# Assignment is below at the bottom

Video 13.1 https://www.youtube.com/watch?v=kIGHE7Cfe1s (https://www.youtube.com/watch?v=kIGHE7Cfe1s)

Video 13.2 https://www.youtube.com/watch?v=Rm9bJcDd1KU (https://www.youtube.com/watch?v=Rm9bJcDd1KU)

Video 13.3 https://youtu.be/6HjZk-3LsjE (https://youtu.be/6HjZk-3LsjE)

In [1]:
```
from keras.callbacks import TensorBoard
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import numpy as np

(xtrain, ytrain), (xtest, ytest) = mnist.load_data()

xtrain = xtrain.astype('float32') / 255.
xtest = xtest.astype('float32') / 255.
xtrain = xtrain.reshape((len(xtrain), np.prod(xtrain.shape[1:])))
xtest = xtest.reshape((len(xtest), np.prod(xtest.shape[1:])))
xtrain.shape, xtest.shape
```

Out[1]:
```
((60000, 784), (10000, 784))
```

In [2]:
```
# this is the size of our encoded representations
encoding_dim = 4  # 32 floats -> compression of factor 24.5, assuming

# this is our input placeholder
x = input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)
encoded = Dense(encoding_dim, activation='relu')(x)


# "decoded" is the lossy reconstruction of the input
x = Dense(128, activation='relu')(encoded)
x = Dense(256, activation='relu')(x)
decoded = Dense(784, activation='sigmoid')(x)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)

encoder = Model(input_img, encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
dcd1 = autoencoder.layers[-1]
dcd2 = autoencoder.layers[-2]
dcd3 = autoencoder.layers[-3]

# create the decoder model
decoder = Model(encoded_input, dcd1(dcd2(dcd3(encoded_input))))
```

```
2023-04-24 00:09:05.497769: I tensorflow/core/platform/cpu_feature_gu
ard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neur
al Network Library (oneDNN) to use the following CPU instructions in
performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appro
priate compiler flags.
```

In [18]:
```
#autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

In [3]:
```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

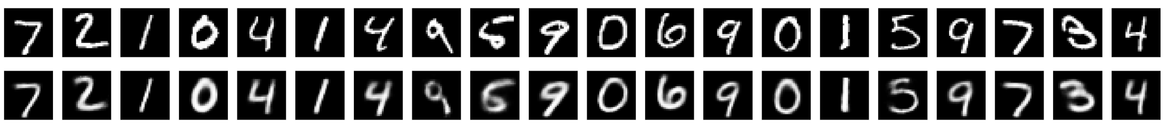
In [4]:
```
autoencoder.fit(xtrain, xtrain,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(xtest, xtest))
                #callbacks=[TensorBoard(log_dir='/tmp/autoencoder')])
```

```
2023-04-24 00:09:20.130155: I tensorflow/compiler/mlir/mlir_graph_opt
imization_pass.cc:185] None of the MLIR Optimization Passes are enabl
ed (registered 2)

Epoch 1/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2508
- val_loss: 0.1879
Epoch 2/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1791
- val_loss: 0.1721
Epoch 3/100
235/235 [==============================] - 2s 6ms/step - loss: 0.1696
- val_loss: 0.1660
Epoch 4/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1643
- val_loss: 0.1620
Epoch 5/100
235/235 [==============================] - 2s 6ms/step - loss: 0.1609
- val_loss: 0.1600
Epoch 6/100
```

