CS7150.37338.202130 Homework 3

Aishwarya Vantipuli, Aveek Choudhury, Harshita Ved

TOTAL POINTS

4.85 / 5

QUESTION 1

Question 1 4 pts

1.1 a 1 / 1

✓ - **0 pts** Correct

- **0.05 pts** Incorrect answer
- **0.1 pts** Miscalculation
- **0.1 pts** Expectation Calculation is inccorect
- **0.025 pts** Can calculate it directly.
- **0.5 pts** Missing work

The correct answer, but you can calculate it directly.

1.2 b 1 / 1

✓ - **0 pts** Correct

1.3 c 1 / 1

✓ - **0 pts** Correct

1.4 d 0.95 / 1

- **0 pts** Correct

✓ - **0.05 pts** Comment on Results

Please make a comment about the change in performance. E.g. higher of lower mean or std.

QUESTION 2

2 Question 2 0.9 / 1

- **0 pts** Correct
- **0.05 pts** Want adversarial example to increase in confidence
- **0.1 pts** Show small perturbation (distortion) separate from the image.
- **0.1 pts** State the method

✓ - **0.1 pts** Misinterped Results

Misinterped results confidence cannot be zero

gradescope

CS 7150: Deep Learning — Spring 2021— Paul Hand

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

Names: **Aishwarya Vantipuli, Aveek Choudhury, Harshita Ved**

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: The **Peak Signal-to-Noise Ratio** is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

From reference link: PSNR is most easily defined via the mean squared error (MSE). Given a noise-free m×n monochrome image I and its noisy approximation K, MSE is defined as:

$$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

The PSNR (in dB) is defined as:

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE) \end{aligned}$$

Here,
x = a color image with values scaled within [0,1] i.e. max pixel value is 1.

y = 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

$\eta[.,.]$ = additive Gaussian noise with mean 0 and variance σ^2. Can also be represented by random variable ν

Using the equation above for PSNR and MSE,

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

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

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

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

Given, expected PSNR between x and y is of approximately 20 dB, we get $\mathbb{E}$[PSNR] = 20

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

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

Using Jensen's inequality:
$\mathbb{E}[-log_{10}\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{m.n}] \geq -log_{10}\mathbb{E}[\frac{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\eta[i,j])^2}{m.n}]$

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

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

$2 \geq -log_{10}\ \frac{1}{m.n}.mn\ \mathbb{E}[(\nu)^2]$

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

Now we know that,
a) $\mathbb{E}[(\nu)^2] = Var[\nu] - \mathbb{E}[(\nu)]^2$
b) $\mathbb{E}[(\nu)]^2$ is 0 as noise is Gaussian with mean 0

Substituting back in equation,

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

$\frac{1}{\sigma^2} \leq 100$

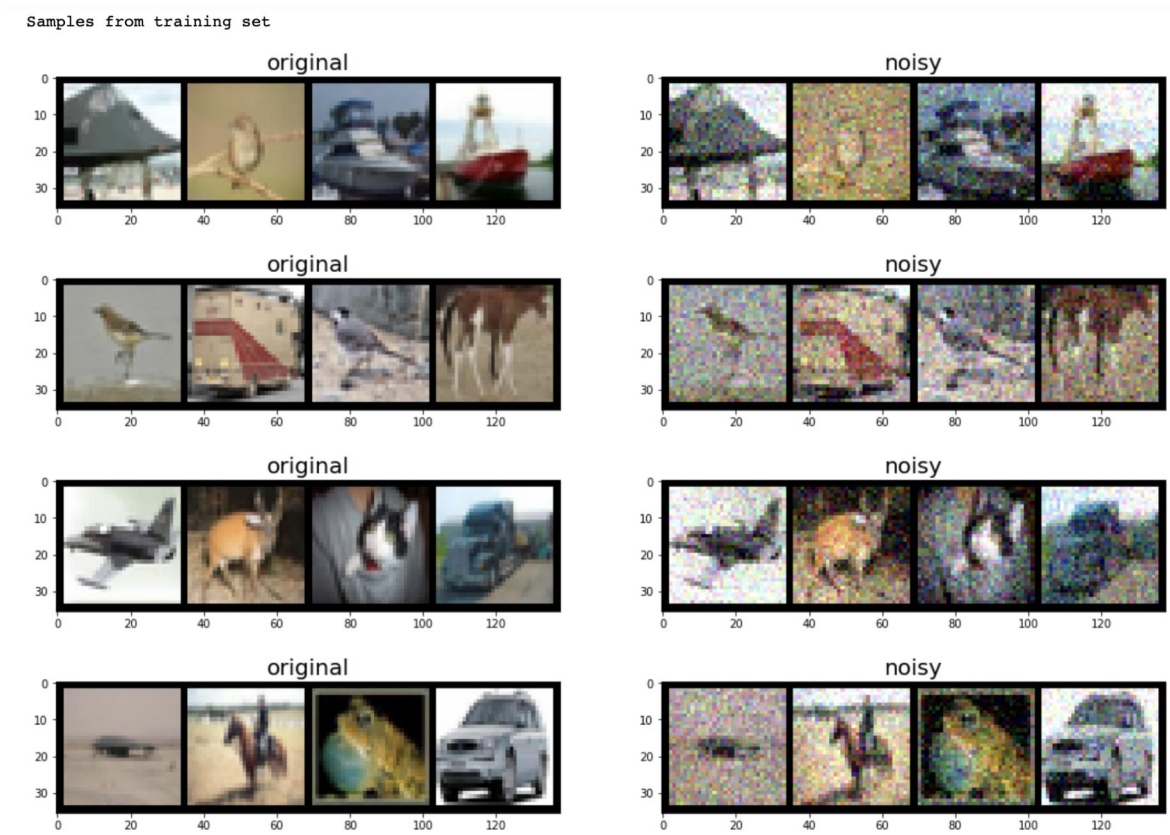
$\sigma \geq 0.1$

Hence value of $\sigma = 0.1$ or greater corresponds to an expected PSNR between x and y of approximately 20 dB.

(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: As seen from above question, value of $\sigma = 0.1$ or greater corresponds to an expected PSNR between x and y of approximately 20 dB. So we created a noisy version of the CIFAR-10 training and test dataset having Gaussian noise with $\sigma = 0.1$, below are the results obtained.

Generated PSNR has Mean: 20.3212 dB and Std Deviation: 0.1620



1.1 a **1 / 1**

✓ **- 0 pts** **Correct**

- 0.05 pts Incorrect answer

- 0.1 pts Miscalculation

- 0.1 pts Expectation Calculation is inccorect

- 0.025 pts Can calculate it directly.

- 0.5 pts Missing work

The correct answer, but you can calculate it directly.

