

NeuralNetwork_FashionMNIST_dataset

December 21, 2023

1 Machine Learning LAB 3

2 IMAGE CLASSIFICATION WITH NEURAL NETWORKS

Course 2022/23: P. Zanuttigh, F. Barbato, M. Mel

The notebook contains some simple tasks to be performed with **NEURAL NETWORKS (NNs)**. Complete all the **required code sections** and **answer to all the questions**.

2.0.1 IMPORTANT for the evaluation score:

1. **Read carefully all cells and follow the instructions**
2. **Re-run all the code from the beginning** to obtain the results for the final version of your notebook, since this is the way we will do it before evaluating your notebooks.
3. Make sure to fill the code in the appropriate places **without modifying the template**, otherwise you risk breaking later cells.
4. Please **submit the jupyter notebook file (.ipynb)**, do not submit python scripts (.py) or plain text files. **Make sure that it runs fine with the `restart&run all` command.**
5. **Answer the questions in the appropriate cells**, not in the ones where the question is presented.

3 Clothes Classification with Neural Networks

In this notebook we are going to explore the Neural Networks for image classification. We are going to use **Fashion MNIST**, a dataset of small images of clothes and accessories.

The dataset labels are the following:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag

Label	Description
9	Ankle boot

```
[1]: # load the required packages and check Scikit-learn version
%matplotlib inline

import numpy as np
from matplotlib import pyplot as plt

import sklearn
print ('scikit-learn version: ', sklearn.__version__)
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

#from sklearnex import patch_sklearn # if you have an Intel CPU and have
#installed the Intel MKL
#patch_sklearn() # library you can uncomment these lines
#for a speedup
```

scikit-learn version: 1.0.2

```
[2]: # helper function to load Fashion MNIST dataset from disk
def load_mnist(path, kind='train'):
    import os
    import gzip
    import numpy as np
    labels_path = os.path.join(path, '%s-labels-idx1-ubyte.gz' % kind)
    images_path = os.path.join(path, '%s-images-idx3-ubyte.gz' % kind)
    with gzip.open(labels_path, 'rb') as lbpath:
        labels = np.frombuffer(lbpath.read(), dtype=np.uint8, offset=8)
    with gzip.open(images_path, 'rb') as imgpath:
        images = np.frombuffer(imgpath.read(), dtype=np.uint8, offset=16).
        reshape(len(labels), 784)
    return images, labels
```

Place your **name** and **ID number** (matricola) in the cell below. Also recall to **save the file as Surname_Name_LAB3.ipynb**

Student name: Tommaso Bergamasco **ID Number:** 2052409

4 A) Hyperparameter Search

TO DO A.0: Set the random **seed** using your **ID**. If you need to change it for testing add a constant explicitly, eg.: ID = 1234567 + 1.

```
[3]: ID = 2052409
      np.random.seed(ID)
```

Before proceeding to the training steps, we **load the dataset**. The **split** is performed after **applying a random permutation** to the dataset, such permutation will **depend on the seed** you set above. **DO NOT CHANGE THE PRE-WRITTEN CODE UNLESS OTHERWISE SPECIFIED**

```
[4]: # load the MNIST dataset
      X, y = load_mnist("data")
      print("Number of samples in the MNIST dataset:", X.shape[0])
      # rescale the data to [0, 1]
      X = X / 255.0
```

Number of samples in the MNIST dataset: 60000

TO DO A.1: Divide the data into training and test set using a **500** samples in the **training** set. Make sure that **each label** is present at **least 10 times** in training frequencies.

```
[5]: # randomly permute the data and split into training and test taking
      # the first 500 data samples as training and the rests as test
      permutation = np.random.permutation(X.shape[0]) # ADD YOUR CODE HERE

      X = X[permutation]
      y = y[permutation]

      m_training = 10000 # ADD YOUR CODE HERE

      X_train, X_test = X[:m_training], X[m_training:] # ADD YOUR CODE HERE
      y_train, y_test = y[:m_training], y[m_training:] # ADD YOUR CODE HERE

      labels, freqs = np.unique(y_train, return_counts=True) # ADD YOUR CODE HERE
      print("Labels in training dataset: ", labels)
      print("Frequencies in training dataset: ", freqs)
```