In [5]:
```
autoencoder.evaluate(xtest, xtest, verbose = 0)
```

Out[5]: 0.14108693599700928

In [6]:
```
encoded_imgs = encoder.predict(xtest)
decoded_imgs = decoder.predict(encoded_imgs)
import matplotlib.pyplot as plt

n = 20  # how many digits we will display
plt.figure(figsize=(40, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(xtest[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



In [7]:
```
encoded_imgs
```

Out[7]:
```
array([[ 8.121867 , 41.15789  , 16.332258 ,  7.376401 ],
       [24.201841 , 21.127634 , 16.845016 ,  6.0990424],
       [36.83489  , 51.335476 , 28.910124 , 18.496061 ],
       ...,
       [12.324904 , 30.052652 ,  9.990883 , 21.742424 ],
       [17.130564 , 16.039772 ,  4.0078583, 14.629419 ],
       [15.10408  , 16.662783 , 11.55282  , 26.731108 ]], dtype=float
32)
```

In [8]:
```
np.max(encoded_imgs)
```

Out[8]:
```
66.474236
```

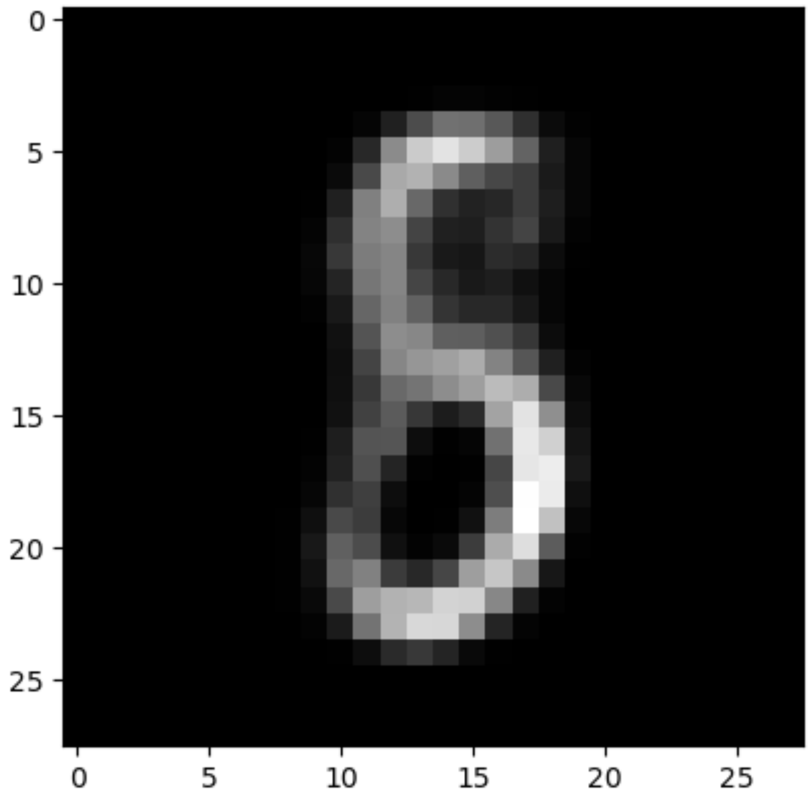
In [9]:
```
noise = np.random.normal(20,4, (4,4))
noise_preds = decoder.predict(noise)
```

In [10]:
```
noise
```

Out[10]:
```
array([[20.07561402, 21.32106027, 21.23946429, 21.70652002],
       [23.88273129, 17.89626404, 22.62913063, 20.43102977],
       [18.65630753, 24.97829953, 23.56176308, 19.2978824 ],
       [21.30138864, 26.72723809, 22.82092237, 25.98322885]])
```

In [11]:
```
plt.imshow(noise_preds[1].reshape(28,28))
```

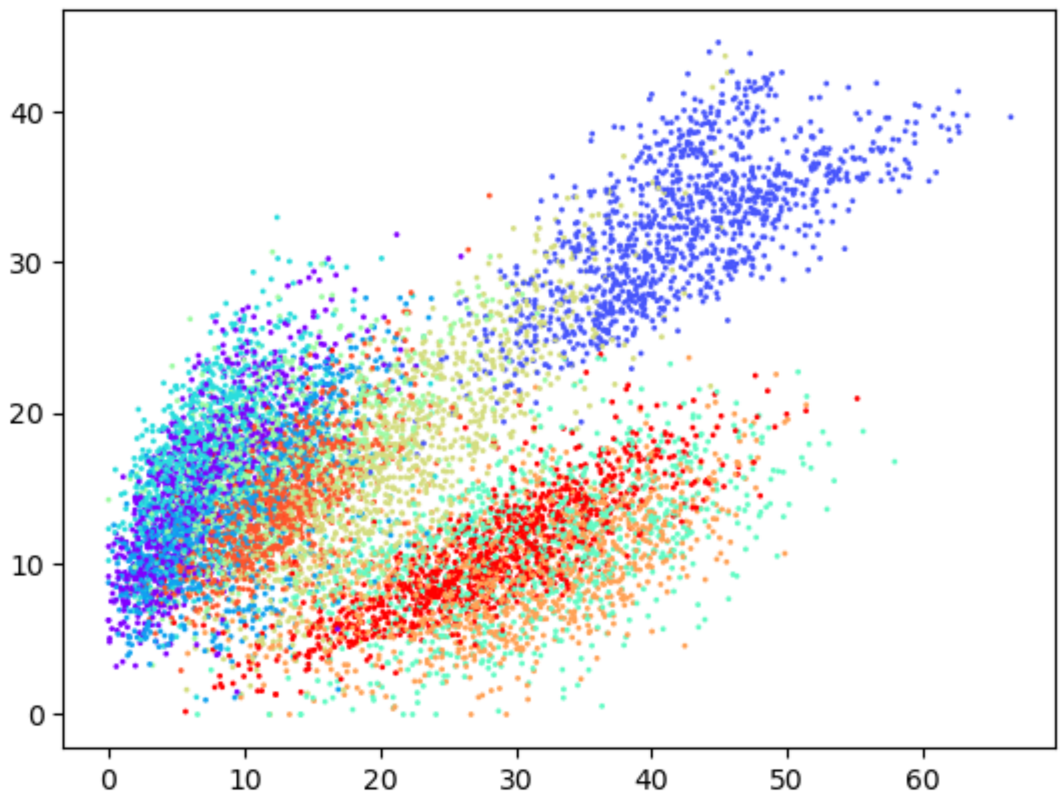
Out[11]: <matplotlib.image.AxesImage at 0x16de97d30>




In [12]:
```
%matplotlib inline
```