gradescope

$\frac{1}{\sigma^2} \leq 100$

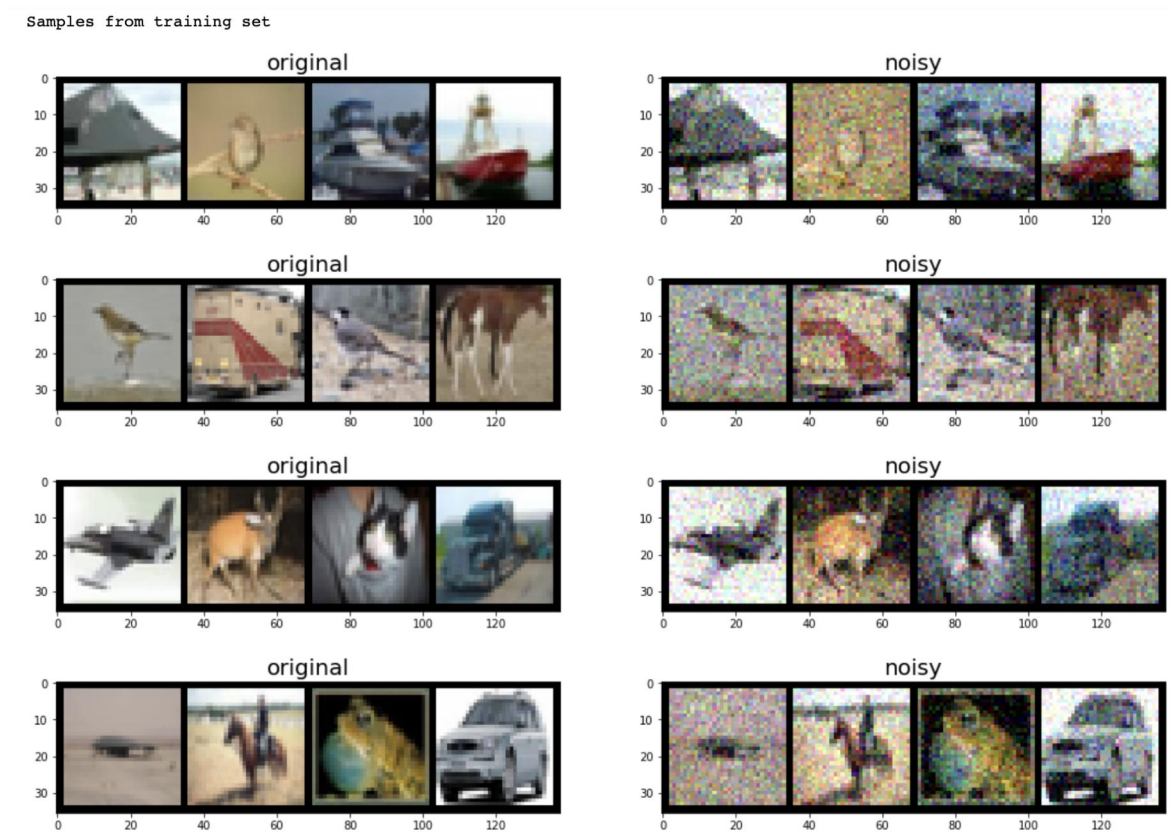
$\sigma \geq 0.1$

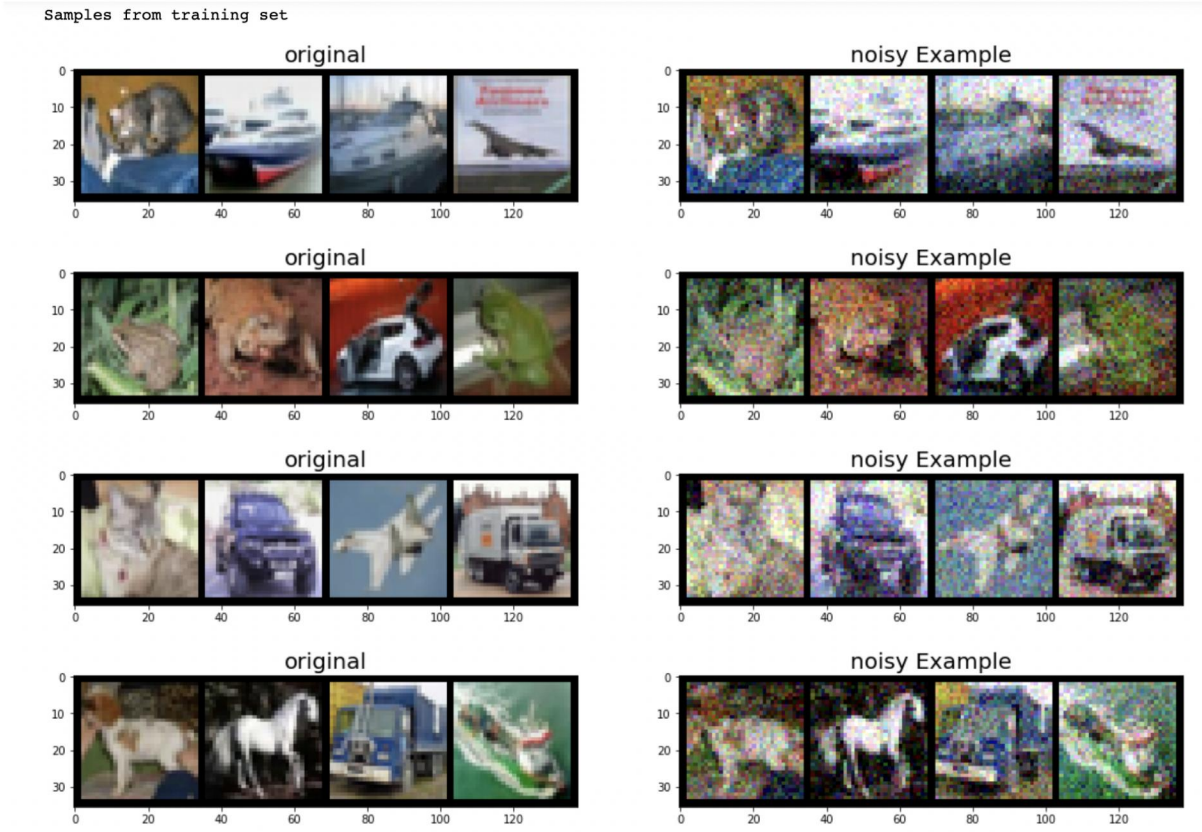
Hence value of $\sigma = 0.1$ or greater corresponds to an expected PSNR between x and y of approximately 20 dB.

(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: As seen from above question, value of $\sigma = 0.1$ or greater corresponds to an expected PSNR between x and y of approximately 20 dB. So we created a noisy version of the CIFAR-10 training and test dataset having Gaussian noise with $\sigma = 0.1$, below are the results obtained.

Generated PSNR has Mean: 20.3212 dB and Std Deviation: 0.1620





(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: We trained a ResNet to denoise noisy CIFAR-10 images and as a result obtained denoised 32x32 px image. We started with a Conv2d layer going from 3 to 16 (later 16 to 32, and 32 to 64), then added BatchNorm and ReLU. We implemented residual blocks in between where first one is to maintain dimensions. We have total of 14 layers. The mean and standard deviation of the recovery PSNRs over the noisy test set is: **Avg. PSNR**: 27.37 dB and **Std. Dev PSNR**: 1.68
Training details:
Optimizer: Adam
Epochs = 10
Learning Rate = 0.001
Batch Size = 4
Below is more detailed description of our architecture.

Q1. Part b and c

March 26, 2021

[1]:
```
import torch
print(torch.__version__)
```

```
1.8.0+cu101
```

[2]:
```
import torchvision
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch import nn, optim
import matplotlib.pyplot as plt
from time import time
import numpy as np
from six.moves import urllib
import random
```

[3]:
```
from skimage.util import random_noise
import pandas as pd
from math import log10
import torch.nn as nn
```

[4]:
```
criterion = nn.MSELoss()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[5]:
```
# patch to allow mnist data download using browser header - security bypass␣
↪since lecun's server moved to cloudflare
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
```

[6]:
```
# Read the train and test sets of CIFAR-10 data
default_transform = transforms.Compose([transforms.ToTensor()])
cifar_trainset = datasets.CIFAR10(root='./data', train=True, download=True,␣
↪transform=default_transform)
cifar_testset = datasets.CIFAR10(root='./data', train=False, download=True,␣
↪transform=default_transform)
print("Training set size:", len(cifar_trainset))
print("Test set size:", len(cifar_testset))
```