Labels in training dataset: [0 1 2 3 4 5 6 7 8 9]

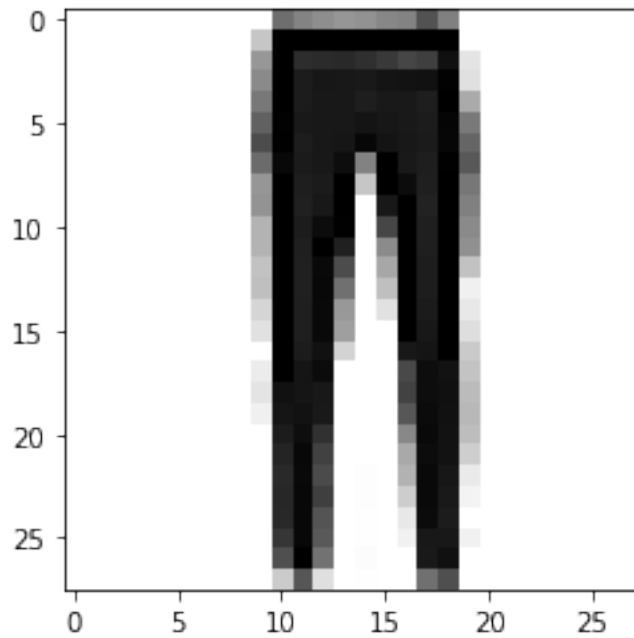
Frequencies in training dataset: [979 991 983 979 983 968 1027 995 1056 1039]

```
[6]: # function for plotting a image and printing the corresponding label
      def plot_input(X_matrix, labels, index):
          print("INPUT:")
          plt.imshow(
              X_matrix[index].reshape(28,28),
              cmap      = plt.cm.gray_r,
              interpolation = "nearest"
          )
          plt.show()
```

```
print("LABEL: %i"%labels[index])  
return
```

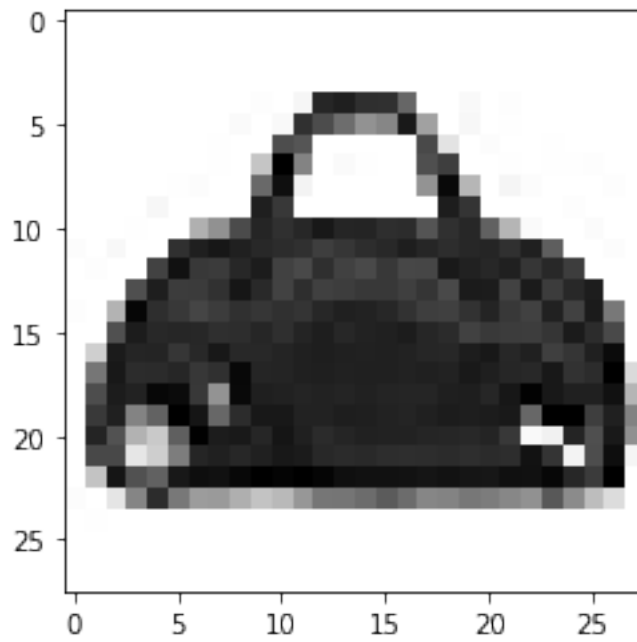
```
[7]: # let's try the plotting function  
plot_input(X_train,y_train,10)  
plot_input(X_test,y_test,100)  
plot_input(X_test,y_test,10000)
```

INPUT:

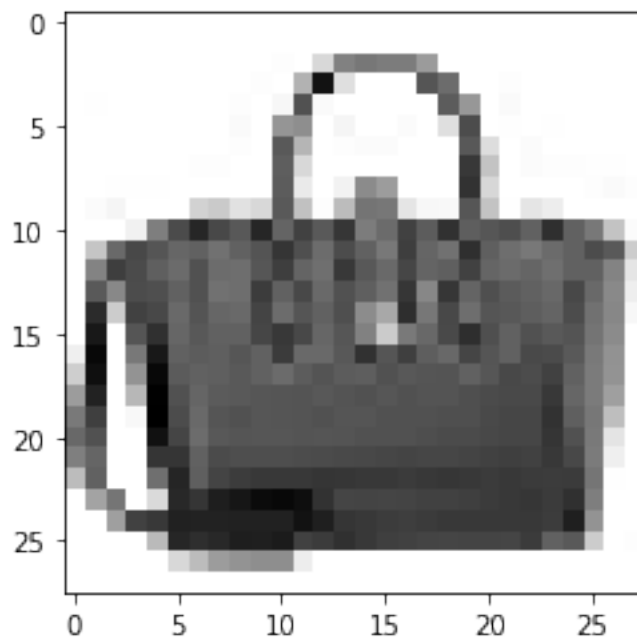


LABEL: 1

INPUT:



LABEL: 8
INPUT:



LABEL: 8

TO DO A.2 Now use a feed-forward Neural Network for prediction. Use the **multi-layer perceptron** classifier, with the following parameters: `max_iter=250`, `alpha=1e-4`, `solver='sgd'`, `tol=1e-4`, `learning_rate_init=.1`, `random_state=ID`. The `random_state` parameter ensures the run is the same even if you run it more than once, while the `alpha` parameter is the regularization term. You might get some **warnings about the convergence**, ignore them (or try to increase the `max_iter` parameter if you have a powerful computer).

Then, using the default activation function, **pick four or five architectures** to consider, with different numbers of hidden layers and different sizes. It is not necessary to create huge neural networks, you can limit to 3 layers and, for each layer, its maximum size can be of 100. Evaluate the architectures you chose using **GridSearchCV with `cv=5`**.

You can reduce the number of iterations if the running time is too long on your computer.

```
[8]: # these are sample values but feel free to change them as you like, try to
      ↪ experiment with different sizes!!
parameters = {'hidden_layer_sizes': [(10,), (25,), (50,), (25,10,), (50,25,10)]
      ↪}

mlp = MLPClassifier(max_iter=500, alpha=1e-4, solver='sgd', tol=1e-4,
      ↪learning_rate_init=.1, random_state=ID) # ADD YOUR CODE HERE

mlp_arch_CV = GridSearchCV(mlp, parameters) # ADD YOUR CODE HERE
mlp_arch_CV.fit(X_train, y_train)

print ('RESULTS FOR NN\n')
print("Best parameters set found:")
# ADD YOUR CODE HERE
print(mlp_arch_CV.best_params_)

print("Score with best parameters:")
# ADD YOUR CODE HERE
print(mlp_arch_CV.best_score_)

print("\nMean scores on the grid:")
# ADD YOUR CODE HERE
print(mlp_arch_CV.cv_results_['mean_test_score'])
```

RESULTS FOR NN

Best parameters set found:
{'hidden_layer_sizes': (50,)}
Score with best parameters:
0.8426

Mean scores on the grid:
[0.8053 0.8235 0.8426 0.8343 0.8389]

TO DO A.3 Now try different batch sizes, while keeping the **best NN** architecture you have

found above. Remember that the batch size was previously set to the default value, i.e., `min(200, n_samples)`.

Recall that a **batch size of 1 corresponds to baseline SGD**, while using all the **400 training samples** (there are 500 samples but in cross validation with 5 folders we use 1/5 of them for validation at each round) corresponds to **standard GD** and using a different mini-batch size lies in the middle between the two extreme cases.

```
[9]: # these are sample values corresponding to baseline SGD, a reasonable
      ↪ mini-batch size and standard GD
      # again feel free to change them as you like, try to experiment with different
      ↪ batch sizes!!
      parameters = {'batch_size': [2, 32, 50]}

      # need to specify that you would like to use the standard k-fold split
      ↪ otherwise sklearn create splits of different sizes
      kf = sklearn.model_selection.KFold() # ADD YOUR CODE HERE, suggestion: http://
      ↪ scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
      best_hidden_layer_sizes = (50,) # ADD YOUR CODE HERE
      mlp = MLPClassifier(hidden_layer_sizes=best_hidden_layer_sizes, max_iter=250,
      ↪ alpha=1e-4, solver='sgd', tol=1e-4, learning_rate_init=.1, random_state=ID)
      ↪ # ADD YOUR CODE HERE

      # recall to use cv=kf to use the k-fold subdivision seen in the lectures
      mlp_batch_CV = GridSearchCV(mlp, parameters) # ADD YOUR CODE HERE
      mlp_batch_CV.fit(X_train, y_train)

      print ('RESULTS FOR NN\n')
      print("Best parameters set found:")
      # ADD YOUR CODE HERE
      print(mlp_batch_CV.best_params_)

      print("Score with best parameters:")
      # ADD YOUR CODE HERE
      print(mlp_batch_CV.best_score_)

      print("\nAll scores on the grid:")
      # ADD YOUR CODE HERE
      print(mlp_batch_CV.cv_results_['mean_test_score'])
```

RESULTS FOR NN

Best parameters set found:

`{'batch_size': 50}`

Score with best parameters:

`8.3570000000000001e-01`

All scores on the grid:

[0.0985 0.8067 0.8357]

TO DO A.Q1

- What do you observe for different architectures and batch sizes?
- How do the number of layers and their sizes affect the performances?
- What do you observe for different batch sizes, in particular what happens to the training convergence for different batch sizes (notice that the algorithm could not converge for some batch sizes)?

ANSWER A.Q1:

- The performance is better with larger batch sizes since the higher the batch size the more “precise” is the gradient descent, while when decreasing the batch size we tend to more and more “random” stochastic GD which decreases the “accuracy”.
- We could expect that increasing the layer/number of neurons we can achieve better performance but we can easily notice that increasing too much these 2 parameters leads to overfitting. Note that a relatively simple architecture (50,) “wins” against much more complicated architectures such as (50,25,10,).
- (I used max_iter=500 and all the batch_sizes converge).

TO DO A.4 Now try to use **different learning rates**, while keeping the **best NN architecture and batch size you have found above**. **Plot the learning curves** (i.e., the variation of the loss over the steps, you can get it from the `loss_curve_` object of sklearn) for the different values of the learning rate. You might get warnings about the convergence, this is expected. Ignore them.

```
[10]: lr_list = [4*10**exp for exp in range(-4,0)]
      scores = {}

      best_hidden_layer_sizes = (50,) # ADD YOUR CODE HERE
      best_batch_size = 50 # ADD YOUR CODE HERE

      for lr in lr_list:
          mlp = MLPClassifier(hidden_layer_sizes=best_hidden_layer_sizes,
                               ↪batch_size=best_batch_size, max_iter=250, alpha=1e-4, solver='sgd',
                               ↪tol=1e-4, learning_rate_init=lr, random_state=ID) # ADD YOUR CODE HERE
          mlp.fit(X_train, y_train)
          scores[lr] = mlp.score(X_test, y_test) # ADD YOUR CODE HERE, score on the
          ↪test set
          plt.plot(mlp.loss_curve_, label='lr: ' + str(lr))

      plt.legend(loc = 1)
      plt.xlabel('Number of iterations')
      plt.ylabel('Loss')
      plt.show()

      print ('RESULTS FOR NN\n')
      print("Best parameters set found:")
      # ADD YOUR CODE HERE
```



```

print(scores)
opt_key = max(scores, key=scores.get)
print(opt_key)

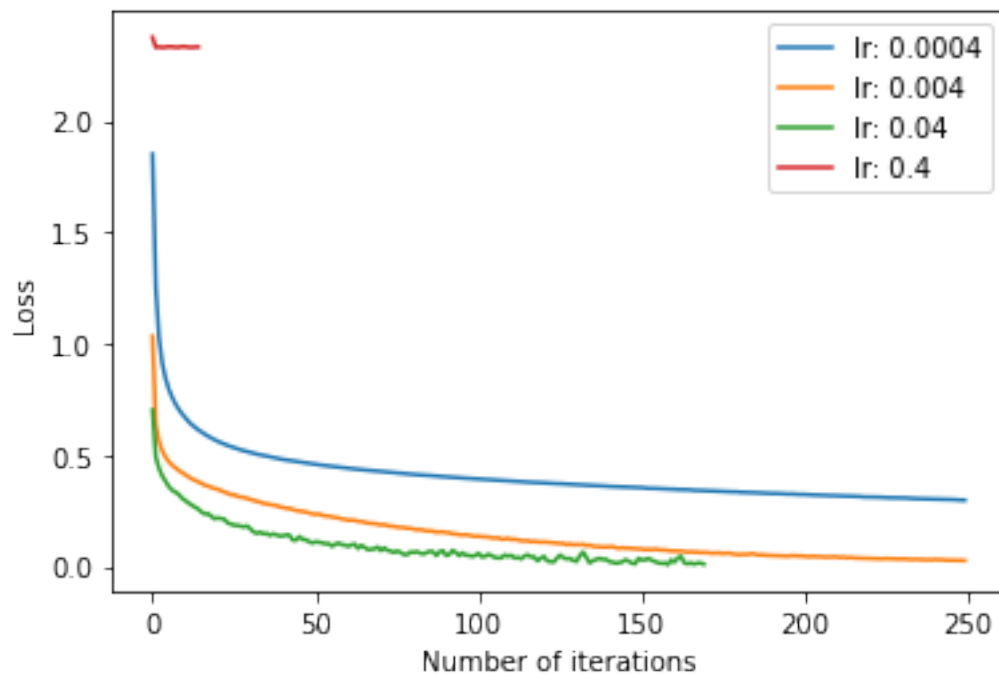
print("Score with best parameters:")
# ADD YOUR CODE HERE
print(scores[opt_key])

```

```

/home/tommaso/anaconda3/lib/python3.9/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:692:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (250) reached and
the optimization hasn't converged yet.
  warnings.warn(
/home/tommaso/anaconda3/lib/python3.9/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:692:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (250) reached and
the optimization hasn't converged yet.
  warnings.warn(

```



RESULTS FOR NN

Best parameters set found:

```
{0.0004: 8.553e-01, 0.004: 0.84294, 0.04: 0.85516, 0.4: 0.1002}
0.0004
```

Score with best parameters:

0.8553

Question A.Q2 Comment about the learning curves (i.e. the variation of the loss over the steps). How does the curve changes for different learning rates in terms of stability and speed of convergence ?

ANSWER A.Q2:

- In general a bigger rate is associated with a faster convergence but an higher sensibility to “noise”, namely oscillating effects due to have a too big learning step.

TO DO A.5 Now get **training and test error** for a NN with best parameters (architecture, batch size and learning rate) from above. Plot the learning curve also for this case. As before, you might get a convergence warning, you can safely ignore it.

```
[11]: #get training and test error for the best NN model from CV
mlp = MLPClassifier(hidden_layer_sizes=best_hidden_layer_sizes,
    ↪batch_size=best_batch_size, max_iter=250, alpha=1e-4, solver='sgd',
    ↪tol=1e-4, learning_rate_init=0.0004, random_state=ID) # ADD YOUR CODE HERE
mlp.fit(X_train, y_train)

training_error = 1-mlp.score(X_train, y_train) # ADD YOUR CODE HERE
test_error = 1-mlp.score(X_test, y_test) # ADD YOUR CODE HERE

print ('\nRESULTS FOR BEST NN\n')

print ("Best NN training error: %f" % training_error)
print ("Best NN test error: %f" % test_error)

plt.plot(mlp.loss_curve_, label="loss")
plt.legend(loc = 1)
plt.xlabel('Number of iterations')
plt.ylabel('Loss')
plt.show()
```

RESULTS FOR BEST NN

Best NN training error: 0.103200

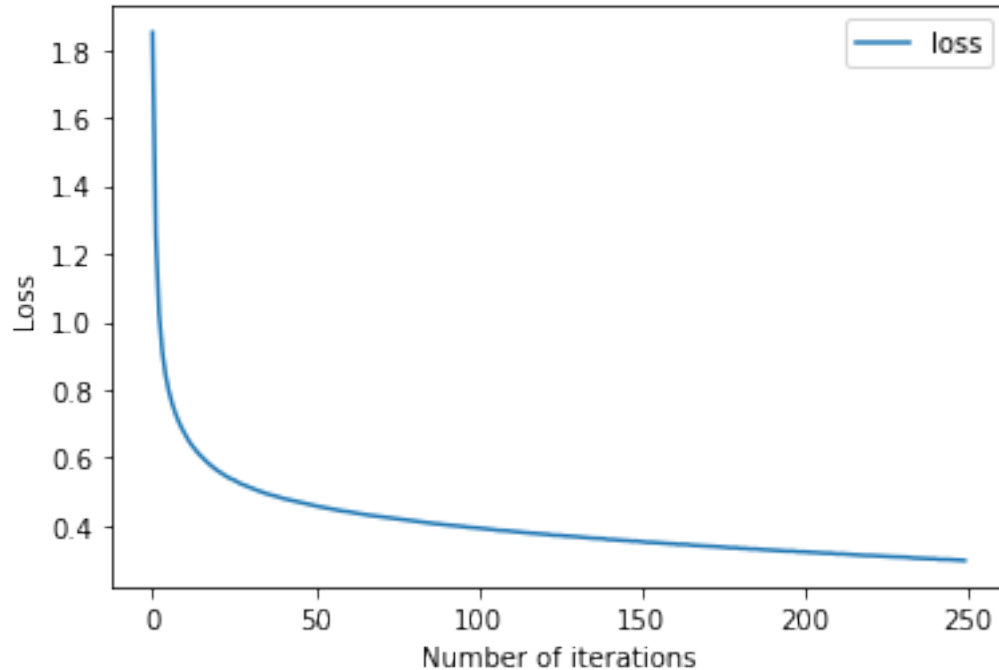
Best NN test error: 0.144700

/home/tommaso/anaconda3/lib/python3.9/site-

packages/sklearn/neural_network/_multilayer_perceptron.py:692:

ConvergenceWarning: Stochastic Optimizer: Maximum iterations (250) reached and the optimization hasn't converged yet.

warnings.warn(



4.1 B) More data

Now let's do the same but **using 20000** (or less if it takes too long on your machine) **data points for training**. Make sure you are **consistent with the choice of `m_training`** in this and the later cells. Use the **same NN architecture** as before, but you can try more if you like and have a powerful computer!!

```
[12]: X = X[permutation]
      y = y[permutation]

      m_training = 20000 # ADD YOUR CODE HERE

      X_train, X_test = X[:m_training], X[m_training:] # ADD YOUR CODE HERE
      y_train, y_test = y[:m_training], y[m_training:] # ADD YOUR CODE HERE

      labels, freqs = np.unique(y_train, return_counts = True) # ADD YOUR CODE HERE
      print("Labels in training dataset: ", labels)
      print("Frequencies in training dataset: ", freqs)
```

Labels in training dataset: [0 1 2 3 4 5 6 7 8 9]

Frequencies in training dataset: [2015 1975 2008 1990 1972 1964 2052 1964 2026 2034]

TO DO B.1 Train the NNs with the added data points using the optimum parameters found above. Eventually, feel free to try different architectures if you like. We suggest that you use `verbose=True` so have an idea of how long it takes to run 1 iteration (eventually reduce also the

number of iterations to 50). This is just for debug purposes, **remember to switch it off** once you found a setup you like. As before, you might get a convergence warning (especially if you reduced the number of iterations), you can safely ignore it.

```
[13]: mlp_large = MLPClassifier(verbose=False, hidden_layer_sizes=best_hidden_layer_sizes,
    ↪ batch_size=best_batch_size, max_iter=350, alpha=1e-4, solver='sgd',
    ↪ tol=1e-4, learning_rate_init=0.0004, random_state=ID) # ADD YOUR CODE HERE
mlp_large.fit(X_train, y_train)

print('\nRESULTS FOR NN\n')

#get training and test error for the NN
training_error = 1-mlp_large.score(X_train, y_train) # ADD YOUR CODE HERE
test_error = 1-mlp_large.score(X_test, y_test) # ADD YOUR CODE HERE
print("NN training error: %f" % training_error)
print("NN test error: %f" % test_error)
```

RESULTS FOR NN

NN training error: 0.078700

NN test error: 0.133125

/home/tommaso/anaconda3/lib/python3.9/site-

packages/sklearn/neural_network/_multilayer_perceptron.py:692:

ConvergenceWarning: Stochastic Optimizer: Maximum iterations (350) reached and the optimization hasn't converged yet.

warnings.warn(

Question B.Q1 Compare the train and test error you got with a large number of samples with the best one you obtained with only 500 data points. Comment about the results you obtained.

