MNIST Dimensionality Reduction [60 Points]

First, we are going to explore two different strategies to reduce the dimensionality of digit images. One, we shall build a CNN-based autoencoder. After, we shall reduce the dimensionality of images using PCA. Again, **avoid any unnecessary loops**, using broadcasting, and writing vectorized code.

Importing Libraries and Data

You may have to select GPU accelaration in Edit > Notebook Settings. No worries if you do not have GPU available, it may just take longer.

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  print('GPU device not found')
else:
  print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
Found GPU at: /device:GPU:0
```

```
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-white')
```

```
import tensorflow_datasets as tfds
(ds_train_org, ds_test_org), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)
```

```
Downloading and preparing dataset 11.06 MiB (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root/tensorflow_datasets/m
 Dl Completed...: 100%                                   5/5 [00:05<00:00,   1.28s/ file]
Dataset mnist downloaded and prepared to /root/tensorflow_datasets/mnist/3.0.1. Subsequent calls will reuse this data.
```

```
def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255.0

def normalize_img_label(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255.0, label

ds_train = ds_train_org.map(
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_test_labels = ds_test_org.map(
    normalize_img_label, num_parallel_calls=tf.data.AUTOTUNE)
```

MNIST CNN Autoencoder [35 Points]

First, we are going to build a neural network autoencoder.

$[\mathbf{10\ points}]$ Let's define the autoencoder neural network.

```
N_batch = 200  # batch size
code_size = 32  # size of autoencoder codes
hid_size = 128 # number of hidden units in network layers
```

```
# encode_nnet should be keras model that has 3 Conv2D layers with hid_size channels
# the filter sizes should be 4 by 4, with a stride of 2 in each dimension
# use a relu activation in the first two layers, none in the last

encode_nnet = keras.models.Sequential([
    keras.layers.Conv2D(hid_size, 4, strides=(2, 2), activation='relu'),
    keras.layers.Conv2D(hid_size, 4, strides=(2, 2), activation='relu'),
    keras.layers.Conv2D(code_size, 4, strides=(2, 2))])

encode_nnet.build([N_batch, 28, 28, 1])


# encode_nnet.output.shape  # should be (N_batch, 1, 1, code_size)!
```

Next we are going to use a very similar type of convolutional operation, the [transposed convolution](), to upsample and reconstruct the original image starting with the encoder codes.

```
# decode_nnet should be keras model that has 3 Conv2DTranspose layers with
# hid_size channels, the filter sizes should be 5 by 5 for the first two layers
# and 4 by 4 for the last, with a stride of 2 in each dimension.
# use a relu activation in the first two layers, sigmoid in the last.

decode_nnet = keras.models.Sequential([
    keras.layers.Conv2DTranspose(hid_size, 5, strides=(2, 2), activation='relu'),
    keras.layers.Conv2DTranspose(hid_size, 5, strides=(2, 2), activation='relu'),
    keras.layers.Conv2DTranspose(1, 4, strides=(2, 2), activation='sigmoid')
]) # TODO

# decode_nnet.build([200,1,1,32])


# decode_nnet.output.shape  # should be (N_batch, 28, 28, 1)
```

$\color{red}{[\textbf{15 points}]}$ Let's implement the loss for batches.

```
def loss(X, encoder, decoder):
  """
  Args:
    X: tensor of input images (N_batch, 28, 28, 1)
    encoder: function that takes in a batch of images and outputs codes
    decoder: function that takes in a batch of codes and outputs reconstructions
  Returns:
    loss: mean of the sum of squared errors (Euclidean distance squared) for
      reconstructions
  """
  # recons_loss should be a mean of the --sum of squared errors-- for
  # reconstructions, averaged over all instances in the batch
  # MAKE SURE TO USE TENSORFLOW FUNCTIONS AND OPERATIONS (IE NOT NUMPY)
  # Hint: make sure to use reduce_mean and reduce_sum over the correct axes.