```
Files already downloaded and verified
Files already downloaded and verified
Training set size: 50000
Test set size: 10000
```

[7]:
```
# Initialize data loader functions
BATCH_SIZE = 4
train_dataLoader = DataLoader(cifar_trainset, batch_size=BATCH_SIZE,␣
↪shuffle=True)
test_dataLoader = DataLoader(cifar_testset, batch_size=BATCH_SIZE,␣
↪shuffle=False)
```

[8]:
```
# Validate shape of the input images
dataiter = iter(train_dataLoader)
images, labels = dataiter.next()

print(images.shape)
print(labels.shape)
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4])
```

0.0.1 1 (b). Generate noisy CIFAR-10 dataset with std. deviation = 0.1 from 1 (a).

[13]:
```
def calculate_psnr(pred, actual):
  psnr = []
  for i, j in zip(pred, actual):
    mse = torch.mean((i - j)**2)
    psnr_val = -10 * torch.log10(mse)
    psnr.append(psnr_val.item())
  return np.mean(psnr), np.std(psnr)
```

[18]:
```
# Add gaussian noise to data -- variance = std_dev^2
def add_noise(dataLoader, noise_mode='gaussian', noise_mean=0, noise_var=0.01):
  original, noisy = [], []
  psnr = []
  for data in dataLoader:
    img, labels = data[0], data[1]
    original.append(img)
    gauss_img = torch.tensor(random_noise(img, mode=noise_mode,␣
↪mean=noise_mean, var=noise_var, clip=True))
    noisy.append(gauss_img)

  avg_psnr, std_psnr = calculate_psnr(noisy, original)
  print("Avg. PSNR between Noisy and Original images: {:.4f} dB \t Std.␣
↪Deviation {:.4f}".format(avg_psnr, std_psnr))
  return original, noisy
```

[72]:
```
# Add noise to train set
image_dict_train = {}
image_dict_train['original'], image_dict_train['noisy'] = ␣
↪add_noise(train_dataLoader)
```

```
Avg. PSNR between Noisy and Original images: 20.3212 dB          Std. Deviation
0.1620
```

[19]:
```
# Add noise to test set
image_dict_test = {}
image_dict_test['original'], image_dict_test['noisy'] = ␣
↪add_noise(test_dataLoader)
```

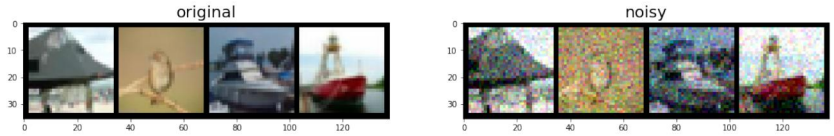
```
Avg. PSNR between Noisy and Original images: 20.3168 dB          Std. Deviation
0.1624
```

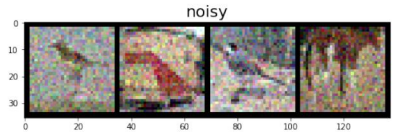
[20]:
```
train_df = pd.DataFrame(image_dict_train)
test_df = pd.DataFrame(image_dict_test)
```

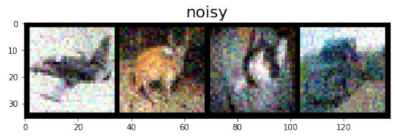
[68]:
```
def display_dataframe(df,cols):
  for i in range(len(df)):
    figure, ax = plt.subplots(1,len(cols), figsize=(18,8))
    for idx in range(len(cols)):
      x = df[cols[idx]][i]
      im = torchvision.utils.make_grid(x)
      x = np.transpose( im.cpu().numpy() , (1,2,0))
      x = np.clip(x, 0, 1)

      ax[idx].imshow(x)
      ax[idx].set_title('{}'.format(cols[idx]), fontsize=20)
```

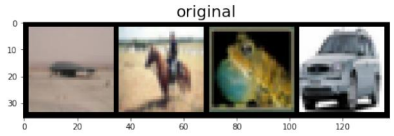
[69]:
```
print("Samples from training set")
display_dataframe(train_df[:4], ['original', 'noisy'])
```

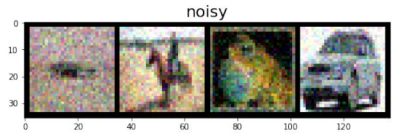
```
Samples from training set
```





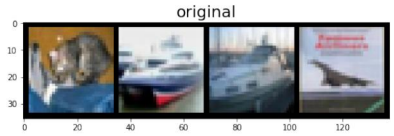


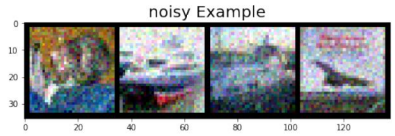


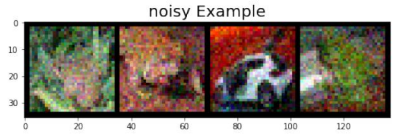


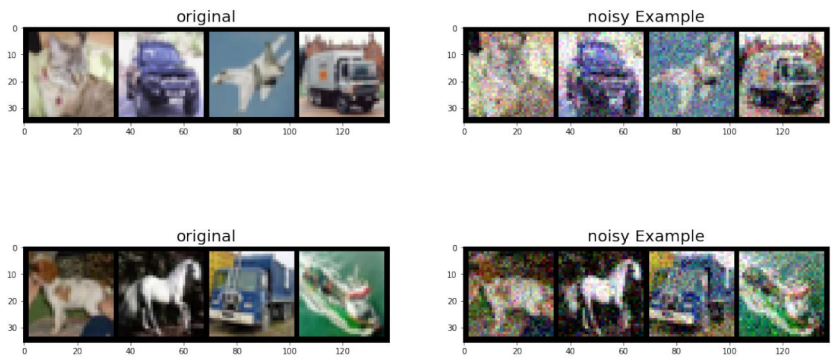
[62]:
```
print("Samples from training set")
display_dataframe(test_df[:4], ['original', 'noisy'])
```

```
Samples from training set
```









0.1 Denoising with ResNets

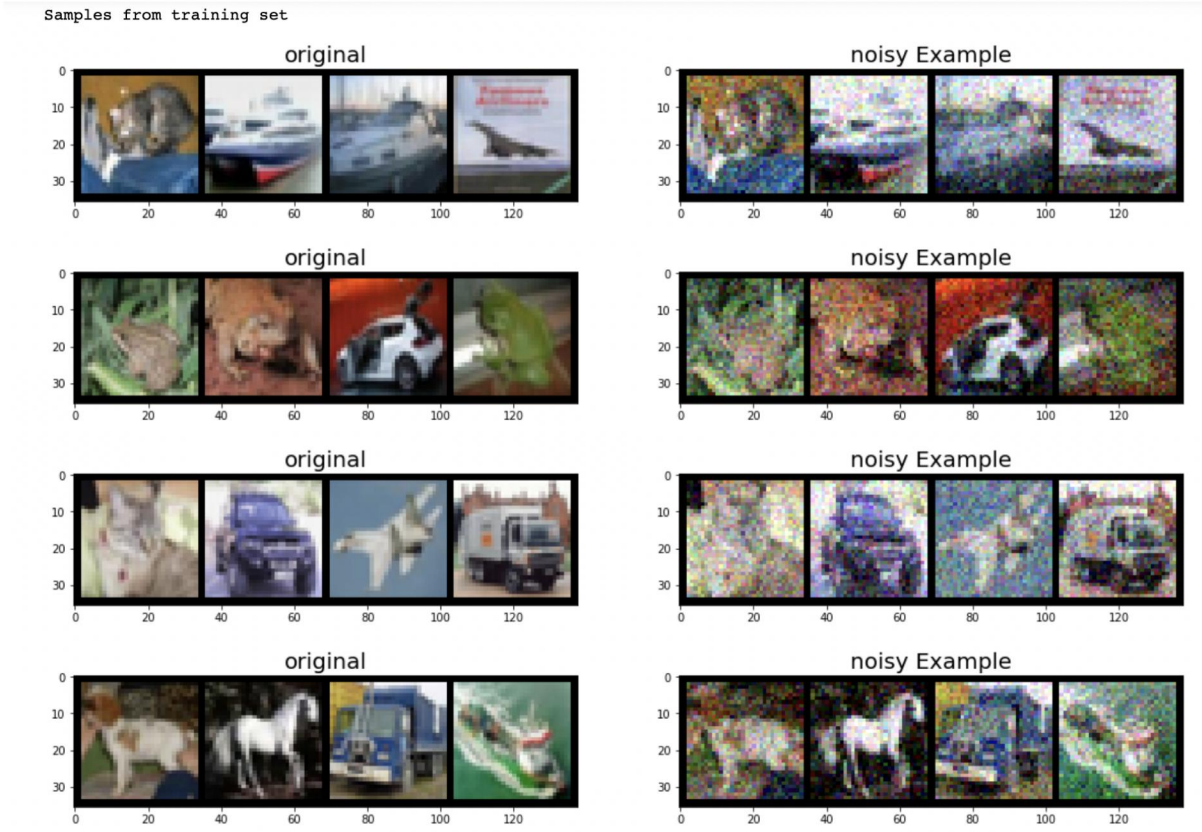
[22]:
```
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)
```

[23]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, skip, 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)
        self.downsample = downsample
        self.skip = skip

    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)
        if self.downsample is not None:
            residual = self.downsample(x)
        if self.skip:
          out += residual
        out = self.relu(out)
```

1.2 b 1 / 1

✓ - 0 pts Correct

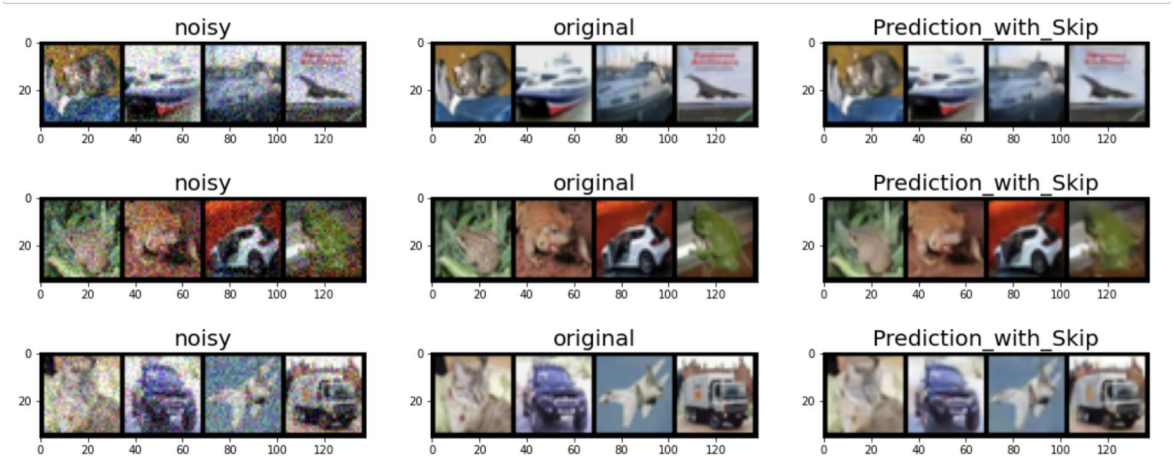
gradescope



(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.

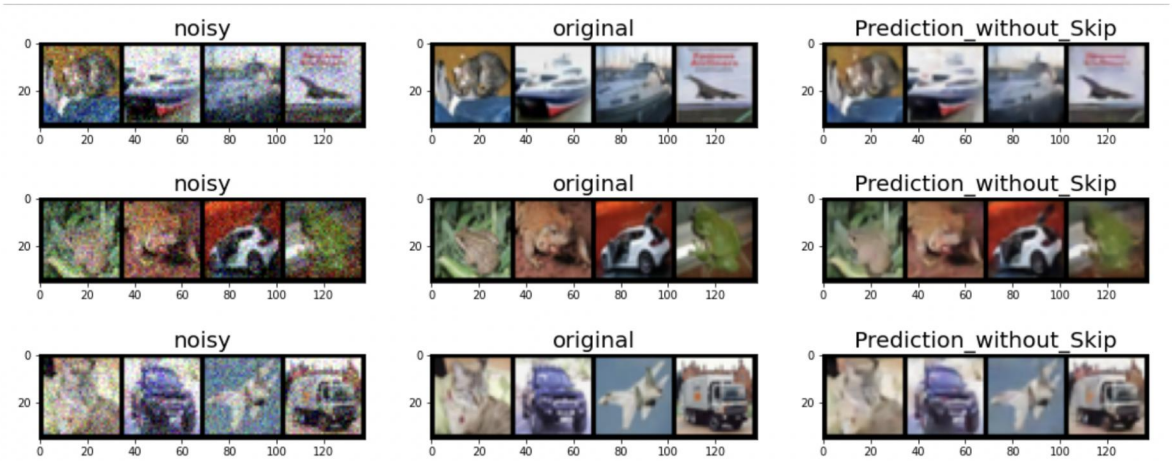
Response: We trained a ResNet to denoise noisy CIFAR-10 images and as a result obtained denoised 32x32 px image. We started with a Conv2d layer going from 3 to 16 (later 16 to 32, and 32 to 64), then added BatchNorm and ReLU. We implemented residual blocks in between where first one is to maintain dimensions. We have total of 14 layers. The mean and standard deviation of the recovery PSNRs over the noisy test set is: **Avg. PSNR**: 27.37 dB and **Std. Dev PSNR**: 1.68
Training details:
Optimizer: Adam
Epochs = 10
Learning Rate = 0.001
Batch Size = 4
Below is more detailed description of our architecture.

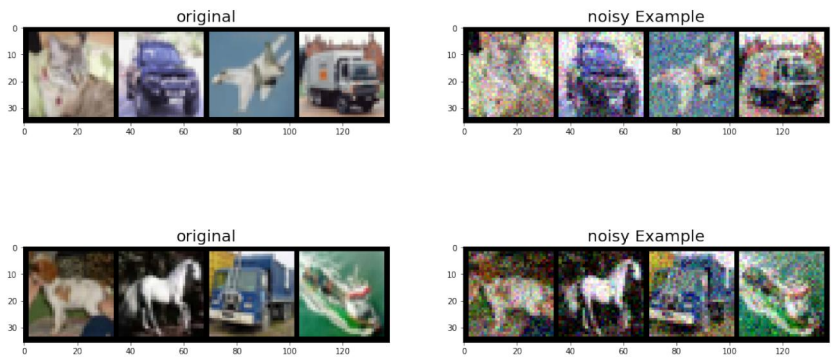
```
ResNet(
  (conv): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): ResidualBlock(
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(32, 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)
      (downsample): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): 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)
    )
  )
  (conv_out): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
```



(d) Repeat the previous task but without the skip connections in your model.

Response: The Model architecture and training details stays the same as above part, we are just removing the skip connection layer and obtaining the results. The mean and standard deviation of the PSNRs over the noisy test set with No skip connection is: **Avg. PSNR**: 27.75 DB and **Std. Dev PSNR**: 0.62





0.1 Denoising with ResNets

[22]:
```
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)
```

[23]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, skip, 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)
        self.downsample = downsample
        self.skip = skip

    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)
        if self.downsample is not None:
            residual = self.downsample(x)
        if self.skip:
          out += residual
        out = self.relu(out)
```

```
        return out
```

[24]:
```
# ResNet (Referenced from Pytorch official tutorial)
class ResNet(nn.Module):
    def __init__(self, block, layers, skip=True):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.skip = skip
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1])
        self.layer3 = self.make_layer(block, 64, layers[2])

        self.conv_out = conv3x3(64,3)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if ((stride != 1) or (self.in_channels != out_channels)):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, self.skip,stride,␣
↪downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels, self.skip))
        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.layer3(out)
        out = self.conv_out(out)

        return out
```

[33]:
```
# Hyper-parameters
num_epochs = 10
learning_rate = 0.001
```

0.1.1 1 (c). Denoising using ResNet with Skip Connections

[34]:
```
model_skip = ResNet(ResidualBlock, [2, 2, 2]).to(device)

# Loss and optimizer
criterion_skip = nn.MSELoss()
optimizer_skip = torch.optim.Adam(model_skip.parameters(), lr=learning_rate)
```

[36]:
```
# Train the model
total_step_skip = len(train_df)
curr_lr_skip = learning_rate
for epoch in range(num_epochs):
    running_loss = 0
    for idx, row in train_df.iterrows():
        y_skip = row['original'].to(device)
        X_skip = row['noisy'].to(device)

