Autoencoders

Objectives:

- Undercomplete Autoencoder
- Denoising Autoencoder
- Sparse Autoencoder

In [14]:

```
import numpy as np
import matplotlib.pyplot as plt
import random
import os
from tqdm import trange

import torch
import torch.utils.data
from torch import nn, optim
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image
%matplotlib inline
```

In [15]:

```
RANDOM_SEED = 100
BATCH_SIZE = 400
DROP_LAST = True
NUM_WORKERS = 1
PIN_MEMORY = True
NUM_EPOCHS = 100
```

In [16]:

```
def check_cuda():
    cuda_available = torch.cuda.is_available()
    device = torch.device('cuda' if cuda_available else 'cpu')
    print('cuda_available: {}, device: {}'.format(cuda_available, device))
    return cuda_available, device
```

In [17]:

```
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
cuda_available, device = check_cuda()

LOADER_KWARGS = {'num_workers': NUM_WORKERS,
                 'pin_memory': PIN_MEMORY} if cuda_available else {}
```

```
cuda_available: True, device: cuda
```

In [18]:

```
# Download the MNIST train and test sets
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())

mnist_train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_siz
e=BATCH_SIZE, shuffle=True, drop_last=DROP_LAST)
mnist_test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=
BATCH_SIZE, shuffle=True, drop_last=DROP_LAST)
```

Undercomplete Autoencoder

Loss function: $L(x, g(f(x)))$

where x is the input, $f()$ is the encoder and $g()$ is the decoder.

In [19]:

```
# Define the structure of the autoencoder

class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 12),
            nn.ReLU(inplace=True),
            nn.Linear(12, 10))
        self.decoder = nn.Sequential(
            nn.Linear(10, 12),
            nn.ReLU(inplace=True),
            nn.Linear(12, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 28 * 28),
            nn.Tanh())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

In [20]:

```
# Instantiate the model and set the loss criterion and optimizer
model = Autoencoder().to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e
-5)
```

In [21]:

```python
# Training function

def train(train_loader, model, criterion, optimizer, num_epochs):
    epoch_losses = []
    for epoch in trange(num_epochs):
        for data in mnist_train_loader:
            img = data[0].to(device)
            # We don't utilize the target data[1]
            img = img.view(img.size(0), -1)
            output = model(img)
            loss = criterion(output, img)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        loss_value = loss.item()
        epoch_losses.append(loss_value)
    return epoch_losses
```

In [22]:

```python
# Train the autoencoder using the mnist train set
epoch_losses = train(mnist_train_loader, model, criterion, optimizer, NUM_
EPOCHS)
```

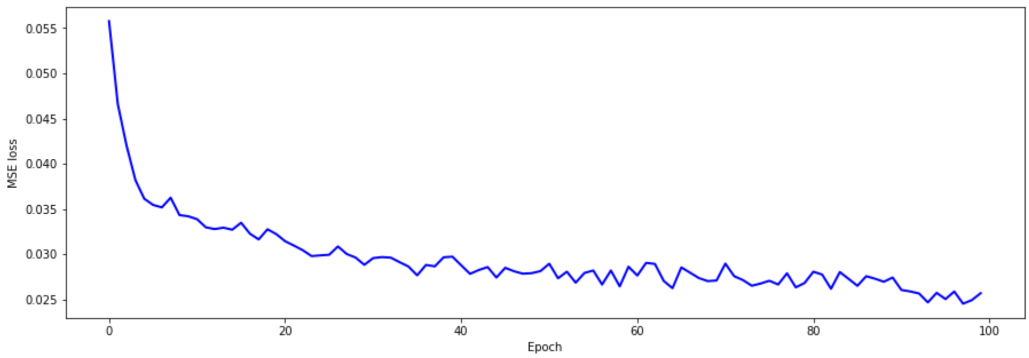
```
100%|██████████| 100/100 [07:33<00:00,  4.54s/it]
```

In [23]:

```python
# Plot the training loss
plt.figure(figsize=(15,5))
plt.plot(epoch_losses, lw=2, color='blue')
plt.ylabel('MSE loss')
plt.xlabel('Epoch')
```

Out[23]:

```
Text(0.5, 0, 'Epoch')
```



In [24]:

```python
# Encode the images from the test set

# Tensor for storing the latent variables from the encoder
testdata_latent = torch.zeros(size=(len(mnist_test_loader), BATCH_SIZE, 10
))