  encoded_X = encoder(X)
  reconstructed_X = decoder(encoded_X)

  squared_errors = tf.square(X - reconstructed_X)
  sum_squared_errors = tf.reduce_sum(squared_errors, axis=[1, 2, 3])

  recons_loss = tf.reduce_mean(sum_squared_errors)  # TODO

  return recons_loss

def grad(X, enc_nnet, dec_nnet):
  with tf.GradientTape() as tape:
    loss_value = loss(X, enc_nnet, dec_nnet)

  return tape.gradient(loss_value, enc_nnet.weights+dec_nnet.weights)


optimizer = tf.keras.optimizers.Adam(learning_rate=0.00001)
# nepochs = number of times to run through the data.
nepochs = 25  # Could probably do better with more epochs, but should suffice
for i in range(nepochs):
  for B_X in ds_train.batch(N_batch, drop_remainder=True):
    grads = grad(B_X, encode_nnet, decode_nnet)
```

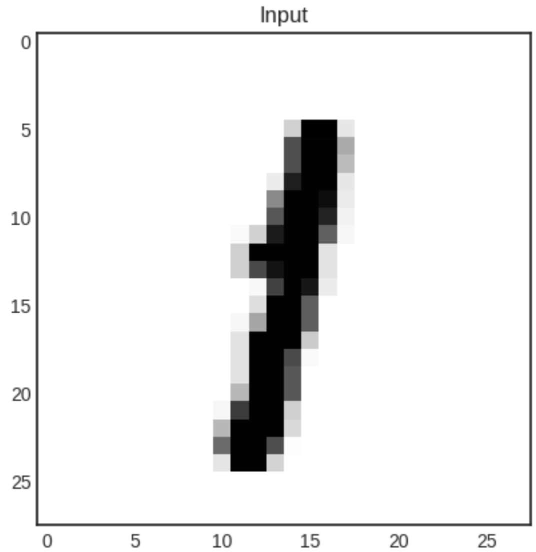
```
    optimizer.apply_gradients(zip(grads, encode_nnet.weights+decode_nnet.weights))  # SGD-type update (w/ Adam)
  print("Loss at epoch {:03d}: {:.3f}".format(i, loss(B_X, encode_nnet, decode_nnet)))
```

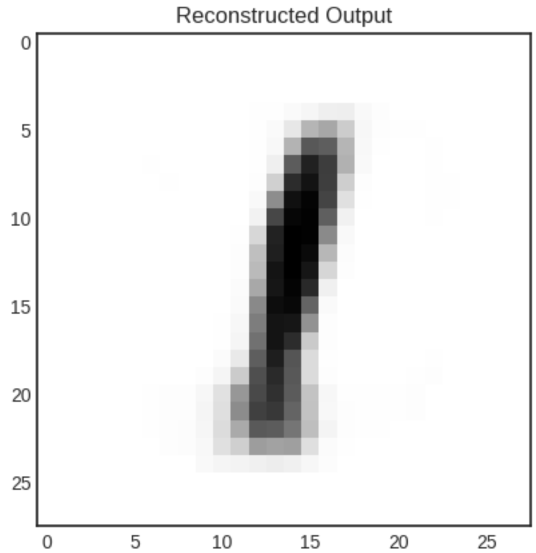
```
Loss at epoch 000: 98.774
Loss at epoch 001: 63.028
Loss at epoch 002: 54.175
Loss at epoch 003: 53.287
Loss at epoch 004: 51.995
Loss at epoch 005: 48.933
Loss at epoch 006: 44.586
Loss at epoch 007: 39.727
Loss at epoch 008: 36.528
Loss at epoch 009: 34.390
Loss at epoch 010: 32.494
Loss at epoch 011: 30.693
Loss at epoch 012: 29.202
Loss at epoch 013: 28.004
Loss at epoch 014: 27.021
Loss at epoch 015: 26.169
Loss at epoch 016: 25.416
Loss at epoch 017: 24.717
Loss at epoch 018: 24.022
Loss at epoch 019: 23.330
Loss at epoch 020: 22.647
Loss at epoch 021: 22.001
Loss at epoch 022: 21.389
Loss at epoch 023: 20.824
Loss at epoch 024: 20.324
```

[5 points] Let's visualize the output for an instance in the batch.

