In [1]:

```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

The MNIST dataset

The MNIST dataset is composed of 70,000 28×28 grayscale images of handwritten digits. It is represented as a $70000 \times 28 \times 28$ numpy array (a "3d matrix").

In [2]:

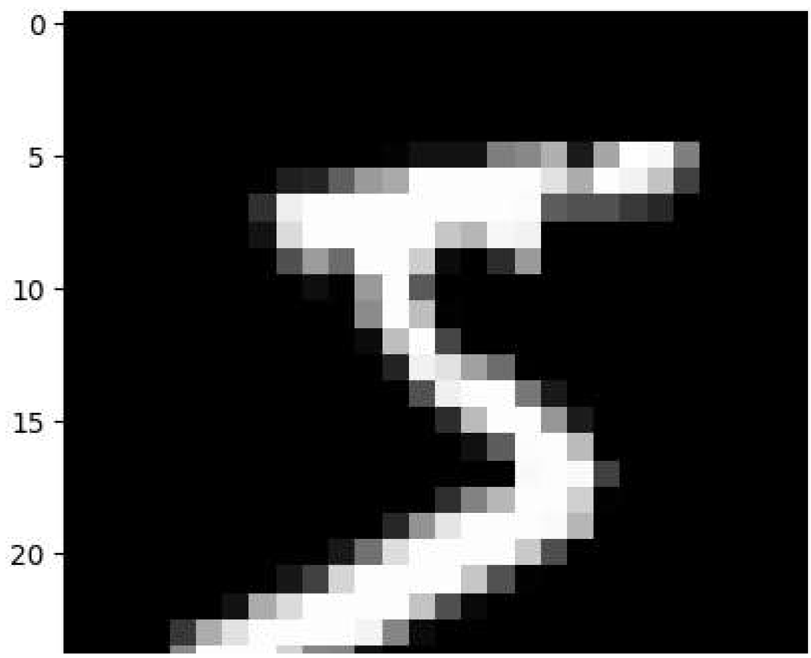
```
x = np.load("mnist.npy")
print(x.shape)
```

```
(70000, 28, 28)
```

Display the first few digits in the dataset.

In [3]:

```
for i in range(5):
    plt.imshow(x[i], cmap="gray")
    plt.show()
```



Computing and diagonalizing the covariance of MNIST

We will interpret each image as a vector in $\mathbb{R}^d$ with $d = 28^2 = 768$. The dataset can thus be seen as a matrix $in \mathbb{R}^{n \times d}$ where $n = 70000$.

In [4]:

```
xx = x.reshape((x.shape[0], -1))
xx
```

Out[4]:

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

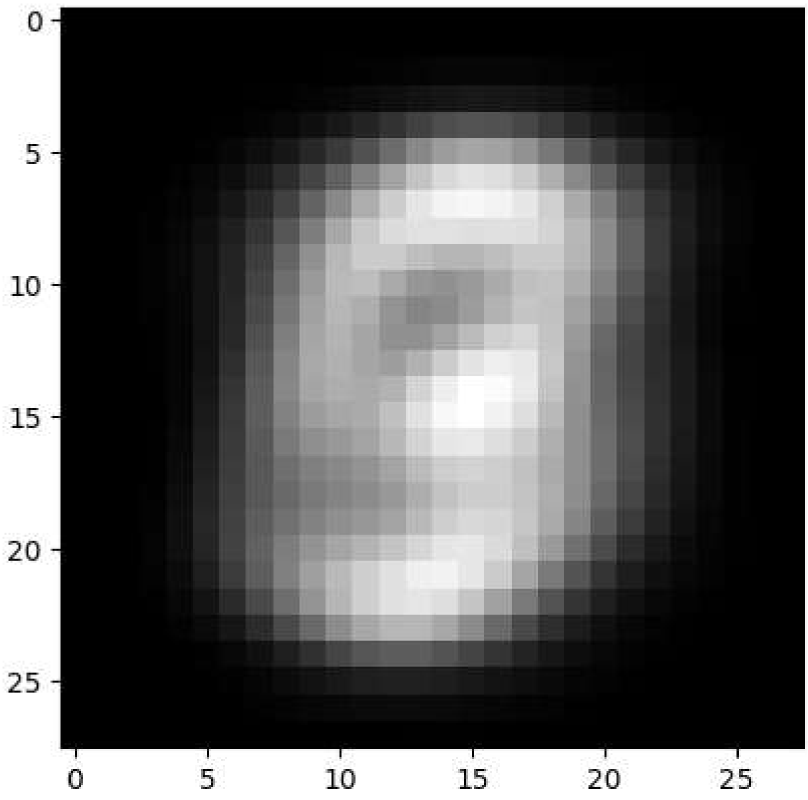
1. Compute the mean $\mu \in \mathbb{R}^d$ of the MNIST dataset and plot it as a 28×28 image.

In [5]:

```
# Your answer here
mean_vector = xx.mean(axis=0)
mean_image = mean_vector.reshape((28,28))
plt.imshow(mean_image, cmap="gray")
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x21641d28130>
```



2. Compute the covariance $\Sigma \in \mathbb{R}^{d \times d}$ of the MNIST dataset and diagonalize it using the function `np.linalg.eigh`.

In [11]:

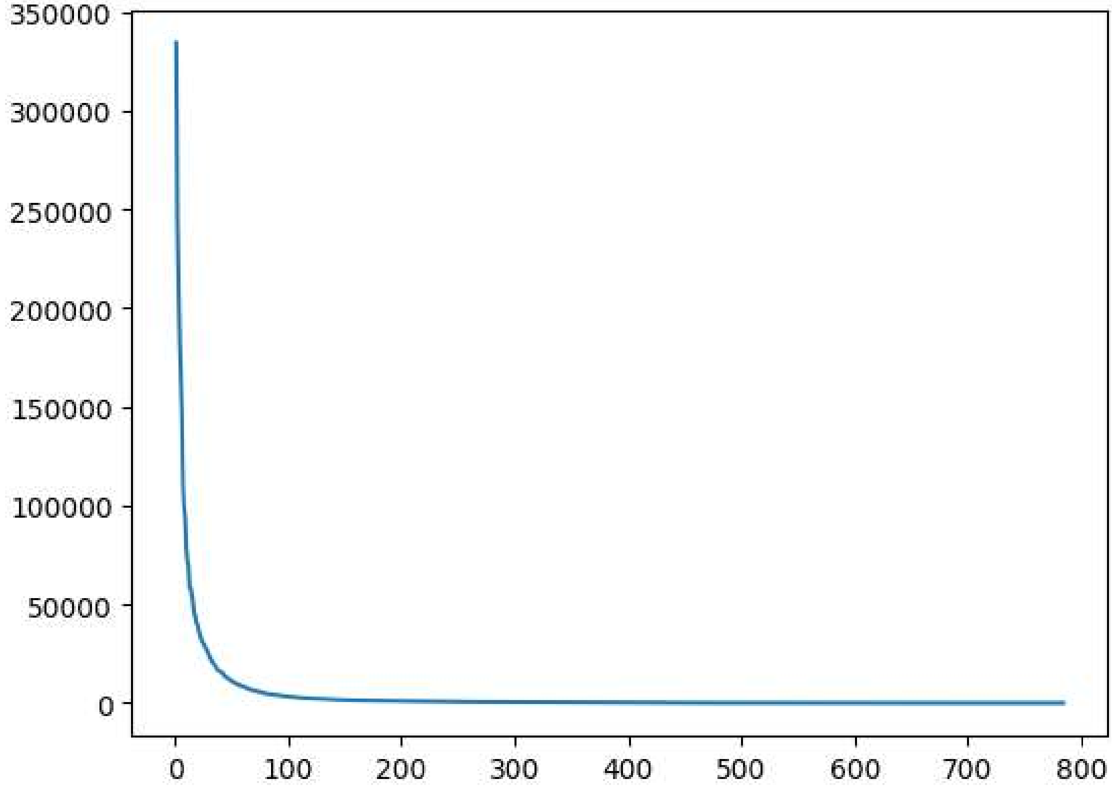
```
# Your answer here
cov_matrix = (1 / x.shape[0]) * (xx - mean_vector).T @ (xx - mean_vector)
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
eigenvalues, eigenvectors
```

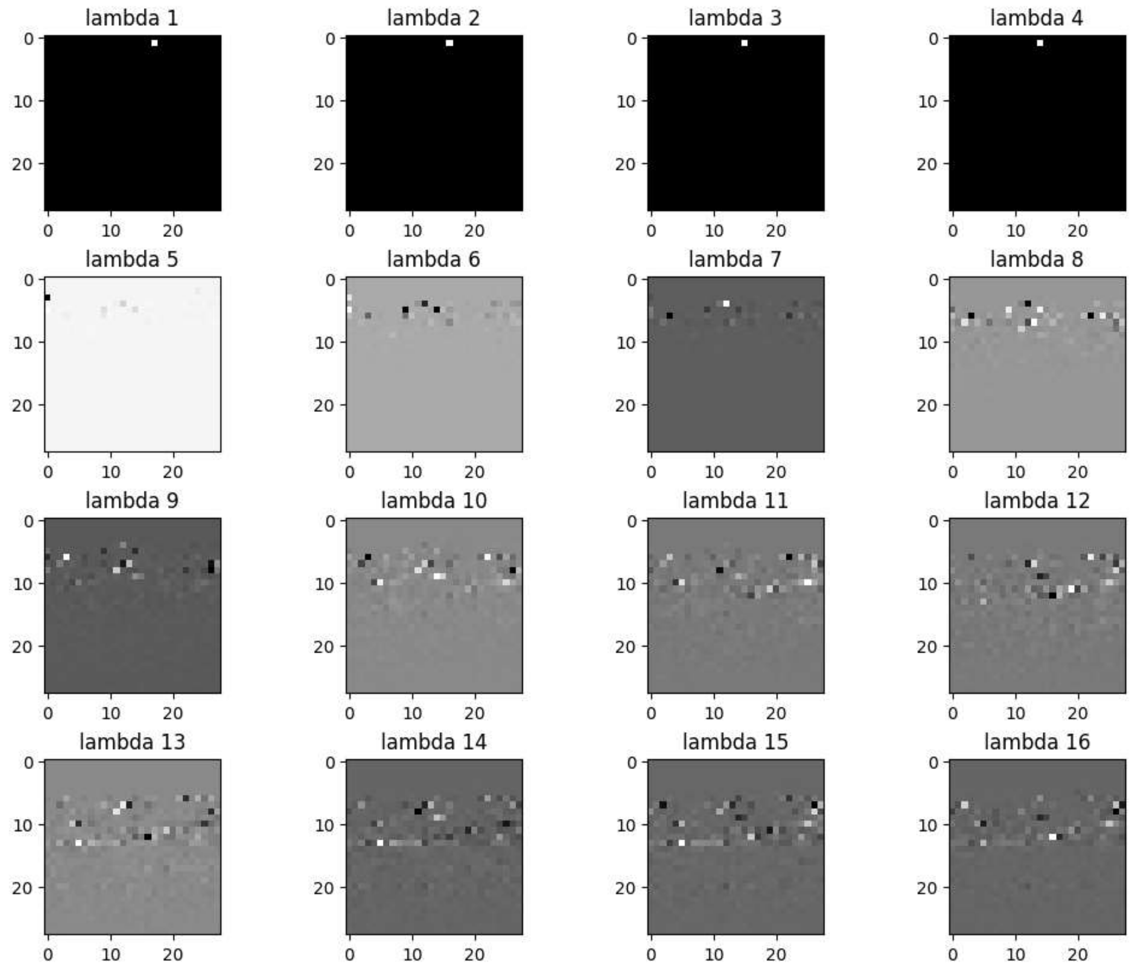
```
        9.24113022e+03,   9.69015100e+03,   9.84750631e+03,   1.01692003e+04,
        1.06378912e+04,   1.08680122e+04,   1.09654969e+04,   1.16184074e+04,
        1.19714546e+04,   1.23968292e+04,   1.28893227e+04,   1.31610777e+04,
        1.35883208e+04,   1.43447224e+04,   1.52605741e+04,   1.55847279e+04,
        1.60383427e+04,   1.64280388e+04,   1.67066997e+04,   1.73116665e+04,
        1.86409047e+04,   1.94395023e+04,   2.00863333e+04,   2.06079485e+04,
        2.21394661e+04,   2.25055712e+04,   2.36673094e+04,   2.53908053e+04,
        2.69499231e+04,   2.77770236e+04,   2.87712502e+04,   3.02965122e+04,
        3.12002508e+04,   3.28986421e+04,   3.46356878e+04,   3.65648922e+04,
        3.95454343e+04,   4.07231430e+04,   4.38698621e+04,   4.52536592e+04,
        5.09812503e+04,   5.43096063e+04,   5.81044457e+04,   5.85518694e+04,
        6.98875556e+04,   7.22588790e+04,   8.03348093e+04,   9.46111872e+04,
        9.91139674e+04,   1.12443533e+05,   1.47668187e+05,   1.67689177e+05,
        1.85334709e+05,   2.10927341e+05,   2.45429921e+05,   3.34289286e+05]),
 array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
```

3. Plot the ordered eigenvalues $\lambda_1 \geq \cdots \geq \lambda_k \geq \cdots$ as a function $k = 1, \ldots, d$ with the x axis in log scale, and the first few eigenvectors $u_1, \ldots, u_k, \ldots$ as 28×28 images.

In [47]:

```
# Your answer here
eigenvalues_tuples = [(i, eigenvalues[i]) for i in range(len(eigenvalues))]
sorted_eigenvalues = sorted(eigenvalues_tuples, key = lambda x: x[1], reverse=True)
sorted_eigenvectors = eigenvectors[list(map(lambda x: x[0], sorted_eigenvalues))]

plt.plot(range(1,len(eigenvalues)+1),sorted(eigenvalues,reverse=True))
fig, axes = plt.subplots(4,4, layout='constrained', figsize=(10, 8))
for i in range(4):
    for j in range(4):
        index = 4 * i + j
        axes[i][j].imshow(sorted_eigenvectors[index].reshape(28, 28), cmap="gray")
        axes[i][j].set_title(f"lambda {index + 1}")
```



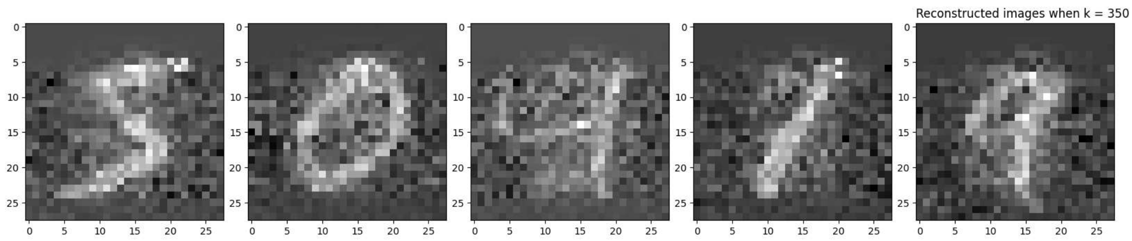


ataset using the
$\cdots + z_{i,k} u_k$,
$, u_k)$. Display
ecognizing the

In [123]:

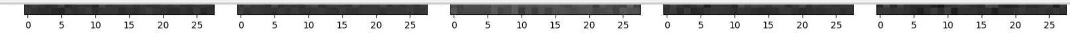
```
# Your answer here
# Compute the dimension of each data point
def dim_data(data):
    return data.shape[1]

# Compute the mean of the sample
def mean_data(data):
    return np.mean(data, axis=0)


# Compute the standard deviation of the sample
def sd_data(data):
    return (data - np.mean(data, axis=0)) / np.svd(data, axis=0)


# Centerize the data for further computation of the covariance matrix
def centerize_data(data):
    # Centerize the data
    return data - np.mean(data, axis=0)

# Compute the eigenbasis with k eigenvectors
def compute_eigenbasis_k(centered_data, k):
    # Compute the top k eigenvectors for our eigenbasis
    cov_matrix = 1 / len(centered_data) * centered_data.T @ centered_data
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

    sorted_eigenvalues = sorted([(i, eigenvalues[i]) for i in range(len(eigenvalues))], key = l
    sorted_eigenvectors = eigenvectors[list(map(lambda x: x[0], sorted_eigenvalues))][:k]

    return sorted_eigenvectors.T


# Main Procedure: PCA
def PCA_procedure(data, k):
    centered_data = centerize_data(data)
    V_k = compute_eigenbasis_k(centered_data, k)

    # Find PCA coordinates
    Z_k = V_k.T @ centered_data.T

    # Z_k is a matrix with (k, 70000), where the coordinates for each data point is organized in
    return Z_k, V_k


# Main Procesure: Inverse PCA
def inverse_PCA_procedure(Z_k, V_k, data, k):
    centered_data = centerize_data(data)
    mu = data.mean(axis=0)

    # Revert to origin
    RC_k = V_k @ Z_k + mu.reshape(dim_data(data),1)


    # RC_k is a matrix with (784, 70000), where the reconstructed coordinates for each data poin
    return RC_k

# Display the reconstructed images
def show_first_five_reconstructed(data, k):
    # Draw the first five reconstructed images
```

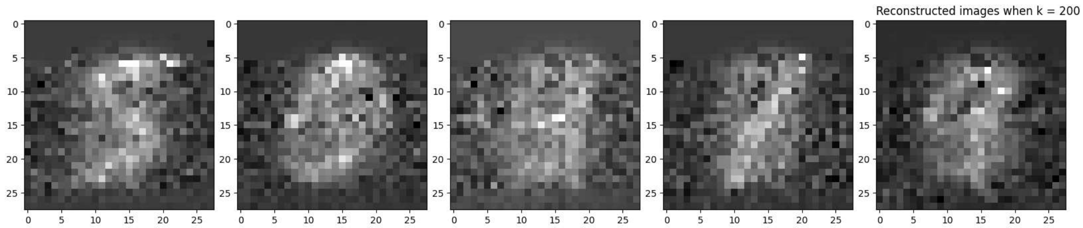
```
    fig, axes = plt.subplots(1, 5, figsize=(16,16), constrained_layout=True)

    Z_k, V_k = PCA_procedure(data, k)
    RC_k = inverse_PCA_procedure(Z_k, V_k, data, k)

    for i in range(5):
        axes[i].imshow(RC_k[:, i].reshape(28,28), cmap="gray")

    plt.title(f"Reconstructed images when k = {k}", loc = "left")


def problem_1(data):
    k = 350
    show_first_five_reconstructed(data, k)


def problem_2(data):
    k_list = [150, 200, 220, 240, 250, 260, 270, 280, 290, 300]
    for k in k_list:
        show_first_five_reconstructed(data, k)


# xx is (70000, 784)
problem_1(xx) # We pick 350, when we could recognize the reconstructed digits by raw eyes, which
```

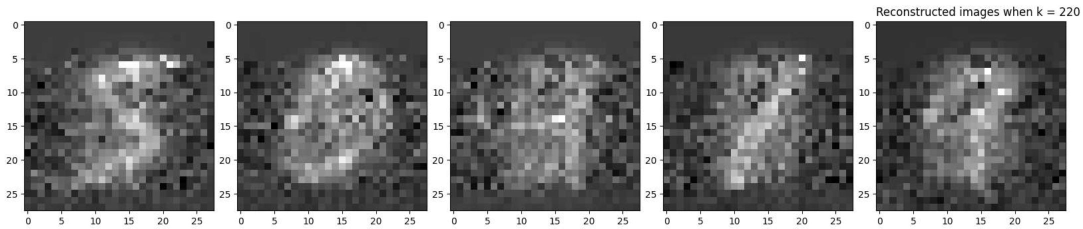


In [124]:

```
problem_2(xx) # We find that 300 is minimum number of eigenvectors that are needed to reconstruc
```







In [ ]: