise = transforms.Compose(
    [transforms.ToTensor(),
])

trainset_noise = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_noise)
trainloader_noise = torch.utils.data.DataLoader(trainset_noise, batch_size=3,
                                          shuffle=False, num_workers=2)

testset_noise = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_noise)
testloader_noise = torch.utils.data.DataLoader(testset_noise, batch_size=3,
                                         shuffle=False, num_workers=2)



trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_no_noise)
trainloader_no_noise = torch.utils.data.DataLoader(trainset, batch_size=3,
                                          shuffle=False, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_no_noise)
testloader_no_noise = torch.utils.data.DataLoader(testset, batch_size=3,
                                         shuffle=False, num_workers=2)
```

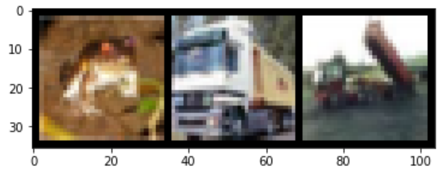
```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

```
# functions to show an image
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

def imshow(img):

    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


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

# show images
imshow(torchvision.utils.make_grid(images))
```

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




```
dataiter = iter(trainloader_no_noise)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
```



Resnet Model with skip connection

```
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)


    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += residual
        out = self.relu(out)
        return out

# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv = conv3x3(3, 64)
        self.bn = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 64, layers[0])
        self.layer2 = self.make_layer(block, 64, layers[1], 1)
        # self.layer3 = self.make_layer(block, 64, layers[2], 1)
        self.conv2 = conv3x3(64, 3)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        layers = []
        layers.append(block(self.in_channels, out_channels, 1, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.conv2(out)
        return out

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


model = ResNet(ResidualBlock, [2, 2, 2]).to(device)
```

Model Architecture for model with Skip Connection

```
model.parameters
```

```
<bound method Module.parameters of ResNet(
  (conv): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (conv2): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)>
```

```
num_epochs = 2
learning_rate = 0.001
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


# Train the model
curr_lr = learning_rate
for epoch in range(num_epochs):
    batch_iteration = 0
    running_loss = 0.00
    for noisy_data, normal_data in zip(trainloader_noise, trainloader_no_noise):
        noisy_inputs, noisy_labels = noisy_data
        normal_inputs, normal_labels = normal_data
        # Forward pass
        outputs = model(noisy_inputs.to(device))
        # print(outputs.shape)
        loss = criterion(outputs, normal_inputs.to(device))

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_iteration %3 == 2:
            print ("[%d %5d] loss: %.3f" %
                   (epoch+1, batch_iteration + 1,running_loss/3))
            running_loss = 0.00
        batch_iteration += 1

print("Finished Training")
PATH = './cifar10_net.pth'
torch.save(model.state_dict(),PATH)
```

```
[1    32] loss: 0.174
[1    64] loss: 0.017
[1    96] loss: 0.012
[1   128] loss: 0.013
[1   160] loss: 0.011
[1   192] loss: 0.009
[1   224] loss: 0.008
[1   256] loss: 0.007
```

```
[1   288] loss: 0.010
[1   320] loss: 0.007
[1   352] loss: 0.007
[1   384] loss: 0.007
[1   416] loss: 0.007
[1   448] loss: 0.007
[1   480] loss: 0.008
[1   512] loss: 0.007
[1   544] loss: 0.006
[1   576] loss: 0.006
[1   608] loss: 0.006
[1   640] loss: 0.009
[1   672] loss: 0.006
[1   704] loss: 0.007
[1   736] loss: 0.007
[1   768] loss: 0.007
[1   800] loss: 0.005
[1   832] loss: 0.005
[1   864] loss: 0.005
[1   896] loss: 0.007
[1   928] loss: 0.005
[1   960] loss: 0.005
[1   992] loss: 0.005
[1  1024] loss: 0.006
[1  1056] loss: 0.006
[1  1088] loss: 0.005
[1  1120] loss: 0.005
[1  1152] loss: 0.006
[1  1184] loss: 0.005
[1  1216] loss: 0.004
[1  1248] loss: 0.005
[1  1280] loss: 0.004
[1  1312] loss: 0.005
[1  1344] loss: 0.005
[1  1376] loss: 0.004
[1  1408] loss: 0.004
[1  1440] loss: 0.005
[1  1472] loss: 0.004
[1  1504] loss: 0.004
[1  1536] loss: 0.005
[2    32] loss: 0.004
[2    64] loss: 0.004
[2    96] loss: 0.004
[2   128] loss: 0.005
[2   160] loss: 0.005
[2   192] loss: 0.004
[2   224] loss: 0.004
[2   256] loss: 0.004
[2   288] loss: 0.005
[2   320] loss: 0.004
[2   352] loss: 0.004
```

```
PATH = './cifar10_net.pth'
model_test = ResNet(ResidualBlock, [2, 2, 2]).to(device)
model_test.load_state_dict(torch.load(PATH))

mean_squared = []

with torch.no_grad():
  for noise, normal in zip(testloader_noise, testloader_no_noise):
    noise_test, _ = noise
    normal_test, _ = normal
    normal_test = normal_test.numpy()
    output = model_test(noise_test.to(device))
    output = output.cpu()
    output = output.numpy()
    mse = []
    for i, j in zip(output, normal_test):
      mse_cal = np.mean(np.square(np.subtract(j,i)))
      mse.append(mse_cal)
    mean_squared.extend(mse)
```

```
print(mean_squared)
PSNR = -10*np.log10(mean_squared)
print("Mean PSNR for the model with SKIP Connections for test set is : {}".format(np.mean(PSNR)))
print("Standard Deviation of PSNR for model with SKIP Connections for the test set is : {}".format(np.std(PSNR)))
```

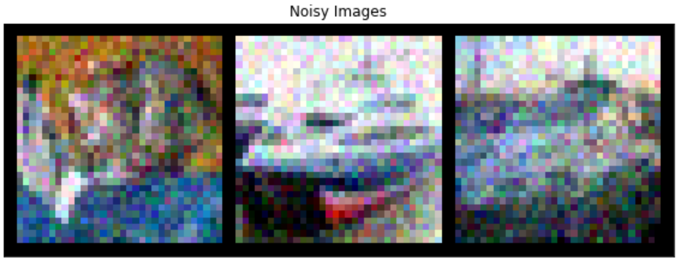
```
[0.0024136526, 0.0020498026, 0.0019262683, 0.0015292036, 0.0022150783, 0.0018906262, 0.0020331796, 0.0014242209, 0.0023035968, 0.0029003744, 0.00163832
Mean PSNR for the model with SKIP Connections for test set is : 27.252046585083008
Standard Deviation of PSNR for model with SKIP Connections for the test set is : 1.3458813428878784
```

```
dataiter = iter(testloader_noise)
noise_image, _ = dataiter.next()
```

```
dataiter_no_noise = iter(testloader_no_noise)
normal_image, _ = dataiter_no_noise.next()

with torch.no_grad():
  denoised_image = model_test(noise_image.to(device))

noisy = torchvision.utils.make_grid(noise_image)
noisy_np = noisy.numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.imshow(np.transpose(noisy_np, (1, 2, 0)))
plt.title("Noisy Images")
plt.show()
```

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

Noisy Images



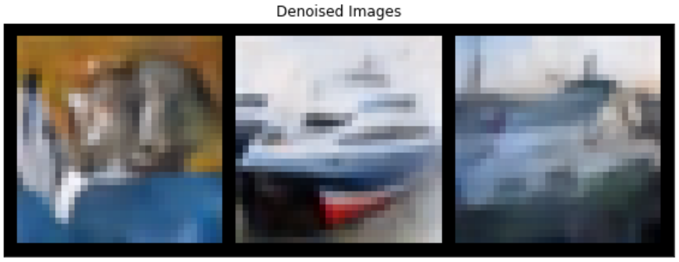
```
normal = torchvision.utils.make_grid(normal_image)
normal_np = normal.numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.title("Original Images")
plt.imshow(np.transpose(normal_np, (1, 2, 0)))
plt.show()
```

Original Images

```
denoised = torchvision.utils.make_grid(denoised_image)
denoised_np = denoised.cpu().numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.title("Denoised Images")
plt.imshow(np.transpose(denoised_np, (1, 2, 0)))
plt.show()
```

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

Denoised Images



1.d Resnet Model with no skip connection -- Removing the Add() layer from the model and observing the output

```
def conv3x3(in channels, out channels, stride=1):
```

```
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)


    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        # out += residual ## ResNet without skip connection
        out = self.relu(out)
        return out

# ResNet
class ResNet_noSkip(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet_noSkip, self).__init__()
        self.in_channels = 64
        self.conv = conv3x3(3, 64)
        self.bn = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 64, layers[0])
        self.layer2 = self.make_layer(block, 64, layers[1], 1)
        # self.layer3 = self.make_layer(block, 64, layers[2], 1)
        self.conv2 = conv3x3(64, 3)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        layers = []
        layers.append(block(self.in_channels, out_channels, 1, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.conv2(out)
        return out


model_no_skip = ResNet_noSkip(ResidualBlock, [2, 2, 2]).to(device)


num_epochs = 2
learning_rate = 0.001
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model_no_skip.parameters(), lr=learning_rate)
```

Model Architecture for model with NOSkip Connection

```
model_no_skip.parameters
```

```
<bound method Module.parameters of ResNet_noSkip(
  (conv): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): ResidualBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): ResidualBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): ResidualBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (conv2): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )>
```

```
# Train the model
curr_lr = learning_rate
for epoch in range(num_epochs):
    batch_iteration = 0
    running_loss = 0.00
    for noisy_data, normal_data in zip(trainloader_noise, trainloader_no_noise):
        noisy_inputs, noisy_labels = noisy_data
        normal_inputs, normal_labels = normal_data
        # print(noisy_inputs.shape, normal_inputs.shape)
        # Forward pass
        outputs = model_no_skip(noisy_inputs.to(device))
        # print(outputs.shape)
        loss = criterion(outputs, normal_inputs.to(device))

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_iteration %3 == 2:
            print ("[%d %5d] loss: %.3f" %
                   (epoch+1, batch_iteration + 1,running_loss/3))
            running_loss = 0.00
        batch_iteration += 1

print("Finished Training")
PATH = './cifar10_net_no_skip.pth'
torch.save(model_no_skip.state_dict(),PATH)
```

```
[2 14865] loss: 0.004
[2 14868] loss: 0.002
[2 14871] loss: 0.003
[2 14874] loss: 0.003
[2 14877] loss: 0.003
[2 14880] loss: 0.003
[2 14883] loss: 0.003
[2 14886] loss: 0.003
[2 14889] loss: 0.002
[2 14892] loss: 0.002
[2 14895] loss: 0.002
[2 14898] loss: 0.002
[2 14901] loss: 0.002
[2 14904] loss: 0.003
[2 14907] loss: 0.003
[2 14910] loss: 0.002
[2 14913] loss: 0.002
[2 14916] loss: 0.003
[2 14919] loss: 0.005
[2 14922] loss: 0.005
[2 14925] loss: 0.004
[2 14928] loss: 0.003
[2 14931] loss: 0.004
[2 14934] loss: 0.002
[2 14937] loss: 0.003
[2 14940] loss: 0.003
[2 14943] loss: 0.002
[2 14946] loss: 0.003
```

```
[2 14949] loss: 0.002
[2 14952] loss: 0.002
[2 14955] loss: 0.002
[2 14958] loss: 0.003
[2 14961] loss: 0.002
[2 14964] loss: 0.002
[2 14967] loss: 0.002
[2 14970] loss: 0.003
[2 14973] loss: 0.002
[2 14976] loss: 0.002
[2 14979] loss: 0.002
[2 14982] loss: 0.002
[2 14985] loss: 0.002
[2 14988] loss: 0.003
[2 14991] loss: 0.002
[2 14994] loss: 0.003
[2 14997] loss: 0.003
[2 15000] loss: 0.003
[2 15003] loss: 0.002
[2 15006] loss: 0.003
[2 15009] loss: 0.003
[2 15012] loss: 0.002
[2 15015] loss: 0.002
[2 15018] loss: 0.003
[2 15021] loss: 0.002
[2 15024] loss: 0.002
[2 15027] loss: 0.002
[2 15030] loss: 0.002
[2 15033] loss: 0.003
[2 15036] loss: 0.002
[2 15039] loss: 0.002
```

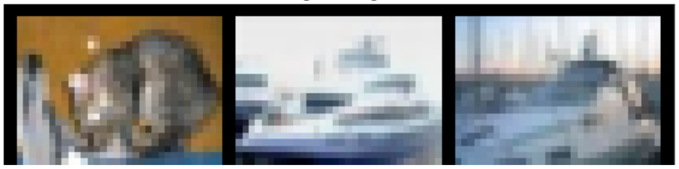
```
torch.save(model_no_skip.state_dict(),PATH)
PATH = './cifar10_net_no_skip.pth'
model_noSkip = ResNet_noSkip(ResidualBlock, [2, 2, 2]).to(device)
model_noSkip.load_state_dict(torch.load(PATH))

mean_squared = []

with torch.no_grad():
  for noise, normal in zip(testloader_noise, testloader_no_noise):
    noise_test, _ = noise
    normal_test, _ = normal
    normal_test = normal_test.numpy()
    output = model_noSkip(noise_test.to(device)).cpu().numpy()

    mse = []
    for i, j in zip(output, normal_test):
      mse_cal = np.mean(np.square(np.subtract(j,i)))
      mse.append(mse_cal)
    mean_squared.extend(mse)
```

```
PSNR = -10*np.log10(mean_squared)
print("Mean PSNR for the model with no SKIP Connections for test set is : {}".format(np.mean(PSNR)))
print("Standard Deviation of PSNR for model with no SKIP Connections for the test set is : {}".format(np.std(PSNR)))
```

```
Mean PSNR for the model with no SKIP Connections for test set is : 26.090673446655273
Standard Deviation of PSNR for model with no SKIP Connections for the test set is : 1.403568148612976
```

```
dataiter = iter(testloader_noise)
noise_image, _ = dataiter.next()

dataiter_no_noise = iter(testloader_no_noise)
normal_image, _ = dataiter_no_noise.next()

with torch.no_grad():
  denoised_image = model_noSkip(noise_image.to(device))

original = torchvision.utils.make_grid(normal_image)
original = original.numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.imshow(np.transpose(original, (1, 2, 0)))
plt.title("Original Images")
plt.show()
```

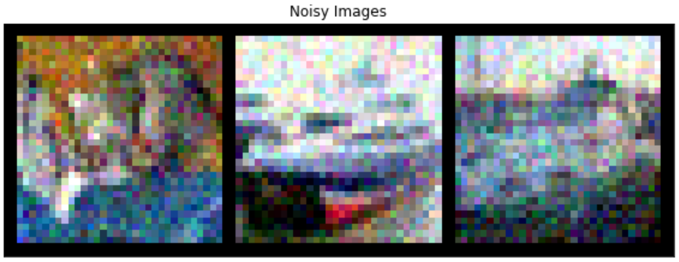
Original Images



```
noise_image = torchvision.utils.make_grid(noise_image)
noise_image = noise_image.numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.imshow(np.transpose(noise_image, (1, 2, 0)))
plt.title("Noisy Images")
plt.show()
```

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

Noisy Images



```
noise_image = torchvision.utils.make_grid(denoised_image)
noise_image = noise_image.cpu().numpy()
plt.figure(figsize=(10,8))
plt.xticks([])
plt.yticks([])
plt.imshow(np.transpose(noise_image, (1, 2, 0)))
plt.title("Denoised Images with No SKip Connections")
plt.show()
```

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

Denoised Images with No SKip Connections

Question 2

```
model_resnet = torch.hub.load('pytorch/vision:v0.6.0', 'resnet101', pretrained=True)
```

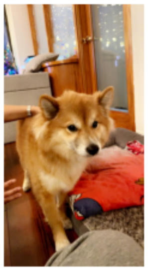
```
Downloading: "https://github.com/pytorch/vision/archive/v0.6.0.zip" to /root/.cache/torch/hub/v0.6.0.zip
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /root/.cache/torch/hub/checkpoints/resnet101-5d3b4d8f.pth
100%                    170M/170M [00:06<00:00, 26.8MB/s]
```

```
#mean and std will remain same irresptive of the model you use
mean=[0.485, 0.456, 0.406]
std=[0.229, 0.224, 0.225]

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)

])
```

```
from PIL import Image
true_img = Image.open('/content/drive/MyDrive/Khoury Courses/Deep Learning/HW/dog.jpg')
```

```
#Display true image
plt.imshow(true_img)
plt.axis("off")
```

```
(-0.5, 1187.5, 2207.5, -0.5)
```



```
#true image
true_transform_img = transform(true_img)
true_batch_t = true_transform_img.unsqueeze(0)

img_variable = Variable(true_batch_t, requires_grad=True) #convert tensor into a variable


def predict_class(output):
  with open('/content/drive/MyDrive/Khoury Courses/Deep Learning/HW/imagenet1000_clsidx_to_labels.txt') as f:
    classes = [line.strip() for line in f.readlines()]
  _, index = torch.max(output, 1)

  percentage = torch.nn.functional.softmax(output, dim = 1)[0] * 100
  print("TOP 1 predictions {} and confidence of : {}".format(classes[index[0]], percentage[index[0].item()]))
  # Finding index where we get the TOP 5 maximum score
  _, predicted = torch.sort(output, descending= True)
  predictions = [classes[idx] for idx in predicted[0][:5]]
  print([(classes[idx], percentage[idx].item()) for idx in predicted[0][:5]])
  return predictions


def pred_restnet(image):
  model_resnet.eval()
  output_resnet = model_resnet(image)
  return predict_class(output_resnet)


pred_resnet = pred_restnet(true_batch_t)
label_idx  = pred_resnet[0].split(':')[0]

output = model_resnet.forward(img_variable)
label_idx = torch.max(output.data, 1)[1][0]   #get an index(class number) of a largest element
print(label_idx)
```

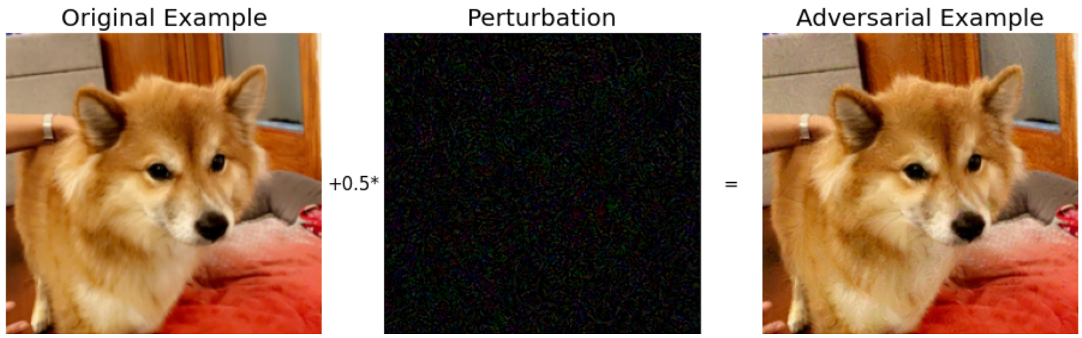
```
TOP 1 predictions 263: 'Pembroke, Pembroke Welsh corgi', and confidence of : 49.826072692871094
[("263: 'Pembroke, Pembroke Welsh corgi',", 49.826072692871094), ("259: 'Pomeranian',", 34.52444076538086), ("260: 'chow, chow chow',", 3.3401818275451
tensor(263)
```

```
def load_classes(label_idx):
  with open('/content/drive/MyDrive/Khoury Courses/Deep Learning/HW/imagenet1000_clsidx_to_labels.txt') as f:
      classes = [line.strip() for line in f.readlines()]
  x_pred = classes[int(label_idx)]
  return x_pred, classes

x_pred, classes = load_classes(label_idx)


#get probability dist over classes
output_probs = F.softmax(output, dim=1)
x_pred_prob =  np.round((torch.max(output_probs.data, 1)[0][0]) * 100,4)


def visualize(x, x_adv, x_grad, epsilon, clean_pred, adv_pred, clean_prob, adv_prob):

    x = x.squeeze(0)
    x = x.mul(torch.FloatTensor(std).view(3,1,1)).add(torch.FloatTensor(mean).view(3,1,1)).numpy()#reverse of normalization op- "unnormal
    x = np.transpose( x , (1,2,0))
    x = np.clip(x, 0, 1)

    x_adv = x_adv.squeeze(0)
    x_adv = x_adv.mul(torch.FloatTensor(std).view(3,1,1)).add(torch.FloatTensor(mean).view(3,1,1)).numpy()#reverse of normalization op
    x_adv = np.transpose( x_adv , (1,2,0))
```

```
    x_adv = np.clip(x_adv, 0, 1)

    x_grad = x_grad.squeeze(0).numpy()
    x_grad = np.transpose(x_grad, (1,2,0))
    x_grad = np.clip(x_grad, 0, 1)

    figure, ax = plt.subplots(1,3, figsize=(16,6))
    ax[0].imshow(x)
    ax[0].set_title('Original Example', fontsize=20)


    ax[1].imshow(x_grad)
    ax[1].set_title('Perturbation', fontsize=20)
    ax[1].set_yticklabels([])
    ax[1].set_xticklabels([])
    ax[1].set_xticks([])
    ax[1].set_yticks([])


    ax[2].imshow(x_adv)
    ax[2].set_title('Adversarial Example', fontsize=20)

    ax[0].axis('off')
    ax[2].axis('off')

    ax[0].text(1.1,0.5, "+{}*".format(round(epsilon,3)), size=15, ha="center",
             transform=ax[0].transAxes)

    ax[0].text(0.5,-0.13, "Prediction: {}\n Probability: {}".format(clean_pred, clean_prob), size=15, ha="center",
         transform=ax[0].transAxes)

    ax[1].text(1.1,0.5, " = ", size=15, ha="center", transform=ax[1].transAxes)

    ax[2].text(0.5,-0.13, "Prediction: {}\n Probability: {}".format(adv_pred, adv_prob), size=15, ha="center",
         transform=ax[2].transAxes)


    plt.show()

y_leastLikely = torch.min(output.data, 1)[1][0]
print(y_leastLikely.item(), classes[int(y_leastLikely)]) #least likely class

y_target = Variable(torch.LongTensor([y_leastLikely.item()]), requires_grad=False)

#Parameters
epsilon = 0.5
num_steps = 10
alpha = 0.025
```

```
324 324: 'cabbage butterfly',
```

```
img_variable.data = true_batch_t


for i in range(num_steps):
  zero_gradients(img_variable)
  output = model_resnet.forward(img_variable)

  loss = torch.nn.CrossEntropyLoss()
  loss_cal = loss(output, y_target)
  loss_cal.backward()
  x_grad = alpha * torch.sign(img_variable.grad.data)
  adv_temp = img_variable.data - x_grad
  total_grad = adv_temp - true_batch_t
  total_grad = torch.clamp(total_grad, -epsilon, epsilon)
  x_adv = true_batch_t + total_grad
  img_variable.data = x_adv


output_adv = model_resnet.forward(img_variable)
x_adv_pred = classes[torch.max(output_adv.data, 1)[1][0]]
output_adv_probs = F.softmax(output_adv, dim=1)
x_adv_pred_prob =  np.round((torch.max(output_adv_probs.data, 1)[0][0]) * 100,4)
visualize(true_batch_t, img_variable.data, total_grad, epsilon, x_pred,x_adv_pred, x_pred_prob,  x_adv_pred_prob)
```



✓ 0s completed at 23:01