```
B_X_plot = B_X
nnet_output = decode_nnet(encode_nnet(B_X))  # TODO: get N_batch x 28 x 28 x 1 tensor of reconstructions
rind = np.random.randint(0, N_batch)  # Select a random instance in batch

plt.figure()
plt.title('Input')
plt.imshow(np.reshape(B_X_plot[rind], (28, 28)))

plt.figure()
plt.title('Reconstructed Output')
x_recon = nnet_output[rind]
plt.imshow(np.reshape(x_recon, (28, 28)))
```

```
<matplotlib.image.AxesImage at 0x7b6ce0bdff10>
```





[5 points] Let's compute the accuracy over the test set.

```
test_batch_MSEs = []
for B_X, B_Ylbls in ds_test_labels.batch(N_batch, drop_remainder=True):
  nnet_output = decode_nnet(encode_nnet(B_X))
  squared_errors = tf.square(B_X - nnet_output)
  sum_squared_errors = tf.reduce_sum(squared_errors, axis=[1, 2, 3])
  test_batch_MSEs.append(tf.reduce_mean(sum_squared_errors))  # TODO: append the MSE for the batch

print('Mean Sum of Squared Errors on test set: {}'.format(np.mean(test_batch_MSEs)))
```

```
Mean Sum of Squared Errors on test set: 20.757062911987305
```

PCA [25 Points]

Next, we are going to learn a linear autoencoder with PCA, utilizing scikit-learn's implementation.

[10 points] Let's fit a PCA model on the MNIST data.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=code_size)


# Hint: list(ds_train.as_numpy_iterator()) is very useful
Xdesign_mat = []
```

```
Xdesign_mat = np.stack(list(ds_train.as_numpy_iterator()), axis=0).reshape(-1,784)
pca.fit_transform(Xdesign_mat)
```

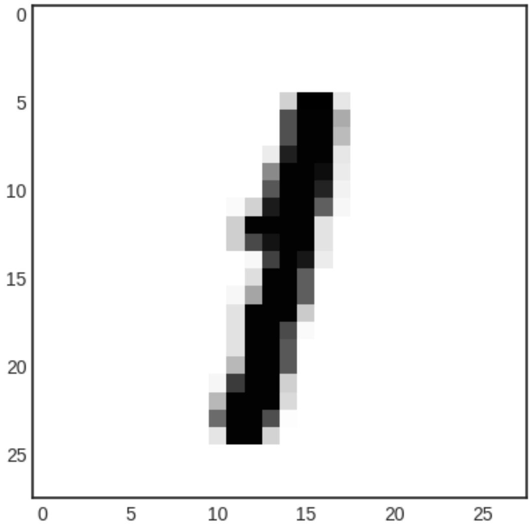
```
array([[-2.7357507 , -0.40374228, -0.32074684, ..., -0.37409383,
        -0.3000402 ,  0.9021928 ],
       [-3.9322762 ,  1.6692578 , -0.44134945, ..., -0.98440295,
        -0.3824799 ,  0.51441216],
       [ 4.5570955 ,  0.9185379 , -3.938978  , ..., -0.05938989,
        -0.25504458, -0.42250964],
       ...,
       [ 2.9310338 , -0.03648547, -0.22359508, ..., -0.26937854,
         0.83677745,  0.04880274],
       [-3.1157043 ,  0.46734783,  0.80377257, ..., -0.18952505,
         0.06372073,  0.5626818 ],
       [-0.4871719 , -0.28548703, -1.1456549 , ...,  1.292084  ,
         1.9250314 ,  0.37560377]], dtype=float32)
```

$[\textbf{5 points}]$ Let's compute MINST reconstructions using PCA.

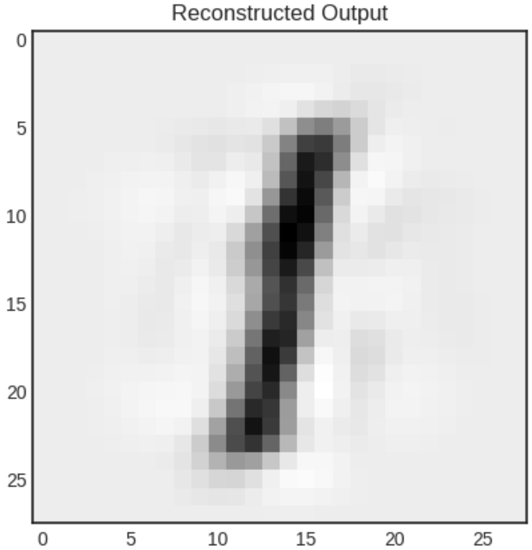
```
B_X_plot = tf.reshape(B_X_plot, [B_X_plot.shape[0], -1]).numpy()
codes = pca.transform(B_X_plot)
recons = np.reshape(pca.inverse_transform(codes), (B_X_plot.shape[0], 28, 28))

plt.figure()
plt.title('Input')
plt.imshow(np.reshape(B_X_plot[rind], (28, 28)))

plt.figure()
plt.title('Reconstructed Output')
x_recon = recons[rind]
plt.imshow(np.reshape(x_recon, (28, 28)))

plt.figure()
plt.title('Truncated Reconstructed Output')
x_recon[x_recon<0.0] = 0.0
x_recon[x_recon>1.0] = 1.0
plt.imshow(np.reshape(x_recon, (28, 28)))
```

```
<matplotlib.image.AxesImage at 0x7b0c5691c4f0>
```

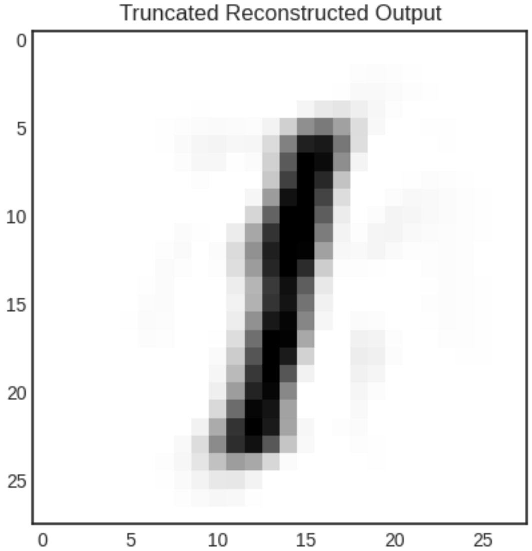
Input



Reconstructed Output



Truncated Reconstructed Output



[10 points] Let's compute MNIST PCA reconstructions errors on the MNIST test set.

```
test_batch_MSEs_pca = []
for B_X, B_Ylbls in ds_test_labels.batch(N_batch, drop_remainder=True):
  # YOU must use loss(X, encoder, decoder) here!!
```

```
  # Do *not* truncate reconstructions.
  # Hint: lambda functions...
  encoder = lambda x: pca.transform(tf.reshape(x, [x.shape[0], -1]).numpy())
  decoder = lambda x: tf.convert_to_tensor(pca.inverse_transform(x).reshape(-1, 28, 28, 1))
  test_batch_MSEs_pca.append(loss(B_X, encoder, decoder))  # TODO: append the MSE for the batch

print('Mean Sum of Squared Errors on test set: {}'.format(
  np.mean(test_batch_MSEs_pca)))
```

```
Mean Sum of Squared Errors on test set: 13.193336486816406
```

Not too bad for a linear model!!

Kmeans from Scratch [40 Points]

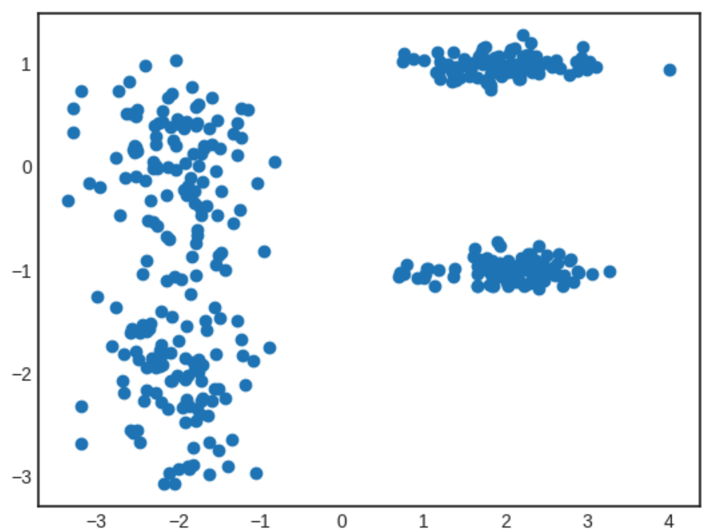
We are now going to implement and visualize kmeans. In addition to getting some more intuition and understanding about kmeans, we are going to be efficient about our implementation. This means **avoid any unnecessary loops**, using broadcasting, and writing vectorized code.

First, let's generate data.

```
np.random.seed(8675309)  # Feel free to fiddle with seed when playing around
X1 = np.concatenate([0.5*np.random.randn(100, 1)-2.0, 0.5*np.random.randn(100, 1)-2.0], 1)
X2 = np.concatenate([0.5*np.random.randn(100, 1)+-2.0, 0.5*np.random.randn(100, 1)], 1)
X3 = np.concatenate([0.5*np.random.randn(100, 1)+2.0, 0.1*np.random.randn(100, 1)-1.0], 1)
X4 = np.concatenate([0.5*np.random.randn(100, 1)+2.0, 0.1*np.random.randn(100, 1)+1.0], 1)
X_all = np.concatenate([X1, X2, X3, X4], 0)
mus_init = np.random.randn(4, 2)
```

```
plt.scatter(X_all[:, 0], X_all[:, 1])
```

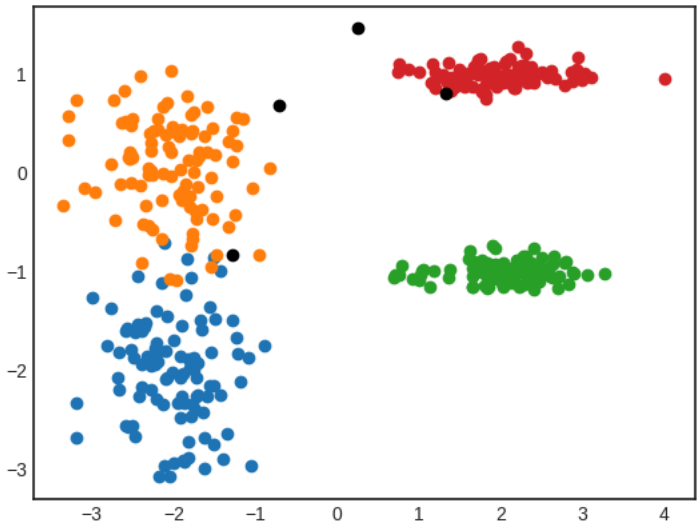
```
<matplotlib.collections.PathCollection at 0x7b6ce0b36ec0>
```



Visualizing the "ground truth" clusters.

```
plt.scatter(X1[:, 0], X1[:, 1])
plt.scatter(X2[:, 0], X2[:, 1])
plt.scatter(X3[:, 0], X3[:, 1])
plt.scatter(X4[:, 0], X4[:, 1])
plt.scatter(mus_init[:, 0], mus_init[:, 1], c='black')
```

<matplotlib.collections.PathCollection at 0x7b6c595c77f0>



[15 points] Let's implement the "distortion" loss that kmeans is minimizing. **Avoid any loops**.

```
def kmeans_distortion(X, mus, zs_onehot):
  """
  Args:
    X: N x d matrix of the points to cluster
    mus: K x d matrix of means
    zs_onehot: N x K matrix of one hot indicators of assignments.
      I.e. zs_onehot[i, j] = 1 if point i is in clust j, 0 otherwise.
  Returns:
    distortion: scalar of loss that Kmeans is trying to minimize
  """
  # hint: use diffs_cluster[i, j] = ||x_i - mu_j||^2 matrix
  # other hint: diffs_cluster[i, j] = ||x_i||^2 -2 x_i^mu_j  + ||mu_j||^2

  X_norm_sq = np.sum(X**2, axis=1, keepdims=True)
  mus_norm_sq = np.sum(mus**2, axis=1)
  diffs_cluster = X_norm_sq - 2 * np.dot(X, mus.T) + mus_norm_sq
  distortion = np.sum(zs_onehot * diffs_cluster)

  return distortion
```

[15 points] Let's now code kmeans. It's a simple algorithm that can actually be coded in a handful of lines. Let's be efficient, and **avoid any other loops**.

```
def kmeans_cluster(X, mus, iters):
  """
  Args:
    X: N x d matrix of the points to cluster
    mus: K x d matrix of initial means
    iters: integer number of iterations to run kmeans updates
  Returns:
    mus: K x d matrix of means
    zs_onehot: N x K matrix of one hot indicators of assignments.
      I.e. zs_onehot[i, j] = 1 if point i is in clust j, 0 otherwise.
    distortions: iters length array of loss that Kmeans is trying to minimize
  """
  K = mus.shape[0]
  eyeK = np.eye(K)

  distortions = []
  for t in range(iters):
    # get distances
    # diffs_cluster[i, j] = ||x_i - mu_j||^2
    X_norm_sq = np.sum(X**2, axis=1, keepdims=True)
    mus_norm_sq = np.sum(mus**2, axis=1)
    diffs_cluster = X_norm_sq - 2 * np.dot(X, mus.T) + mus_norm_sq # should be N x k matrix

    # assignments
    # zs[i] \in {1, .., K}, cluster x_i is in
```

```
    zs = np.argmin(diffs_cluster, axis=1)  # N length array
    # zs_onehot[i, j] = x_i \in cluster_j
    zs_onehot = eyeK[zs]  # should be N x k matrix

    # record distortion after new assignments
    distortions.append(kmeans_distortion(X, mus, zs_onehot))

    # plot assignments
    plt.figure()
    plt.title('Iter {}'.format(t))
    plt.scatter(X[:, 0], X[:, 1], c=zs)
    plt.scatter(mus[:, 0], mus[:, 1], c='red')

    # update mus
    # mus[j, l] = l-th dimension of cluster j
    for j in range(K):
      if np.sum(zs_onehot[:, j]) > 0:  # Avoid division by zero
        mus[j] = np.sum(X * zs_onehot[:, j][:, np.newaxis], axis=0) / np.sum(zs_onehot[:, j])
    # TODO: should be K x d matrix

    # plot new means
    plt.scatter(mus[:, 0], mus[:, 1], c='orange')

    # record distortion after new means
    distortions.append(kmeans_distortion(X, mus, zs_onehot))

  plt.figure()
  plt.title('Distortions')
  plt.plot(distortions)
  return mus, zs_onehot, distortions
```

+ Code + Text

```
plt.style.use('default')
```

```
mus_kmeans, zs_onehot_kmeans, distortions_kmeans = kmeans_cluster(X_all, mus_init, 6)
```