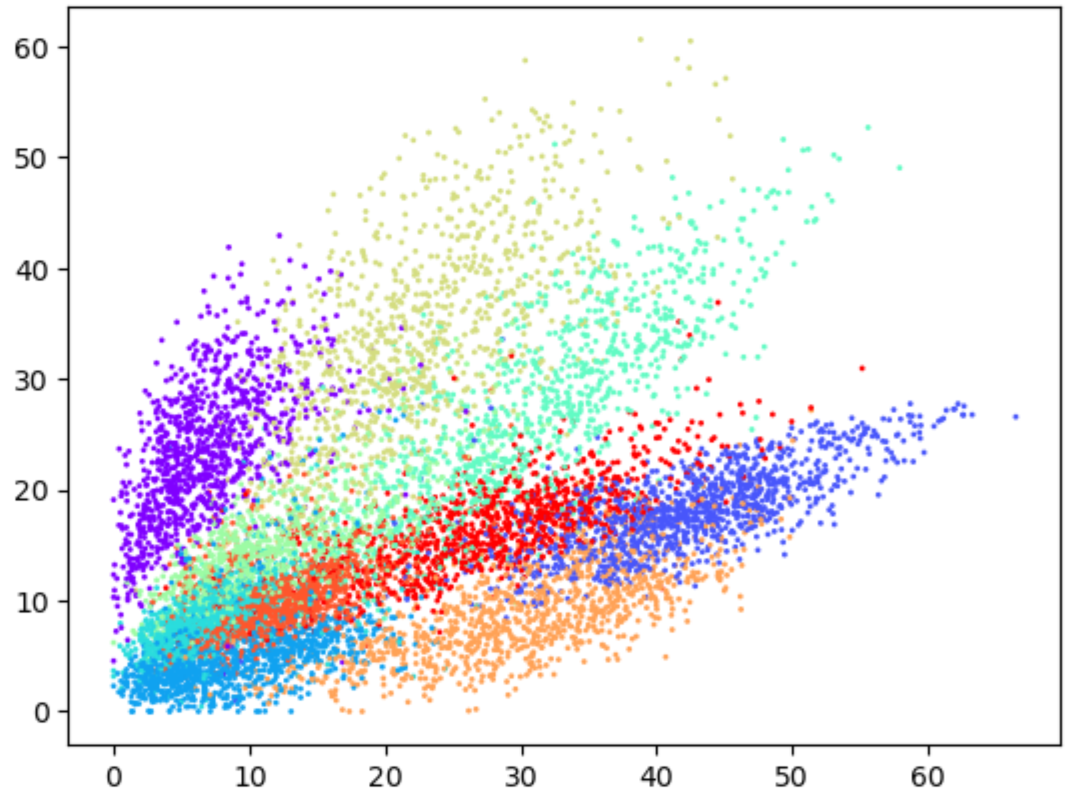
In [13]:
```
plt.scatter(encoded_imgs[:,1], encoded_imgs[:,0], s=1, c=ytest, cmap='
# plt.show()
```

Out[13]:
```
<matplotlib.collections.PathCollection at 0x16dc95fd0>
```



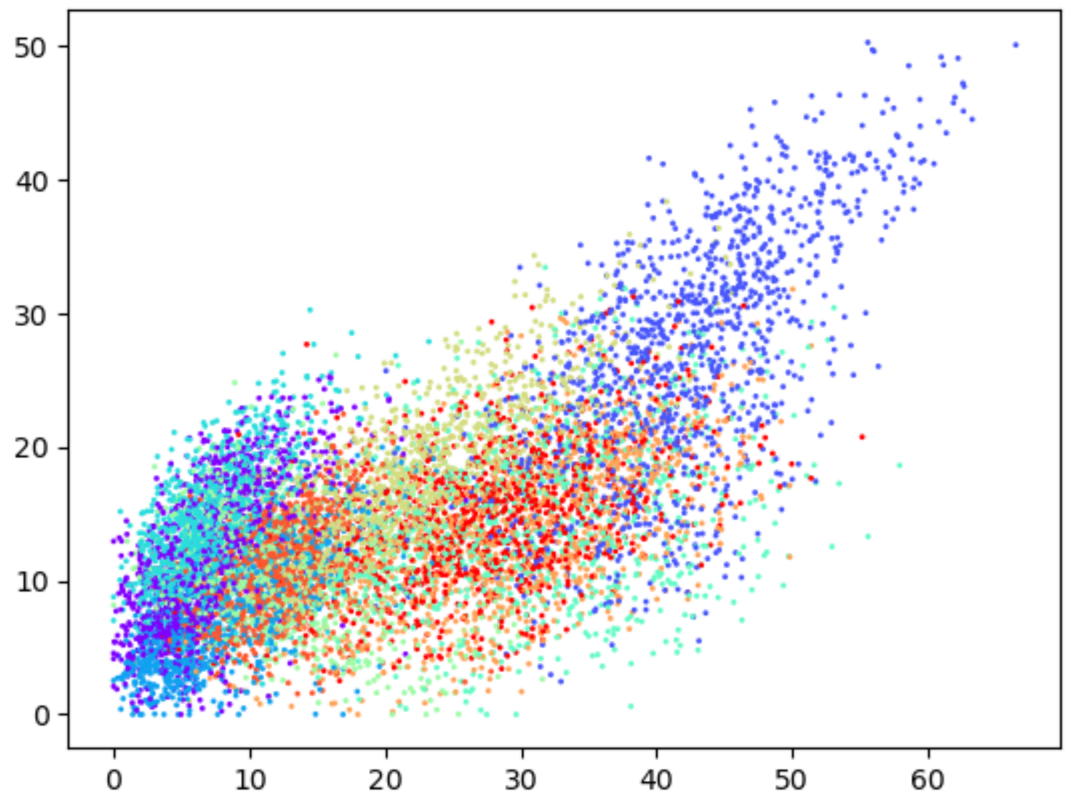
In [14]:
```
plt.scatter(encoded_imgs[:,1], encoded_imgs[:,3], s=1, c=ytest, cmap='
# plt.show()
```

Out[14]: <matplotlib.collections.PathCollection at 0x17062cb50>



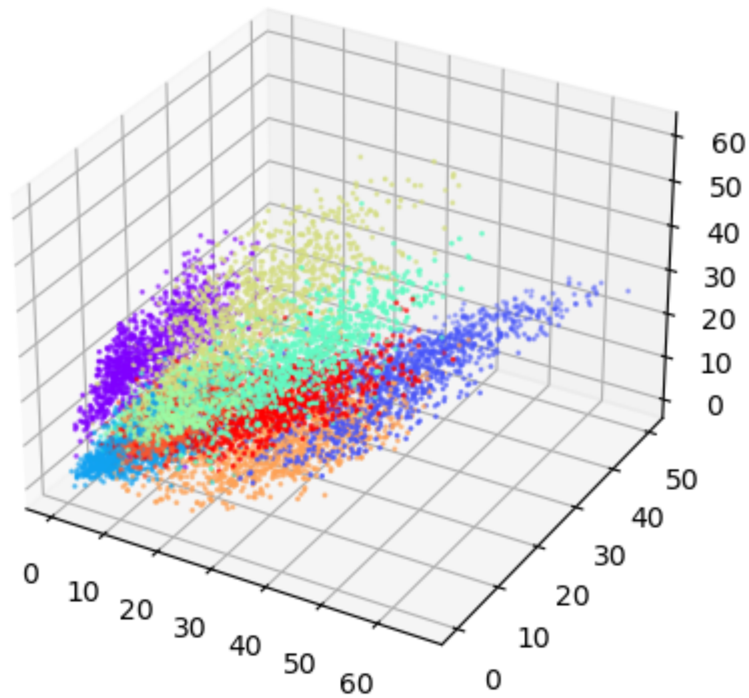
In [15]:
```
plt.scatter(encoded_imgs[:,1], encoded_imgs[:,2], s=1, c=ytest, cmap='
# plt.show()
```

Out[15]: <matplotlib.collections.PathCollection at 0x1768f3970>



In [16]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(encoded_imgs[:,1], encoded_imgs[:,2], encoded_imgs[:,3], c=
```

Out[16]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x171cd7880>
```




# Assignment

1. change the `encoding_dim` through various values ( `range(2,18,2)` and store or keep track of the best loss you can get. Plot the 8 pairs of dimensions vs loss on a scatter plot

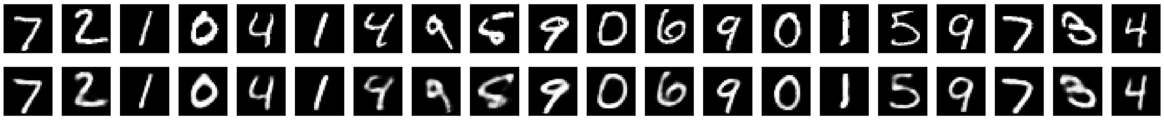
In [17]:
```
losses = []
dimensions = range(2,18,2)

for encoding_dim in dimensions:

    print(encoding_dim)

    # this is our input placeholder
    x = input_img = Input(shape=(784,))
    # "encoded" is the encoded representation of the input
    x = Dense(256, activation='relu')(x)
    x = Dense(128, activation='relu')(x)
    encoded = Dense(encoding_dim, activation='relu')(x)
    # "decoded" is the lossy reconstruction of the input
    x = Dense(128, activation='relu')(encoded)
    x = Dense(256, activation='relu')(x)
    decoded = Dense(784, activation='sigmoid')(x)

    # this model maps an input to its reconstruction
    autoencoder = Model(input_img, decoded)
    encoder = Model(input_img, encoded)

    # create a placeholder for an encoded (32-dimensional) input
    encoded_input = Input(shape=(encoding_dim,))
    # retrieve the last layer of the autoencoder model
    dcd1 = autoencoder.layers[-1]
    dcd2 = autoencoder.layers[-2]
    dcd3 = autoencoder.layers[-3]