        # Forward pass
        outputs_skip = model_skip(X_skip.float())
        loss = criterion_skip(outputs_skip, y_skip)

        # Backward and optimize
        optimizer_skip.zero_grad()
        loss.backward()
        optimizer_skip.step()
        running_loss += loss.item()

        if (idx+1) % 2000 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, idx+1, total_step_skip, loss.
 ↪item()))
    print("Epoch: {} Running Loss: {:.4f}".format(epoch+1, running_loss/
 ↪total_step_skip))
```

```
Epoch [1/10], Step [2000/12500] Loss: 0.0059
Epoch [1/10], Step [4000/12500] Loss: 0.0048
Epoch [1/10], Step [6000/12500] Loss: 0.0034
Epoch [1/10], Step [8000/12500] Loss: 0.0041
Epoch [1/10], Step [10000/12500] Loss: 0.0023
Epoch [1/10], Step [12000/12500] Loss: 0.0028
Epoch: 1 Running Loss: 0.0052
Epoch [2/10], Step [2000/12500] Loss: 0.0028
Epoch [2/10], Step [4000/12500] Loss: 0.0035
Epoch [2/10], Step [6000/12500] Loss: 0.0029
Epoch [2/10], Step [8000/12500] Loss: 0.0020
Epoch [2/10], Step [10000/12500] Loss: 0.0017
Epoch [2/10], Step [12000/12500] Loss: 0.0022
Epoch: 2 Running Loss: 0.0025
```

```
Epoch [3/10], Step [2000/12500] Loss: 0.0019
Epoch [3/10], Step [4000/12500] Loss: 0.0031
Epoch [3/10], Step [6000/12500] Loss: 0.0028
Epoch [3/10], Step [8000/12500] Loss: 0.0019
Epoch [3/10], Step [10000/12500] Loss: 0.0017
Epoch [3/10], Step [12000/12500] Loss: 0.0021
Epoch: 3 Running Loss: 0.0021
Epoch [4/10], Step [2000/12500] Loss: 0.0017
Epoch [4/10], Step [4000/12500] Loss: 0.0028
Epoch [4/10], Step [6000/12500] Loss: 0.0030
Epoch [4/10], Step [8000/12500] Loss: 0.0017
Epoch [4/10], Step [10000/12500] Loss: 0.0016
Epoch [4/10], Step [12000/12500] Loss: 0.0020
Epoch: 4 Running Loss: 0.0019
Epoch [5/10], Step [2000/12500] Loss: 0.0018
Epoch [5/10], Step [4000/12500] Loss: 0.0024
Epoch [5/10], Step [6000/12500] Loss: 0.0031
Epoch [5/10], Step [8000/12500] Loss: 0.0018
Epoch [5/10], Step [10000/12500] Loss: 0.0016
Epoch [5/10], Step [12000/12500] Loss: 0.0020
Epoch: 5 Running Loss: 0.0018
Epoch [6/10], Step [2000/12500] Loss: 0.0017
Epoch [6/10], Step [4000/12500] Loss: 0.0021
Epoch [6/10], Step [6000/12500] Loss: 0.0029
Epoch [6/10], Step [8000/12500] Loss: 0.0018
Epoch [6/10], Step [10000/12500] Loss: 0.0016
Epoch [6/10], Step [12000/12500] Loss: 0.0019
Epoch: 6 Running Loss: 0.0018
Epoch [7/10], Step [2000/12500] Loss: 0.0017
Epoch [7/10], Step [4000/12500] Loss: 0.0021
Epoch [7/10], Step [6000/12500] Loss: 0.0029
Epoch [7/10], Step [8000/12500] Loss: 0.0019
Epoch [7/10], Step [10000/12500] Loss: 0.0016
Epoch [7/10], Step [12000/12500] Loss: 0.0019
Epoch: 7 Running Loss: 0.0017
Epoch [8/10], Step [2000/12500] Loss: 0.0016
Epoch [8/10], Step [4000/12500] Loss: 0.0020
Epoch [8/10], Step [6000/12500] Loss: 0.0029
Epoch [8/10], Step [8000/12500] Loss: 0.0019
Epoch [8/10], Step [10000/12500] Loss: 0.0016
Epoch [8/10], Step [12000/12500] Loss: 0.0019
Epoch: 8 Running Loss: 0.0017
Epoch [9/10], Step [2000/12500] Loss: 0.0015
Epoch [9/10], Step [4000/12500] Loss: 0.0020
Epoch [9/10], Step [6000/12500] Loss: 0.0031
Epoch [9/10], Step [8000/12500] Loss: 0.0019
Epoch [9/10], Step [10000/12500] Loss: 0.0016
Epoch [9/10], Step [12000/12500] Loss: 0.0019
```

```
Epoch: 9 Running Loss: 0.0017
Epoch [10/10], Step [2000/12500] Loss: 0.0016
Epoch [10/10], Step [4000/12500] Loss: 0.0018
Epoch [10/10], Step [6000/12500] Loss: 0.0028
Epoch [10/10], Step [8000/12500] Loss: 0.0018
Epoch [10/10], Step [10000/12500] Loss: 0.0015
Epoch [10/10], Step [12000/12500] Loss: 0.0018
Epoch: 10 Running Loss: 0.0016
```

[37]:
```
# Get predictions on test set - model with skip connections
model_skip.eval()
with torch.no_grad():
    y_pred_skip = []
    y_test_skip = []

    for idx, row in test_df.iterrows():
        y_test_skip.append(row['original'].to(device))
        x_test_skip = row['noisy'].to(device)
        outputs_skip = model_skip(x_test_skip.float())
        y_pred_skip.append(outputs_skip)
```

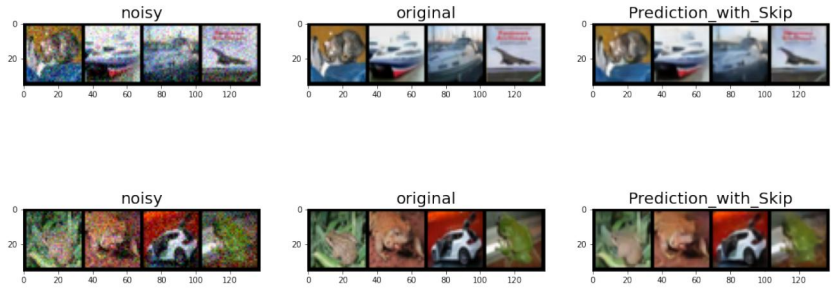
[39]:
```
avg_psnr_skip, std_psnr_skip = calculate_psnr(y_pred_skip, y_test_skip)
print("Avg. PSNR between Noisy and Original images: {:.4f} dB \t Std. Deviation␣
↪{:.4f}".format(avg_psnr_skip, std_psnr_skip))
```

```
Avg. PSNR between Noisy and Original images: 27.3717 dB          Std. Deviation
1.6864
```

[40]:
```
# save model
torch.save(model_skip.state_dict(), 'resnet_with_skip_v2.pt')
```

[63]:
```
test_df['Prediction_with_Skip'] = y_pred_skip
```

[70]:
```
display_dataframe(test_df[:3], ['noisy', 'original', 'Prediction_with_Skip'])
```



0.1.2 1 (d). Denoising using ResNet without Skip Connections

[41]:
```
model_noSkip = ResNet(ResidualBlock, [2, 2, 2], skip=False).to(device)


# Loss and optimizer
criterion_noSkip = nn.MSELoss()
optimizer_noSkip = torch.optim.Adam(model_noSkip.parameters(), lr=learning_rate)
```

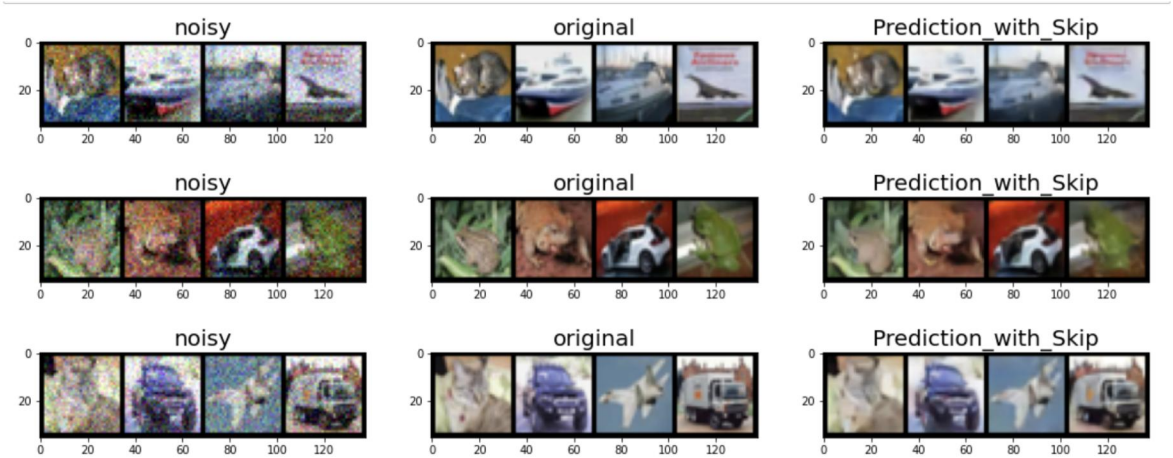
[42]:
```
# Train the model
total_step_noSkip = len(train_df)
curr_lr_noSkip = learning_rate

for epoch in range(num_epochs):
    running_loss = 0
    for idx, row in train_df.iterrows():
        y_noSkip = row['original'].to(device)
        X_noSkip = row['noisy'].to(device)

        # Forward pass
        outputs_noSkip = model_noSkip(X_noSkip.float())
        loss = criterion_noSkip(outputs_noSkip, y_noSkip)

        # Backward and optimize
        optimizer_noSkip.zero_grad()
        loss.backward()
        optimizer_noSkip.step()
        running_loss += loss.item()

        if (idx+1) % 2000 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, idx+1, total_step_noSkip, loss.
 ↪item()))
    print("Epoch: {} Running Loss: {:.4f}".format(epoch+1, running_loss/
 ↪total_step_noSkip))
```

```
Epoch [1/10], Step [2000/12500] Loss: 0.0045
Epoch [1/10], Step [4000/12500] Loss: 0.0046
Epoch [1/10], Step [6000/12500] Loss: 0.0078
```

1.3 C **1 / 1**

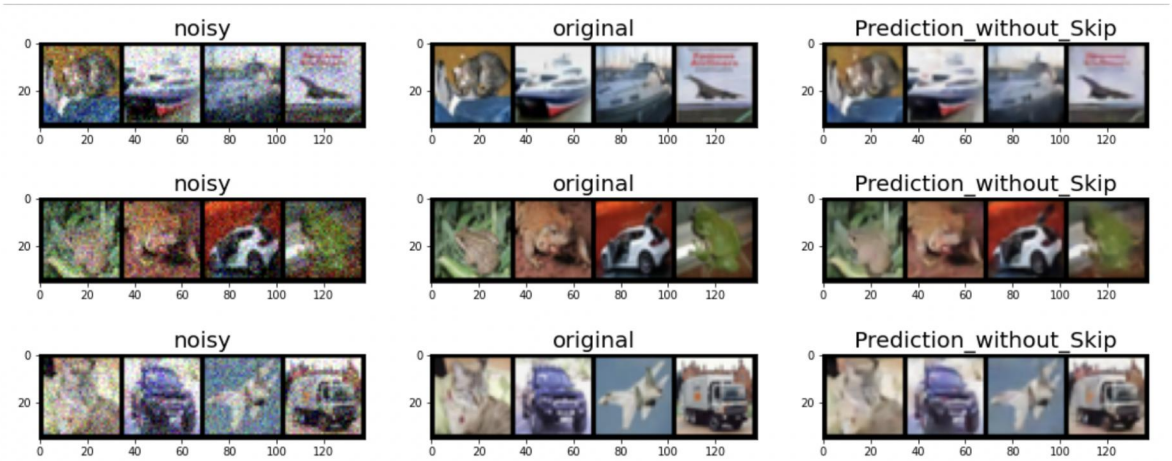
✓ **- 0 pts** Correct

gradescope



(d) Repeat the previous task but without the skip connections in your model.

Response: The Model architecture and training details stays the same as above part, we are just removing the skip connection layer and obtaining the results. The mean and standard deviation of the PSNRs over the noisy test set with No skip connection is: **Avg. PSNR**: 27.75 DB and **Std. Dev PSNR**: 0.62



0.1.2 1 (d). Denoising using ResNet without Skip Connections

[41]:
```
model_noSkip = ResNet(ResidualBlock, [2, 2, 2], skip=False).to(device)


# Loss and optimizer
criterion_noSkip = nn.MSELoss()
optimizer_noSkip = torch.optim.Adam(model_noSkip.parameters(), lr=learning_rate)
```

[42]:
```
# Train the model
total_step_noSkip = len(train_df)
curr_lr_noSkip = learning_rate

for epoch in range(num_epochs):
    running_loss = 0
    for idx, row in train_df.iterrows():
        y_noSkip = row['original'].to(device)
        X_noSkip = row['noisy'].to(device)

        # Forward pass
        outputs_noSkip = model_noSkip(X_noSkip.float())
        loss = criterion_noSkip(outputs_noSkip, y_noSkip)

        # Backward and optimize
        optimizer_noSkip.zero_grad()
        loss.backward()
        optimizer_noSkip.step()
        running_loss += loss.item()

        if (idx+1) % 2000 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, idx+1, total_step_noSkip, loss.
 ↪item()))
    print("Epoch: {} Running Loss: {:.4f}".format(epoch+1, running_loss/
 ↪total_step_noSkip))
```

```
Epoch [1/10], Step [2000/12500] Loss: 0.0045
Epoch [1/10], Step [4000/12500] Loss: 0.0046
Epoch [1/10], Step [6000/12500] Loss: 0.0078
```

```
Epoch [1/10], Step [8000/12500] Loss: 0.0048
Epoch [1/10], Step [10000/12500] Loss: 0.0036
Epoch [1/10], Step [12000/12500] Loss: 0.0046
Epoch: 1 Running Loss: 0.0069
Epoch [2/10], Step [2000/12500] Loss: 0.0026
Epoch [2/10], Step [4000/12500] Loss: 0.0039
Epoch [2/10], Step [6000/12500] Loss: 0.0061
Epoch [2/10], Step [8000/12500] Loss: 0.0024
Epoch [2/10], Step [10000/12500] Loss: 0.0023
Epoch [2/10], Step [12000/12500] Loss: 0.0028
Epoch: 2 Running Loss: 0.0034
Epoch [3/10], Step [2000/12500] Loss: 0.0022
Epoch [3/10], Step [4000/12500] Loss: 0.0037
Epoch [3/10], Step [6000/12500] Loss: 0.0046
Epoch [3/10], Step [8000/12500] Loss: 0.0022
Epoch [3/10], Step [10000/12500] Loss: 0.0023
Epoch [3/10], Step [12000/12500] Loss: 0.0023
Epoch: 3 Running Loss: 0.0028
Epoch [4/10], Step [2000/12500] Loss: 0.0021
Epoch [4/10], Step [4000/12500] Loss: 0.0030
Epoch [4/10], Step [6000/12500] Loss: 0.0044
Epoch [4/10], Step [8000/12500] Loss: 0.0022
Epoch [4/10], Step [10000/12500] Loss: 0.0023
Epoch [4/10], Step [12000/12500] Loss: 0.0022
Epoch: 4 Running Loss: 0.0025
Epoch [5/10], Step [2000/12500] Loss: 0.0020
Epoch [5/10], Step [4000/12500] Loss: 0.0028
Epoch [5/10], Step [6000/12500] Loss: 0.0038
Epoch [5/10], Step [8000/12500] Loss: 0.0019
Epoch [5/10], Step [10000/12500] Loss: 0.0024
Epoch [5/10], Step [12000/12500] Loss: 0.0021
Epoch: 5 Running Loss: 0.0023
Epoch [6/10], Step [2000/12500] Loss: 0.0020
Epoch [6/10], Step [4000/12500] Loss: 0.0027
Epoch [6/10], Step [6000/12500] Loss: 0.0040
Epoch [6/10], Step [8000/12500] Loss: 0.0018
Epoch [6/10], Step [10000/12500] Loss: 0.0021
Epoch [6/10], Step [12000/12500] Loss: 0.0020
Epoch: 6 Running Loss: 0.0022
Epoch [7/10], Step [2000/12500] Loss: 0.0021
Epoch [7/10], Step [4000/12500] Loss: 0.0026
Epoch [7/10], Step [6000/12500] Loss: 0.0035
Epoch [7/10], Step [8000/12500] Loss: 0.0019
Epoch [7/10], Step [10000/12500] Loss: 0.0019
Epoch [7/10], Step [12000/12500] Loss: 0.0020
Epoch: 7 Running Loss: 0.0021
Epoch [8/10], Step [2000/12500] Loss: 0.0022
Epoch [8/10], Step [4000/12500] Loss: 0.0025
```

```
Epoch [8/10], Step [6000/12500] Loss: 0.0035
Epoch [8/10], Step [8000/12500] Loss: 0.0018
Epoch [8/10], Step [10000/12500] Loss: 0.0018
Epoch [8/10], Step [12000/12500] Loss: 0.0020
Epoch: 8 Running Loss: 0.0020
Epoch [9/10], Step [2000/12500] Loss: 0.0021
Epoch [9/10], Step [4000/12500] Loss: 0.0023
Epoch [9/10], Step [6000/12500] Loss: 0.0032
Epoch [9/10], Step [8000/12500] Loss: 0.0018
Epoch [9/10], Step [10000/12500] Loss: 0.0018
Epoch [9/10], Step [12000/12500] Loss: 0.0021
Epoch: 9 Running Loss: 0.0020
Epoch [10/10], Step [2000/12500] Loss: 0.0021
Epoch [10/10], Step [4000/12500] Loss: 0.0022
Epoch [10/10], Step [6000/12500] Loss: 0.0031
Epoch [10/10], Step [8000/12500] Loss: 0.0017
Epoch [10/10], Step [10000/12500] Loss: 0.0018
Epoch [10/10], Step [12000/12500] Loss: 0.0021
Epoch: 10 Running Loss: 0.0019
```

[43]:
```
# Get predictions on test set - model without skip connections
model_noSkip.eval()
with torch.no_grad():
    y_pred_noSkip = []
    y_test_noSkip = []

    for idx, row in test_df.iterrows():
        y_test_noSkip.append(row['original'].to(device))
        x_test_noSkip = row['noisy'].to(device)
        outputs_noSkip = model_noSkip(x_test_noSkip.float())
        y_pred_noSkip.append(outputs_noSkip)
```

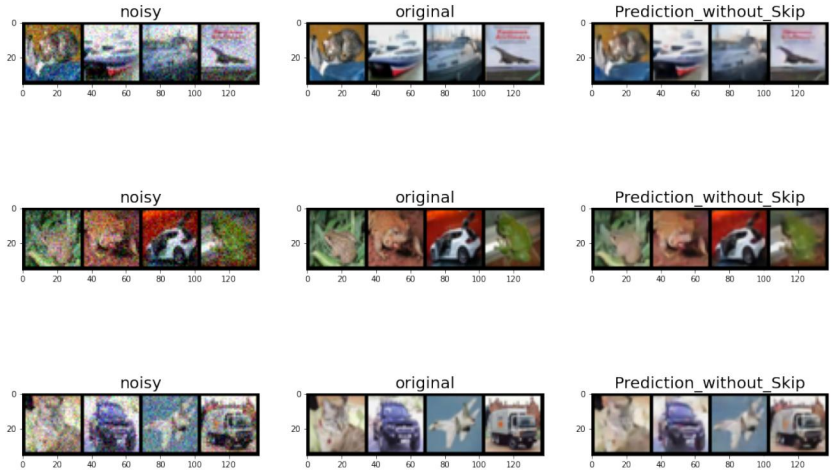
[46]:
```
avg_psnr_noSkip, std_psnr_noSkip = calculate_psnr(y_pred_noSkip, y_test_noSkip)
print("Avg. PSNR between Noisy and Original images: {:.4f} dB \t Std. Deviation␣
↪{:.4f}".format(avg_psnr_noSkip, std_psnr_noSkip))
```

```
Avg. PSNR between Noisy and Original images: 27.7498 dB          Std. Deviation
0.6202
```

[47]:
```
# save model
torch.save(model_noSkip.state_dict(), 'resnet_without_skip_v2.pt')
```

[65]:
```
test_df['Prediction_without_Skip'] = y_pred_noSkip
```

[71]:
```
display_dataframe(test_df[:3], ['noisy', 'original', 'Prediction_without_Skip'])
```


noisy
original
Prediction_without_Skip
noisy
original
Prediction_without_Skip
noisy
original
Prediction_without_Skip

13

1.4 d **0.95 / 1**

- **0 pts** Correct

✓ - **0.05 pts** **Comment on Results**

Please make a comment about the change in performance. E.g. higher of lower mean or std.

gradescope

Question 2. *Adversarial examples*

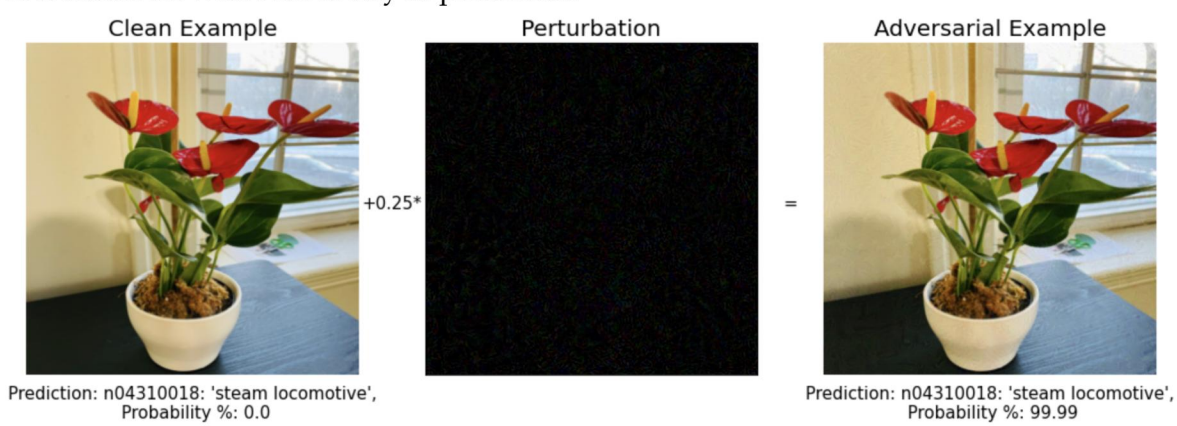
Obtain a pretrained classifier for ImageNet, such as AlexNet or ResNet101 from TorchVision. Using a camera, take a picture of an object that belongs to one of the ImageNet classes. Resize it as appropriate. Select a target class that is different from the image's true class. Compute an adversarial perturbation 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: We have used pre-trained ResNet model. Resized and Normalized the input image. Then classified the image using Resnet and obtained Predicted Class: n03991062: 'pot, flowerpot' with Probability: 54.9%.
For selecting a target class that is different from the image's true class, we chose least probable prediction i.e. n04310018: 'steam locomotive', having Probability: 0.0%.
We calculated an adversarial perturbation that is barely perceptible to the human eye and the results generated are being misclassified as the target class, as shown below.
The adversarial perturbation is created by Basic Iterative Method and as a result, the image is mis-classified as n04310018: 'steam locomotive', having Probability: 99.99%.

Basic Iterative Method: This method is an extension of Fat Gradient Sign Method (FSGM) where it iteratively applies FSGM multiple times with small step size. Adversial image is initialized as original image and then at each iteration, we take one step along the gradient.

$$X_0^{adv} = X$$
$$X_{N+1}^{adv} = Clip_{X,\,\epsilon}\,(\,X_N^{adv} + \alpha\, sign(\,\nabla_X J(X_N^{adv}, Y_{true}\,))$$

Example:
Hyperparameters: epsilon = 0.25
steps = 5
alpha = 0.025 – When we increase alpha, perturbation is more clearly visible than in image below but it trades off with Probability of prediction.



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 itas 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.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount,
call drive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
#Imports
import torch
import torchvision
import torch.nn
from torch.autograd.gradcheck import zero_gradients
from torchvision import datasets, transforms, models
import torch.nn.functional as F
import os
from PIL import Image
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
```

In [ ]:
```
#Setting the environment variable
os.environ['TORCH_HOME'] = '/content/drive/MyDrive/Colab Notebooks/CS 7150
```

In [ ]:
```
# Pretrained ResNet
resnet = torchvision.models.resnet18(pretrained=True)
# print(resnet)
```

In [ ]:
```
#Image of a plant from camera
plantImg = Image.open("/content/drive/MyDrive/Colab Notebooks/CS 7150 Deep
plantImg
```

Out[ ]:

In [ ]:
```
#Resize the image to 256×256 pixels and Convert the image to PyTorch Tensor
mean=[0.485, 0.456, 0.406]
std=[0.229, 0.224, 0.225]
transforms = transforms.Compose([ transforms.Resize((299,299)), transforms.
plantImg_transformed = transforms(plantImg)
# print(plantImg_transformed)
```

In [ ]:
```
#To prepare a batch to be passed through the network
plantbatch_transformed = torch.unsqueeze(plantImg_transformed, 0)
# plantbatch_transformed
```

In [ ]:
```
#Converting tensor as a variable
img_var = Variable(plantbatch_transformed, requires_grad=True) #convert ten
```

In [ ]:
```
with open('/content/drive/MyDrive/Colab Notebooks/CS 7150 Deep Learning/HW1
  classes = [line.strip() for line in f.readlines()]
```

In [ ]:
```
#resnet Model now in eval mode
resnet.eval()
```

Out[ ]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ce
il_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (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): BasicBlock(
      (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): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
```

```
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fal
se)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fal
se)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): BasicBlock(
```

```
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

In [ ]:
```
plant_resN = resnet(plantbatch_transformed)
print(plant_resN.shape)
```

```
torch.Size([1, 1000])
```

In [ ]:
```
with open('/content/drive/MyDrive/Colab Notebooks/CS 7150 Deep Learning/HW1
  classes = [line.strip() for line in f.readlines()]
```

In [ ]:
```
#ResNet Top 5 Predictions for plant image
print("ResNet prediction for Anthurium Plant")
_, indices = torch.sort(plant_resN, descending=True)
probabilities = torch.nn.functional.softmax(plant_resN, dim=1)[0]
[print(classes[idx], "{0:.2f}".format(probabilities[idx].item())) for idx i
```

```
ResNet prediction for Anthurium Plant
n03991062: 'pot, flowerpot', 0.55
n04522168: 'vase', 0.09
n03457902: 'greenhouse, nursery, glasshouse', 0.03
n07747607: 'orange', 0.02
n03944341: 'pinwheel', 0.01
```

Out[ ]:
```
[None, None, None, None, None]
```

In [ ]:
```
#Resnet Top predicted class
label_idx = torch.max(plant_resN.data, 1)[1][0].item()
print('Predicted Class:', classes[label_idx])
tgt_prob =  round(probabilities[indices[0][0]].item() * 100,2)
print('Probability %:', tgt_prob)
```

```
Predicted Class: n03991062: 'pot, flowerpot',
Probability %: 54.89
```

Using Basic Iterative Method to generate the perturbation : Instead of applying the perturbation in a single step, it is applied multiple times with a small step size.

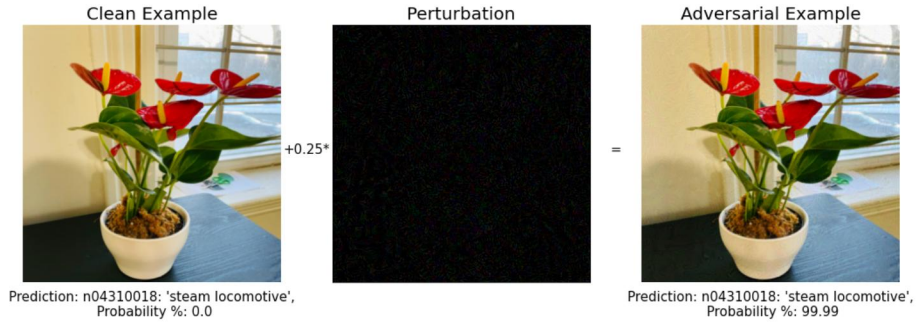
In [ ]:
```
# Selecting target class that is different from the image's true class i.e.
label_idx = torch.min(plant_resN.data, 1)[1][0].item()
tgt_cls = classes[label_idx]
tgt_prob =  round(probabilities[indices[0][-1]].item() * 100,2)
y_tgt = Variable(torch.LongTensor([label_idx]), requires_grad=False)
print('Originally least Predicted Class:', classes[label_idx], 'having Prob
```

```
Originally least Predicted Class: n04310018: 'steam locomotive', having P
robability %: 0.0
```

In [ ]:
```
# Hyperparameters for the attack
eps = 0.25
steps = 5
alpha = 0.025

img_var.data = plantbatch_transformed
for i in range(0, steps):
  zero_gradients(img_var)
  out = resnet(img_var)
  loss_bck = torch.nn.CrossEntropyLoss()(out, y_tgt).backward()
  x_grad = alpha * torch.sign(img_var.grad.data)
  total_grad = img_var.data - x_grad - plantbatch_transformed
  total_grad = torch.clamp(total_grad, -eps, eps)
  adv = plantbatch_transformed + total_grad
  img_var.data = adv

out_adv = resnet(img_var)
adv_img = classes[torch.max(out_adv.data, 1)[1][0]]
out_adv_prob = F.softmax(out_adv, dim=1)
adv_img_prob =  round((torch.max(out_adv_prob.data, 1)[0][0]).item() * 100,
print('Predicted Class:', adv_img)
print('Probability %:' , adv_img_prob)
```

In [ ]:
```
def visualize(x, x_adv, x_grad, epsilon, clean_pred, adv_pred, clean_prob,

    x = x.squeeze(0)     #remove batch dimension # B X C H X W ==> C X H X
    x = x.mul(torch.FloatTensor(std).view(3,1,1)).add(torch.FloatTensor(mea
    x = np.transpose( x , (1,2,0))   # C X H X W  ==>   H X W X C
    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.FloatTe
    x_adv = np.transpose( x_adv , (1,2,0))   # C X H X W  ==>   H X W X C
    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=(18,8))
    ax[0].imshow(x)
    ax[0].set_title('Clean 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="cente
             transform=ax[0].transAxes)

    ax[0].text(0.5,-0.13, "Prediction: {}\n Probability %: {}".format(clean
         transform=ax[0].transAxes)

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

    ax[2].text(0.5,-0.13, "Prediction: {}\n Probability %: {}".format(adv_p
         transform=ax[2].transAxes)


    plt.show()
```

In [19]:
```
visualize(plantbatch_transformed, img_var.data, total_grad, eps, classes[la
```



2 Question 2 0.9 / 1

- **0 pts** Correct
- **0.05 pts** Want adversarial example to increase in confidence
- **0.1 pts** Show small perturbation (distortion) separate from the image.
- **0.1 pts** State the method
- ✓ **0.1 pts** **Misinterped Results**

Misinterped results confidence cannot be zero

gradescope