# Week 2: Principal Component Analysis

**Isaac Yoho, Brandon Causing, Natalia Montalvo**

In this workshop, we will work through a set of problems on dimensionality reduction -- a cannonical form of unsupervised learning. Within the machine learning pipeline, dimensionality reduction is an important tool, which can used in EDA to understand patterns in the data, feature engineering to create a low-dimensional representation of the inputs, and/or in the final phase when you are presenting and visualizing your solution.

As usual, the worksheets will be completed in teams of 2-3, using **pair programming**, and we have provided cues to switch roles between driver and navigator. When completing worksheets:

> - You will have tasks tagged by (CORE) and (EXTRA).
> - Your primary aim is to complete the (CORE) components during the WS session, afterwards you can try to complete the (EXTRA) tasks for your self-learning process.
> - Look for the 🏁 as cue to switch roles between driver and navigator.
> - In some Exercises, you will see some beneficial hints at the bottom of questions.

Instructions for submitting your workshops can be found at the end of worksheet. As a reminder, you must submit a pdf of your notebook on Learn by 16:00 PM on the Friday of the week the workshop was given.

As you work through the problems it will help to refer to your lecture notes (navigator). The exercises here are designed to reinforce the topics covered this week. Please discuss with the tutors if you get stuck, even early on!

## Outline

# Problem Definition and Setup

## Packages

First, lets load in some packages to get us started.

In [50]:
```
import matplotlib.pyplot as plt  # plotting API (Matplotlib)
import seaborn as sns  # statistical plotting styles/helpers (Seaborn)
import numpy as np  # numerical arrays and linear algebra (NumPy)
import pandas as pd  # tabular data structures (pandas)
from sklearn.decomposition import PCA  # principal component analysis transforme
from sklearn.preprocessing import StandardScaler  # z-score standardization tran
```

## Data

Our dataset will be the famous MNIST dataset of handwritten digits, which we will download from sklearn. The dataset consists of a set of greyscale images of the numbers 0-9 and corresponding labels. Usually the goal is to train a classifier (i.e. given an image, what digit does it correspond to?). Here we will throw away the labels and focus on the images themselves. Specifically, we will use dimensionality reduction to explore the images and underlying patterns and find a low-dimensional representation.

First, load the data:

In [51]:
```
from sklearn.datasets import fetch_openml  # utility to download datasets from O

mnist = fetch_openml('mnist_784', parser='auto')  # load MNIST (784 features/pix
X = mnist.data  # feature matrix: rows are images, columns are pixel intensities
y = mnist.target  # labels/targets: digit class for each image (often stored as
```

### 🚩 Exercise 1 (CORE)

What is stored in `X` and `y` in the command above? What is the shape/datatype etc if an array?

In [52]:
```
X

# X is a dataframe of pixel intensities at certain locations
# There are 70000 28x28 images and they are flattened into vectors of length 784
```

Out[52]:

| | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | pixel10 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **69995** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **69996** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **69997** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **69998** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| **69999** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |

70000 rows × 784 columns

In [53]:
```
y

# Each entry in y corresponds to a row index in X
# It tells you what number that vector corresponds with
```

Out[53]:
```
0        5
1        0
2        4
3        1
4        9
        ..
69995    2
69996    3
69997    4
69998    5
69999    6
Name: class, Length: 70000, dtype: category
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']
```

For X:

```
X is a dataframe of pixel intensities at certain locations.
There are 70000 28x28 images and they are flattened into vectors
of length 784.
```

For Y:

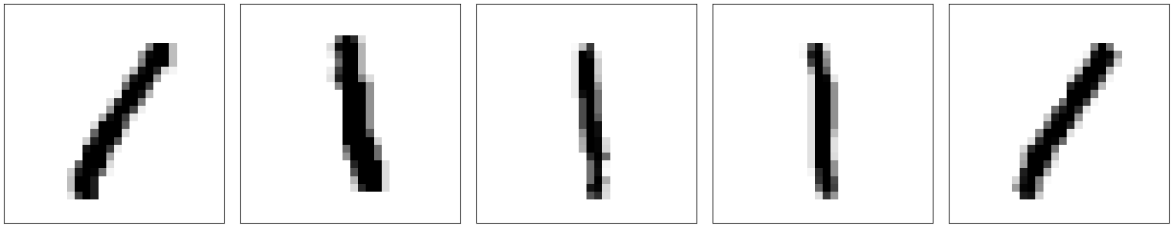
```
Each entry in y corresponds to a row index in X. It tells you
what number that vector corresponds with.
```

Now, let's create a dictionary, with the digit classes (0-9) as keys, where the correponding values are the set of all images corresponding to that particular label.

In [54]:
```
digits_dict = {}  # initialise dict: label (0-9) -> list of image vectors
X_ = X.values  # convert the pandas DataFrame `X` into a NumPy array for fast ro
count = 0  # row counter to keep X_ aligned with the current label in `y`

for label in y:  # iterate over each label in the same order as the rows of `X`
  if label in digits_dict:  # if we've seen this label before, append to its lis
    digits_dict[label] += [X_[count]]  # add the current image (row) to the labe
  else:  # if this is the first time we see this label, create the list
    digits_dict[label] = [X_[count]]  # start a new list containing the current
  count += 1  # increment row index so the next label matches the next image
```

Next let's visualize some of the images. We will start by picking a label and plotting a few images from within the dictionary. Note that each image contains a total of 784 pixels (28 by 28) and we will need to `reshape` the image to plot with `imshow(...,cmap='gray_r')`. Try also changing the label to view different digits.

In [55]:
```
mylabel = '1'  # choose which digit label to visualise (as stored in `y`, often
n_images_per_label = 5  # how many examples of this digit to plot

fig = plt.figure(figsize=(4 * n_images_per_label, 4))  # create a figure sized t
for j in range(n_images_per_label):  # loop over the first `n_images_per_label`
    ax_number = 1 + j  # subplot index (Matplotlib subplots are 1-indexed here)
    ax = fig.add_subplot(1, n_images_per_label, ax_number)  # add a subplot in a
    ax.imshow(digits_dict[mylabel][j].reshape((28, 28)), cmap='gray_r')  # resha
    ax.set_xticks([])  # hide x-axis tick marks for cleaner image display
    ax.set_yticks([])  # hide y-axis tick marks for cleaner image display
fig.tight_layout()  # automatically adjust spacing so subplots don't overlap
```



## Exercise 2 (EXTRA)

Edit the code above to plot a few images for multiple labels.

▶ **Hint**

In [9]:
```
# Code for your answer here!
```

## Exercise 3 (CORE)

Now focus on the 3s only and create a data matrix called `X_threes`. Define also `N` (# datapoints) and `D` (# features).

What are the features in this problem? How many features and data points are there?

In [56]:
```
X_threes = np.array(X[y == "3"].reset_index(drop=True))
```

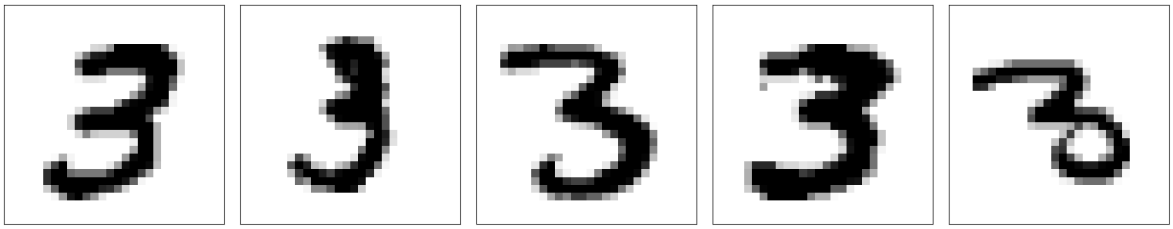
```
N = X_threes.shape[0]
D = X_threes.shape[1]
```

The features are the different pixel locations. There are 784 features and 7141 data points.

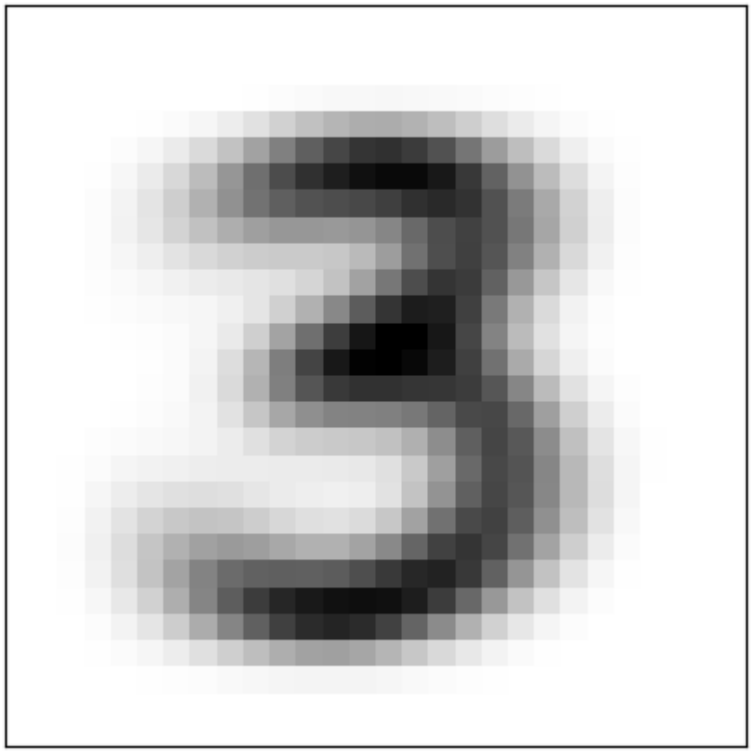
## Exercise 4 (CORE)

Now compute and plot the mean image of three.

In [61]:
```
mylabel = '3'  # choose which digit label to visualise (as stored in `y`, often
n_images_per_label = 5  # how many examples of this digit to plot

fig = plt.figure(figsize=(4 * n_images_per_label, 4))  # create a figure sized t
for j in range(n_images_per_label):  # loop over the first `n_images_per_label`
    ax_number = 1 + j  # subplot index (Matplotlib subplots are 1-indexed here)
    ax = fig.add_subplot(1, n_images_per_label, ax_number)  # add a subplot in a
    ax.imshow(digits_dict[mylabel][j].reshape((28, 28)), cmap='gray_r')  # resha
    ax.set_xticks([])  # hide x-axis tick marks for cleaner image display
    ax.set_yticks([])  # hide y-axis tick marks for cleaner image display
fig.tight_layout()  # automatically adjust spacing so subplots don't overlap
```

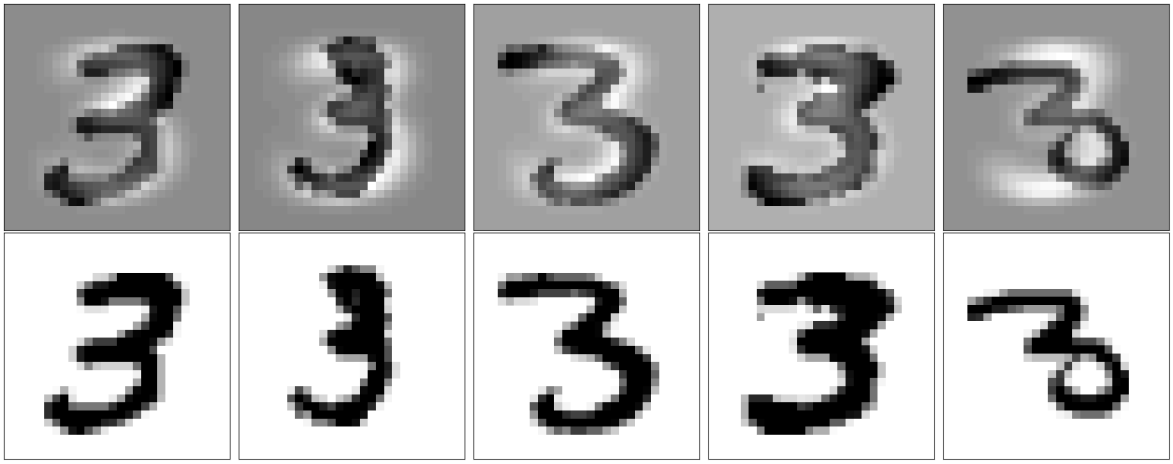


In [62]:
```
X_three_mean = np.array(X_threes.mean(axis=0))

fig, ax = plt.subplots()  # add a subplot in a 1 x n_images_per_label grid
ax.imshow(X_three_mean.reshape((28, 28)), cmap='gray_r')  # reshape vector -> 28
ax.set_xticks([])  # hide x-axis tick marks for cleaner image display
ax.set_yticks([])

plt.show()
```



Run the following code to first create a new data matrix that centers the data by subtracting the mean image, and then visualise some of the images and compare to the original data. Note: you will need to replace `X_three_mean` with the name you gave the mean image in the computation above.

In [63]:
```
X_three_centred = X_threes - X_three_mean  # centre each image by subtracting th

n_images = 5  # number of example images to show

fig = plt.figure(figsize=(4 * n_images, 4 * 2))  # create a figure with 2 rows (
for j in range(n_images):  # loop over a few images to visualise
  ax = fig.add_subplot(2, n_images, j + 1)  # top row: centred images
  ax.imshow(X_three_centred[j, :].reshape((28, 28)), cmap='gray_r')  # reshape a
  ax.set_xticks([])  # hide x ticks
  ax.set_yticks([])  # hide y ticks

  ax = fig.add_subplot(2, n_images, j + 1 + n_images)  # bottom row: original (u
  ax.imshow(X_threes[j, :].reshape((28, 28)), cmap='gray_r')  # reshape and plot
  ax.set_xticks([])  # hide x ticks
  ax.set_yticks([])  # hide y ticks
fig.tight_layout()  # adjust subplot spacing
```



In [64]:
```
X_threes - X_three_mean
```

Out[64]:
```
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.]], shape=(7141, 784))
```

### 🚩 Exercise 5 (CORE)

Comment on whether or not the images need to be standardized before using PCA

The images do not need to be standardized because all of the features are in the same "units".

🏁 **Now, is a good point to switch driver and navigator**

# PCA

Now, we will perform PCA to summarize the main patterns in the images. We will use the `PCA()` transformer from the `sklearn.decomposition` package:

- As we saw last week, we start by creating our transformer object, specifying any parameters as desired. For example, we can specify the number of components with the option `n_components`. If omitted, all components are kept.
- Note that by default the `PCA()` transform centers the variables to have zero mean (but does not scale them).
- After calling `.fit()`, our fitted object has a number of attributes, including:
  - the mean accessible through the attribute `mean_`.
  - the basis vectors (principal components) accesible through the `components_` attribute.
- There are also a number of methods for the fitted object, including `.transform()` to obtain the low-dimensional representation (or also `fit_transform` combining

both together).

First, let's create the PCA transformer object and call `.fit()`:

In [65]:
```
pca_threes = PCA(n_components=200)  # create PCA object keeping the first 200 pr
pca_threes.fit(X_threes)  # fit PCA to the digit-3 data (computes mean and compo
```

Out[65]:

▼ PCA ⓘ ?

▶ Parameters

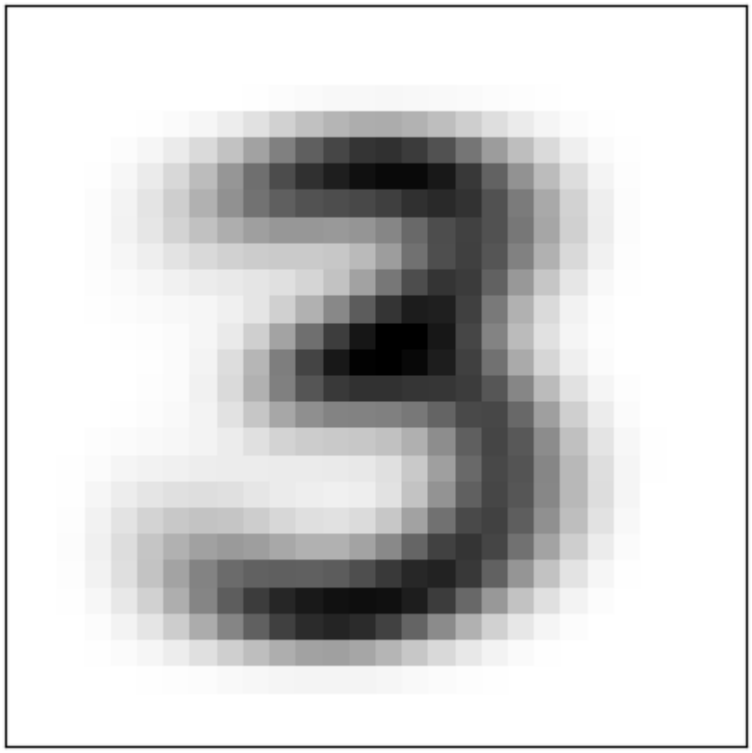
# Examining the Basis Vectors and Scores

## Exercise 6 (EXTRA)

Plot the mean image by accessing the `mean_` attribute and check that it is the same as above.

In [66]:
```
mean = pca_threes.mean_

fig, ax = plt.subplots()  # add a subplot in a 1 x n_images_per_label grid
ax.imshow(mean.reshape((28, 28)), cmap='gray_r')  # reshape vector -> 28x28 and
ax.set_xticks([]) # hide x-axis tick marks for cleaner image display
ax.set_yticks([])
```

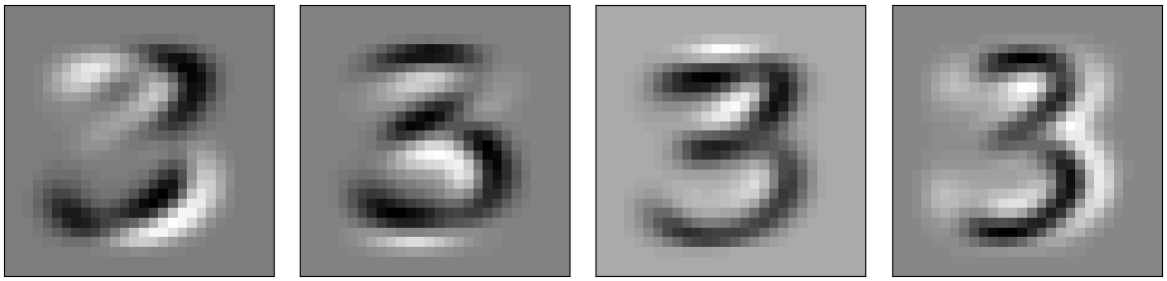
Out[66]:
```
[]
```



## Exercise 7 (CORE)

Plot the the first four basis vectors as images by accessing the `components_` attribute. What patterns do they seem to describe?

In [67]:
```
comp = pca_threes.components_  # The components of the PCA

fig = plt.figure(figsize=(12, 3))  # Optional: set figure size

for j, img in enumerate(comp[:4]):  # Use enumerate to get index j
    ax = fig.add_subplot(1, 4, j + 1)  # top row: centred images
    ax.imshow(img.reshape((28, 28)), cmap='gray_r')  # reshape vector -> 28x28 a
    ax.set_xticks([])  # hide x-axis tick marks for cleaner image display
    ax.set_yticks([])

fig.tight_layout()  # adjust subplot spacing
plt.show()
```

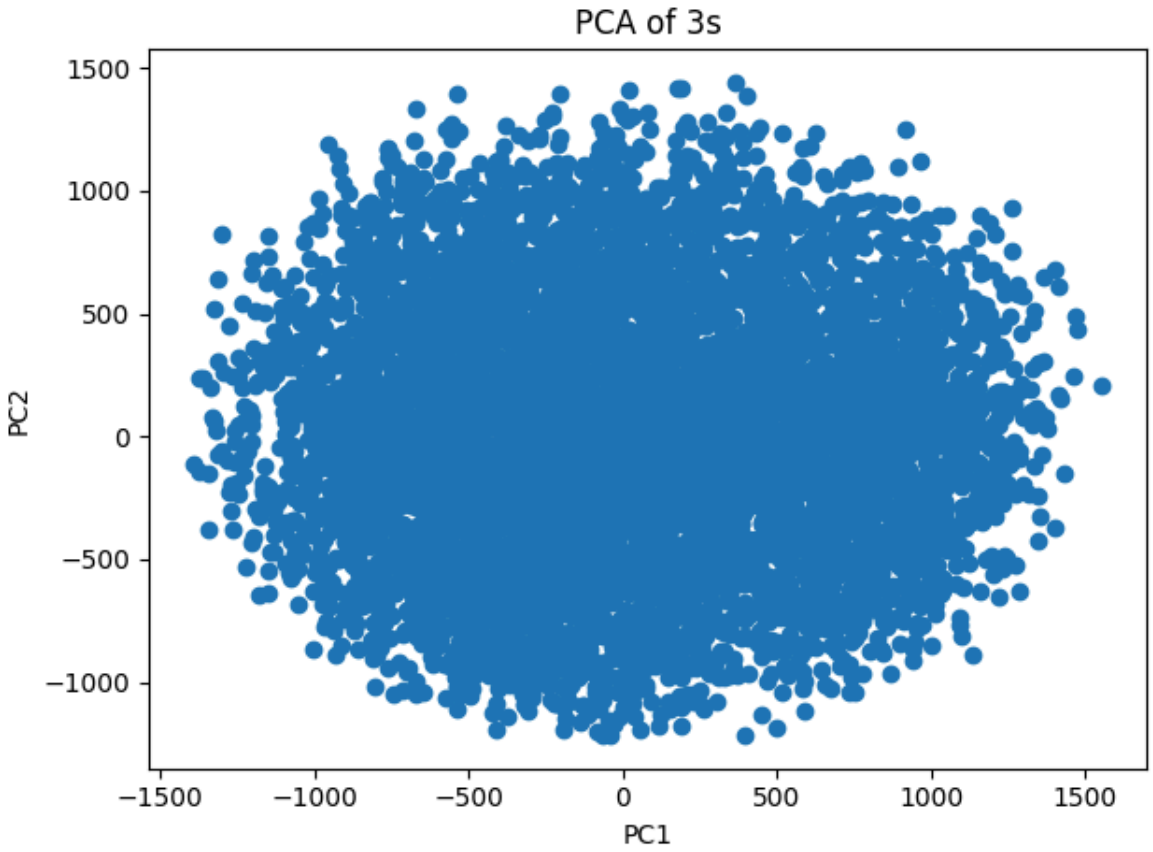


The images seem to capture a significant portion of the variability explained by the principal components. In other words, the first few principal components will deviate the most from the mean 3.

## Exercise 8 (CORE)

a) Use the `transform()` method to compute the PCA scores and save them in an object called `scores`. Then, plot the data points in the low-dimensional space spanned by the first two principal components.

In [68]:
```
scores = pca_threes.transform(X_threes) # compute pca scores

fig, ax = plt.subplots()

pc1 = scores[:,0]
pc2 = scores[:,1]

scatter = ax.scatter(pc1, pc2)
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_title('PCA of 3s')
plt.tight_layout()
plt.show()
```




To better interpret the latent dimensions, let's look at some projected points along each dimension and the corresponding images. Specifically, run the following code to:

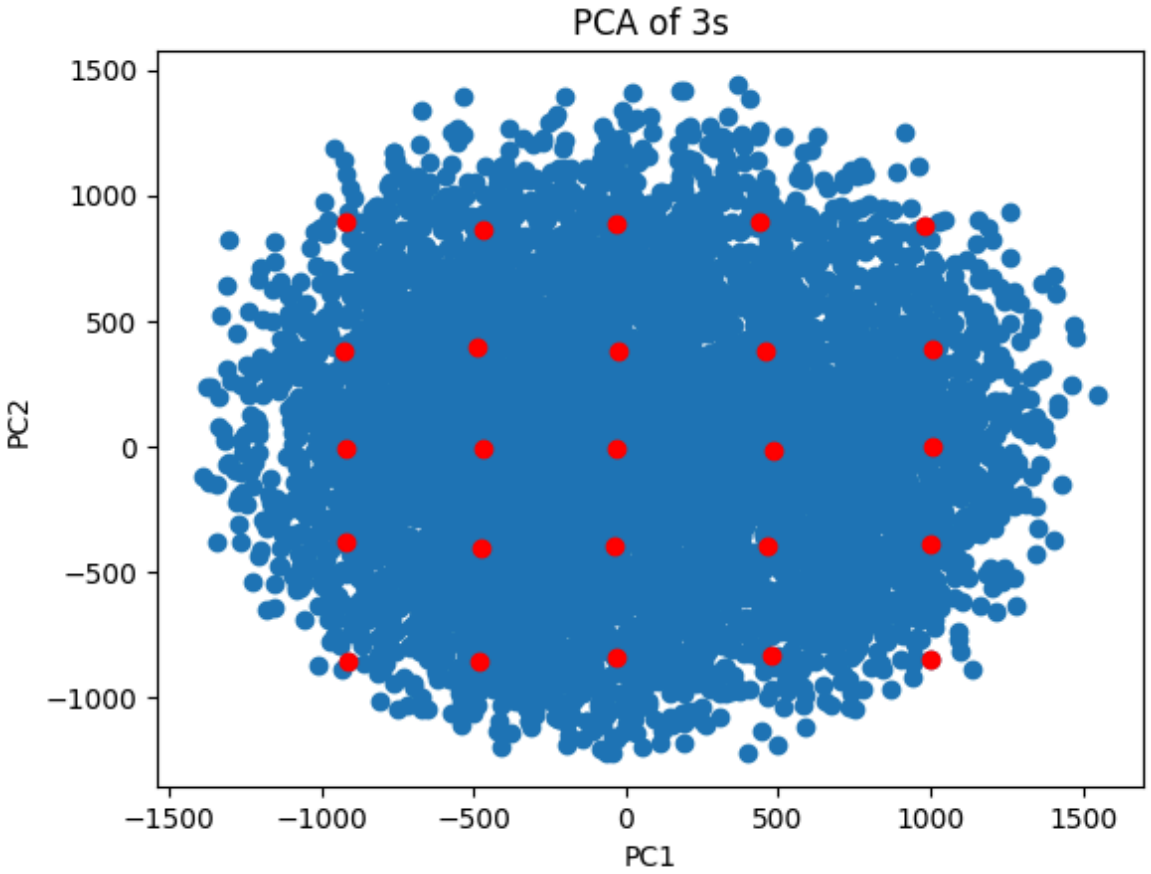
- first compute the $5, 25, 50, 75, 95\%$ quantiles of the scores for the first two dimensions
- then find the data point whose projection is closest to each combination of quantiles.

In [69]:
```
s1q = np.quantile(scores[:, 0], [.05, .25, .5, .75, .95])  # compute 5/25/50/75/
s2q = np.quantile(scores[:, 1], [.05, .25, .5, .75, .95])  # compute 5/25/50/75/

idx = np.zeros([len(s1q), len(s2q)])  # allocate array to store indices of close

for i in range(len(s1q)):  # loop over PC1 quantiles
    for j in range(len(s2q)):  # loop over PC2 quantiles
        aux = ((scores[:, 0] - s1q[i]) ** 2 + (scores[:, 1] - s2q[j]) ** 2).resh
        idx[i, j] = np.where(aux == min(aux))[0][0]  # index of the point with m

idx = idx.astype(int)  # cast indices to integers for array indexing
```

b) Now, add these points in red to your plot above in.

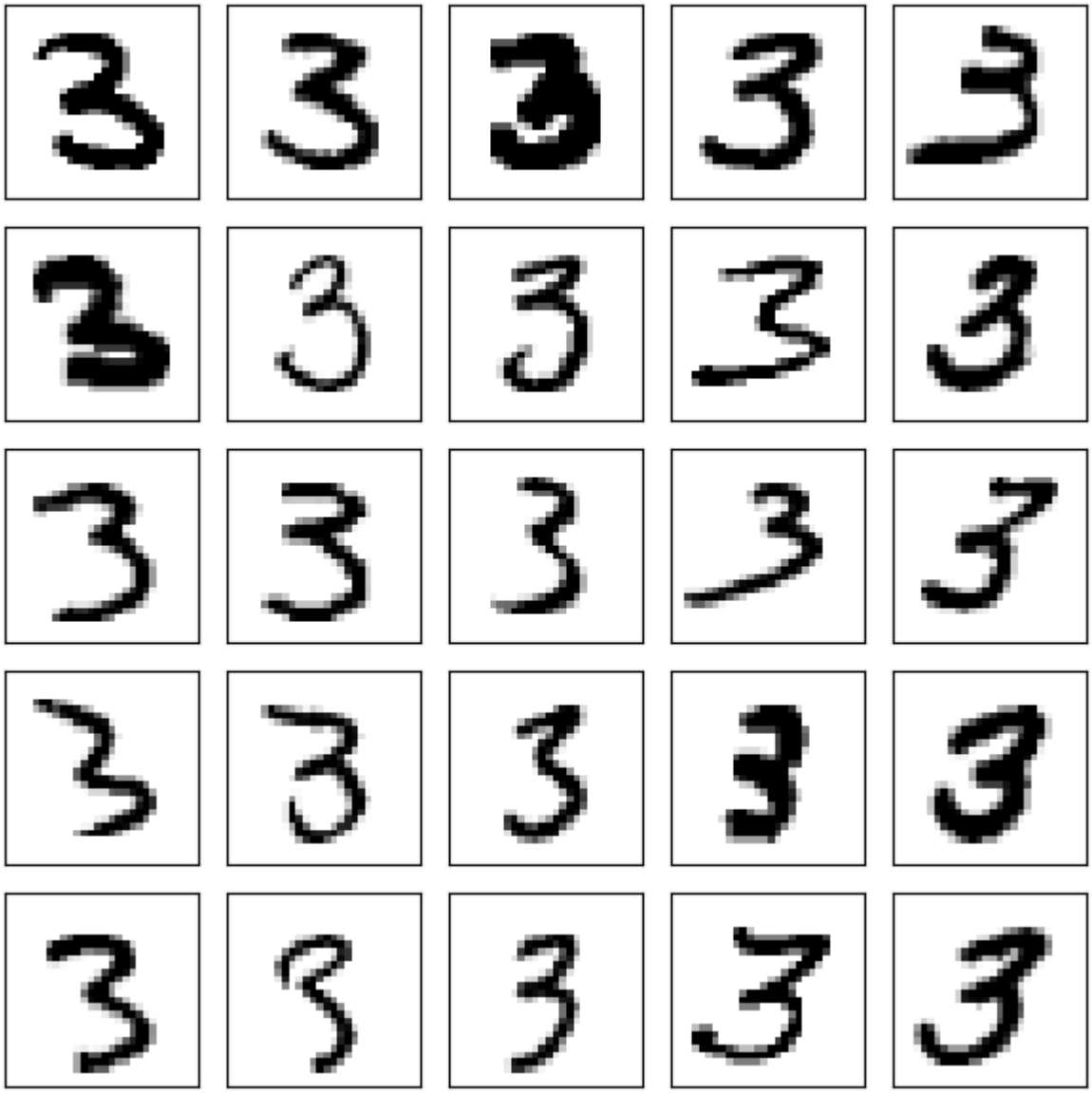
In [70]:
```
scores = pca_threes.transform(X_threes)

pc1 = scores[:,0]
pc2 = scores[:,1]

fig, ax = plt.subplots()

scatter = ax.scatter(scores[:,0], scores[:,1])
red = ax.scatter(pc1[idx.flatten()], pc2[idx.flatten()], color='red')
```

```
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_title('PCA of 3s')
plt.show()
```




c) Run the following code to plot the images corresponding to this grid of points. Describe the general pattern of the first (left to right) and second (down to up) principal component.

In [72]:
```
fig, ax = plt.subplots(len(s1q), len(s2q), figsize=(6, 6))  # create a grid of s
for i in range(len(s1q)):  # iterate over PC1 quantile levels (columns)
    for j in range(len(s2q)):  # iterate over PC2 quantile levels (rows)
        ax[len(s2q) - 1 - j, i].imshow(X_threes[idx[i, j], :].reshape((28, 28)),
plt.setp(ax, xticks=[], yticks=[])  # remove axis ticks for all subplots
fig.tight_layout()  # reduce overlap and improve spacing
```



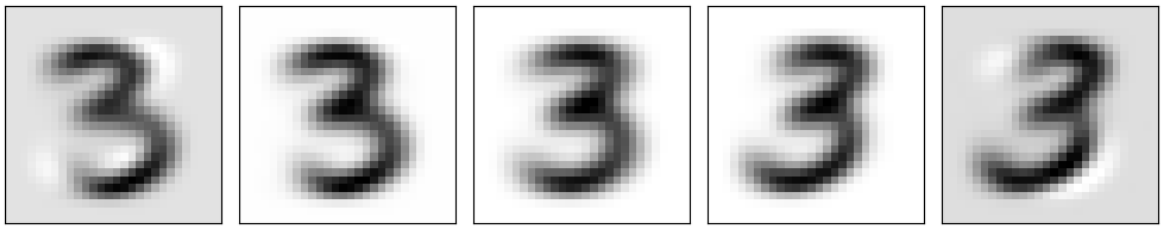
PC1(Left to Right): On the left the lines look thinner than the right side. Maybe the PC1 is capturing the instesity of the written number

PC2(Bottom to Top): On the bottom we can see more rounded lines, and in the top sharper strokes. PC2 looks like it is capturing the style of the 3.

You can also try to create some artificial images, by fixing different values of the weights. This can also help to interpret the latent dimensions.

In [73]:
```
weight1 = np.quantile(scores[:, 0], [.05, .25, .5, .75, .95])  # choose represen
weight2 = 0  # fix PC2 weight to 0 so we vary only along the first component

images_pc1 = np.zeros([len(weight1), D])  # allocate array for synthetic images

count = 0  # counter to place each generated image into `images_pc1`
for w in weight1:  # iterate over selected PC1 weights
    images_pc1[count, :] = (pca_threes.mean_ + pca_threes.components_[0, :] * w
    count += 1  # move to next row in `images_pc1`


fig, ax = plt.subplots(1, len(weight1), figsize=(10, 6))  # create a row of subp
for i in range(len(weight1)):  # loop over the generated images
```

```
    ax[i].imshow(images_pc1[i, :].reshape((28, 28)), cmap='gray_r')  # reshape a
plt.setp(ax, xticks=[], yticks=[])  # remove ticks for cleaner presentation
fig.tight_layout()  # adjust layout to prevent overlap
```

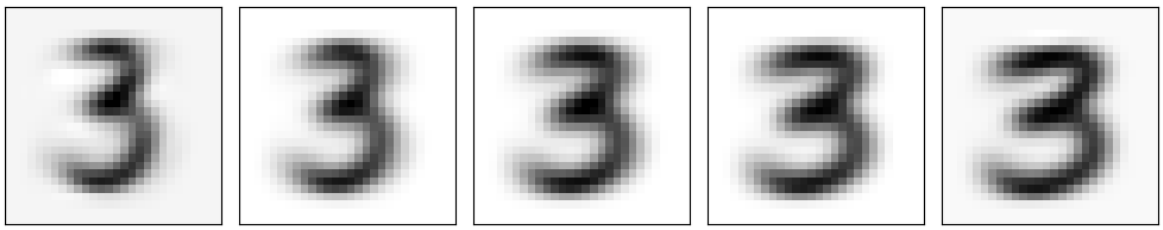


## Exercise 9 (CORE)

Repeat this to describe the third principal component.

In [74]:
```
weight3 = np.quantile(scores[:, 2], [.05, .25, .5, .75, .95])  # choose represen
weight2 = 0  # fix PC2 weight to 0 so we vary only along the third component
weight1 = 0  # fix PC1 weight to 0 so we vary only along the third component

images_pc3 = np.zeros([len(weight3), D])  # allocate array for synthetic images

count = 0  # counter to place each generated image into `images_pc1`
for w in weight3:  # iterate over selected PC1 weights
    images_pc3[count, :] = (
        pca_threes.mean_
        + pca_threes.components_[0, :] * weight1
        + pca_threes.components_[1, :] * weight2
        + pca_threes.components_[2, :] * w)
    count += 1  # move to next row in `images_pc1`


fig, ax = plt.subplots(1, len(weight3), figsize=(10, 6))  # create a row of subp
for i in range(len(weight3)):  # loop over the generated images
    ax[i].imshow(images_pc3[i, :].reshape((28, 28)), cmap='gray_r')  # reshape a
plt.setp(ax, xticks=[], yticks=[])  # remove ticks for cleaner presentation
fig.tight_layout()  # adjust layout to prevent overlap

weight3
```

Out[74]:
```
array([-743.35121548, -361.71351265,  -23.30865372,  332.53616945,
        835.30763364])
```



When the weights from PC1 and PC2 the images generated from PC3 align with the mean image, when we increase the weights for PC1 and PC2 we can see more variance in the images.

## Exercise 10 (EXTRA)

In lecture, we saw that we can also compute the basis vectors from an SVD decomposition of the data matrix. Use the `svd` function in `scipy.linalg` to compute the first three basis vectors and verify that they are the same (up to a change in sign -- note that the signs may be flipped because each principal component specifies a direction in the $D$-dimensional space and flipping the sign has no effect as the direction does not change).

Does `PCA()` perform principal component analysis using an eigendecomposition of the empirical covariance matrix or using a SVD decomposition of the data matrix?

In [ ]:
```
# Code for your answer here!
from scipy.linalg import svd  # import singular value decomposition (SVD) for co
```

*Type your answer here!*

🏁 **Now, is a good point to switch driver and navigator**

# Selecting the Number of Components

## 🚩 Exercise 11 (CORE)

Next, let's investigate how many components are needed by considering how much variance is explained by each component.

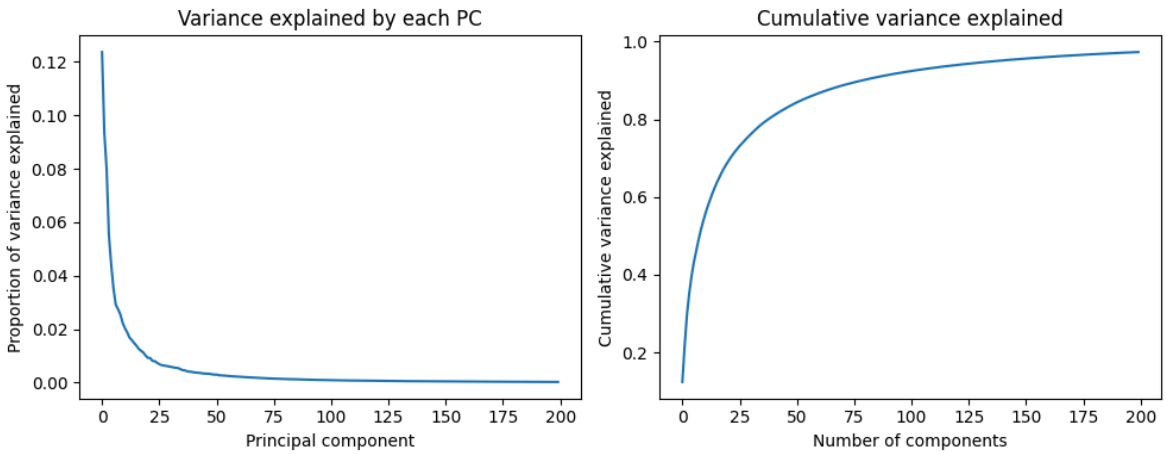
Note that the `pca_threes` object has an attribute `explained_variance_` (variance of each component) and `explained_variance_ratio_` (proportion of variance explained by each component).

Plot both the proportion of variance explained and the cummulative proportion of variance explained. Provide a suggestion of how many components to use. How much variance is explained by the suggest number of components? Comment on why we may be able to use this number of components in relation to the total number of features.

▶ **Hint**

In [75]:
```
var = pca_threes.explained_variance_
var_r = pca_threes.explained_variance_ratio_

cum_var_r = np.cumsum(var_r)

fig, ax = plt.subplots(1, 2, figsize=(10, 4))

# Individual variance explained
ax[0].plot(var_r)
ax[0].set_xlabel('Principal component')
ax[0].set_ylabel('Proportion of variance explained')
ax[0].set_title('Variance explained by each PC')

# Cumulative variance explained
ax[1].plot(cum_var_r)
```

```
ax[1].set_xlabel('Number of components')
ax[1].set_ylabel('Cumulative variance explained')
ax[1].set_title('Cumulative variance explained')

plt.tight_layout()
plt.show()
```




From the Cumulative variance explained graph we can see that we only need around 100 components to get at least 90% of the variance.

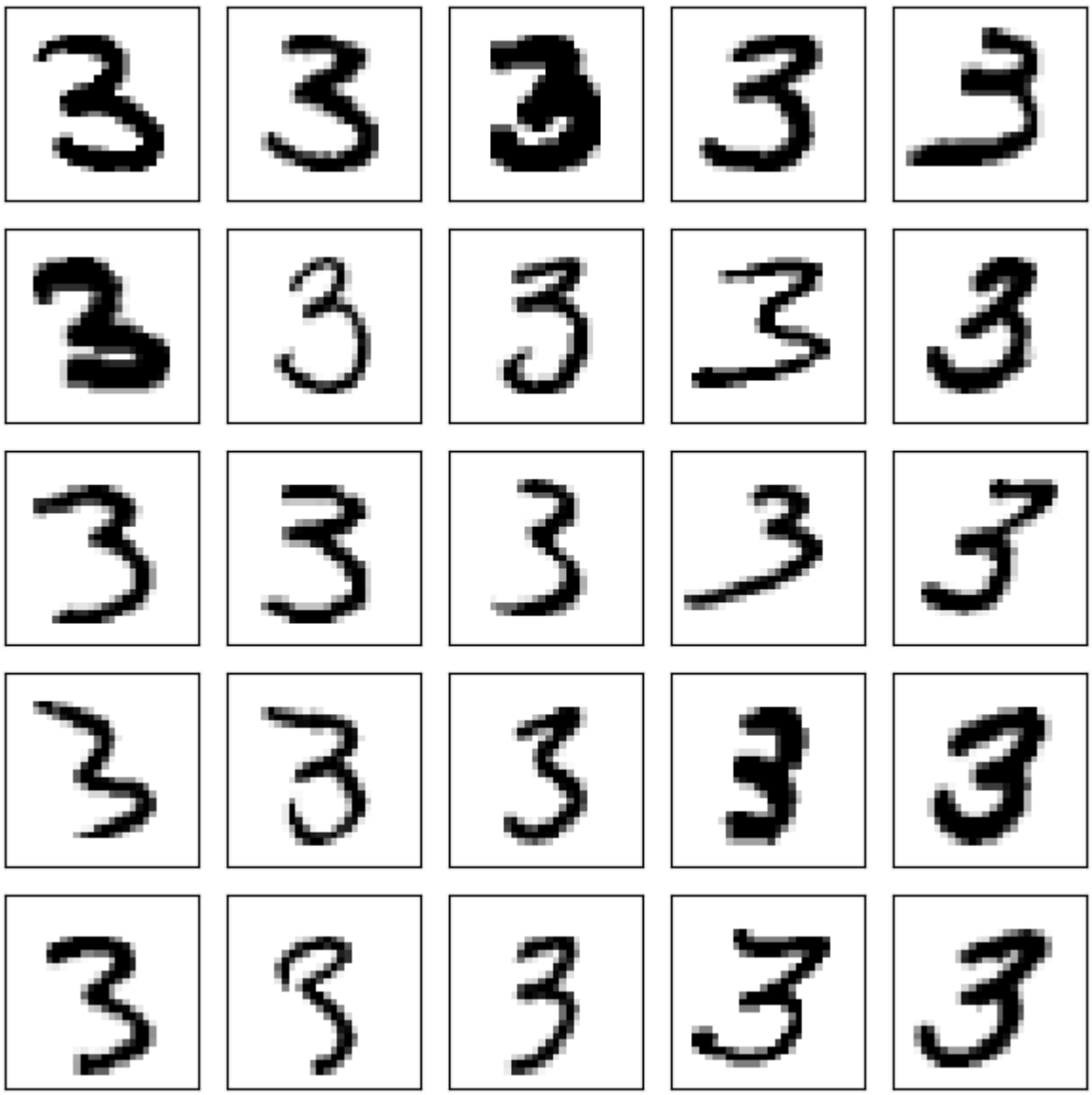
## Exercise 12 (CORE)

For your selected number of components, compute the reconstruted images. Plot the reconstruction for a few images and compare with the original images. Comment on the results.

▶ **Hint**

In [76]:
```
pca_threes = PCA(n_components=100)  # create PCA object keeping the first 200 pr
scores = pca_threes.fit_transform(X_threes)  # fit PCA to the digit-3 data (comp

pc1 = scores[:, 0]
pc2 = scores[:, 1]

s1q = np.quantile(pc1, [.05, .25, .5, .75, .95])  # compute 5/25/50/75/95% quant
s2q = np.quantile(pc2, [.05, .25, .5, .75, .95])  # compute 5/25/50/75/95% quant

idx = np.zeros([len(s1q), len(s2q)])  # allocate array to store indices of close

for i in range(len(s1q)):  # loop over PC1 quantiles
    for j in range(len(s2q)):  # loop over PC2 quantiles
        aux = ((scores[:, 0] - s1q[i]) ** 2 + (scores[:, 1] - s2q[j]) ** 2).resh
        idx[i, j] = np.where(aux == min(aux))[0][0]  # index of the point with m

idx = idx.astype(int)  # cast indices to integers for array indexing


fig, ax = plt.subplots(len(s1q), len(s2q), figsize=(6, 6))  # create a grid of s
for i in range(len(s1q)):  # iterate over PC1 quantile levels (columns)
    for j in range(len(s2q)):  # iterate over PC2 quantile levels (rows)
        ax[len(s2q) - 1 - j, i].imshow(X_threes[idx[i, j], :].reshape((28, 28)),
```

```
plt.setp(ax, xticks=[], yticks=[])  # remove axis ticks for all subplots
fig.tight_layout()  # reduce overlap and improve spacing
```



It looks very similar to the other images we got with the 200 components as we would expect, as with only 100 components we are capturing most of the variance.

🏁 **Now, is a good point to switch driver and navigator**

# Other Digits

Now, let's consider another digit.

## 🚩 Exercise 13 (CORE)

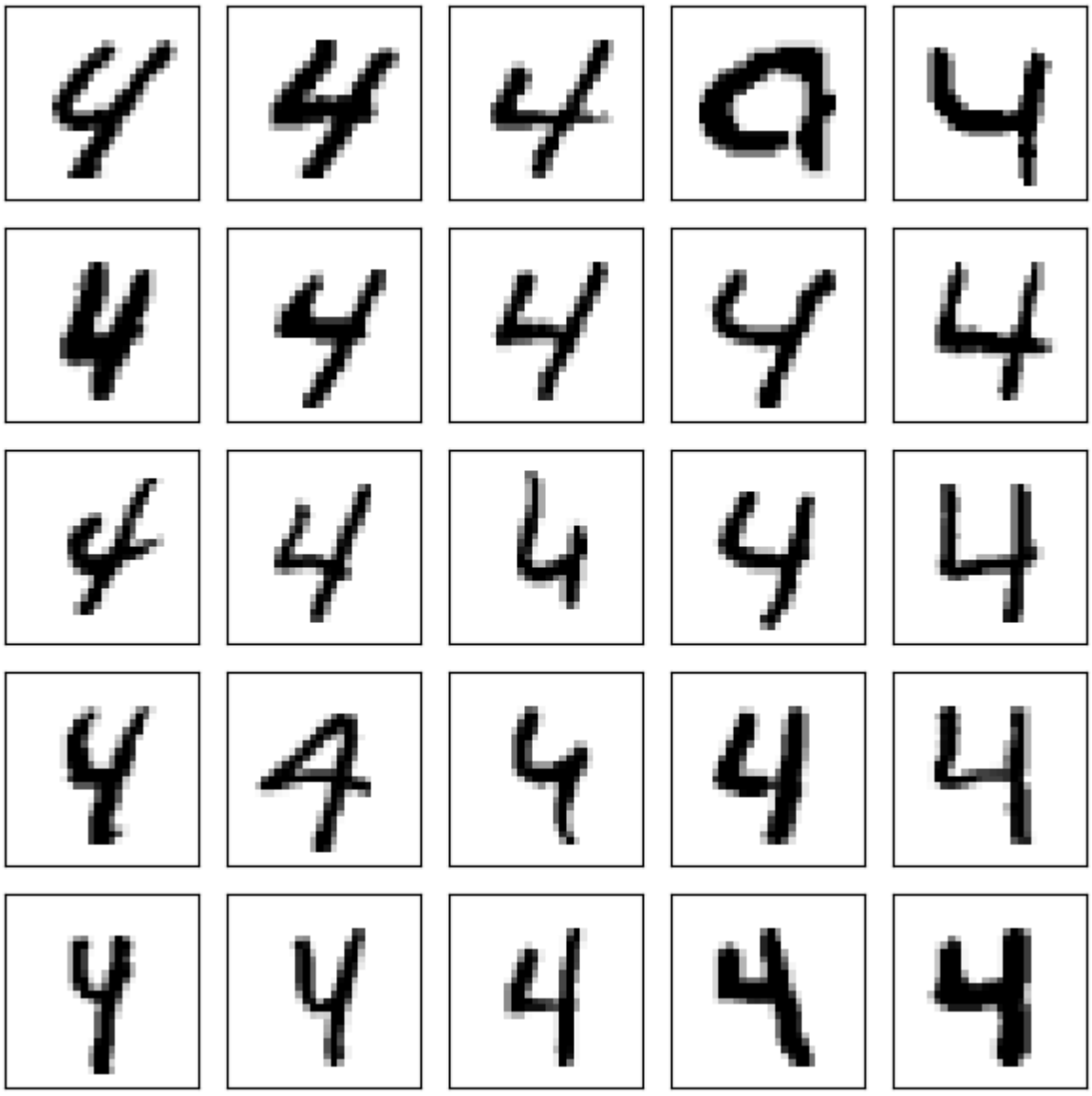
Perform PCA for another choice of digit. What do the first two components describe? Do some digits have better approximations than others? Comment on why this may be.

In [77]:
```
X_four = np.array(X[y == "4"].reset_index(drop=True))

N = X_four.shape[0]
```

```
pca_six = PCA(n_components=100)  # create PCA object keeping the first 200 princ
scores = pca_six.fit_transform(X_four)  # fit PCA to the digit-3 data (computes

pc1 = scores[:, 0]
pc2 = scores[:, 1]
```

In [78]:
```
s1q = np.quantile(pc1, [.05, .25, .5, .75, .95])  # compute 5/25/50/75/95% quant
s2q = np.quantile(pc2, [.05, .25, .5, .75, .95])  # compute 5/25/50/75/95% quant

idx = np.zeros([len(s1q), len(s2q)])  # allocate array to store indices of close

for i in range(len(s1q)):  # loop over PC1 quantiles
    for j in range(len(s2q)):  # loop over PC2 quantiles
        aux = ((scores[:, 0] - s1q[i]) ** 2 + (scores[:, 1] - s2q[j]) ** 2).resh
        idx[i, j] = np.where(aux == min(aux))[0][0]  # index of the point with m

idx = idx.astype(int)  # cast indices to integers for array indexing
```

In [79]:
```
fig, ax = plt.subplots(len(s1q), len(s2q), figsize=(6, 6))  # create a grid of s
for i in range(len(s1q)):  # iterate over PC1 quantile levels (columns)
    for j in range(len(s2q)):  # iterate over PC2 quantile levels (rows)
        ax[len(s2q) - 1 - j, i].imshow(X_four[idx[i, j], :].reshape((28, 28)), c
plt.setp(ax, xticks=[], yticks=[])  # remove axis ticks for all subplots
fig.tight_layout()  # reduce overlap and improve spacing
```



PC1 seems to be capturing the style of the number.

PC2 looks like it is capturing the thickness of the number.

For numbers that can be written in different ways such as 4, the approximations seem to be worse than the other digits that are written in more consisten ways.

## Exercise 14 (EXTRA)

Finally, consider now two digits of your choice (edit the code below if you wish to pick different digits).

In [80]:
```
# Extract data
X_twodigits = np.concatenate((digits_dict['3'], digits_dict['8']))  # stack all
N, D = X_twodigits.shape  # store dataset size: N images (rows), D pixels/featur
```

Run the following code to compute and plot the mean and some of the principle components for this dataset.

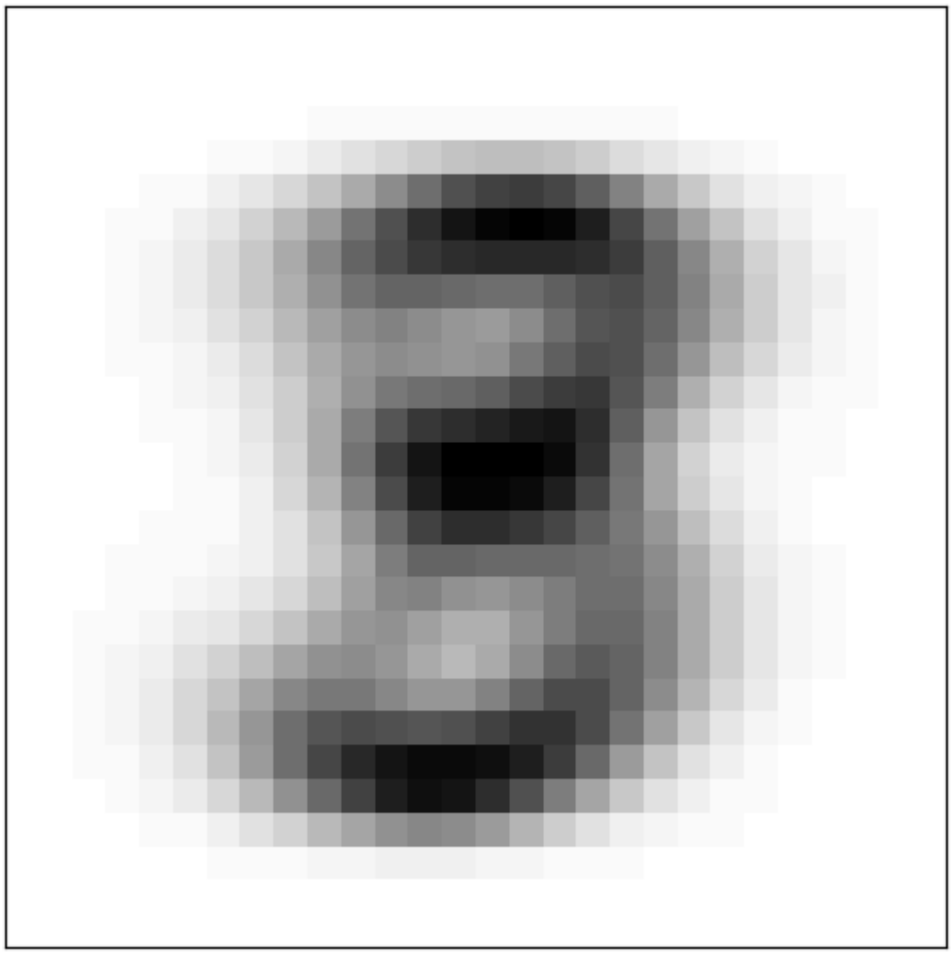
In [81]:
```
# Fit PCA
pca_digits = PCA(n_components=50)  # create PCA transformer keeping the first 50
pca_digits.fit(X_twodigits)  # fit PCA on the two-digit dataset (learns mean and
```

Out[81]:

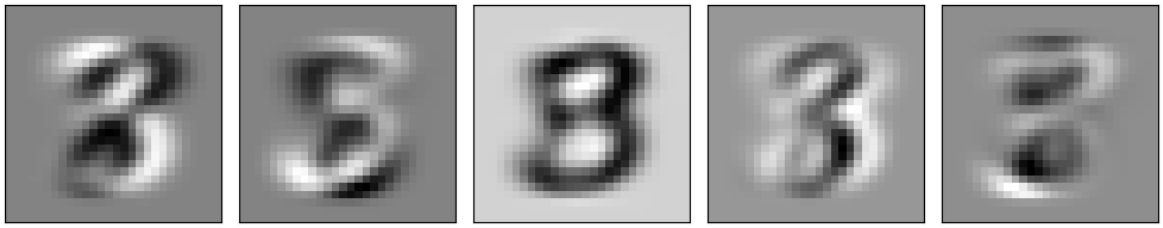
▾ PCA ⓘ ?

▸ Parameters

In [82]:
```
# Plot the mean image
fig = plt.figure(figsize=(5, 5))  # create a square figure for the mean digit im
ax = fig.add_subplot(111)  # add a single subplot occupying the whole figure
ax.imshow(pca_digits.mean_.reshape(28, 28), cmap='gray_r')  # reshape mean vecto
ax.set_xticks([])  # remove x-axis ticks
ax.set_yticks([])  # remove y-axis ticks
fig.tight_layout()  # tidy spacing
```



In [83]:
```
# Plot basis vectors
n_plot = 5  # number of principal components to visualise
fig, ax = plt.subplots(1, 5, figsize=(10, 4))  # create a row of 5 subplots for
for n in range(n_plot):  # loop over the first `n_plot` principal components
  ax[n].imshow(pca_digits.components_[n, :].reshape((28, 28)), cmap='gray_r')  #
plt.setp(ax, xticks=[], yticks=[])  # remove ticks on all subplots
fig.tight_layout()  # adjust spacing
```



Plot the projection of the data in the latent space and color the data by the labels. What do you observe?

In [ ]:
```
# Code for your answer here!
```

*Type your answer here!*

Try also to generate artificial images and decsribe how images change along the PCs.

In [ ]:
```
# Code for your answer here!
```

*Type your answer here!*

# Kernel PCA

Now, let's try using kernel PCA, which is available through sklearn's `KernelPCA` transformer. As usual we start by creating our object and specifying parameters (see documentation to learn more about the optional parameters). Then, we use the methods `.fit()` and `.transform()` to fit the object and obtain the lower-dimensional representation.

In the code below, we use the radial basis function kernel, with the inverse bandwith parameter `gamma` set to 0.05. Setting, the option `fit_inverse_transform=True` will allow us to reconstruct the images later (and `alpha` is regularization used when inversing the transforming).

*Note:* we first subsampled the data, as kernel PCA can be slow on large datasets.

In [ ]:
```
from sklearn.decomposition import KernelPCA  # kernel PCA transformer for nonlin
from sklearn.model_selection import train_test_split  # utility to split arrays
from sklearn.preprocessing import MinMaxScaler  # scaler to map features into a

# Prepare data
y_twodigits = y_twodigits.astype(int)  # ensure labels are integer-coded (useful
X_twodigits = MinMaxScaler().fit_transform(X_twodigits)  # scale pixel values to

# Subsample the images (for speed)
X_twodigits_subsampled, X_twodigits_test, y_twodigits_subsampled, y_twodigits_te
    X_twodigits,  # full feature matrix for the two digits
    y_twodigits,  # corresponding digit labels
    stratify=y_twodigits,  # preserve class proportions in both splits
    random_state=0,  # seed for reproducibility
    train_size=500,  # number of samples to keep for fitting (subsample)
    test_size=100  # number of samples to reserve for evaluation/denoising
)  # end train/test split call

# Define our KPCA and PCA transformers
n_components = 10  # number of latent dimensions/components to keep
kpca = KernelPCA(  # construct the kernel PCA model
    n_components=n_components,  # dimensionality of the KPCA embedding
    kernel="rbf",  # radial basis function (Gaussian) kernel
    gamma=0.05,  # inverse bandwidth for the RBF kernel (controls nonlinearity)
    fit_inverse_transform=True,  # enable approximate inverse mapping for recons
    random_state=0,  # seed (used in some solver paths)
    alpha=0.01  # regularisation for the inverse transform
)  # end KPCA model definition

pca = PCA(n_components=n_components)  # define standard (linear) PCA baseline wi

# Fit and transform the data
scores_kpca = kpca.fit_transform(X_twodigits_subsampled)  # fit KPCA on the subs
scores_pca = pca.fit_transform(X_twodigits_subsampled)  # fit PCA on the subsamp
```

Next, let's plot the images in the space of the first two components for both kernel PCA and standard PCA.

In [ ]:
```
# Plot the images in the space of the first two components, colored by digit
i, j = 0, 1  # component indices to plot (0-based: first vs second component)
yu = [3, 8]  # the digit classes to plot
fig, ax = plt.subplots(1, 2, figsize=(10, 5))  # create a 1x2 panel: KPCA scatte
for dig in yu:  # loop over the digit classes so each class gets its own colour/
    ax[0].scatter(  # scatter plot for kernel PCA scores
        scores_kpca[y_twodigits_subsampled == dig, i],  # x-coordinates: compone
        scores_kpca[y_twodigits_subsampled == dig, j],  # y-coordinates: compone
        c=colors[dig],  # point colour for this digit class
        label=dig  # legend label for this digit class
    )  # end KPCA scatter
    ax[1].scatter(  # scatter plot for standard PCA scores
        scores_pca[y_twodigits_subsampled == dig, i],  # x-coordinates: componen
        scores_pca[y_twodigits_subsampled == dig, j],  # y-coordinates: componen
        c=colors[dig],  # point colour for this digit class
        label=dig  # legend label for this digit class
    )  # end PCA scatter
ax[0].legend()  # show legend for the kernel PCA panel
ax[0].set_xlabel('PCA%d' % (i + 1))  # label x-axis with component number (1-bas
ax[0].set_ylabel('PCA%d' % (j + 1))  # label y-axis with component number (1-bas
ax[0].set_title('Kernel PCA')  # title for the kernel PCA subplot
ax[1].legend()  # show legend for the standard PCA panel
ax[1].set_xlabel('PCA%d' % (i + 1))  # label x-axis for the PCA subplot
ax[1].set_ylabel('PCA%d' % (j + 1))  # label y-axis for the PCA subplot
ax[1].set_title('Standard PCA')  # title for the standard PCA subplot
plt.show()  # render the figure
```

## Image Denoising

Let's add some noise to our test images that weren't used in the fitting. We will then encode the noisy images into the latent space and then reconstruct our images, to see how well both methods are able to denoise the images.

In [ ]:
```
# Add noise to the test images
np.random.seed(0)  # set RNG seed so the added noise is reproducible
noise = np.random.normal(0, 0.1, X_twodigits_test.shape)  # sample Gaussian nois
X_twodigits_test_noisy = X_twodigits_test + noise  # create noisy test images by

# Plot some noisy test images
n_images = 5  # number of test images to visualise
fig, ax = plt.subplots(2, n_images, figsize=(2 * n_images, 4))  # create a 2-row
for j in range(n_images):  # loop over a few example images
    ax[0, j].imshow(X_twodigits_test[j].reshape((28, 28)), cmap='gray_r')  # plo
    ax[1, j].imshow(X_twodigits_test_noisy[j].reshape((28, 28)), cmap='gray_r')
plt.setp(ax, xticks=[], yticks=[])  # remove ticks on all axes for cleaner displ
# Add titles
ax[0, 2].set_title('Original Images')  # title for the original row (placed on t
ax[1, 2].set_title('Noisy Images')  # title for the noisy row (placed on the mid
fig.tight_layout()  # adjust spacing to avoid overlaps
```

In [ ]:
```
# Now transform the noisy test images using both PCA and KernelPCA
scores_kpca_test = kpca.transform(X_twodigits_test_noisy)  # encode noisy test i
scores_pca_test = pca.transform(X_twodigits_test_noisy)  # encode noisy test ima

# And reconstruct the noisy test images using both PCA and KernelPCA
X_reconstructed_kpca = kpca.inverse_transform(  # decode KPCA scores back to pix
    scores_kpca_test  # the KPCA scores for noisy test images
```

```
)  # end KPCA reconstruction
X_reconstructed_pca = pca.inverse_transform(  # decode PCA scores back to pixel
    scores_pca_test  # the PCA scores for noisy test images
)  # end PCA reconstruction

# Plot some reconstructed images
n_images = 5  # number of images to show in each row
fig, ax = plt.subplots(4, n_images, figsize=(2 * n_images, 8))  # create a 4-row
for j in range(n_images):  # loop over the selected test images
    ax[0, j].imshow(X_twodigits_test[j].reshape((28, 28)), cmap='gray_r')  # plo
    ax[1, j].imshow(X_twodigits_test_noisy[j].reshape((28, 28)), cmap='gray_r')
    ax[2, j].imshow(X_reconstructed_kpca[j].reshape((28, 28)), cmap='gray_r')  #
    ax[3, j].imshow(X_reconstructed_pca[j].reshape((28, 28)), cmap='gray_r')  #
plt.setp(ax, xticks=[], yticks=[])  # remove ticks on all subplots
# Add titles
ax[0, 2].set_title('Original Images')  # title for the original row (placed in t
ax[1, 2].set_title('Noisy Images')  # title for the noisy row
ax[2, 2].set_title('Reconstructed Images (Kernel PCA)')  # title for the KPCA re
ax[3, 2].set_title('Reconstructed Images (Standard PCA)')  # title for the PCA r
fig.tight_layout()  # adjust subplot spacing
```

## Exercise 15 (EXTRA)

a) Try changing the `gamma`. What happens when you increase, e.g. `gamma=0.1`? Or decrease `gamma=0.01`?

b) Try changing the number of components. How does this affect the reconstructed images for both PCA and kernel PCA?

c) Which method would you prefer for this dataset?

*Type your answer here!*

# Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please **change 'Student 1' and 'Student 2' at the top of the notebook to include your name(s)**.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF.

In [ ]:
```
# !jupyter nbconvert --to pdf mlp_week02_annotated.ipynb

!jupyter nbconvert mlp_week02_annotated.ipynb --to html
```