ANSWER B.Q1:

- I obtained a strange result here: the bigger NN achieves an higher Generalization gap which is not desirable and means overfitting. This is weird since using a bigger training set should decrease the overfitting.

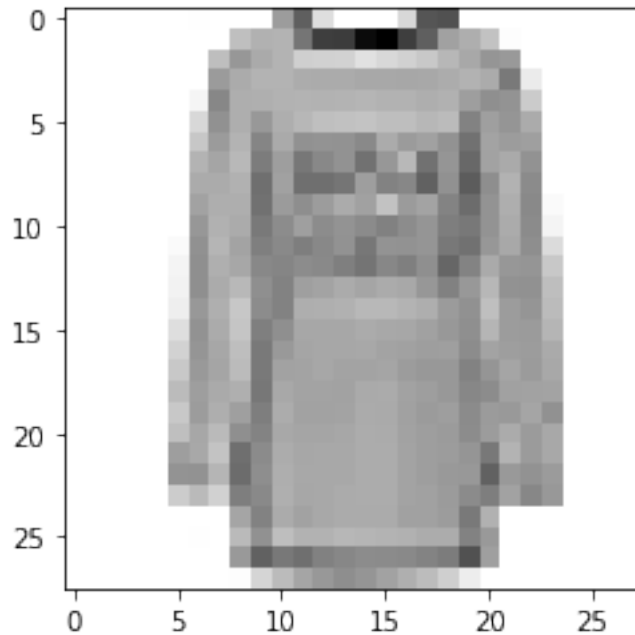
TO DO B.2 Plot an example that was missclassified by NN with $m=500$ training data points and it is now instead correctly classified by NN with $m=20000$ training data points.

```
[14]: NN_prediction = mlp.predict(X_test) # ADD YOUR CODE HERE
large_NN_prediction = mlp_large.predict(X_test) # ADD YOUR CODE HERE

for i, (p, pl, l) in enumerate(zip(NN_prediction, large_NN_prediction, y_test)):
    if p != l and pl == l:
        plot_input(X_test, y_test, i)
        print("NN prediction for m=500:", p)
        print("NN prediction for m=20000:", pl)
```

```
break
```

INPUT:



LABEL: 2

NN prediction for m=500: 3

NN prediction for m=20000: 2

TO DO B.3 Plot the weights of the multi-layer perceptron classifier, for the best NN we get with 500 data points and with 20000 data points. Note that the code is provided, you just need to change the `mlp` variable name to the one used before - if you changed it.

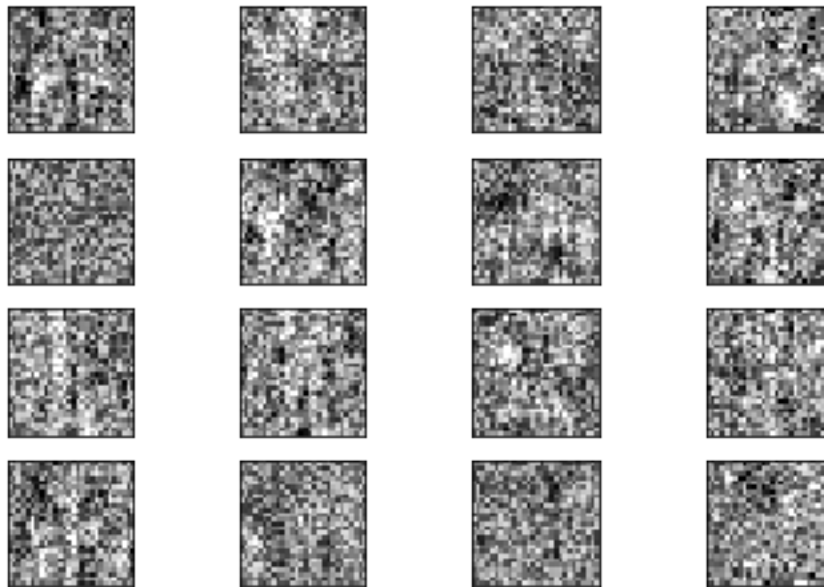
```
[15]: print("Weights with 500 data points:")
fig, axes = plt.subplots(4, 4)
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
                vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()

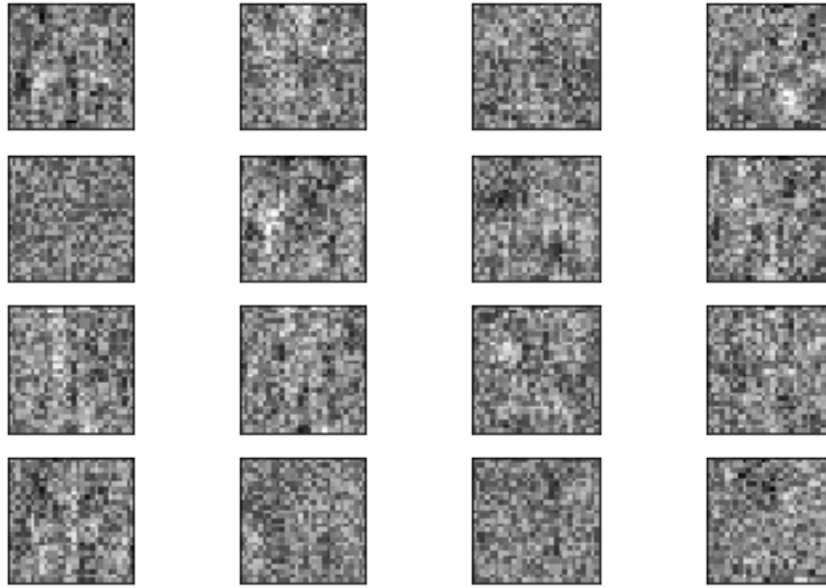
print("Weights with 20000 data points:")
fig, axes = plt.subplots(4, 4)
vmin, vmax = mlp_large.coefs_[0].min(), mlp_large.coefs_[0].max()
for coef, ax in zip(mlp_large.coefs_[0].T, axes.ravel()):
```

```
ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,  
            vmax=.5 * vmax)  
ax.set_xticks(())  
ax.set_yticks(())  
plt.show()
```

Weights with 500 data points:



Weights with 20000 data points:



Question B.Q2 Describe what do you observe by looking at the weights

ANSWER B.Q2:

We can notice in the first case more “visual” contrast, while when increasing the number of training samples we get a more uniform visualization. This is quite intuitive since an higher visual contrast is associated with a more “discriminating” prediction which is associated with overfitting (which occurs more intensely when the training data set is small).

TO DO B.4 Create a SVM classifier with the following parameters: `kernel='rbf'`, `C=10`, `gamma=0.01`. Fit it on a few data points and compute its training and test scores.

```
[16]: m_training = 500 # ADD YOUR CODE HERE

X_train, X_test = X[:m_training], X[m_training:]
y_train, y_test = y[:m_training], y[m_training:]

# best parameters found in the SVM notebook
SVM = SVC(kernel='rbf', C=10, gamma=0.01) # ADD YOUR CODE HERE
SVM.fit(X_train, y_train)

print ('RESULTS FOR SVM')
SVM_training_error = 1-SVM.score(X_train, y_train) # ADD YOUR CODE HERE

print("Training score SVM:")
print(SVM_training_error)

SVM_test_error = 1-SVM.score(X_test,y_test) # ADD YOUR CODE HERE
```

```
print("Test score SVM:")  
print(SVM_test_error)
```

RESULTS FOR SVM

Training score SVM:

0.0

Test score SVM:

1.965546218487395e-01

Question B.Q3 Compare the results of SVM and of NN. Which one achieves the best performances?

ANSWER B.Q3:

The SVM obtains $L_S = 0$ which means overfitting and in facts we can see that it performs worse than both the 2 NN when predicting the labels in the test set.

[]: