Machine Learning
MSE FTP MachLe
Christoph Würsch (mailto:christoph.wuersch@ost.ch)

Lab10, A5 – Dimensionality Reduction

Exercice 5: Stochastic Neighbour Embedding on MNIST dataset

Setup

First, let's make sure this notebook works well in both python 2 and 3, import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

In [1]:
```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "dim_reduction"

def save_fig(fig_id, tight_layout=True):
    path = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID, fig_id + ".png")
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format='png', dpi=600)
```

(a) Using t-SNE to reduce the dimensionality to two dimensions

Exercise: Use t-SNE to reduce the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class.

In [2]:
```
from sklearn.decomposition import PCA
```

Let's start by loading the MNIST dataset:

In [3]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')

#custom_data_home='C:/temp'
#mnist = fetch_openml('mnist_784', data_home=custom_data_home)
mnist.DESCR
```

Out[3]:
```
"**Author**: Yann LeCun, Corinna Cortes, Christopher J.C. Burges  \n**Source**: [MNIST Websit
e](http://yann.lecun.com/exdb/mnist/) - Date unknown  \n**Please cite**:  \n\nThe MNIST datab
ase of handwritten digits with 784 features, raw data available at: http://yann.lecun.com/exd
b/mnist/. (http://yann.lecun.com/exdb/mnist/.) It can be split in a training set of the first
60,000 examples, and a test set of 10,000 examples  \n\nIt is a subset of a larger set availa
ble from NIST. The digits have been size-normalized and centered in a fixed-size image. It is
a good database for people who want to try learning techniques and pattern recognition method
s on real-world data while spending minimal efforts on preprocessing and formatting. The orig
inal black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel
box while preserving their aspect ratio. The resulting images contain grey levels as a result
of the anti-aliasing technique used by the normalization algorithm. the images were centered
in a 28x28 image by computing the center of mass of the pixels, and translating the image so
as to position this point at the center of the 28x28 field.  \n\nWith some classification met
hods (particularly template-based methods, such as SVM and K-nearest neighbors), the error ra
te improves when the digits are centered by bounding box rather than center of mass. If you d
o this kind of pre-processing, you should report it in your publications. The MNIST database
was constructed from NIST's NIST originally designated SD-3 as their training set and SD-1 as
their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason f
or this can be found on the fact that SD-3 was collected among Census Bureau employees, while
SD-1 was collected among high-school students. Drawing sensible conclusions from learning exp
eriments requires that the result be independent of the choice of training set and test among
the complete set of samples. Therefore it was necessary to build a new database by mixing NIS
T's datasets.  \n\nThe MNIST training set is composed of 30,000 patterns from SD-3 and 30,000
patterns from SD-1. Our test set was composed of 5,000 patterns from SD-3 and 5,000 patterns
from SD-1. The 60,000 pattern training set contained examples from approximately 250 writers.
We made sure that the sets of writers of the training set and test set were disjoint. SD-1 co
ntains 58,527 digit images written by 500 different writers. In contrast to SD-3, where block
s of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identi
ties for SD-1 is available and we used this information to unscramble the writers. We then sp
lit SD-1 in two: characters written by the first 250 writers went into our new training set.
The remaining 250 writers were placed in our test set. Thus we had two sets with nearly 30,00
0 examples each. The new training set was completed with enough examples from SD-3, starting
at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set w
as completed with SD-3 examples starting at pattern # 35,000 to make a full set with 60,000 t
est patterns. Only a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3) is av
ailable on this site. The full 60,000 sample training set is available.\n\nDownloaded from op
enml.org."
```

In [ ]:
```

```

Dimensionality reduction on the full 60,000 images takes a very long time, so let's only do this on a random subset of 5,000 images:

In [6]:
```
np.random.seed(42)

m = 5000
idx = np.random.permutation(60000)[:m]
```

In [9]:
```
X = mnist.data.iloc[idx,:].values
y = mnist.target.iloc[idx].values

y[0:5]
```

Out[9]:
```
['7', '3', '8', '9', '3']
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']
```

In [8]:
```
#we save the data to disk so that we can fetch it fast if we need it using the next cell:

import pandas as pd
mnist.data.to_pickle('mnist_data_784.pkl')
mnist.target.to_pickle('mnist_label_784.pkl');
```

In [7]:
```
#read data from pickle file if there is no internet connection
# Xp=pd.read_pickle('mnist_data_784.pkl').values
# yp=pd.read_pickle('mnist_label_784.pkl').values

# m = 5000
# idx = np.random.permutation(60000)[:m]
# X = Xp[idx,:]
# y = yp[idx]
```

Now let's use t-SNE to reduce dimensionality down to 2D so we can plot the dataset (This can take quite a while...):

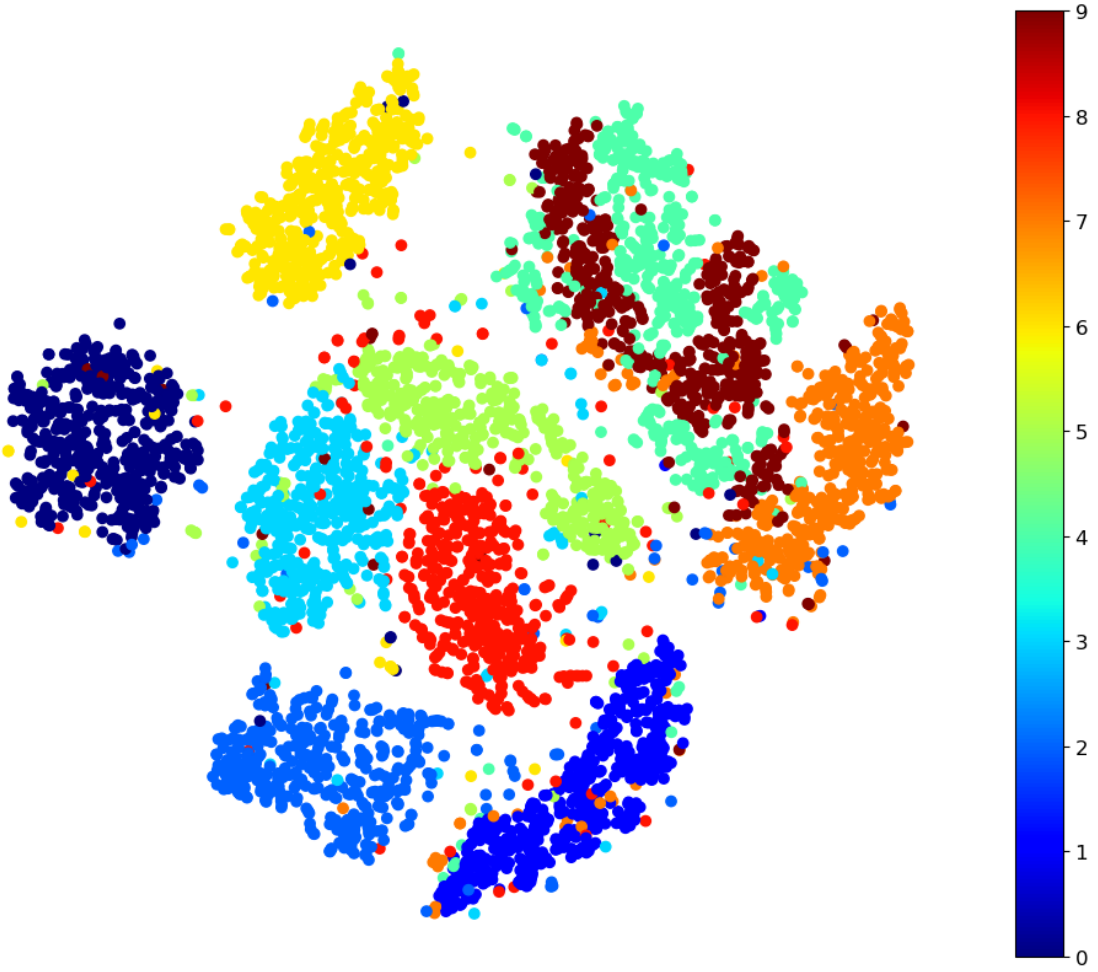
In [10]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_reduced = tsne.fit_transform(X)
```

```
C:\Users\christoph.wuersch\.conda\envs\ML\lib\site-packages\sklearn\manifold\_t_sne.py:800: F
utureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
C:\Users\christoph.wuersch\.conda\envs\ML\lib\site-packages\sklearn\manifold\_t_sne.py:810: F
utureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```

Now let's use Matplotlib's `scatter()` function to plot a scatterplot, using a different color for each digit:

In [11]:
```
plt.figure(figsize=(13,10))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y.astype(int), cmap="jet")
plt.axis('off')
plt.colorbar()
plt.show()
```

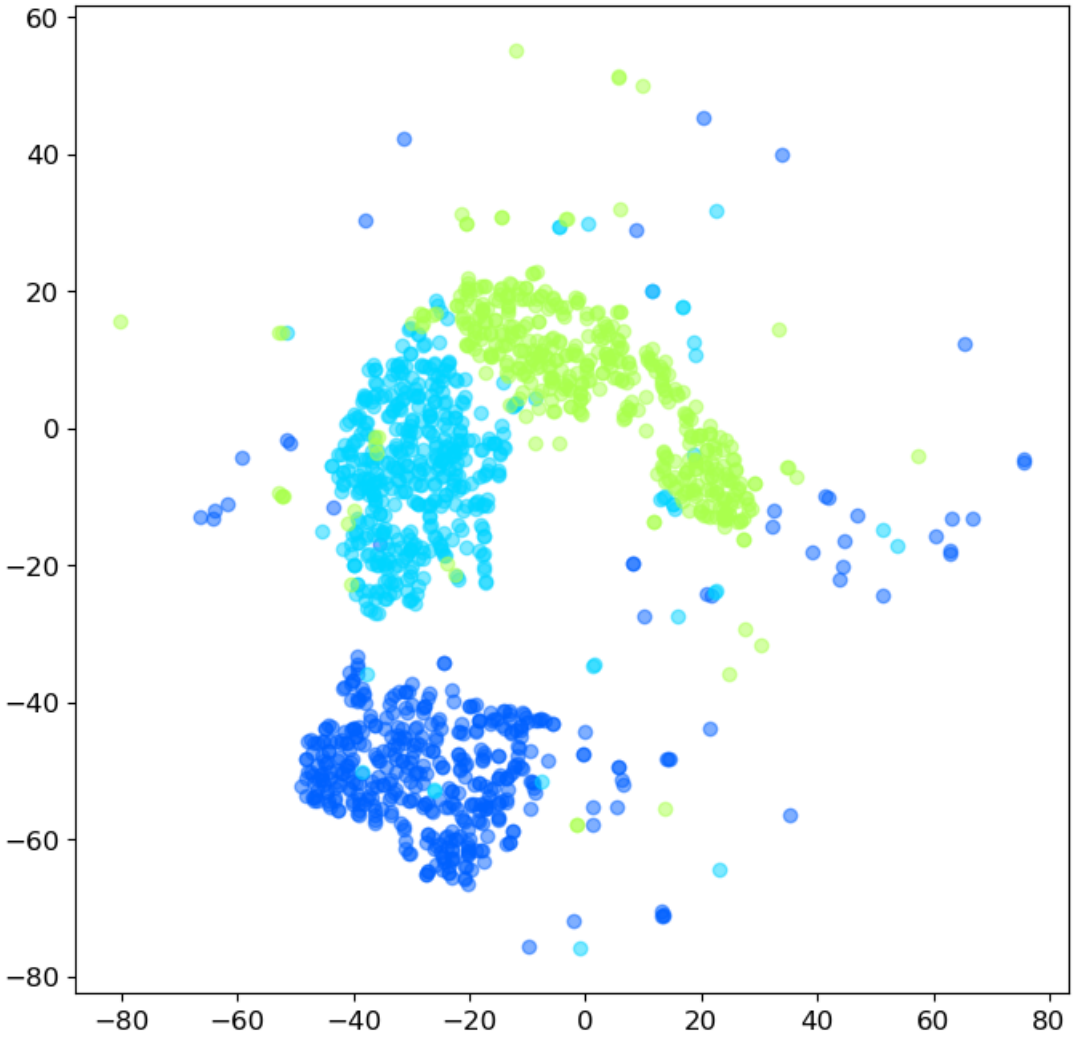


Isn't this just beautiful? :) This plot tells us which numbers are easily distinguishable from the others (e.g., 0s, 6s, and most 8s are rather well separated clusters), and it also tells us which numbers are often hard to distinguish (e.g., 4s and 9s, 5s and 3s, and so on).

(b) Labelling the Clusters

Let's focus on digits 2, 3 and 5, which seem to overlap a lot.

In [12]:
```
plt.figure(figsize=(8,8))
cmap = matplotlib.cm.get_cmap("jet")
for digit in (2, 3, 5):
    color=np.array(cmap(digit / 9)).reshape(1,4)
    plt.scatter(X_reduced[y.astype(int) == digit, 0], X_reduced[y.astype(int) == digit, 1], c=
#plt.axis('off')
plt.show()
```



t-SNE instead of PCA

Let's see if we can produce a nicer image by running t-SNE on these 3 digits:

In [64]:
```
y
```

Out[64]:
```
['1', '3', '4', '6', '3', ..., '8', '2', '5', '2', '1']
Length: 5000
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']
```

In [65]:
```
idx = (y == '2') | (y == '3') | (y == '5')
X_subset = X[idx]
y_subset = y[idx]

tsne_subset = TSNE(n_components=2, random_state=42)
X_subset_reduced = tsne_subset.fit_transform(X_subset)
```

In [66]:
```
plt.figure(figsize=(9,9))
for digit in (2, 3, 5):
    color=np.array(cmap(digit / 9)).reshape(1,4)
    plt.scatter(X_subset_reduced[y_subset.astype(int) == digit, 0],
                X_subset_reduced[y_subset.astype(int) == digit, 1], c=color)
plt.axis('off')
plt.show()
```

Much better, now the clusters have far less overlap. But some 3s are all over the place. Plus, there are two distinct clusters of 2s, and also two distinct clusters of 5s. It would be nice if we could visualize a few digits from each cluster, to understand why this is the case. Let's do that now.

Exercise: Alternatively, you can write colored digits at the location of each instance, or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits.

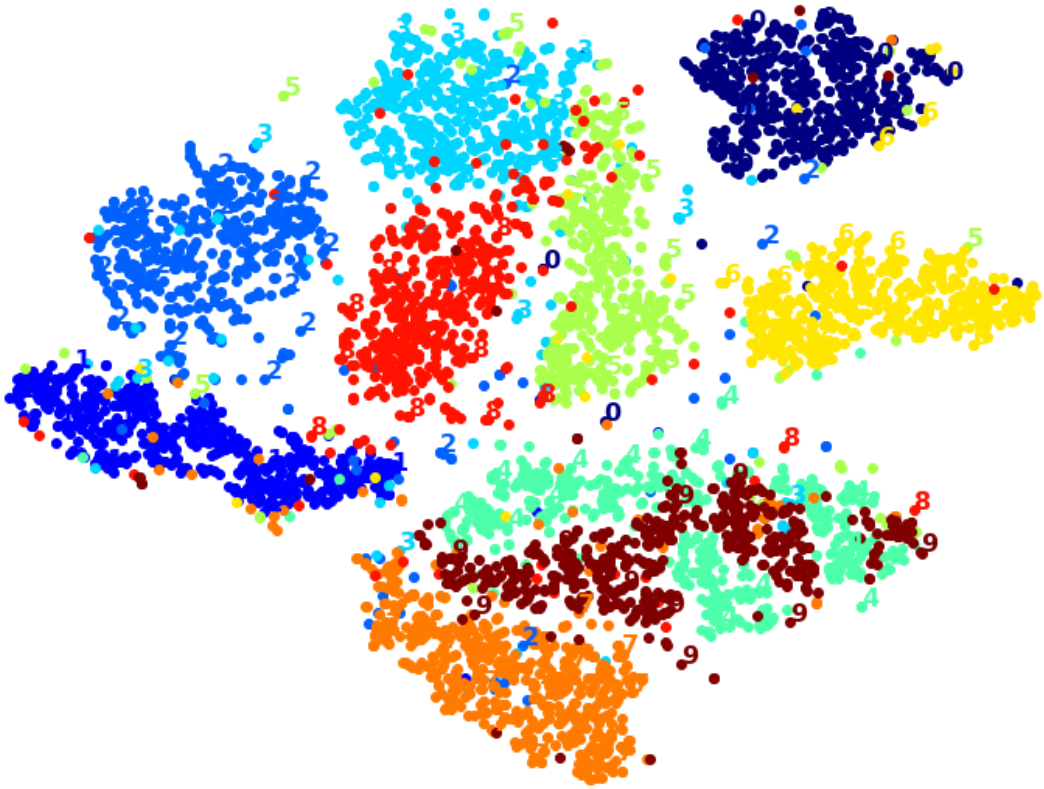
Let's create a `plot_digits()` function that will draw a scatterplot (similar to the above scatterplots) plus write colored digits, with a minimum distance guaranteed between these digits. If the digit images are provided, they are plotted instead. This implementation was inspired from one of Scikit-Learn's excellent examples (plot_lle_digits (http://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html), based on a different digit dataset).

In [67]:
```
from sklearn.preprocessing import MinMaxScaler
from matplotlib.offsetbox import AnnotationBbox, OffsetImage

def plot_digits(X, y, min_distance=0.05, images=None, figsize=(13, 10)):
    # Let's scale the input features so that they range from 0 to 1
    X_normalized = MinMaxScaler().fit_transform(X)
    # Now we create the list of coordinates of the digits plotted so far.
    # We pretend that one is already plotted far away at the start, to
    # avoid `if` statements in the loop below
    neighbors = np.array([[10., 10.]])
    # The rest should be self-explanatory
    plt.figure(figsize=figsize)
    cmap = matplotlib.cm.get_cmap("jet")
    digits = np.unique(y)
    for digit in digits:
        color=np.array(cmap(digit / 9)).reshape(1,4)
        plt.scatter(X_normalized[y == digit, 0], X_normalized[y == digit, 1], c=color)
    plt.axis("off")
    ax = plt.gcf().gca()  # get current axes in current figure
    for index, image_coord in enumerate(X_normalized):
        closest_distance = np.linalg.norm(np.array(neighbors) - image_coord, axis=1).min()
        if closest_distance > min_distance:
            neighbors = np.r_[neighbors, [image_coord]]
            if images is None:
                plt.text(image_coord[0], image_coord[1], str(int(y[index])),
                         color=cmap(y[index] / 9), fontdict={"weight": "bold", "size": 16})
            else:
                image = images[index].reshape(28, 28)
                imagebox = AnnotationBbox(OffsetImage(image, cmap="binary"), image_coord)
                ax.add_artist(imagebox)
```

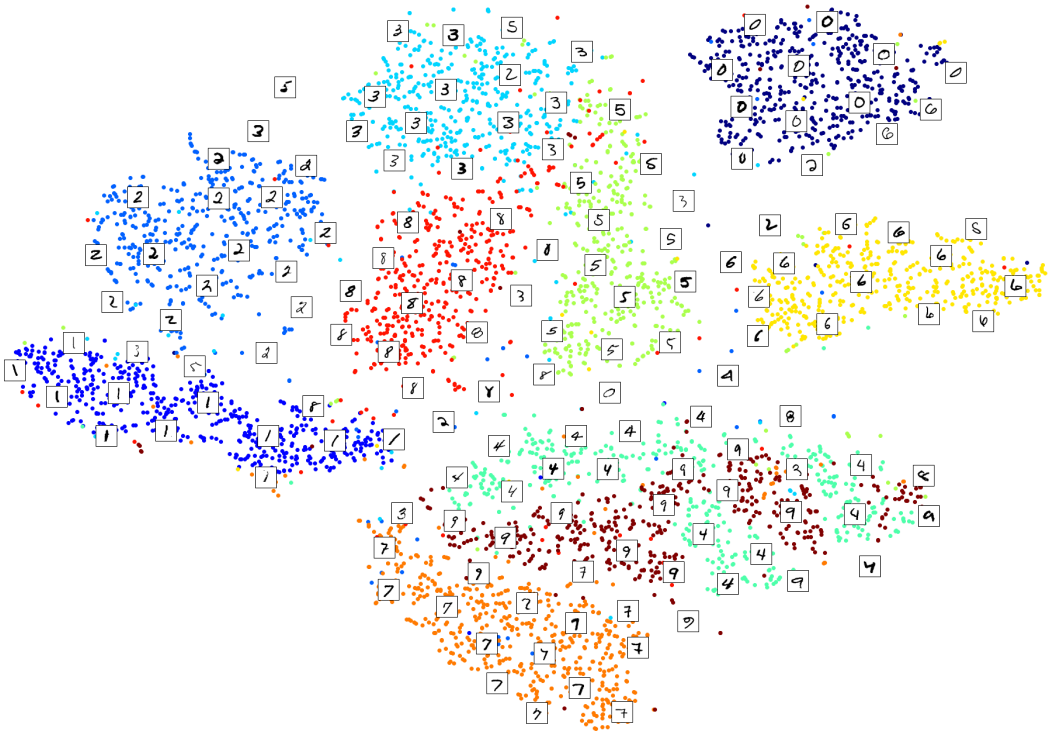
Let's try it! First let's just write colored digits:

In [68]:
```
plot_digits(X_reduced, y.astype(int))
```

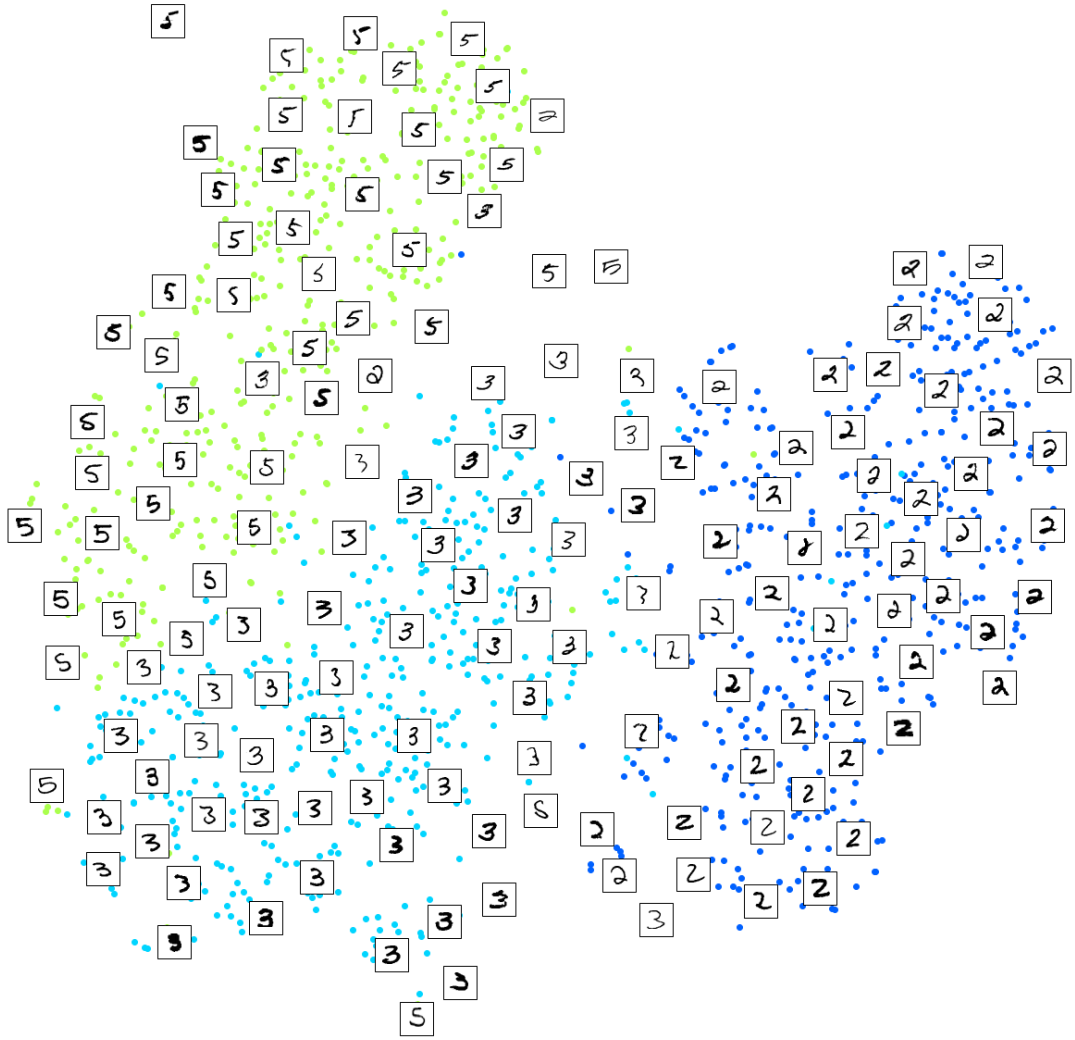


Well that's okay, but not that beautiful. Let's try with the digit images:

In [69]:
```
plot_digits(X_reduced, y.astype(int), images=X, figsize=(35, 25))
```



In [70]:
```
plot_digits(X_subset_reduced, y_subset.astype(int), images=X_subset, figsize=(22, 22))
```



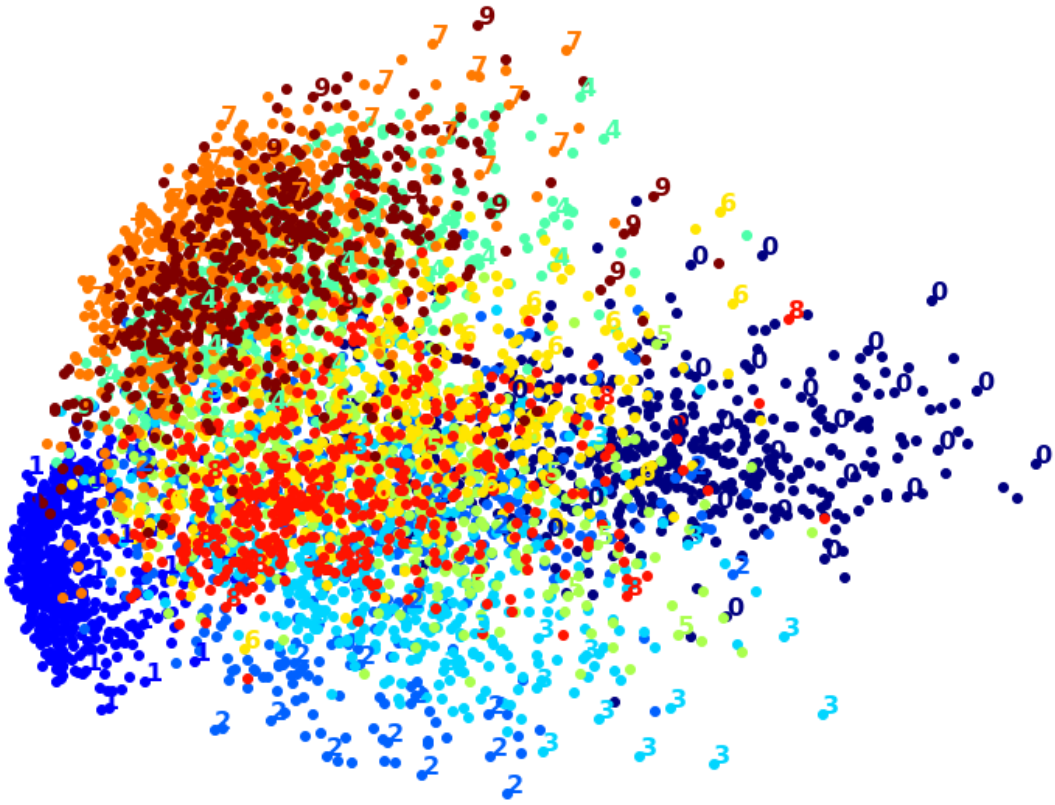
Exercise: Try using other dimensionality reduction algorithms such as PCA, LLE, or MDS and compare the resulting visualizations.

Let's start with PCA. We will also time how long it takes:

In [71]:
```
from sklearn.decomposition import PCA
import time

t0 = time.time()
X_pca_reduced = PCA(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("PCA took {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_reduced, y.astype(int))
plt.show()
```

```
PCA took 0.2s.
```



Wow, PCA is blazingly fast! But although we do see a few clusters, there's way too much overlap. Let's try LLE:

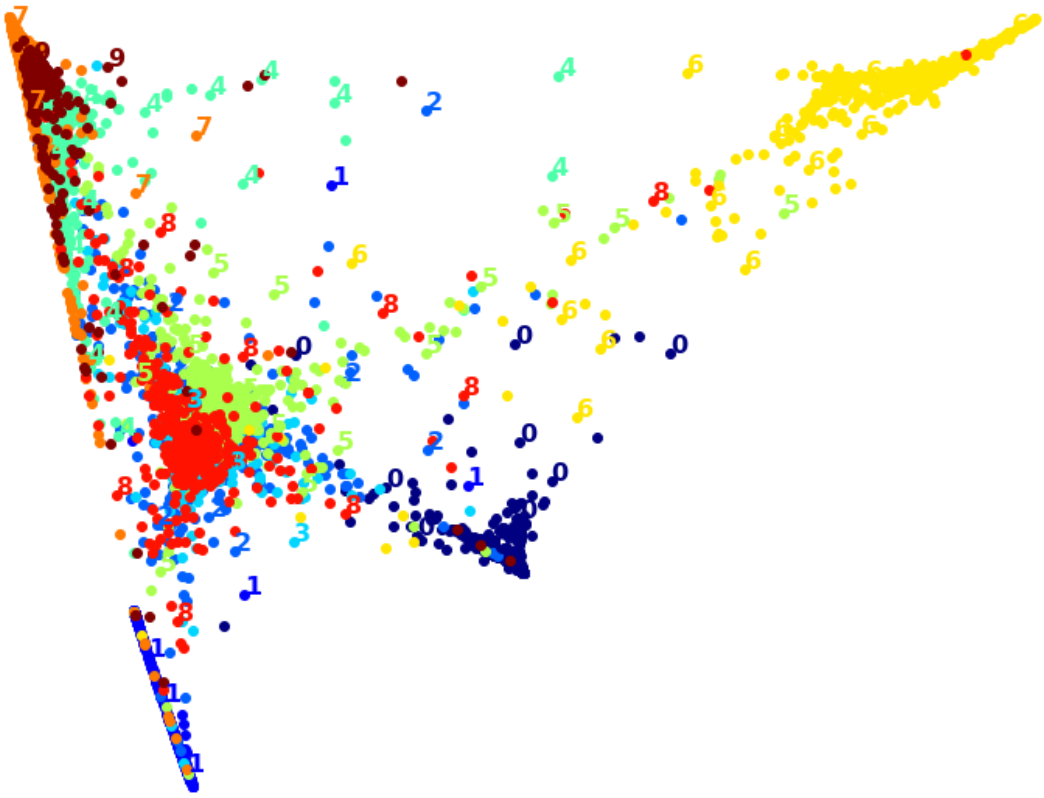
(c) Using other dimensionality reduction algorithms such as LLE, MDS and t-SNE

First we start with LLE, i.e. Local Linear Embedding which is part of the mainfold class of sklearn.

In [72]:
```
from sklearn.manifold import LocallyLinearEmbedding

t0 = time.time()
X_lle_reduced = LocallyLinearEmbedding(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("LLE took {:.1f}s.".format(t1 - t0))
plot_digits(X_lle_reduced, y.astype(int))
plt.show()
```

```
LLE took 2.9s.
```

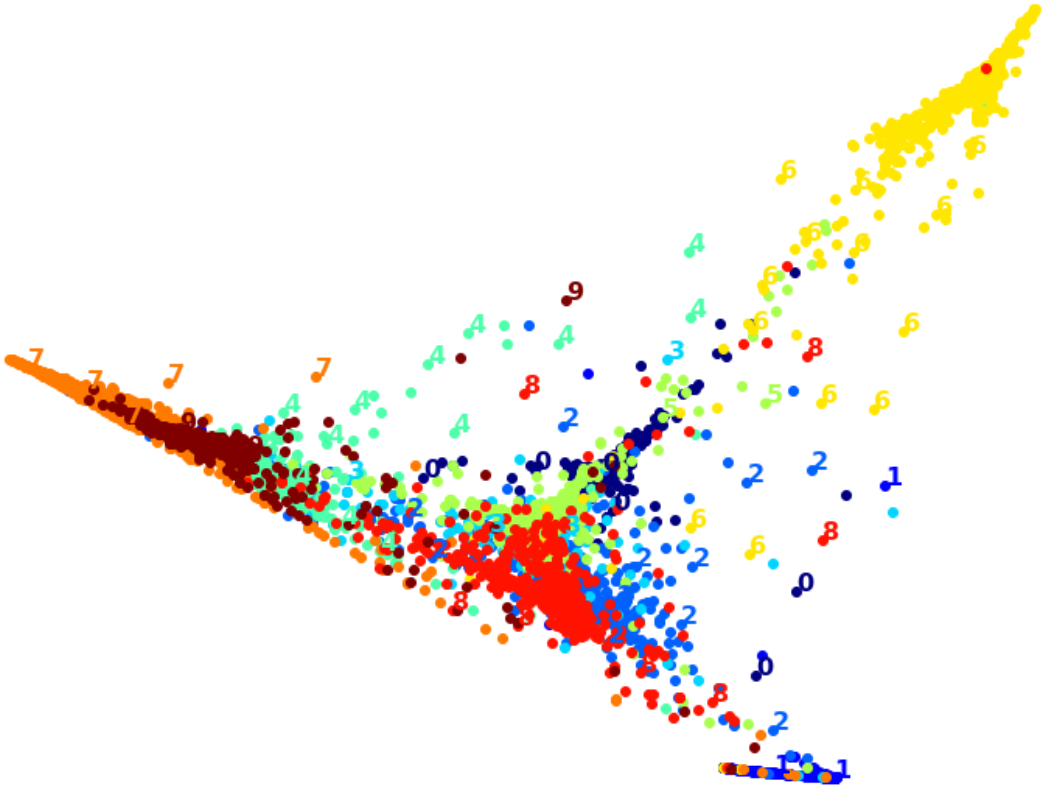


That took a while, and the result does not look too good. Let's see what happens if we apply PCA first, preserving 95% of the variance:

In [73]:
```
from sklearn.pipeline import Pipeline

pca_lle = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("lle", LocallyLinearEmbedding(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_lle_reduced = pca_lle.fit_transform(X)
t1 = time.time()
print("PCA+LLE took {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_lle_reduced, y.astype(int))
plt.show()
```

```
PCA+LLE took 3.4s.
```



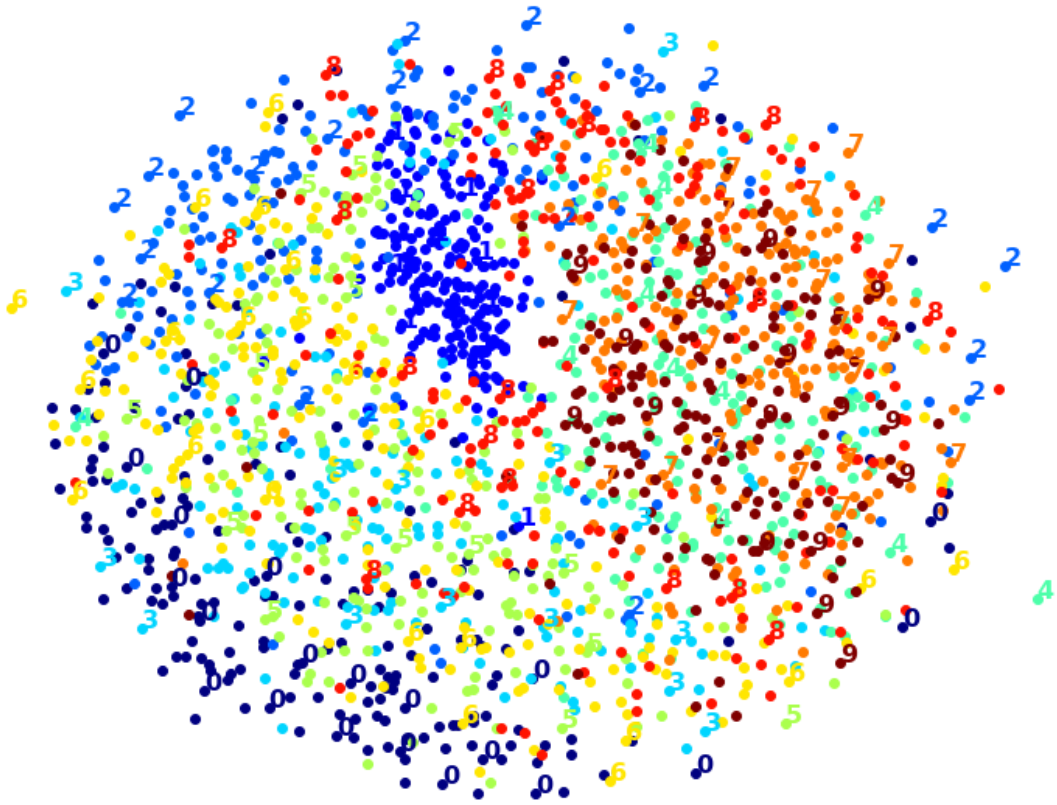
The result is more or less the same, but this time it was almost 4× faster.

Let's try MDS. It's much too long if we run it on 10,000 instances, so let's just try 2,000 for now:

In [74]:
```
from sklearn.manifold import MDS

m = 2000
t0 = time.time()
X_mds_reduced = MDS(n_components=2, random_state=42).fit_transform(X[:m])
t1 = time.time()
print("MDS took {:.1f}s (on just 2,000 MNIST images instead of 10,000).".format(t1 - t0))
plot_digits(X_mds_reduced, y[:m].astype(int))
plt.show()
```

```
MDS took 226.3s (on just 2,000 MNIST images instead of 10,000).
```

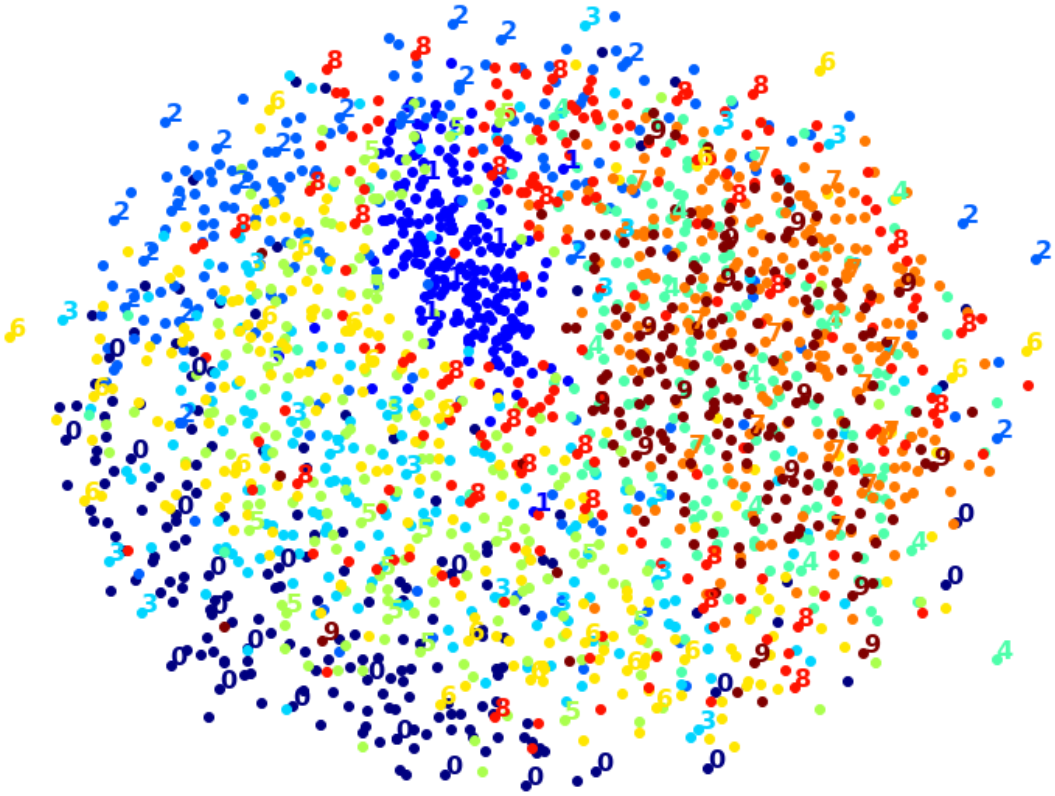


Meh. This does not look great, all clusters overlap too much. Let's try with PCA first, perhaps it will be faster?

In [75]:
```
from sklearn.pipeline import Pipeline

pca_mds = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("mds", MDS(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_mds_reduced = pca_mds.fit_transform(X[:2000])
t1 = time.time()
print("PCA+MDS took {:.1f}s (on 2,000 MNIST images).".format(t1 - t0))
plot_digits(X_pca_mds_reduced, y[:2000].astype(int))
plt.show()
```

```
PCA+MDS took 246.6s (on 2,000 MNIST images).
```



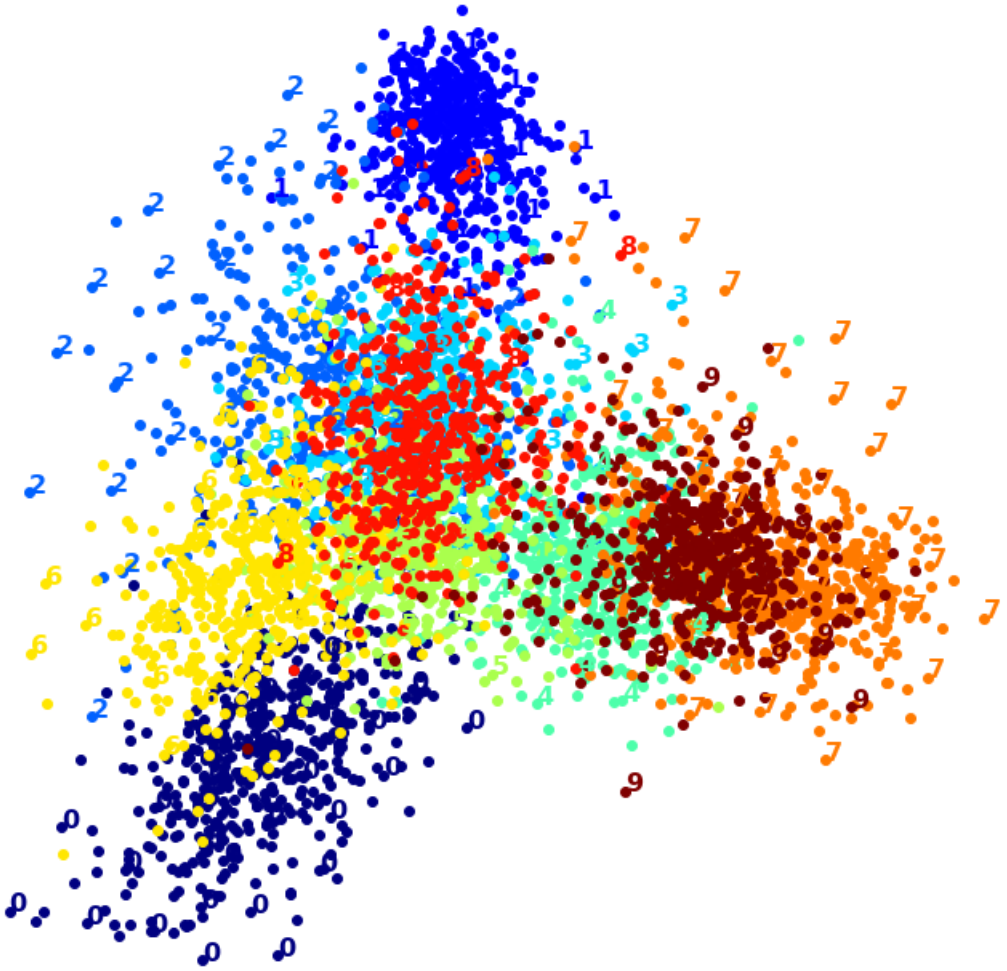
Same result, and no speedup: PCA did not help (or hurt).

Let's try LDA:

In [76]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

t0 = time.time()
X_lda_reduced = LinearDiscriminantAnalysis(n_components=2).fit_transform(X, y)
t1 = time.time()
print("LDA took {:.1f}s.".format(t1 - t0))
plot_digits(X_lda_reduced, y.astype(int), figsize=(12,12))
plt.show()
```

```
LDA took 1.2s.
```



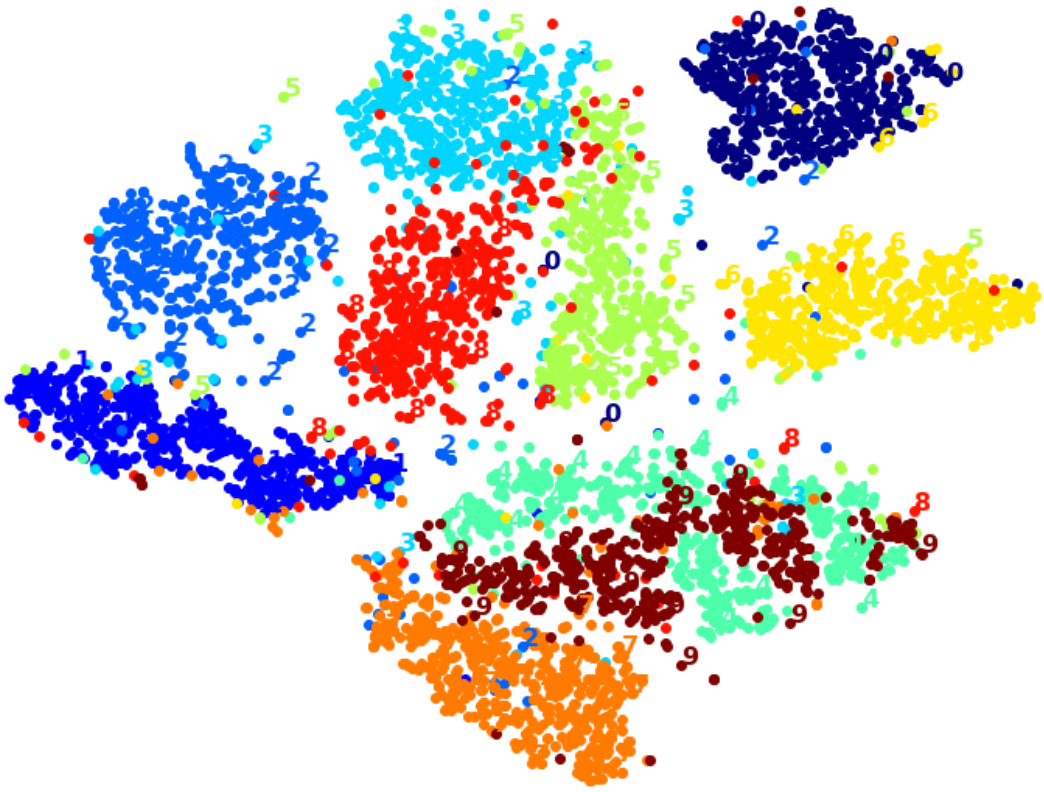
This one is very fast, and it looks nice at first, until you realize that several clusters overlap severely.

Well, it's pretty clear that t-SNE won this little competition, wouldn't you agree? We did not time it, so let's do that now:

In [77]:
```
from sklearn.manifold import TSNE

t0 = time.time()
X_tsne_reduced = TSNE(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("t-SNE took {:.1f}s.".format(t1 - t0))
plot_digits(X_tsne_reduced, y.astype(int))
plt.show()
```

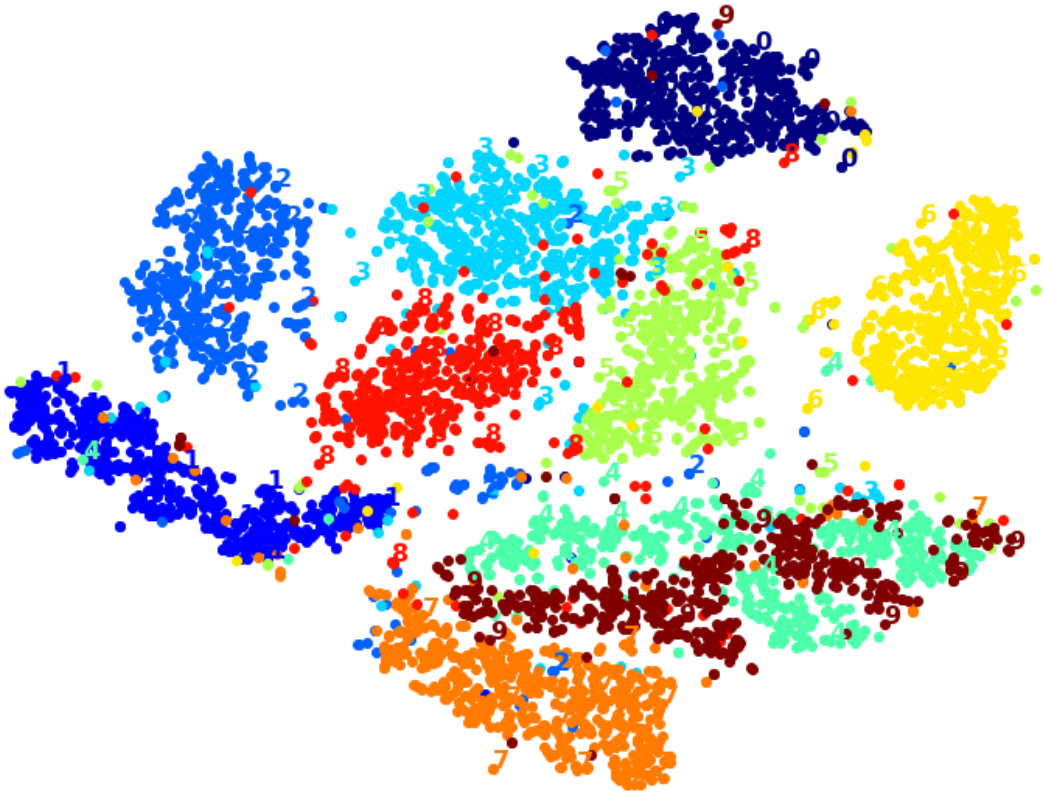
```
t-SNE took 48.1s.
```



It's twice slower than LLE, but still much faster than MDS, and the result looks great. Let's see if a bit of PCA can speed it up:

In [78]:
```
pca_tsne = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("tsne", TSNE(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_tsne_reduced = pca_tsne.fit_transform(X)
t1 = time.time()
print("PCA+t-SNE took {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_tsne_reduced, y.astype(int))
plt.show()
```

```
PCA+t-SNE took 55.3s.
```



Yes, PCA roughly gave us a 25% speedup, without damaging the result. We have a winner!

In [ ]:
```
```