# Array to store the original mnist test images
testdata_input = np.zeros((len(mnist_test_loader), BATCH_SIZE, 1, 28, 28))
```

```
# Encode the images of the test set using the encoder trained on the train
set
for i,test_data in enumerate(mnist_test_loader):
    data, label = test_data[0].to(device), test_data[1].to(device)
    img = data.view(data.size(0), -1)
    latent = model.encoder(img)
    testdata_latent[i] = latent
    testdata_input[i] = data.detach().cpu().numpy()
```

In [25]:

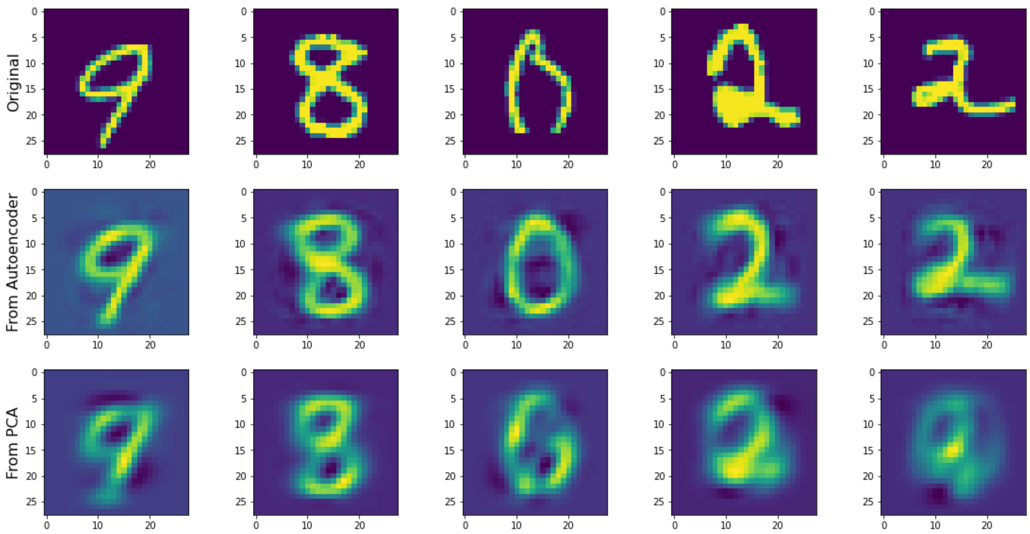
```
# Instantiate the Kernel PCA object
from sklearn.decomposition import PCA, KernelPCA
kpca = KernelPCA(n_components=10, kernel='rbf', fit_inverse_transform=True
)

# Reshape the test set into an array with 2 dimensions
testdata_input_pca = testdata_input.reshape(testdata_input.shape[0]*testda
ta_input.shape[1], -1)

# Normalizing the input
testdata_input_pca_mean = testdata_input_pca.mean()
testdata_input_pca_std = testdata_input_pca.std()
testdata_input_pca_normalized = (testdata_input_pca - testdata_input_pca_m
ean)/testdata_input_pca_std

# Apply PCA to find the compressed representation of the test set images
test_data_pca = kpca.fit_transform(testdata_input_pca_normalized)
print('Shape of compressed data after applying PCA: {}'.format(test_data_p
ca.shape))

# Reconstruct the test set images from the compressed vectors
test_data_pca_reconstructed = kpca.inverse_transform(test_data_pca)
print('Shape of image data after applying the inverse transform of PCA: {}
'.format(test_data_pca_reconstructed.shape))
```

```
Shape of compressed data after applying PCA: (10000, 10)
Shape of image data after applying the inverse transform of P
CA: (10000, 784)
```

In [26]:

```
# Set the autoencoder network to evaluation mode
model.eval()

# Select a particular test batch
test_batch_num = 5

# Show 5 images from the selected batch (original, reconstructed from the
 autoencoder, and from PCA)
fig, axes = plt.subplots(3, 5, figsize=(16,8))

# Select the appropriate latent variables computed by the autoencoder
latents = testdata_latent[test_batch_num][0:5]

# Plot the images
for i, latent in enumerate(latents):

    # Set row titles
    if i==0:
```

```
        axes[0, i].set_ylabel('Original', fontsize=16)
        axes[1, i].set_ylabel('From Autoencoder', fontsize=16)
        axes[2, i].set_ylabel('From PCA', fontsize=16)

    # Display original image
    #axes[0, i].axis('off')
    axes[0, i].imshow(testdata_input[test_batch_num][i].reshape(28,28))

    # Display decoded image from Autoencoder
    img = model.decoder(latent.to(device))
    #axes[1, i].axis('off')
    axes[1, i].imshow(img.cpu().detach().numpy().reshape(28,28))

    # Display decoded image from PCA
    #axes[2, i].axis('off')
    pca_img = test_data_pca_reconstructed[test_batch_num*BATCH_SIZE + i].r
eshape(28,28)
    axes[2, i].imshow(pca_img)

fig.tight_layout()
```



Denoising Autoencoder

Loss function: $L(x, g(f(\tilde{x})))$

where x is the input, $\tilde{x}$ is the noisy input, $f()$ is the encoder and $g()$ is the decoder.

In [27]:

```
def add_noise(inputs):
     noise = torch.randn_like(inputs)*0.2
     return inputs + noise
```

In [38]:

```
lambda1 = 0.001 # sparsity factor
noise_mean = 0.1
noise_std = 0.2
def train_denosing(train_loader, model, criterion, optimizer, num_epochs):
    epoch_losses = []
    for epoch in trange(num_epochs):
```

```
        for data in mnist_train_loader:
            img = data[0].to(device)
            img = img.view(img.size(0), -1)
            noise = add_noise(img)
            #noise_o = img.data.new(img.size()).normal_(noise_mean, noise_
std).float().to(device)
            #noise = torch.clamp((img + noise).data,0,1).float().to(devic
e)
            output = model(noise)
            mse_loss = criterion(output, img)
            optimizer.zero_grad()
            mse_loss.backward()
            optimizer.step()
        loss_value = mse_loss.item()
        epoch_losses.append(loss_value)
    return epoch_losses
```

In [39]:

```
# Train the autoencoder using the mnist train set
epoch_losses = train_denosing(mnist_train_loader, model, criterion, optimi
zer, NUM_EPOCHS)
```

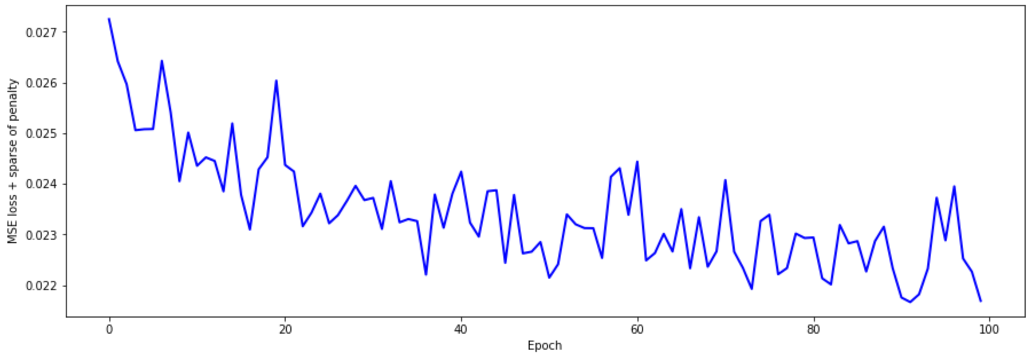
```
100%|██████████| 100/100 [07:39<00:00,  4.59s/it]
```

In [40]:

```
# Plot the training loss
plt.figure(figsize=(15,5))
plt.plot(epoch_losses, lw=2, color='blue')
plt.ylabel('MSE loss + sparse of penalty')
plt.xlabel('Epoch')
```

Out[40]:

```
Text(0.5, 0, 'Epoch')
```



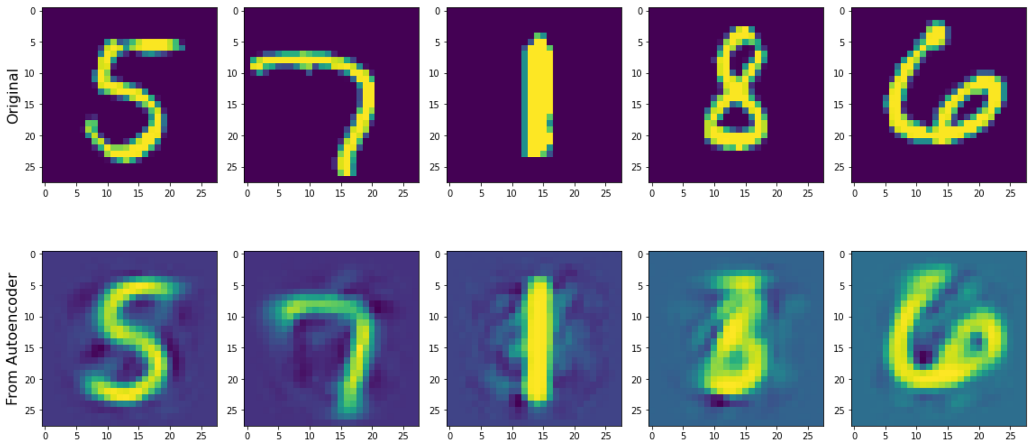
In [31]:

```
# Encode the images from the test set

# Tensor for storing the latent variables from the encoder
testdata_latent = torch.zeros(size=(len(mnist_test_loader), BATCH_SIZE, 10
))

# Array to store the original mnist test images
testdata_input = np.zeros((len(mnist_test_loader), BATCH_SIZE, 1, 28, 28))

# Encode the images of the test set using the encoder trained on the train
```

```
set
for i,test_data in enumerate(mnist_test_loader):
    data, label = test_data[0].to(device), test_data[1].to(device)
    img = data.view(data.size(0), -1)
    latent = model.encoder(img)
    testdata_latent[i] = latent
    testdata_input[i] = data.detach().cpu().numpy()
```

In [32]:

```
# Set the autoencoder network to evaluation mode
model.eval()

# Select a particular test batch
test_batch_num = 5

# Show 5 images from the selected batch (original, reconstructed from the
 autoencoder, and from PCA)
fig, axes = plt.subplots(2, 5, figsize=(16,8))

# Select the appropriate latent variables computed by the autoencoder
latents = testdata_latent[test_batch_num][0:5]

# Plot the images
for i, latent in enumerate(latents):

    # Set row titles
    if i==0:
        axes[0, i].set_ylabel('Original', fontsize=16)
        axes[1, i].set_ylabel('From Autoencoder', fontsize=16)

    # Display original image
    #axes[0, i].axis('off')
    axes[0, i].imshow(testdata_input[test_batch_num][i].reshape(28,28))

    # Display decoded image from Autoencoder
    img = model.decoder(latent.to(device))
    axes[1, i].imshow(img.cpu().detach().numpy().reshape(28,28))

fig.tight_layout()
```



Sparse Autoencoder

Loss function: $L(x, g(f(x))) + \Omega(h)$

where x is the input, $f()$ is the encoder, $g()$ is the decoder, h is the latent output (output of encoder) and $\Omega(h)$ is a sparsity penalty on h.

In [33]:

```
lambda1 = 0.001 # sparsity factor
def train_sparse(train_loader, model, criterion, optimizer, num_epochs):
    epoch_losses = []
    for epoch in trange(num_epochs):
        for data in mnist_train_loader:
            img = data[0].to(device)
            # We don't utilize the target data[1]
            img = img.view(img.size(0), -1)
            output = model(img)
            mse_loss = criterion(output, img)
            l1_norm = lambda1*torch.norm(model.encoder[0].weight, p=1) # L
1 penalty for the encoder
            total_loss = mse_loss+l1_norm
            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()
        loss_value = total_loss.item()
        epoch_losses.append(loss_value)
    print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,
num_epochs, total_loss))
    return epoch_losses
```

In [34]:

```
# Train the autoencoder using the mnist train set
epoch_losses = train_sparse(mnist_train_loader, model, criterion, optimize
r, NUM_EPOCHS)
```

```
100%|██████████| 100/100 [07:42<00:00,  4.62s/it]
```

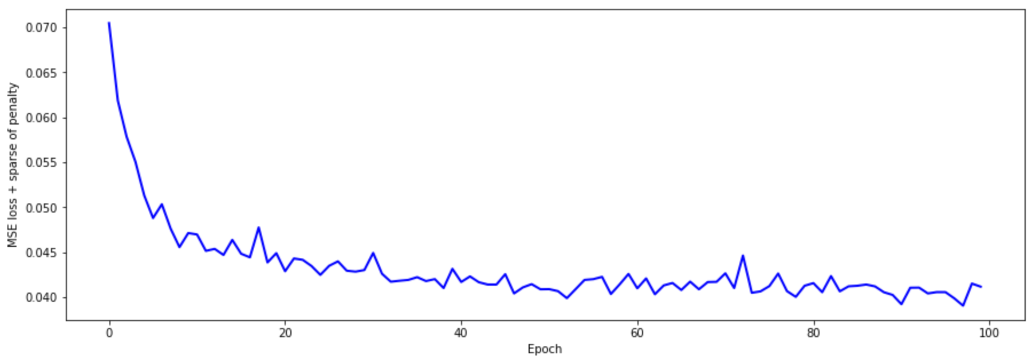
```
At Iteration : 100 / 100  ;  Mean-Squared Error : 0.041184
```

In [35]:

```
# Plot the training loss
plt.figure(figsize=(15,5))
plt.plot(epoch_losses, lw=2, color='blue')
plt.ylabel('MSE loss + sparse of penalty')
plt.xlabel('Epoch')
```

Out[35]:

```
Text(0.5, 0, 'Epoch')
```

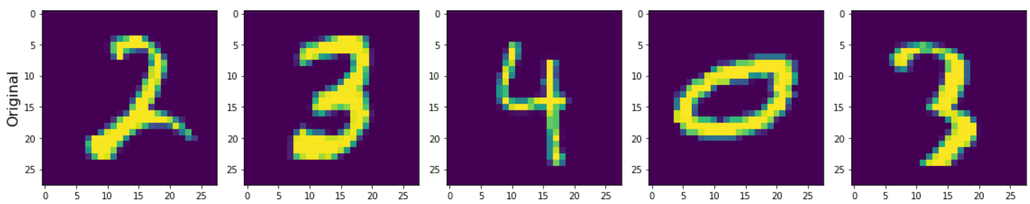


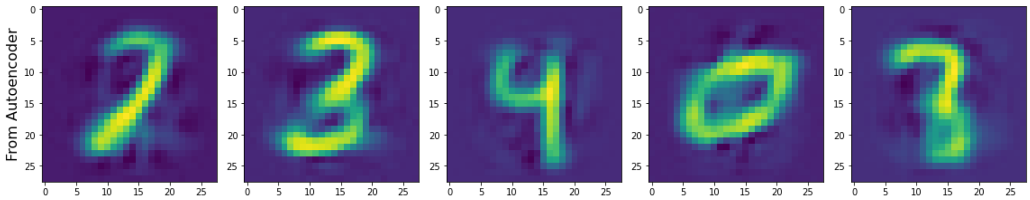
In [36]:

```
# Encode the images from the test set

# Tensor for storing the latent variables from the encoder
testdata_latent = torch.zeros(size=(len(mnist_test_loader), BATCH_SIZE, 10
))

# Array to store the original mnist test images
testdata_input = np.zeros((len(mnist_test_loader), BATCH_SIZE, 1, 28, 28))

# Encode the images of the test set using the encoder trained on the train
set
for i,test_data in enumerate(mnist_test_loader):
    data, label = test_data[0].to(device), test_data[1].to(device)
    img = data.view(data.size(0), -1)
    latent = model.encoder(img)
    testdata_latent[i] = latent
    testdata_input[i] = data.detach().cpu().numpy()
```

In [37]:

```
# Set the autoencoder network to evaluation mode
model.eval()

# Select a particular test batch
test_batch_num = 5

# Show 5 images from the selected batch (original, reconstructed from the
 autoencoder, and from PCA)
fig, axes = plt.subplots(2, 5, figsize=(16,8))

# Select the appropriate latent variables computed by the autoencoder
latents = testdata_latent[test_batch_num][0:5]

# Plot the images
for i, latent in enumerate(latents):

    # Set row titles
    if i==0:
        axes[0, i].set_ylabel('Original', fontsize=16)
        axes[1, i].set_ylabel('From Autoencoder', fontsize=16)

    # Display original image
    #axes[0, i].axis('off')
    axes[0, i].imshow(testdata_input[test_batch_num][i].reshape(28,28))

    # Display decoded image from Autoencoder
    img = model.decoder(latent.to(device))
    axes[1, i].imshow(img.cpu().detach().numpy().reshape(28,28))

fig.tight_layout()
```





In [37]: