ISMIN Machine Learning Project Report

Axel Comparetto-Berthier

Gabriel Micoud

Started on the 26th November 2020

Table of content

Introduction

Context

This report marks the end of the machine learning project included in the second year of the ISMIN course in the Ecole des Mines de Saint-Etienne. This project is based on the use of the MNIST dataset, which contains pictures of handwritten digits.

The project was conducted using Python 3.7.2 and various machine learning libraries, including numpy, Matplotlib, Scikit Learn (a.k.a. Sklearn, version 0.21.0) and Tensorflow (version 2.3.1). Please note that some processes resort to random generation, so the results can slightly vary between each run of the script associated.

Syntax

The problem presents several questions in each part that we must answer. Those questions are shown with the keyword "**WORK:**". The code is written with a `specific font`, the code blocks will have `the same specific font highlighted` and console results are introduced with ">>".

I) The dataset: MNIST

The aim of this part of the report is to describe the MNIST dataset. This dataset contains pictures of handwritten digits. We can load the dataset with the following code:

```
X = np.load("MNIST_X_28x28.npy") #images
y = np.load("MNIST_y.npy") #labels
```

WORK: What is the shape of the data? Display samples from the dataset.

The shape of the dataset is $70000 \times 28 \times 28$. Decomposing it, 28×28 is the size of every picture in pixels. 70000 is the number of images in the whole dataset. Here are some samples:

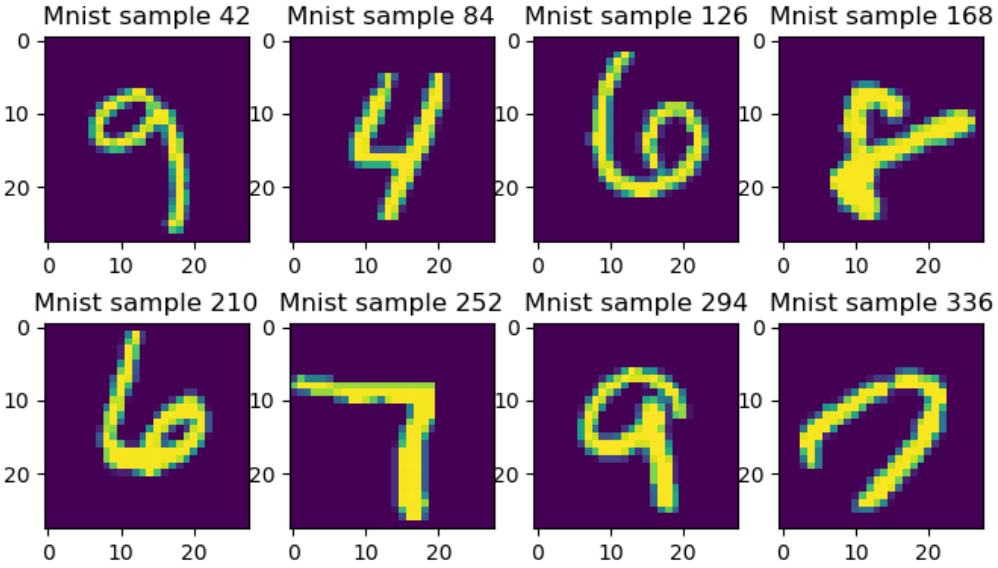


Figure 1: samples from MNIST dataset

WORK: Is the dataset well balanced?

To answer this question, we can plot the frequency of every digit:

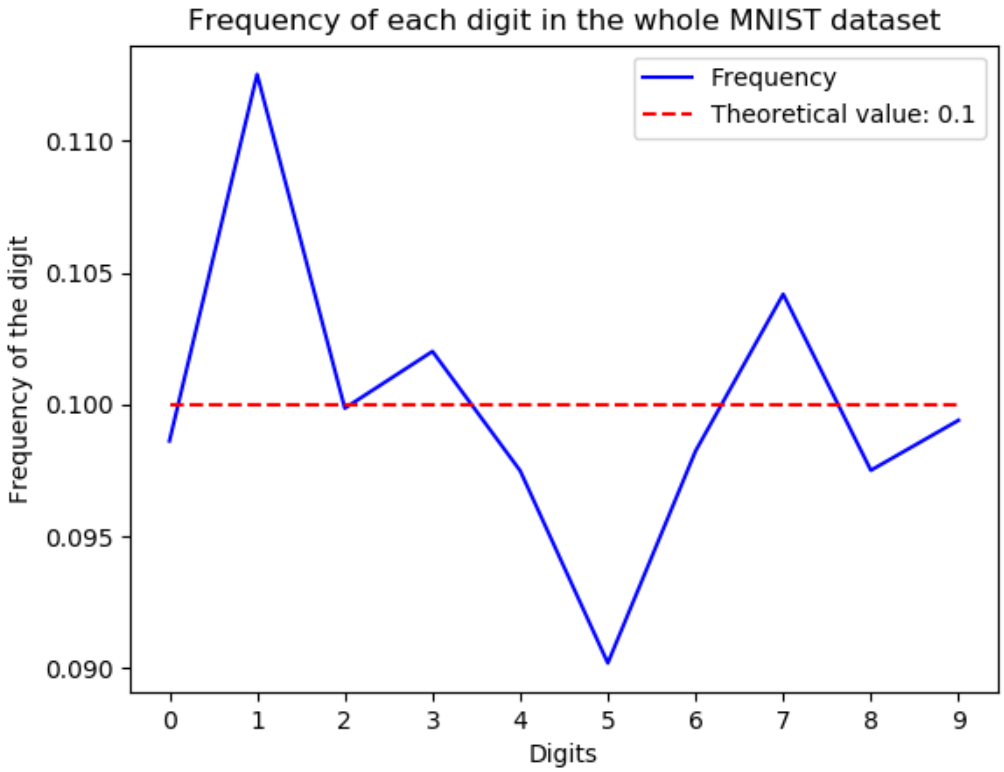


Figure 2: frequency of each digit

We observe there is no more than 1.5% excess for the label that has the most difference when compared to the theoretical value, which is acceptable.

Another insight to answer the question is the mean and the standard deviation of the proportion of each digit in the dataset:

```
>> Mean: 0.1 ; Standard deviation: 0.005411774766418957
```

The mean is (almost) identical to its theoretical value, and the standard deviation is very low.

All these elements tend to prove the dataset is well balanced.

WORK: Split the dataset in one train set and one test set

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2) #20% test set, 80% training set
```

We chose to keep 80% of the data as training set and to test with 20% of the data. This is a common ratio when splitting a dataset. We will use these test and training sets for the rest of the project.

II) Unsupervised machine learning

II)1) Dimensionality reduction

Working with 28x28 gray scale picture means we are working with 784 dimensions (each pixel corresponds to a dimension), which is not unprecedented but still quite a lot. The aim of this part is to perform principal component analysis (PCA) to consider only the most relevant dimensions.

WORK: Perform a Principal Component Analysis (PCA) with sklearn.

```
nsamples, dimx, dimy = X.shape
d2_X = X.reshape((nsamples, dimx*dimy)) #flatten the pictures
pca = PCA(n_components=3)
pca.fit(d2_X)
```

The code above performs a PCA on the dataset. Note that we first need to "flatten" the dataset (from 28x28 matrices to 784 arrays) as required by `PCA::fit`. Here the PCA is performed with only the 3 most relevant dimensions, which is absolutely not enough for the purpose of this exercise. Let us try different values of `n_components`.

WORK: Try different `n_components`

We tried to use the 5, 20, 50 and 100 principal components. Here are the results.

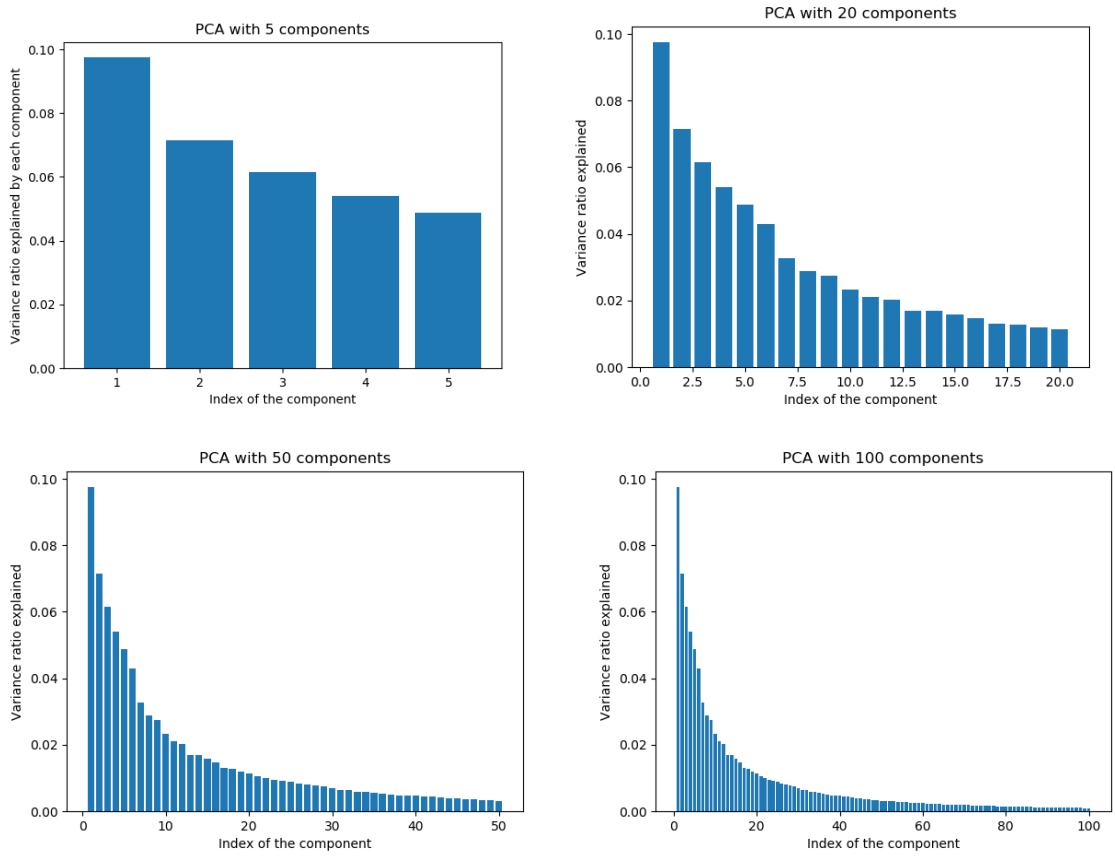


Figure 3: PCA on MNIST with several principal components

Adding components to the PCA does not alter the first components. Then we can use the graph above to compute the explained variance ratio of the first `n_components`:

```
>> Variation explained by the first 5 components: 0.3334340972252255
>> Variation explained by the first 20 components: 0.644706791661695
>> Variation explained by the first 50 components: 0.825418838829714
>> Variation explained by the first 100 components: 0.91496660849744
```

Now, we understand that the 100 most relevant dimensions hold about 90% of the total information, which allow to gain time when processing enormous datasets.

WORK: Try to display some MNIST pictures with different `n_components`.

Another interesting property of PCA is that it allows to reconstruct the original data using less dimensions. Doing so, we can display images of the MNIST dataset when using a given number of principal components and compare them to the original images.

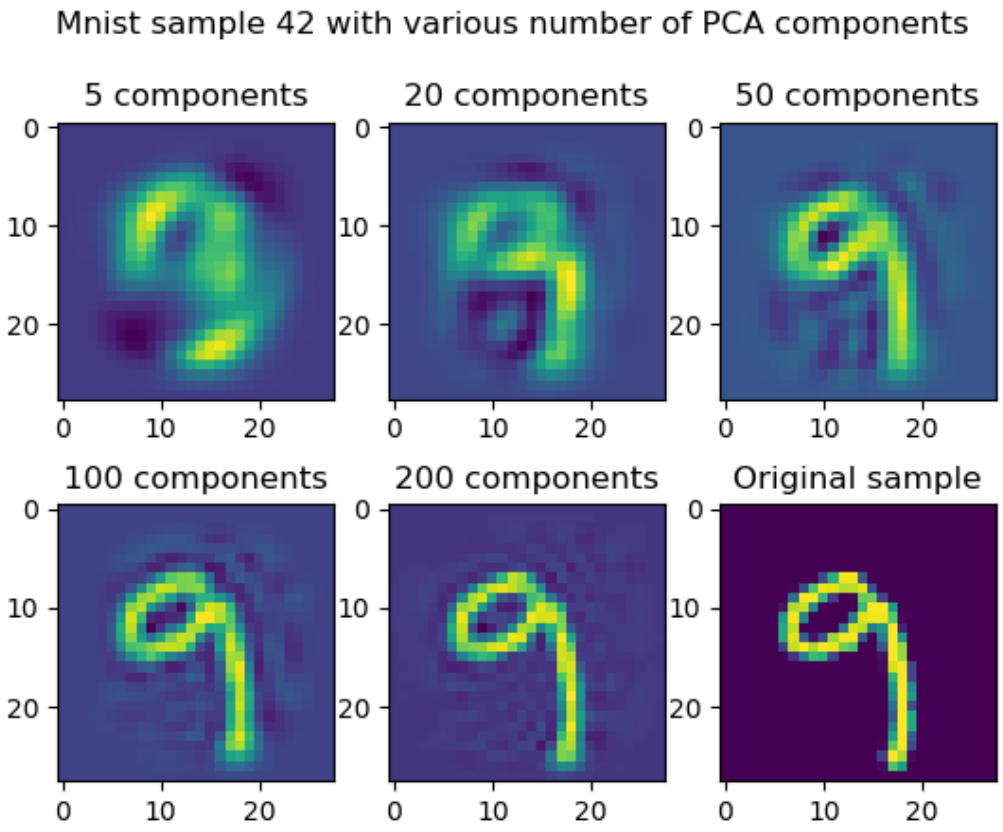


Figure 4: reconstruction of a MNIST sample

WORK: An interesting feature is `PCA::explained_variance_ratio_`. Explain these values according to your understanding of PCA and use these values to fit `n_components`

`PCA::explained_variance_ratio_` is a sorted array storing the ratio of information contained by each principal component. The cumulated sum of its elements gives the amount of information held by the first n principal components:

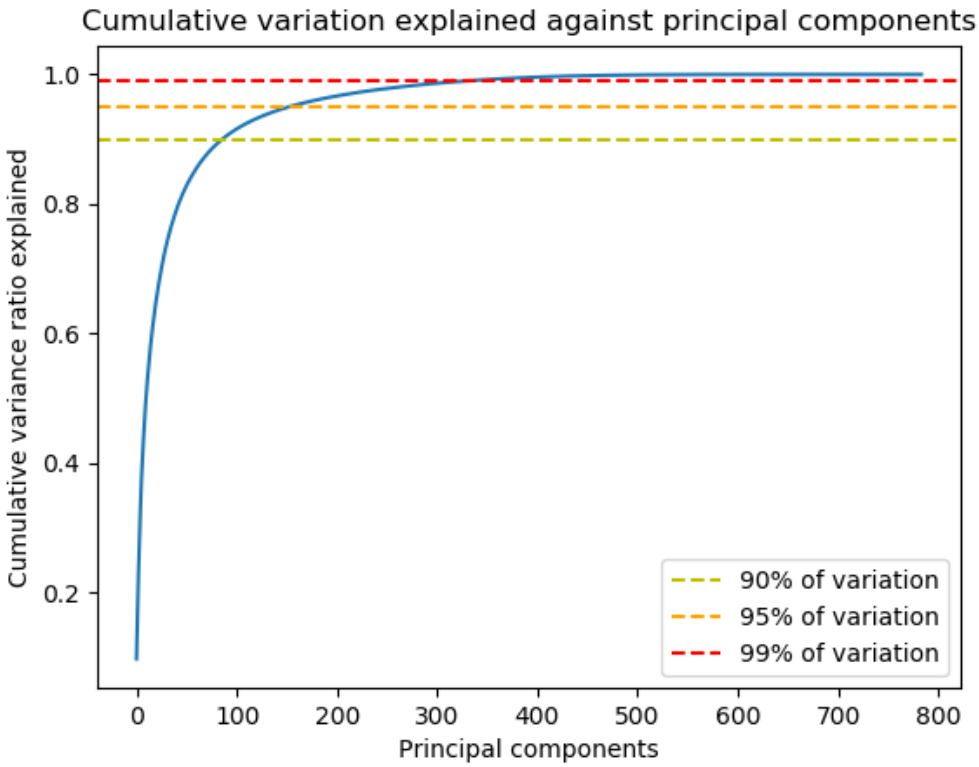


Figure 5: variance ratio explained

We can then use this array to reduce the dimension of the dataset with a given information loss, e.g., 5%. The code below does that and give the number of principal components required:

```
pca = PCA(0.95)
pca.fit(d2_X)
>> Number of components required to keep 95% of the information: 154
```

Using only 154 components (out of 784, about a fifth of the original dataset), we can achieve a 5% loss in information, thanks to PCA. We will use this value for training our algorithms when required.

II)2) Clustering

Clustering (i.e., classify groups among the data) is one of the main tasks of unsupervised machine learning. For this project, we use two different methods of clustering: K-means and Expectation-Maximization Gaussian Mixture.

WORK: Split X (and y) in a train/test sets with the sklearn method: `split_train_test`.

We keep the same training (80%) and test set (20%) that we made before. We need to reshape them for them to be used by the methods of K-means and Expectation-Maximization Gaussian Mixture.

WORK: With sklearn, perform K-MEANS. Play with the parameter K as well as the initialization (KMEANS++, random, or fixed array).

```
for ncluster in range(8,13):
  model = KMeans(n_clusters=ncluster, init='k-means++')
  kmeans = model.fit(d2_X_train)
  y_pred = kmeans.predict(d2_X_test)
```

The code above performs the K-Means with `ncluster` being the number of clusters we want to establish. `init` parameter can have several values:

- `k-means++` means sklearn will look for the best start centroids
- `random` means the `ncluster` starting centroids will be chosen at random.
- a fixed array allows the user to choose the centroids.

This parameter changes the results, as well as the time required to train the model: random or fixed centroids may lead to a slower convergence than those chosen by sklearn.

We then have to fit the model to our training set `d2_X_train`, and finally test it by predicting clusters in `d2_X_test`.

WORK: For the correct K, evaluate how good is this partition (with the knowledge of y).

Since we are predicting digits, the correct K is ten (there are 10 digits to be predicted). We can perform the K-Means on the whole train set. We chose the following metrics to evaluate the performance of our clustering:

- Completeness is satisfied if all the points of a given cluster belong to the same class
- Homogeneity is satisfied if every cluster contains only one class of data.
- V-measure is a synthesis (the harmonic mean) of the two previous metrics.

After fitting the model and making the predictions, we have the following results:

```
>> Evaluation of the clustering (K-Means, all information):
>> Homogeneity within clusters: 0.4990983487132732
>> Completeness score: 0.506252399437294
>> V-measure:  0.5026499200524681
```

The results are not already pretty accurate, but not yet good enough for any reliable application to be based on this process.

WORK: Using the PCA performed in section 2.1. apply K-MEANS with K=10 and `n_components` = 2. Display the partition and comment.

As seen in the results below, the two main components are not enough to grasp the complexity of the data: in earnest some classes tend to reunite (the handwritten digits are written in a similar way) so that K-Means is not appropriate, especially when considering only the two principal components. Furthermore, most classes appear to be badly located on the K-Means plot, probably due to a bad initialization (however, we did not find parameters to have a better result).

```
>> Evaluation of the clustering (K-Means, 2 principal components):
>> Homogeneity within clusters: 0.35623260677269564
>> Completeness score: 0.3624291942651569
>> V-measure:  0.3593041857996409
```

Figure 6: KMeans with 2 principle components

WORK: Briefly explain what the main difference between K-MEANS and EM with Gaussian Mixture is.

The K-means algorithm assigns every sample to a cluster, which is very straightforward and efficient in simple cases but does not acknowledge the complexity of the data. Expectation-Maximization with Gaussian mixture is a probabilistic clustering algorithm that is more nuanced than K-Means since it assigns every sample a probability to belong to each cluster.

WORK: Do the same job with the EM-clustering using the good K parameter (10 for MNIST). Comment your results.

Here are the results with an Expectation-Maximization Gaussian Mixture clustering. They are slightly better than K-means with 2 principle components but way worse than K-mean with all the information (and obviously nowhere near good enough for a reliable application). We were not able to display the probability for every sample to belong to each cluster. We can see that the clusters have a more flexible shape than with K-Means, but still do not match the real data: clustering is not a method powerful enough to classify the images of the MNIST dataset.

```
>> Evaluation of the clustering (EMGM, 2 principal components):
>> Homogeneity within clusters: 0.3619996777168655
>> Completeness score: 0.37152452566195515
>> V-measure:  0.36670026137932976
```

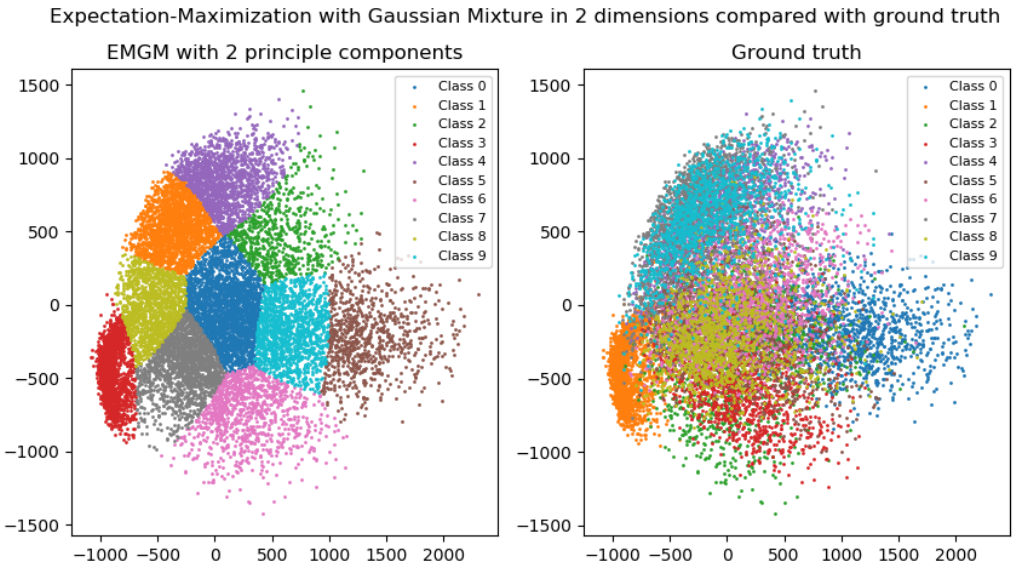


Figure 7: EMGM with 2 principle component

III) Supervised Machine Learning

Supervised machine learning consists of training a model knowing both the data and the ground truth for the training set.

III)1) Decision Tree, SVM and Logistic Regression

WORK: Why the train/test separation is compulsory?

We need to split the dataset in a training and test set so that our model does not develop biases when we evaluate it: if we train and test on the same set, the performance will be very high, but not replicable on other data input.

WORK: Why do we need to analyze the performance of the model at training and testing time?

As stated before, we must compute the score on both test and training dataset to ensure there is no bias in training: if the training error and the test error are too far away from each other (usually because training accuracy is very high), that may imply there is a flaw in the training process.

WORK: What is the major difference between Naive Bayes Classifier and Support Vector Machine (or Logistic Regression)?

Naive Bayes classifier is a generative model (it evaluates $P(X|y)$ to deduce $P(y|X)$) whereas SVM and logistic regression are discriminative models (they directly estimate $P(y|X)$).

For the two following questions, we chose to perform all four methods suggested, as it is a great exercise to compare them! Our observations for each method are reported below.

WORK: With sklearn, perform a classification using your favorite methods. With the documentation, check how to modify the parameters and comment how it influences the results

WORK: With the score method, compute the accuracy of the model on the training and the test datasets.

III)1)i) Decision Tree

For decision trees, the most relevant parameter to modify is the depth (maximum number of nodes between the root and a leaf). It mostly influences the run time and the accuracy (a deeper tree is longer to fit but more accurate on the training set). We first trained a model without restricting the depth of the tree ("unrestricted model" below), then using a variety of imposed depths ("restricted model"). Here are the results for the unrestricted model:

```
>> Model tree classifier: unrestricted model
>> Depth:  44, Number of leaves:  3806
>> Accuracy on the training set:  1.0
>> Accuracy on the test set:  0.8727857142857143
```

We then compare this unrestricted model to models restricted in depth. The number of leaves (actually log2 of this number, because decision trees are binary trees) and accuracy seemed to be the most relevant metrics to evaluate.

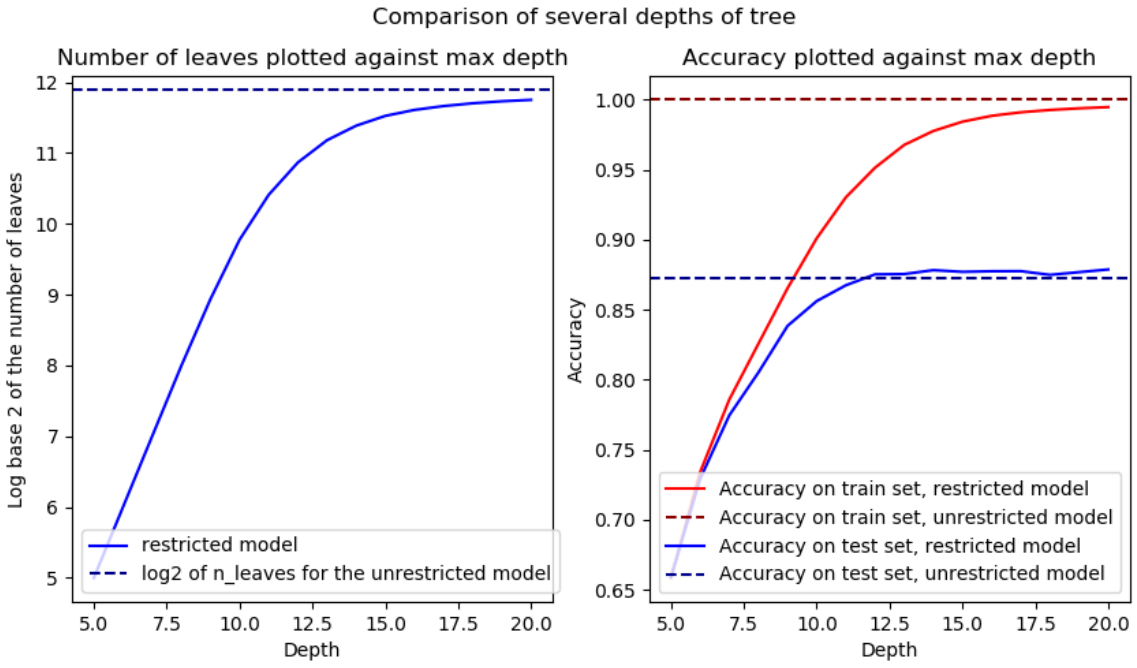


Figure 8: comparison of tree depth

We observe a perfect correlation between the number of leaves and the accuracy on the training set, because it is the way the tree is built. We also observe that the decision tree model reaches its peak accuracy on the test set with a depth slightly greater than 10 and stagnate for depths beyond. That means that any depth beyond 10 (including our first unrestricted model with a depth over 40) is a form of overfitting.

Using a decision tree is more accurate than both clustering methods we used earlier, and more easily interpreted as well, since a decision tree can consider all the 784 dimensions of the images and still be interpreted, whereas we need to consider only the two or three most relevant dimensions for clustering to be readable. It is still to be perfected though, because the accuracy on the test set is lower than 90%.

III)1)ii) Support Vector Machine (SVM)

The principle of support vector machine for classification is to find an hyperplane in the data that splits the data in two: those "above" and those "under" the hyperplane (be careful with the notion of above and under in dimension 784). For multi classes problems, the algorithm must work out as many hyperplanes as there are pair of classes (one versus one approach).

The two most important parameters for SVM are the kernel's shape and the regularization parameter C. C is the proportional to the inverse of the regularization strength. A high value of C means the model will sacrifice flexibility in favor of training accuracy.

SVM is one of the most appreciated machine learning method that is not deep learning but suffers one major drawback here: its complexity. It requires to compute the distance between every pair of two samples, which means it is at least quadratic regarding the number of samples (the number of classes is irrelevant here, as it is not modified by our method). Let m be the number of features (i.e., the dimensionality of the problem) and n the number of samples, the complexity of the fitting is thus: $O(\mathrm{m} \times \mathrm{n}^2)$.

It took way over an hour to fit the model, so we implemented two methods there: PCA and K-cross validation. For the PCA, we used the number of components computed earlier to keep 95% of the information, reducing the dimensionality so that $\mathrm{m}' \approx \frac{m}{5}$ (154 components out of 784). We also used K-cross validation with K=20, so that the new number of training samples is $\mathrm{n}' = \frac{n}{K}$ meaning that we have less data available to fit a model, but we are going to fit K models with different parameters. The total complexity is finally:

$$\mathrm{K} \times O(m'^2 \times n'^2) = \mathrm{K} \times O\left(\frac{m}{5} \times \left(\frac{n}{K}\right)^2\right) = \frac{1}{5K} \times O(m \times n^2)$$

Using PCA and cross validation, we made out fitting a hundred times faster that it would have been otherwise. But was it worthwhile?

```
>> Best SVM: kernel poly, C=1,
>> Accuracy on validation set:  0.9531954887218045
>> Accuracy on test set:  0.9499285714285715
```

The cross validation was used to find the best kernel among gaussian kernel, polynomial kernel, linear kernel, or sigmoid kernel. The parameter C was also selected thanks to the cross-validation. With the model fitted, the results are pretty good with almost 95% accuracy on the test set, and do not seem to present overfitting.

III)1)iii) Logistic Regression

Logistic regression is widely used for classification. The most relevant parameter to modify while fitting a logistic regression is the penalty mode: penalty can use L1 or L2 norm. So we tested both. We actually had a problem about convergence not being achieved, even after increasing the number of iterations (up to 1000 instead of 100) and changing the solver (lbfgs for L2, liblinear for L1), although that does not seem very detrimental to the model.

```
>> Logistic regression (penalty L1):
>> Accuracy on train set:  0.9342321428571428
>> Accuracy on test set:  0.9132857142857143
>>
>> Logistic regression (penalty L2):
>> Accuracy on train set:  0.9432321428571429
>> Accuracy on test set:  0.913
```

Both penalties seem to perform quite the same.

III)1)iv) Naive Bayes Classifier

Naive Bayes Classifier is really easy to implement, and very fast to fit. The principle is to train the model to evaluate $P_{y_{train}}(X_{train}) = \mathrm{k}$ for each class k, then using Bayes formula ($\mathrm{P(A)} \times \mathrm{P_A(B)} = \mathrm{P(A \cap B)} = \mathrm{P(B)} \times \mathrm{P_B(A)}$) the model predicts the most probable class, i.e. $\max_{\mathrm{k \in classes}}\left(\mathrm{P_{X_{test}}(y_{test}) = k}\right)$.

```
>> Naive Bayes Classifier:
>> Accuracy on train set:  0.5664642857142858
>> Accuracy on test set:  0.566
```

This model is pretty simple, but makes a strong assumption: the probability $\mathrm{P_{y_{train}}(X_{train})}$ follows a normal distribution. Considering the poor results Naive Bayes Classifier gives us, we can conclude that (as we saw on the clustering graphs) the samples of the MNIST dataset are not normally distributed.

WORK: In section 2.1. you applied a PCA to X so that the projected set – hereafter Xred – lies on a "reduced" space. Among the supervised methods you chose, select one method and apply your code to Xred. Does the PCA influence the performance of the classification (according to the intensity of the reduction)?

Since we had no point of comparison when we used PCA on SVM, we now use PCA on decision trees. This is a particular model, because of its tendency to overfit, so we wanted to know the effects of PCA on such a model. We tried different configurations. Here are the results:

% information	Tree depth	N° of leaves	Accuracy train set	Accuracy test set
0.66	43	4947	1.0	0.84992
0.8	46	4670	1.0	0.84514
0.9	49	4582	1.0	0.8375
0.95	53	4501	1.0	0.828

The results are pretty interesting. Of course, the model clearly overfits regardless of PCA or not, but when considering the most relevant components, the accuracy on the test set is increased: the models overfits less, although the number of leaves is increased. That is quite unintuitive.

III)2) Deep Learning

III)2)i) Objective and tool

To build the neural networks used in this part of the report, we used the functional API of Tensorflow Keras. This API is intuitive and less verbose than its sequential counterpart. The code usually looks like that: a plug-in sequence to define the layers, then the methods `compile` and `fit`, and ultimately the `evaluate` method.

```
inputs = keras.Input(shape=(d2_X_train.shape[1], ))
x = Dense(32, activation='relu') (inputs)
outputs = Dense(10, activation='softmax') (x)
mlp = Model(inputs, outputs, name="MLP0")
mlp.compile( … ) #details about compile below
mlp_hist=mlp.fit(d2_X_train,y_train,epochs=10, validation_split=0.2)
mlp_score = mlp.evaluate(d2_X_test, y_test, verbose=0)
```

We will test two types of neural networks: the MultiLayer Perceptron (MLP) and convolutional Neural Networks (CNN).

III)2)ii) Multilayer Perceptron (MLP)

The MLP is a feed forward fully connected neural network.

WORK: What is the size of the input tensor? What is the size of the output layer?

MLP does not consider spatial representation of the data, an image is represented by a 784 array. The input tensor is thus a $N \times 784 \times 1$ with N being the number of samples.

The output layer must have one neuron per possible outcome. Since we are predicting digits, there are ten possible outcomes, the output has ten neurons.

WORK: How many epochs do you use? What does it mean? What is the batch_size? What does it mean? Why do we define a validation set (for example: validation_split=0.2)? What kind of parameters do we set with the compile method? And with the fit method? For each one, explain why it is an important parameter and what is the influence on the training process?

An epoch is a training phase during which all the data available for training has been used once. Neural networks with a great capacity (lot of neurons per layer, many layers) will need less epochs to achieve a great performance, but epochs will last longer in time. Training on this dataset could last 10 to 30 epochs approximately, depending on the model.

The batch size is the number of samples to be processed through the model between each update of the model internal parameters. Defining a low batch size will make the model training longer, but allegedly more effectively.

The validation set is used to evaluate the performance of the training after every epoch. The data of the validation set is not used for training.

The parameters of the compile method are the loss and the optimizer used by the model (Sparse categorical cross entropy and Adam for this project) and the metrics we want to evaluate the model on (we chose to evaluate the accuracy of the model). These parameters ultimately define the ability of the model to perform a given task.

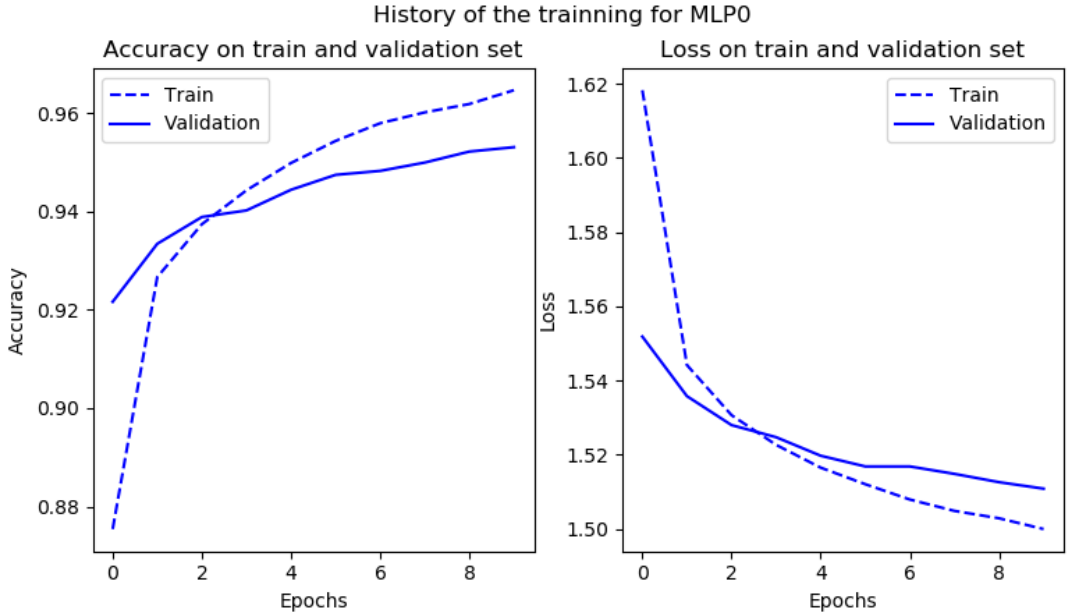
The parameters of the fit method are the training data and their labels (because deep learning is supervised learning), the proportion of training data that will be used as validation set and the number of epochs during which the model will be trained. This last parameter is very important, because the longer we train a model, the better it will perform on the training set (and up to a certain point, on the validation and test set too). Note that this method returns the historic of the training.

WORK: Comment the training and the results.

The first model we build (named `MLP0`) is quite simple, with only one hidden layer:

`Model: "MLP0"`		
Layer type	**Output shape**	**Param #**
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 32)	25 120
dense_1 (Dense)	(None, 10)	330

Total params	25 450
Trainable params	25 450
Non-trainable params	0




```
>> MLP0: Test loss: 1.513702392578; Test accuracy: 0.9505000114440
```

With about 95% accuracy on the test set, this model is already quite good, but could clearly be better: we are underfitting. That can be solved by adding trainable parameters (e.g. adding layers) and/or training longer.

WORK: Is there any overfitting? Why? If yes, what could be the causes? How to fix this issue?

There is no overfitting yet: the gap between train and validation performance is explained by the lack of capacity of the model. It is not overfitting, as the performance on validation set was still progressing after 10 epochs.

WORK: If you do not observe overfitting, how can you make your model overfit? Try and demonstrate the overfitting.

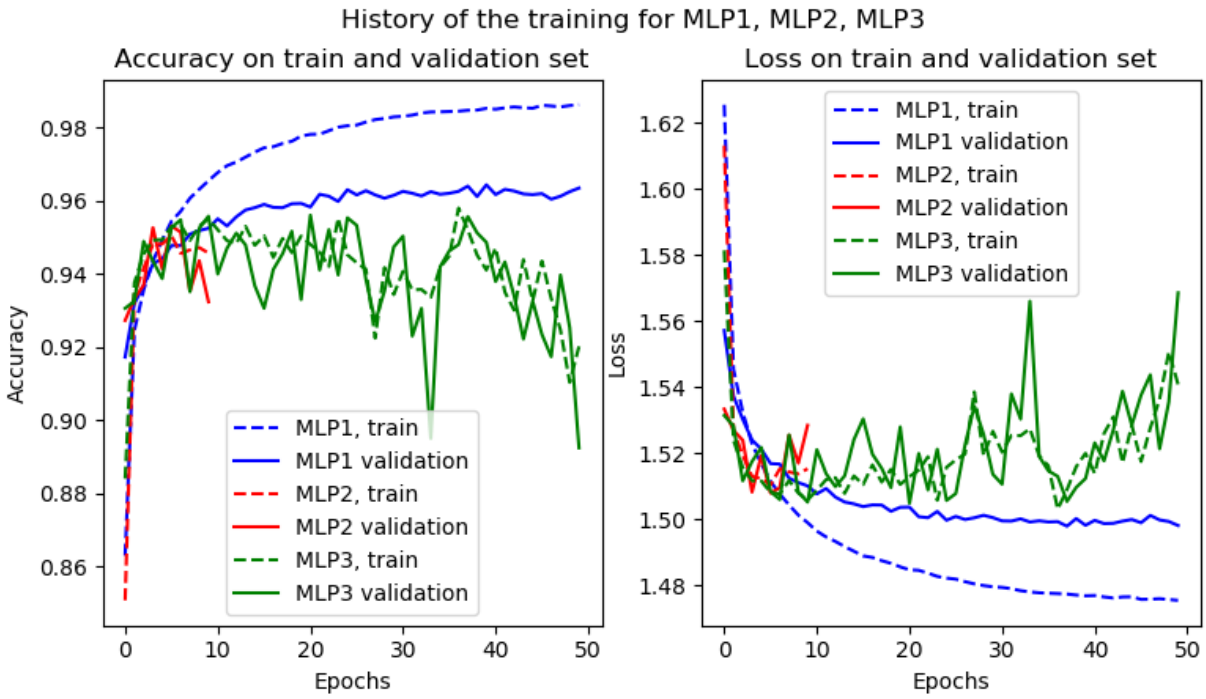
As said before, adding trainable parameters (either by adding layers or increasing the number of neurons in current layers) or training longer will improve the results of the model up to a certain point, beyond which the model overfits.

We can try to demonstrate overfitting with three models: one that will be trained too long, another that will have too much capacity and the last that would combine both.

* `MLP1` has the same structure as `MLP0` but is trained during 50 epochs (instead of 10).

* `MLP2` has more parameters (4 hidden layers with 128 neurons each) than `MLP0` and is trained during 10 epochs too.

* `MLP3` has the same structure as `MLP2` and the same training time as `MLP1`.




```
>> MLP1: Test loss: 1.5010445117950; Test accuracy: 0.9604285955429
>> MLP2: Test loss: 1.5360097885131; Test accuracy: 0.9248571395874
>> MPL3: Test loss: 1.5732859373092; Test accuracy: 0.8878571391105
```

`MLP1` overfits a bit: while the train accuracy continuously increases, the validation accuracy hits a threshold at 96%: we train the model for too long. `MLP2` has way too much capacity, so much that the model breaks: it cannot learn properly. This is an extreme case of overfitting. It is even worse with `MLP3` that has both problems: the model is beyond broken and does not seem to learn anything.

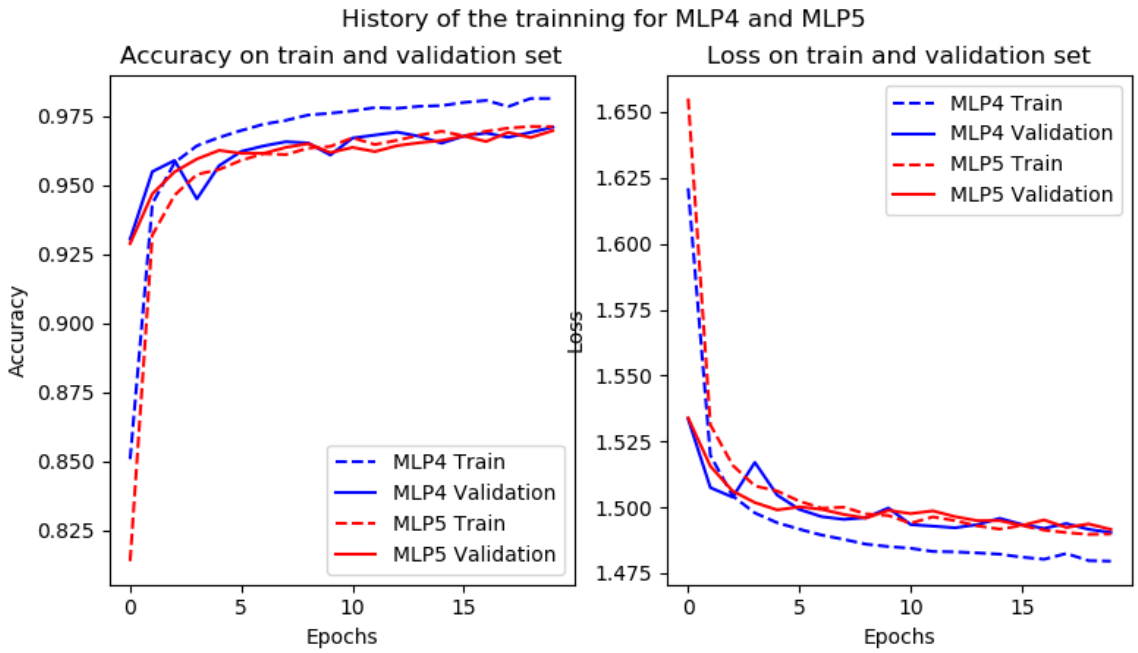
WORK: According to this first performance, change the architecture of the MLP (change parameters, add/remove layers…) as well as hyper-parameters, explain why, what are the influence on the results…?

Now we need to find the right training time and structure to obtain better results. We need to have more than 1 hidden layer, but 4 were too much. 50 epochs were too much, and 10 not so bad: maybe around 20 epochs is the way to go. After a few attempts, we came up with this model:

Model: "MLP4"		
Layer type	**Output shape**	**Param #**
input_5 (InputLayer)	[(None, 784)]	0
dense_14 (Dense)	(None, 128)	100 480
dense_15 (Dense)	(None, 64)	8 256
dense_16 (Dense)	(None, 32)	2 080
dense_17 (Dense)	(None, 10)	330

Total params	111 146
Trainable params	111 146
Non-trainable params	0

And MLP5 that has the same cone structure as MLP4 except it has a dropout of 15% between the dense layers, as a mean to limit overfitting and regularize the training. Here are the results:




```
>> MLP4: Test loss: 1.49190425872802; Test accuracy: 0.96928572654724
>> MLP5: Test loss: 1.49315071105957; Test accuracy: 0.96792858839035
```

Both models perform quite good, with more than 96% of accuracy, but the dropout was not useless: MLP4 presents some overfitting (training accuracy is substantially better than validation accuracy), which is not the case with MLP5.

III)2)iii) Convolutional Neural Network (CNN)

WORK: Why this type of neural networks is (probably) more relevant for our task? Is the input tensor size the same as for MLP? Why?

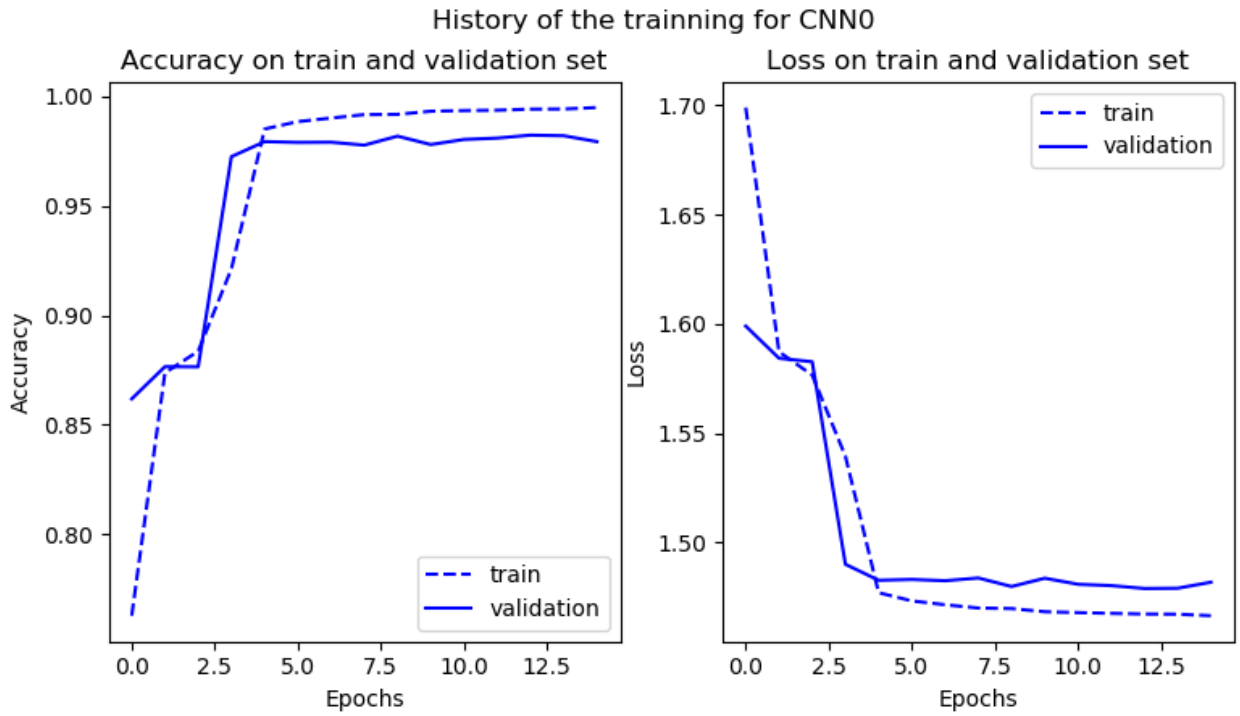
Unlike MLP, CNN acknowledge for the spatial organization of data (such as pixels in an image), making them more relevant than MLP for image classification. Unlike MLP (for which we flattened every 28×28 image in a 784 array), CNN process raw images. The images are grayscale (1 channel) 28×28 pictures: the size of the input tensor is then $N \times 28 \times 28 \times 1$ with N being the number of samples.

WORK: Build a first CNN model with only one convolutional block. Comment the training and the results.

For the first model, we define only one convolutional layer with 64 filters.

Model: "CNN0"		
Layer type	**Output shape**	**Param #**
input_7 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 26, 26, 64)	640
flatten (Flatten)	(None, 43264)	0
dense_22 (Dense)	(None, 64)	2 768 960
dense_23 (Dense)	(None, 10)	650

Total params	2 770 250
Trainable params	2 770 250
Non-trainable params	0




```
>> CNN0: Test loss: 1.48226356506347 Test accuracy: 0.9788571596145
```

The results are already pretty good with almost 98% accuracy, but there is a slight overfitting that we will try to fix.

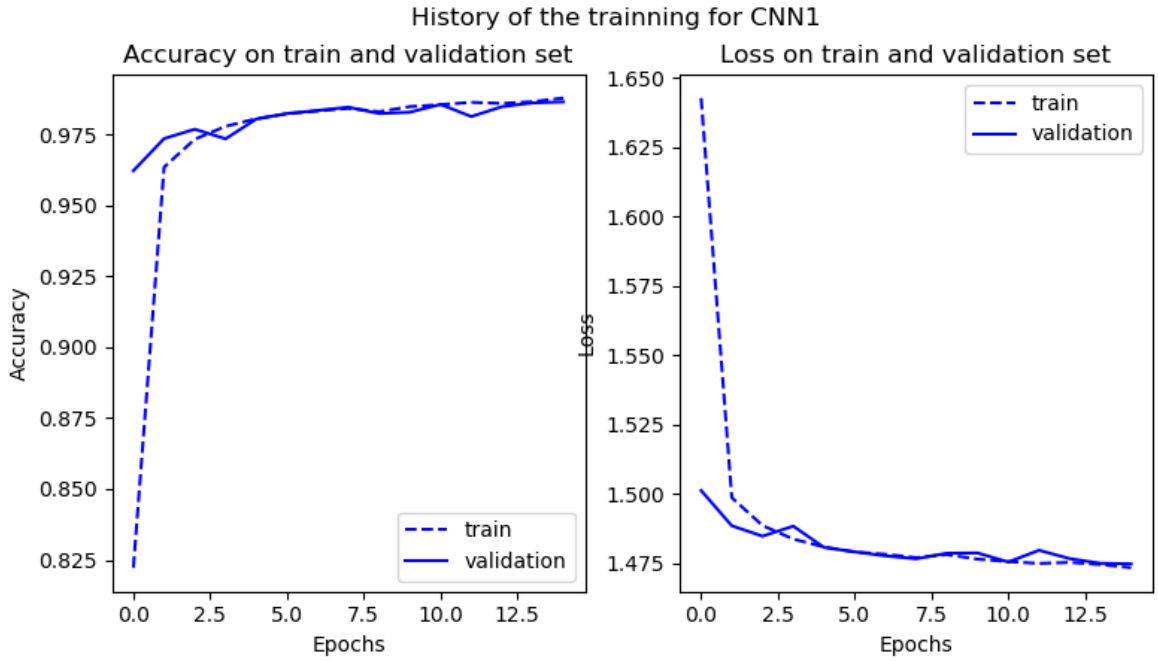
WORK: Change the architecture of the CNN (change parameters, add layers…) as well as hyper-parameters, explain the influence on the results

We change the model by adding convolutional layers, thus increasing pattern recognition.

Model: "CNN1"		
Layer type	**Output shape**	**Param #**
input_8 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 3, 3, 32)	18464
flatten_1 (Flatten)	(None, 288)	0
dense_24 (Dense)	(None, 64)	18 496
dropout_3 (Dropout)	(None, 64)	0
dense_25 (Dense)	(None, 10)	650

Total params	56 426
Trainable params	56 426
Non-trainable params	0

The use of multiple MaxPooling2D layers allowed to drastically decrease the number of parameters required for the model.




```
>> CNN1: Test loss: 1.47617197036743; Test accuracy: 0.98485714197158
```

The results are better: the accuracy is improved and there is almost no overfitting left. This is our best model so far, although it could certainly be improved: we still have much to learn about the creation of neural networks.

Conclusion

Our project was to classify image of handwritten digits contained in the dataset MNIST. To do so, we used several methods of unsupervised and supervised machine learning. Our results are in the table below: (PC stands for “principal component”, HL for “hidden layer”, CL for “convolutional layer”)

Model	Parameters	Accuracy (train)	Accuracy (val)	Accuracy (test)
K-means	10 clusters, All information	-	-	0.50264992005
K-means	10 clusters, 2 PC	-	-	0.35930418579
EM Gaussian Mix	10 clusters, 2 PC	-	-	0.366700261379
Decision Tree	Unrestricted max depth, All information	1.0	-	0.87278571428
Decision Tree	Max depth: 15, All information	0.9844285714	-	0.87714285714
SVM	K-cross validation with 20 folds, 154 PC, Kernel poly, C=1	-	0.95319548872	0.94992857142
Logistic Regression	Penalty L1, All information	0.93423214285	-	0.91328571428
Logistic Regression	Penalty L2, All information	0.94323214285	-	0.913
Gaussian Naïve Bayes	All information	0.56646428571	-	0.566
Decision Tree	Unrestricted max depth, 66% information	1.0	-	0.84992857142
MLP	1HL, 25450 params, 10 epochs	0.96462053060	0.95303571224	0.95050001144
MLP	1 HL, 25450 params, 50 epochs	0.98624998331	0.9634821414	0.96042859554
MLP	4 HL, 151306 params, 10 epochs	0.94571429491	0.9324107170	0.92485713958
MLP	4 HL, 151306 params, 50 epochs	0.92022323608	0.89249998331	0.88785713911
MLP	3 HL, 111146 params; 20 epochs	0.98147320747	0.97107142210	0.96928572654
MLP	3 HL, 111146 params; 20 epochs, dropout 15%	0.9712946414	0.96982145309	0.96792858839
CNN	1 CL, 2770250 params, 15 epochs	0.99484372138	0.9792857170	0.9788571596
CNN	3 CL, 56426 params, 15 epochs, dropout 10%	0.98962050676	0.98562502861	0.98485714197

This project was interesting, although the results we display here can certainly be improved. As our first work with high-dimensional dataset, it gave us insights about how to process such quantity of data