    # create the decoder model
    decoder = Model(encoded_input, dcd1(dcd2(dcd3(encoded_input))))

    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    autoencoder.fit(xtrain, xtrain,
                    epochs=100,
                    batch_size=256,
                    shuffle=True,
                    validation_data=(xtest, xtest))

    loss = autoencoder.evaluate(xtest, xtest, verbose=0)
    losses.append(loss)
```

```
2
Epoch 1/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2755
- val_loss: 0.2494
Epoch 2/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2450
- val_loss: 0.2387
Epoch 3/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2356
- val_loss: 0.2315
Epoch 4/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2299
```

```
- val_loss: 0.2275
Epoch 5/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2272
- val_loss: 0.2259
Epoch 6/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2251
- val_loss: 0.2238
```

In [18]:

```python
# example with encoding dim at 16 after previous block is run
encoded_imgs = encoder.predict(xtest)
decoded_imgs = decoder.predict(encoded_imgs)
import matplotlib.pyplot as plt

n = 20  # how many digits we will display
plt.figure(figsize=(40, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(xtest[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



In [19]:

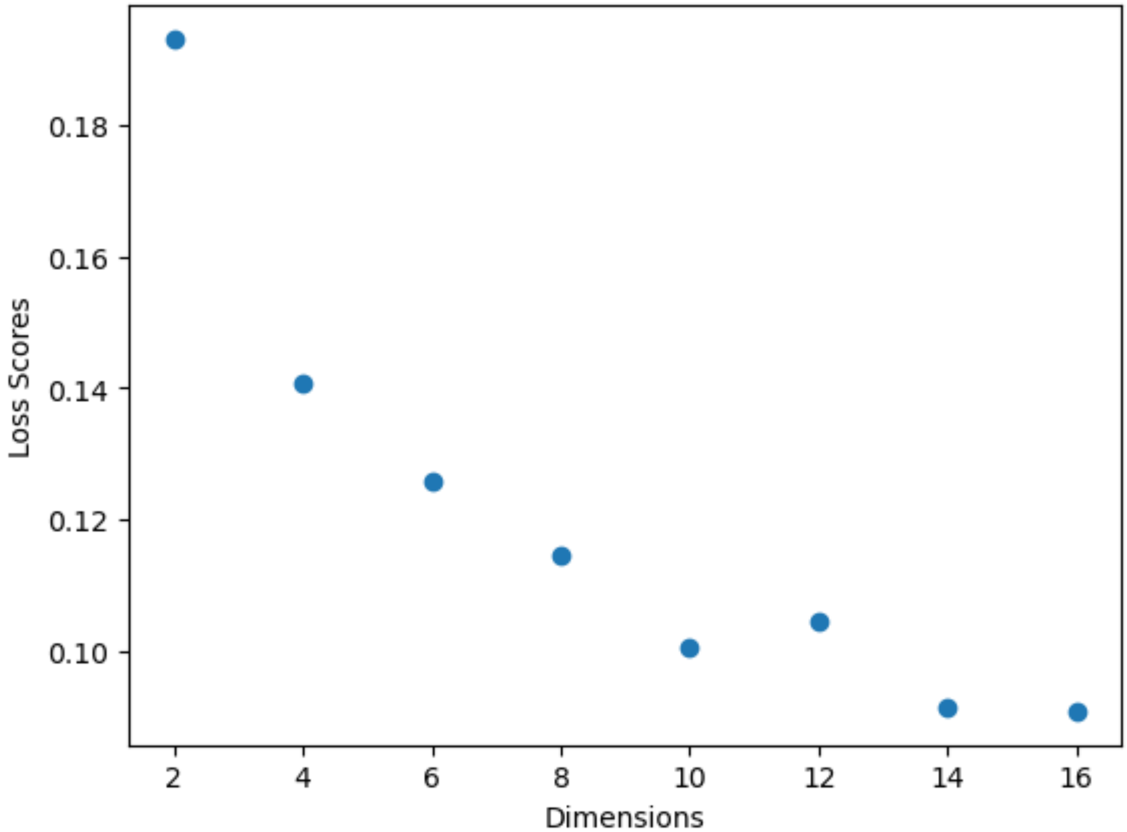
```python
losses
```

Out[19]:

```
[0.1930532604455948,
 0.1406082808971405,
 0.12566663324832916,
 0.11441220343112946,
 0.10058099031448364,
 0.10441082715988159,
 0.09129456430673599,
 0.0907890573143959]
```

In [20]:
```
plt.figure()
plt.scatter(dimensions, losses)
plt.xlabel("Dimensions")
plt.ylabel("Loss Scores")
```

Out[20]: Text(0, 0.5, 'Loss Scores')




2. ***After*** training an autoencoder with `encoding_dim=8` , apply noise (like the previous assignment) to *only* the input of the trained autoencoder (not the output). The output images should be without noise.

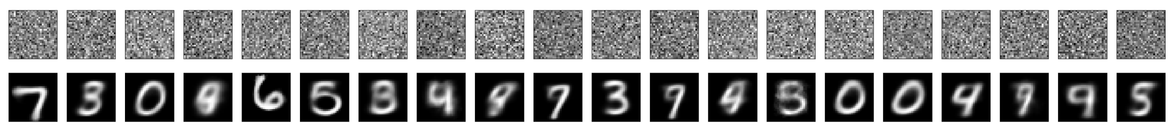
Print a few noisy images along with the output images to show they don't have noise.

In [21]:
```
losses = []
encoding_dim = 8
scales = [.1, .5, 1.0, 2.0, 4.0]

# this is our input placeholder
x = input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)
encoded = Dense(encoding_dim, activation='relu')(x)
# "decoded" is the lossy reconstruction of the input
x = Dense(128, activation='relu')(encoded)
x = Dense(256, activation='relu')(x)
decoded = Dense(784, activation='sigmoid')(x)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
dcd1 = autoencoder.layers[-1]
dcd2 = autoencoder.layers[-2]
dcd3 = autoencoder.layers[-3]

# create the decoder model
decoder = Model(encoded_input, dcd1(dcd2(dcd3(encoded_input))))

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

for scale in scales:
    print(scale)

    noise = np.random.normal(loc=1, scale=scale, size=xtrain.shape)
    xtrain_noisy = xtrain + noise

    noise = np.random.normal(loc=1, scale=scale, size=xtest.shape)
    xtest_noisy = xtest + noise

    autoencoder.fit(xtrain_noisy, xtrain,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(xtest_noisy, xtest))

    loss = autoencoder.evaluate(xtest_noisy, xtest, verbose=0)
    losses.append(loss)

    encoded_imgs = encoder.predict(xtest_noisy)
    decoded_imgs = decoder.predict(encoded_imgs)

    n = 5  # how many digits we will display
    plt.figure(figsize=(40, 4))
    for i in range(n):
```

```
        # display original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(xtest_noisy[i].reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        # display reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(decoded_imgs[i].reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()
```

```
0.1
Epoch 1/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2653
- val_loss: 0.2331
Epoch 2/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2193
- val_loss: 0.2111
Epoch 3/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2020
- val_loss: 0.1928
Epoch 4/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1870
- val_loss: 0.1816
Epoch 5/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1777
- val_loss: 0.1734
Epoch 6/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1673
- val_loss: 0.1618
Epoch 7/100
```

In [22]:
```
# example with the noise scale at 4.0 after the previous block is run
encoded_imgs = encoder.predict(xtest_noisy)
decoded_imgs = decoder.predict(encoded_imgs)

n = 20  # how many digits we will display
plt.figure(figsize=(40, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(xtest_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



In [23]:
```
losses
```

Out[23]:
```
[0.1256238967180252,
 0.14162208139896393,
 0.17681051790714264,
 0.2561157345771789 6,
 0.35957613587379456]
```

In [24]:
```
plt.figure()
plt.scatter(scales, losses)
plt.xlabel("Noise Scales")
plt.ylabel("Loss Scores")
```

Out[24]: Text(0, 0.5, 'Loss Scores')