tp-nn-cohen-stephen

June 11, 2024

1 TP: MNIST with Neural Networks (NN)

1.1 Stephen Cohen

[23]:
```
import numpy as np
import tensorflow as tf
import keras
print("Using tensorflow version " + str(tf.__version__))
print("Using keras version " + str(keras.__version__))
```

```
Using tensorflow version 2.16.1
Using keras version 3.3.3
```

1.2 Loading and preparing the MNIST dataset

Load the MNIST dataset made available by keras.datasets. Check the size of the training and testing sets.

Code:

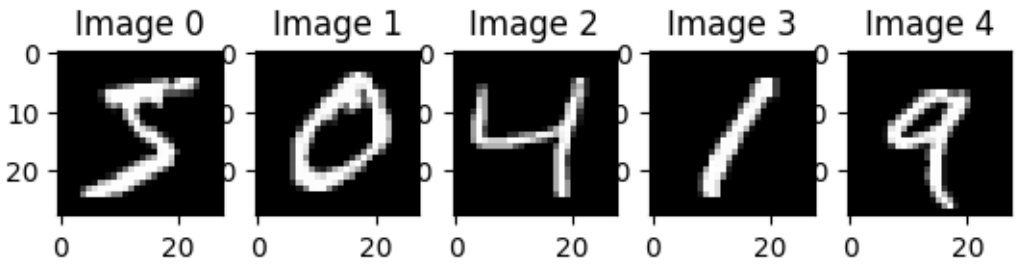
[24]:
```
# The MNSIT dataset is ready to be imported from Keras into RAM
# Warning: you cannot do that for larger databases (e.g., ImageNet)
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

The MNIST database contains 60,000 training images and 10,000 testing images. Using the pyplot package, visualize the first sample of the training set:

Code:

[25]:
```
# Let us visualize the first training sample using the Matplotlib library with␣
 ↪the imshow function
from matplotlib import pyplot as plt

# Create a figure and set of subplots
fig, axes = plt.subplots(1, 5)

# Plot each image on its respective subplot
for i in range(5):
    axes[i].imshow(train_images[i], cmap='gray')
```

```
    axes[i].set_title('Image '+str(i))


# Display the plot
plt.show()
```



The database contains images of handwritten digits. Hence, they belong to one of 10 categories, depending on the digit they represent. Reminder: in order to do multi-class classification, we use the softmax function, which outputs a multinomial probability distribution. That means that the output to our model will be a vector of size 10, containing probabilities (meaning that the elements of the vector will be positive sum to 1). For easy computation, we want to true labels to be represented with the same format: that is what we call **one-hot encoding**. For example, if an image $\mathbf{x}$ represents the digit 5, we have the corresponding one_hot label (careful, 0 will be the first digit):

$$\mathbf{y} = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$$

Here, you need to turn train and test labels to one-hot encoding using the following function:

Code:

[26]:
```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels, num_classes=10)
test_labels= to_categorical(test_labels, num_classes=10)
```

Images are black and white, with size 28×28. We will work with them using a simple linear classification model, meaning that we will have them as vectors of size (784). You should then transform the images to the size (784) using the numpy function `reshape`.

Then, after casting the pixels to floats, normalize the images so that they have zero-mean and unitary deviation. Be careful to your methodology: while you have access to training data, you may not have access to testing data, and must avoid using any statistic on the testing dataset.

Code:

[27]:
```
# Reshape images to vectors of pixels
img_rows, img_cols = train_images.shape[1], train_images.shape[2]
train_images = train_images.reshape((len(train_images),784))
```

```
imgt_rows, imgt_cols = test_images.shape[1], test_images.shape[2]
test_images = test_images.reshape((len(test_images),784))


# Cast pixels from uint8 to float32
train_images = train_images.astype('float32')
# Cast pixels from uint8 to float32
test_images = test_images.astype('float32')

# Now let us normalize the images so that they have zero mean and standard␣
 ↪deviation
# Hint: are real testing data statistics known at training time ?

mean = np.mean(train_images)
std = np.std(train_images)
train_images = (train_images - mean) / std
test_images = (test_images - mean) / std
```

2 First part: working with Numpy

Look at this cheatsheet for some basic information on how to use numpy.

2.0.1 Defining the model

We will here create a simple, linear classification model. We will take each pixel in the image as an input feature (making the size of the input to be 784) and transform these features with a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$. Since there is 10 possible classes, we want to obtain 10 scores. Then,

$$\mathbf{W} \in \mathbb{R}^{784\times 10}$$

$$\mathbf{b} \in \mathbb{R}^{10}$$

and our scores are obtained with:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$$

where $\mathbf{x} \in \mathbb{R}^{784}$ is the input vector representing an image. We note $\mathbf{y} \in \mathbb{R}^{10}$ as the target one_hot vector.

Here, you fist need to initialize $\mathbf{W}$ and $\mathbf{b}$ using `np.random.normal` and `np.zeros`, then compute $\mathbf{z}$.

Code:

[28]:
```
# To avoid implementing a complicated gradient back-propagation,
# we will try a very simple architecture with one layer
def initLayer(n_input,n_output):
    """
    Initialize the weights, return the number of parameters
    Inputs: n_input: the number of input units - int
```

```
        : n_output: the number of output units - int
    Outputs: W: a matrix of weights for the layer - numpy ndarray
           : b: a vector bias for the layer - numpy ndarray
           : nb_params: the number of parameters  - int
    """
    # Create W at the right size with a normal distribution
    W = np.random.normal(size=(n_input,n_output))
    # Create b at the right size, with zeros
    b = np.zeros(n_output)
    nb_params = n_input
    return W, b, nb_params
```

[29]:
```
n_training = train_images.shape[0]
n_feature = img_rows*img_cols
n_labels = 10
W, b, nb_params = initLayer(n_feature, n_labels)

print(W,b,nb_params)
```

```
[[-0.49227489  0.80169441 -0.90627028 … -0.44849144  1.2215934
   2.81360742]
 [ 0.87246766 -0.84341206 -0.20340011 …  0.32724713 -1.17339643
   0.68078533]
 [ 1.77070295 -0.56468523 -0.73206008 … -1.29304491 -0.59211111
  -1.5425716 ]
 …
 [-0.65675724 -0.66153526 -0.10087695 … -1.21702192 -1.50204698
  -0.47267804]
 [-2.43240665 -0.65215219  0.38965722 … -0.32427541  1.21704887
  -0.51846064]
 [-0.32642781  0.64752945 -0.24166367 …  0.95817025 -0.86280059
   0.44323615]] [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] 784
```

Code:

[30]:
```
def forward(W, b, X):
    """
    Perform the forward propagation
    Inputs: W: the weights - numpy ndarray
          : b: the bias - numpy ndarray
          : X: the batch - numpy ndarray
    Outputs: z: outputs - numpy ndarray
    """

    z = X@W+b
    return z
```

2.0.2 Computing the output

To obtain classification probabilities, we use the softmax function:

$$\mathbf{o} = softmax(\mathbf{z}) \text{ with } o_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

The usual difficulty with the softmax function is the possibility of overflow when the scores z_i are already large. Since a softmax is not affected by a shift affecting the whole vector $\mathbf{z}$:

$$\frac{\exp(z_i - c)}{\sum_{j=0}^{9} \exp(z_j - c)} = \frac{\exp(c)\exp(z_i)}{\exp(c)\sum_{j=0}^{9} \exp(z_j)} = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

what trick can we use to ensure we will not encounter any overflow ?

Code:

[31]:
```
def softmax(z):
    """
    Perform the softmax transformation to the pre-activation values
    Inputs: z: the pre-activation values - numpy ndarray
    Outputs: out: the activation values - numpy ndarray
    """
    c = np.max(z)
    out = np.exp([zi - c for zi in z])/np.sum(np.exp([zi-c for zi in z]))
    return out
```

2.0.3 Making updates

We define a learning rate η. The goal is to be able to apply updates:

$$\mathbf{W}^{t+1} = \mathbf{W}^t + \nabla_{\mathbf{W}} l_{MLE}$$

In order to do this, we will compute this gradient (and the bias) in the function `update`. In the next function `updateParams`, we will actually apply the update with regularization.

Reminder: the gradient $\nabla_{\mathbf{W}} l_{MLE}$ is the matrix containing the partial derivatives

$$\left[\frac{\delta l_{MLE}}{\delta W_{ij}}\right]_{i=1..784, j=1..10}$$

Remark: Careful, the usual way of implementing this in python has the dimensions of $\mathbf{W}$ reversed compared to the notation of the slides.

Coordinate by coordinate, we obtain the following update:

$$W_{ij}^{t+1} = W_{ij}^t + \eta \frac{\delta l_{MLE}}{\delta W_{ij}}$$

Via the chain rule, we obtain, for an input feature $i \in [0, 783]$ and a output class $j \in [0, 9]$:

$$\frac{\delta l_{MLE}}{\delta W_{ij}} = \frac{\delta l_{MLE}}{\delta z_j} \frac{\delta z_j}{\delta W_{ij}}$$

It's easy to compute that $\frac{\delta z_j}{\delta W_{ij}} = x_i$

We compute the softmax derivative, to obtain:

$$\nabla_{\mathbf{z}} l_{MLE} = \mathbf{o} - \mathbf{y}$$

Hence, $\frac{\delta l_{MLE}}{\delta z_j} = o_j - y_j$ and we obtain that

$$\frac{\delta l_{MLE}}{\delta W_{ij}} = (o_j - y_j)x_i$$

This can easily be written as a scalar product, and a similar computation (even easier, actually) can be done for $\mathbf{b}$. Noting $\nabla_{\mathbf{z}} l_{MLE} = \mathbf{o} - \mathbf{y}$ as `grad` in the following function, compute the gradients $\nabla_{\mathbf{W}} l_{MLE}$ and $\nabla_{\mathbf{b}} l_{MLE}$ in order to call the function `updateParams`.

Note: the regularizer and the weight_decay λ are used in `updateParams`.

Code:

[32]:
```
def update(eta, W, b, grad, X, regularizer, weight_decay):
    """
    Perform the update of the parameters
    Inputs: eta: the step-size of the gradient descent - float
          : W: the weights - ndarray
          : b: the bias -  ndarray
          : grad: the gradient of the activations w.r.t. to the loss -  list of␣
 ↪ndarray
          : X: the data -  ndarray
          : regularizer: 'L2' or None - the regularizer to be used in␣
 ↪updateParams
          : weight_decay: the weight decay to be used in updateParams - float
    Outputs: W: the weights updated -  ndarray
           : b: the bias updated -  ndarray
    """
    grad_w = np.array([np.array([grad[i]*X[j] for i in range(len(grad))]) for j␣
 ↪in range(len(X))])
    grad_b = grad

    W = updateParams(W, grad_w, eta, regularizer, weight_decay)
    b = updateParams(b, grad_b, eta, regularizer, weight_decay)
    return W, b
```

The update rule is affected by regularization. We implement two cases: No regularization, or L2 regularization. Use the two possible update rules to implement the following function:

Code:

[33]:
```
def updateParams(param, grad_param, eta, regularizer=None, weight_decay=0.):
    """
    Perform the update of the parameters
```

```
    Inputs: param: the network parameters - ndarray
          : grad_param: the updates of the parameters - ndarray
          : eta: the step-size of the gradient descent - float
          : weight_decay: the weight-decay - float
    Outputs: the parameters updated - ndarray
    """
    if regularizer==None:
        grad = param-eta*grad_param
        return grad
    elif regularizer=='L2':
        grad = param-eta*grad_param+weight_decay*param
        return grad
    else:
        raise NotImplementedError
```

2.0.4 Computing the Accuracy

Here, we simply use the model to predict the class (by taking the argmax of the output !) for every example in `X`, and count the number of times the model is right, to output the accuracy.

Code:

[34]:
```
def computeAcc(W, b, X, labels):
    """
    Compute the loss value of the current network on the full batch
    Inputs: act_func: the activation function - function
          : W: the weights - list of ndarray
          : B: the bias - list of ndarray
          : X: the batch - ndarray
          : labels: the labels corresponding to the batch
    Outputs: loss: the negative log-likelihood - float
           : accuracy: the ratio of examples that are well-classified - float
    """
      # Forward propagation
    z = np.array([forward(W, b, x) for x in X])
    # Compute the softmax and the prediction
    out = np.array([softmax(zl) for zl in z])
    pred = np.array([np.argmax(zl) for zl in z])
    pred= to_categorical(pred, num_classes=10)
    '''
    for j in range(10):
      print(pred[j],labels[j],(labels[j]==pred[j]).all())
      '''
    # Compute the accuracy
    accuracy = np.sum([1 if (labels[i]==pred[i]).all() else 0 for i in␣
 ↪range(len(labels))])

    return accuracy
```

```
#train_accuracy = computeAcc(W, b, train_images, train_labels)
#print(train_accuracy)
```

2.0.5 Preparing training

The following hyperparameters are given. Next, we can assemble all the function previously defined to implement a training loop. We will train the classifier on **one epoch**, meaning that the model will see each training example once.

[35]:
```
# Optimization
eta = 0.01
regularizer = 'L2'
weight_decay = 0.0001

# Training
log_interval = 5000
```

Code:

[36]:
```
# Data structures for plotting
g_train_acc=[]
g_valid_acc=[]

########################
### Learning process ##
########################
#Took approximately 3 min to execute
for j in range(n_training):
    # Getting the example
    X, y = train_images[j],train_labels[j]

    # Forward propagation
    z = forward(W, b, X)

    # Compute the softmax
    out = softmax(z)

    # Compute the gradient at the top layer
    derror = out - y # This is o - y
    # Update the parameters
    W, b = update(eta,W,b,derror,X,regularizer,weight_decay)
    if j % log_interval == 0:
        # Every log_interval examples, look at the training accuracy
        train_accuracy = computeAcc(W, b, train_images, train_labels)

        # And the testing accuracy
        test_accuracy = computeAcc(W, b, test_images, test_labels)
```

```
        g_train_acc.append(train_accuracy)
        g_valid_acc.append(test_accuracy)
        result_line = str(int(j)) + " " + str(train_accuracy) + " " +␣
 ↪str(test_accuracy) + " " + str(eta)
        print(result_line)

g_train_acc.append(train_accuracy)
g_valid_acc.append(test_accuracy)
result_line = "Final result:" + " " + str(train_accuracy) + " " +␣
 ↪str(test_accuracy) + " " + str(eta)
print(result_line)
```

```
0 5244 866 0.01
5000 48205 8088 0.01
10000 49909 8359 0.01
15000 50687 8495 0.01
20000 51027 8549 0.01
25000 51187 8576 0.01
30000 51304 8605 0.01
35000 51373 8614 0.01
40000 51428 8602 0.01
45000 51476 8605 0.01
50000 51475 8603 0.01
55000 51487 8612 0.01
Final result: 51487 8612 0.01
```

What can you say about the performance of this simple linear classifier ?

Answer:

On dirait que l'erreur est au minimum de 10% donc une accuracy de maximum 90% (On ne semble pas pouvoir dépasser ce seuil) De plus, le temps de calcul est très long (environ 3mn et 45s)

3 Second part: Autoencoder with Keras

3.1 Autoencoder and PCA

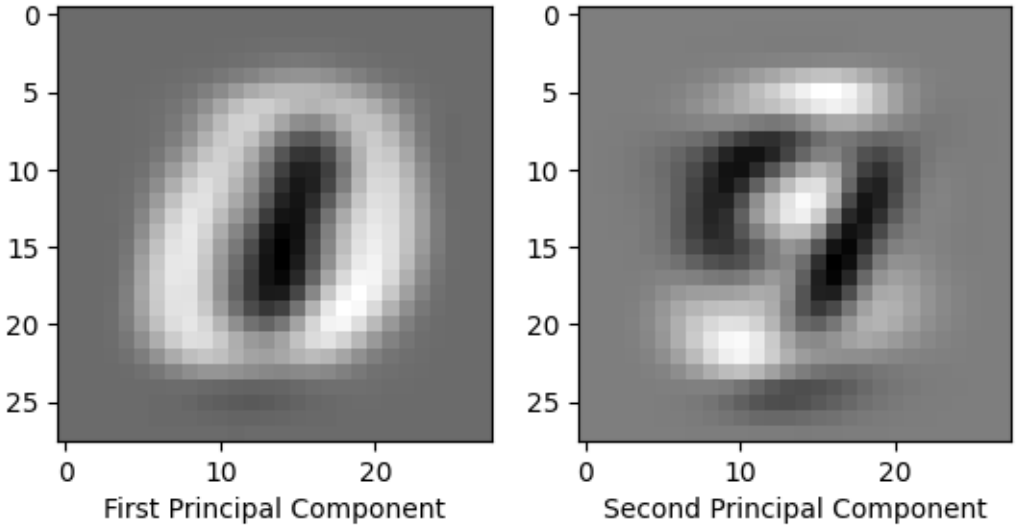
First, we will try to connect the representation produced by Principal Component Analysis with what is learnt by a simple, linear, autoencoder. We will use the `scikit-learn` implementation of the `PCA` to obtain the two first components (hint: use the attribute `.components_`), and visualize them:

Code:

[37]:
```
from sklearn.decomposition import PCA

# Let's find the first 2 PCA components
num_components = 2
pca = PCA(num_components).fit(train_images)
```

```
# Reshape so they resemble images and we can print them
eigen_mnist = pca.components_.reshape(2,28,28)

# Show the reshaped principal components
f, ax = plt.subplots(1,2)
ax[0].imshow(eigen_mnist[0], cmap='gray')
ax[0].set_xlabel('First Principal Component')
ax[1].imshow(eigen_mnist[1], cmap='gray')
ax[1].set_xlabel('Second Principal Component')
```

[37]: `Text(0.5, 0, 'Second Principal Component')`



[38]:
```
# Print the variance explained by those components
print(pca.explained_variance_)
print(pca.explained_variance_ratio_)
```

```
[53.903347 39.41201 ]
[0.0970372  0.07094979]
```

Comment on the visualization in relation to the variance explained by only keeping the two principal components:

Answer:

On remarque que chacun des deux chiffres semblent expliquer 10% du data set global (au total 20% ici). Ce qui pourrait s'expliquer par le fait que chacun des 10 chiffres explique 10% des données (ce qui parait intuitif)

3.1.1 Implementing the Autoencoder with Keras

[39]:
```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import SGD
```

Now, we will use Keras to implement the autoencoder. You can take a look at this cheatsheet for some basic commands to use keras.

In this first case, we implement a **simple linear autoencoder**. Build it in order to have the same capacity as the PCA decomposition (2 hidden dimensions !) we made just above.

Code:

[40]:
```
# Input layer
input_layer = Input(shape=(784,))

# Encoding layer
latent_view = Dense(2, activation='linear', name='layer')(input_layer)

# Decoding layer
output_layer = Dense(784, activation='linear',name='output')(latent_view)

ae_model = Model(input_layer, output_layer, name='ae_model')
ae_model.summary()
```

Model: "ae_model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
layer (Dense)	(None, 2)	1,570
output (Dense)	(None, 784)	2,352

Total params: 3,922 (15.32 KB)

Trainable params: 3,922 (15.32 KB)

Non-trainable params: 0 (0.00 B)

What loss shoud we use ? Choose the usual one and import it directly from Keras. You can use a simple `SGD` optimizer, and then compile the model; finally, train it to rebuild images from the original examples.

Code:

[41]:
```
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.optimizers import SGD

loss = MeanSquaredError()

# Utilisez 'learning_rate' au lieu de 'lr'
optimizer = SGD(learning_rate=1e-1)

ae_model.compile(optimizer=optimizer, loss=loss)

batch_size = 128
epochs = 10

history = ae_model.fit(train_images,
                       train_images,
                       epochs=epochs,
                       batch_size=batch_size,
                       verbose=1,
                       shuffle=True,
                       validation_data=(test_images, test_images))
```

```
Epoch 1/10
469/469             1s 2ms/step -
loss: 0.8875 - val_loss: 0.6742
Epoch 2/10
469/469             1s 2ms/step -
loss: 0.6621 - val_loss: 0.6221
Epoch 3/10
469/469             1s 2ms/step -
loss: 0.6228 - val_loss: 0.6173
Epoch 4/10
469/469             1s 2ms/step -
loss: 0.6194 - val_loss: 0.6164
Epoch 5/10
469/469             1s 2ms/step -
loss: 0.6172 - val_loss: 0.6160
Epoch 6/10
469/469             1s 2ms/step -
loss: 0.6163 - val_loss: 0.6158
Epoch 7/10
469/469             1s 2ms/step -
loss: 0.6167 - val_loss: 0.6155
Epoch 8/10
469/469             1s 2ms/step -
loss: 0.6164 - val_loss: 0.6152
Epoch 9/10
```

```
469/469             1s 2ms/step -
loss: 0.6159 - val_loss: 0.6152
Epoch 10/10
469/469             1s 2ms/step -
loss: 0.6161 - val_loss: 0.6150
```

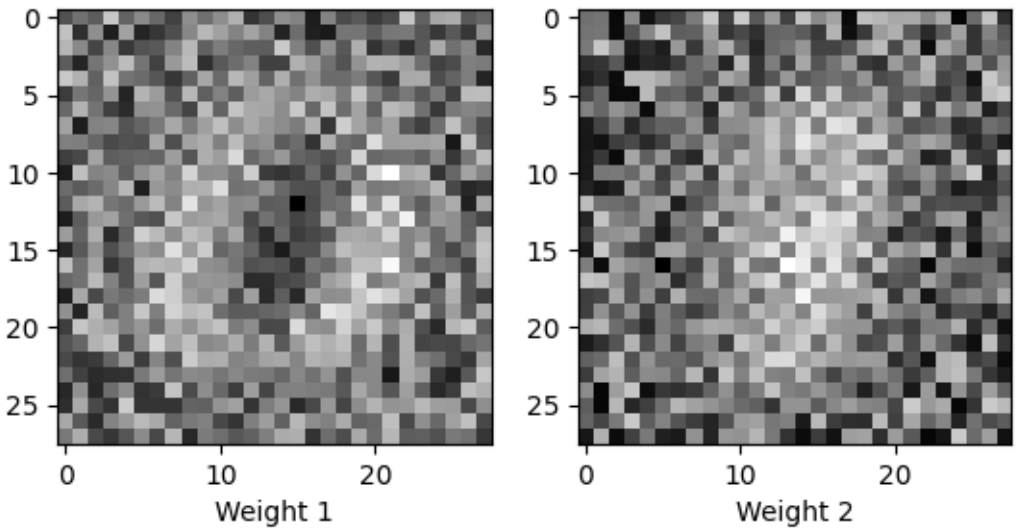
Assuming that the name of your layer (obtained through the command `model.summary()`) is `'layer'`, here is the way to obtained the weights. Visualize the weights of the encoder and compare them to the two components obtained through the PCA.

[42]:
```
weights, bias = ae_model.get_layer('layer').get_weights()

weights = [[x[0] for x in weights],[x[1] for x in weights]]
```

Code:

[43]:
```
# Show the two dimensions of the encoder, in a similar manner to the principal␣
 ↪components
# (after reshaping them as images !)

f, ax = plt.subplots(1,2)
ax[0].imshow(np.array(weights[0]).reshape(28,28), cmap='gray')
ax[0].set_xlabel('Weight 1')
ax[1].imshow(np.array(weights[1]).reshape(28,28), cmap='gray')
ax[1].set_xlabel('Weight 2')
```

[43]:
```
Text(0.5, 0, 'Weight 2')
```

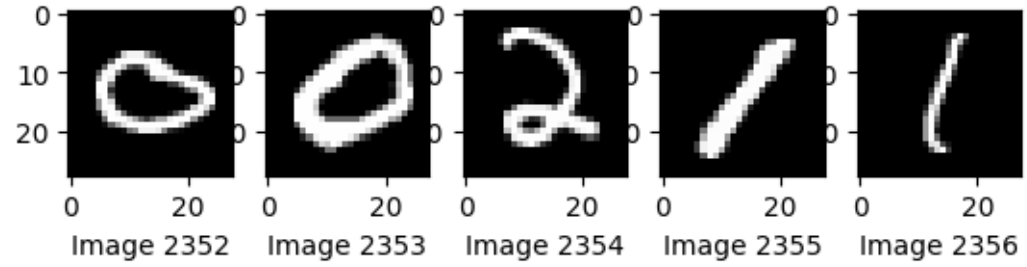


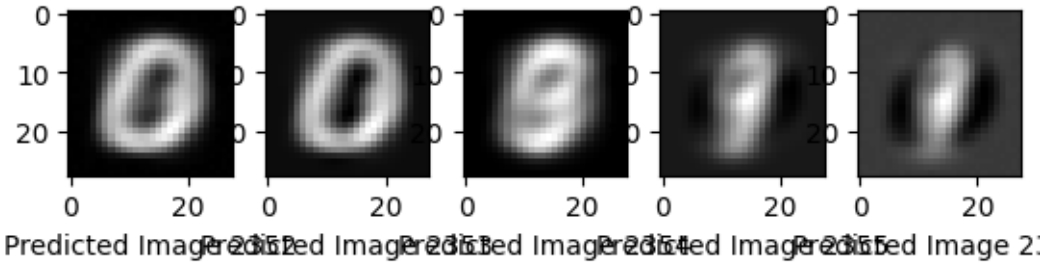
Now, visualize the images rebuilt by the network !

Code:

[44]:
```
# Select a few images at random: look from n
n = np.random.randint(0,len(test_images)-5)

# Plot a few images from n
f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(train_images[a].reshape((28,28)), cmap='gray')
    ax[i].set_xlabel('Image '+str(a))

# Get the prediction from the model

# ... and plot them
f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow((ae_model.predict(train_images[a].reshape(1,784)).
 ↪reshape((28,28))), cmap='gray')
    ax[i].set_xlabel('Predicted Image '+str(a))

plt.show()
```

```
1/1             0s 32ms/step
1/1             0s 16ms/step
1/1             0s 32ms/step
1/1             0s 16ms/step
1/1             0s 16ms/step
```





Do the same (= build a new model) with a latent dimension that is largely higher than 2. Compare the visualizations and the images that are rebuilt.

Code:

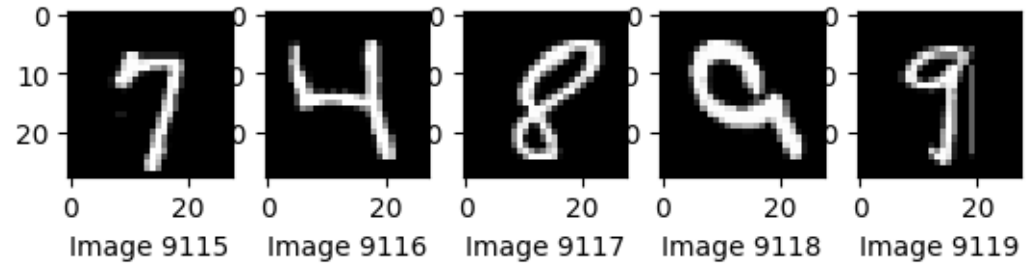
[45]:
```
input_layer = Input(shape=(784,)) #Input

latent_view = Dense(64, activation='linear')(input_layer) #Encoder

output_layer = Dense(784, activation='linear')(latent_view) #Decoder

ae_model = Model(input_layer, output_layer, name='ae_model')

ae_model.summary() #Affichage

optimizer = SGD(learning_rate=1e-1)
ae_model.compile(optimizer='adam', loss='mean_squared_error')
batch_size = 128
epochs = 10
# No noise here - we want to train a simple auto-encoder and compare visually␣
 ↪with PCA
history = ae_model.fit(train_images,
                       train_images,
                       epochs=epochs,
                       batch_size=batch_size,
                       verbose=1,
                       shuffle=True,
                       validation_data=(test_images,test_images))

n = np.random.randint(0,len(test_images)-5)


f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(train_images[a].reshape((28,28)), cmap='gray')
    ax[i].set_xlabel('Image '+str(a))


f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow((ae_model.predict(train_images[a].reshape(1,784)).
 ↪reshape((28,28))), cmap='gray')
    ax[i].set_xlabel('Predicted Image '+str(a))

plt.show()
```

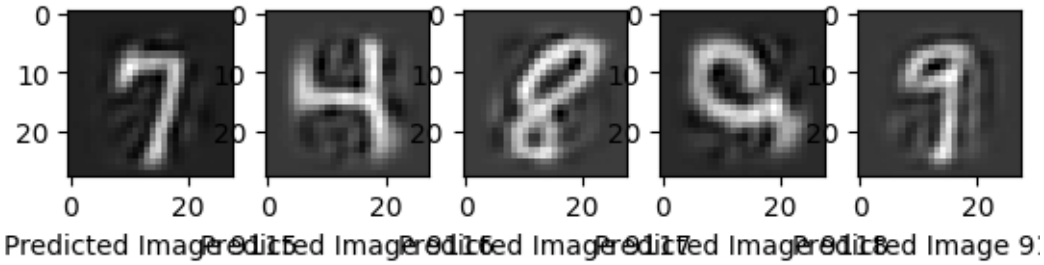
```
Model: "ae_model"
```

```
 Layer (type)                    Output Shape                  Param #

 input_layer_1 (InputLayer)      (None, 784)                         0

 dense (Dense)                   (None, 64)                     50,240

 dense_1 (Dense)                 (None, 784)                    50,960


 Total params: 101,200 (395.31 KB)

 Trainable params: 101,200 (395.31 KB)

 Non-trainable params: 0 (0.00 B)
```

```
Epoch 1/10
469/469             2s 3ms/step -
loss: 0.4344 - val_loss: 0.1194
Epoch 2/10
469/469             1s 2ms/step -
loss: 0.1147 - val_loss: 0.1014
Epoch 3/10
469/469             1s 2ms/step -
loss: 0.1034 - val_loss: 0.0989
Epoch 4/10
469/469             1s 3ms/step -
loss: 0.1013 - val_loss: 0.0982
Epoch 5/10
469/469             2s 4ms/step -
loss: 0.1007 - val_loss: 0.0979
Epoch 6/10
469/469             2s 4ms/step -
loss: 0.1006 - val_loss: 0.0975
Epoch 7/10
469/469             2s 4ms/step -
loss: 0.1001 - val_loss: 0.0971
Epoch 8/10
469/469             2s 3ms/step -
loss: 0.1002 - val_loss: 0.0968
Epoch 9/10
469/469             1s 2ms/step -
loss: 0.0996 - val_loss: 0.0968
Epoch 10/10
469/469             1s 2ms/step -
loss: 0.0995 - val_loss: 0.0968
```

```
1/1             0s 31ms/step
1/1             0s 16ms/step
1/1             0s 16ms/step
1/1             0s 16ms/step
1/1             0s 16ms/step
```





3.1.2 Bonus: De-noising Autoencoder

Now, we can implement a **de-noising autoencoder**. The following function will transform an array of images by adding it random noise. Create a new autoencoder model, this time with **more layers** and **non-linear activations** (like the ReLU) and train it to rebuild the de-noised images. Display some testing images, with noise, and re-built.

[46]:
```python
def noise(array):
    """
    Adds random noise to each image in the supplied array.
    """
    noise_factor = 0.4
    noisy_array = array + noise_factor * np.random.normal(
        loc=0.0, scale=1.0, size=array.shape
    )
    return noisy_array
```

[47]:
```python
# Create a copy of the data with added noise
noisy_train_images = noise(train_images)
```

```
noisy_test_images = noise(test_images)
```

Code:

[48]:
```
# Visualize some of the images with noise against the originals


# Build a new model with more layers and Relu activations


# Compile it but here, use noised data as inputs !


# Visualize the images rebuilt by the model !


input_layer = Input(shape=(784,))


latent_view = Dense(256, activation='relu')(input_layer)
latent_view_encore = Dense(128, activation='sigmoid')(latent_view)
latent_view_toujours = Dense(64, activation='linear')(latent_view_encore)

output_layer = Dense(784, activation='linear')(latent_view_toujours)

noise_ae_model = Model(input_layer, output_layer, name='ae_model')


noise_ae_model.summary()

optimizer = SGD(learning_rate=1e-1)
noise_ae_model.compile(optimizer='adam', loss='mean_squared_error')
batch_size = 128
epochs = 10

history = noise_ae_model.fit(noisy_train_images,
                        train_images,
                        epochs=epochs,
                        batch_size=batch_size,
                        verbose=1,
                        shuffle=True,
                        validation_data=(noisy_test_images,test_images))


n = np.random.randint(0,len(test_images)-5)
```

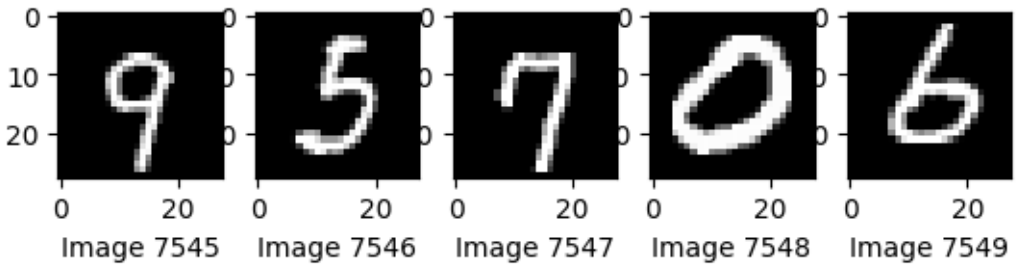
```
f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(train_images[a].reshape((28,28)), cmap='gray')
    ax[i].set_xlabel('Image '+str(a))




f, ax = plt.subplots(1,5)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow((noise_ae_model.predict(train_images[a].reshape(1,784)).
 ↪reshape((28,28))), cmap='gray')
    ax[i].set_xlabel('Predicted Image '+str(a))

plt.show()
```

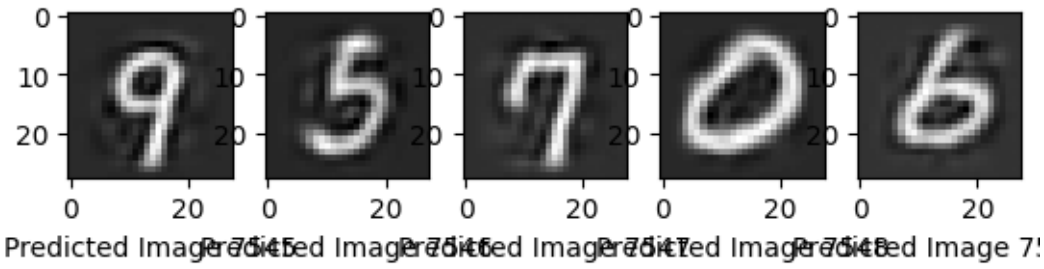
```
Model: "ae_model"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 784)	0
dense_2 (Dense)	(None, 256)	200,960
dense_3 (Dense)	(None, 128)	32,896
dense_4 (Dense)	(None, 64)	8,256
dense_5 (Dense)	(None, 784)	50,960

```
 Total params: 293,072 (1.12 MB)

 Trainable params: 293,072 (1.12 MB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/10
469/469            3s 4ms/step -
loss: 0.4868 - val_loss: 0.2005
Epoch 2/10
469/469            2s 4ms/step -
loss: 0.1884 - val_loss: 0.1561
Epoch 3/10
```

```
469/469             2s 4ms/step -
loss: 0.1528 - val_loss: 0.1377
Epoch 4/10
469/469             2s 4ms/step -
loss: 0.1374 - val_loss: 0.1291
Epoch 5/10
469/469             2s 4ms/step -
loss: 0.1288 - val_loss: 0.1213
Epoch 6/10
469/469             2s 4ms/step -
loss: 0.1232 - val_loss: 0.1180
Epoch 7/10
469/469             2s 4ms/step -
loss: 0.1193 - val_loss: 0.1161
Epoch 8/10
469/469             2s 4ms/step -
loss: 0.1182 - val_loss: 0.1162
Epoch 9/10
469/469             2s 4ms/step -
loss: 0.1170 - val_loss: 0.1144
Epoch 10/10
469/469             2s 4ms/step -
loss: 0.1157 - val_loss: 0.1133
1/1              0s 47ms/step
1/1              0s 16ms/step
1/1              0s 16ms/step
1/1              0s 16ms/step
1/1              0s 14ms/step
```





Assuming that we normalize the images to be in the 0-1 range, what other loss function could we use ?

`Answer:</div>`

On pourrait utiliser la fonction de perte binaire entropique.