École des Mines de Saint-Etienne

Computer Science
Practical Session
Report

Machine Learning Report

Students :
Hugo Besançon
Vincent Lalanne

Teacher :
Pierre-Alain Moellic

31 janvier 2023

Table des matières

1 The dataset : MNIST **2**
1.1 Shape of the data . . . 2
1.2 Display of samples from the dataset . . . 2
1.3 Dataset split . . . 3
1.4 Importance of the test/train split . . . 4
1.5 Balance of train and test sets and its importance . . . 4

2 Unsupervised Machine Learning **5**
2.1 Principal Component Analysis (PCA) . . . 5
2.2 Explained Variance Ratio . . . 5
2.3 Display of some MNIST pictures with different values of n_components . . . 8
2.4 Data clustering . . . 9

3 Supervised Machine Learning **12**
3.1 Difference between Naïve Bayes Classifier and Support Vector Machine . . . 12
3.2 Classification using SVM . . . 12
3.3 SVM and Gaussian Naïve Bayes comparison . . . 14
3.4 Logistic Regression . . . 14
3.5 Decision Tree . . . 16
3.6 Need of analyzing the performance of the test and train sets . . . 18
3.7 VSM classification after PCA . . . 18
3.8 MultiLayer Perceptron . . . 20
3.9 Convolutional Neural Network . . . 22

1 The dataset : MNIST

1.1 Shape of the data

The shape of a dataset refers to the number of rows and columns it has. It is represented by a tuple of the form (`n_samples`, `n_features`), where `n_samples` is the number of rows (also known as the number of samples or observations) and `n_features` is the number of columns (also known as the number of features). In our case, the shape is (70000, 784).

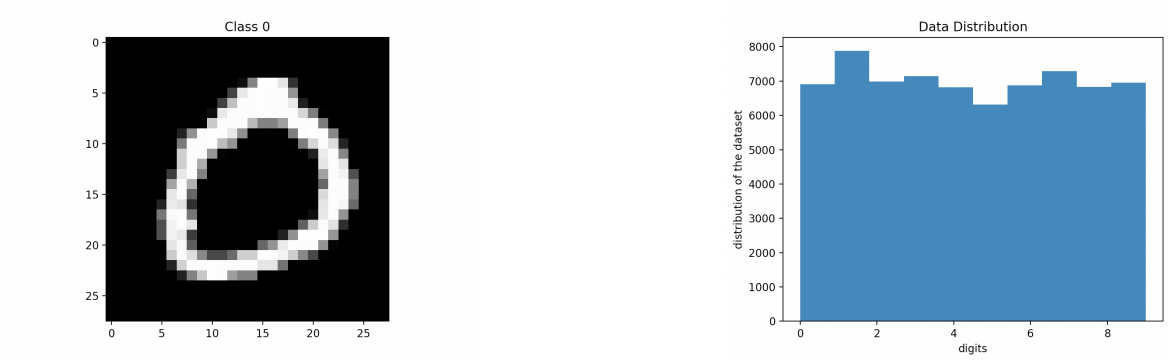
The shape of a dataset determines the size of the dataset and the number of features that can be used to train a machine learning model. The size of the dataset impact the computational efficiency of the model, and the number of features impact the model's ability to learn from the data.

1.2 Display of samples from the dataset

We use the following Python code to import and scale the dataset, and display 5 random samples from it.

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split

data_X = np.load('MNIST_X_28x28.npy')
data_Y = np.load('MNIST_y.npy')
data_X_1d = data_X.reshape(len(data_X), -1)

# scale data 0 1
data_X_1d = data_X_1d/255.0

# plot 5 random images
p = np.random.permutation(len(data_X))
p = p[:5]
for i in p:
    plt.imshow(data_X[i], cmap='gray')
    img_title = 'Class ' + str(data_Y[i])
    plt.title(img_title)
    plt.show()

# plot the distribution of the dataset
plt.hist(data_Y)
plt.xlabel('digits')
plt.ylabel('distribution of the dataset')
plt.title('Data Distribution')
plt.show()
```

An example of output is given in the subfigure 1a, and the dataset's distribution is shown by the subfigure 1b.



(a) Sample from the dataset (b) Distribution of the dataset

FIGURE 1 – Dataset samples display

1.3 Dataset split

The following code performs the split of the dataset using the sklearn method `train_test _split`. It results in the creation of training and test sets. The goal is to create separate sets of data for training and testing a machine learning model (80% train, 20% test). The training set is used to fit the machine learning model, and the test set is used to evaluate the model's performance.

```
X_train, X_test, y_train, y_test = train_test_split(data_X_1d, data_Y,
    test_size=0.2, random_state=42)

# print the size of the train and test set to check if the split was
    done correctly (80% train, 20% test)
print('X_train size: '+ str(len(X_train)))
print('X_test size: '+ str(len(X_test)))
print('y_train size: '+ str(len(y_train)))
print('y_test size: '+ str(len(y_test)))
```

The output of the algorithm is :

```
X_train size: 56000
X_test size: 14000
y_train size: 56000
y_test size: 14000
```

The `train_test_split` function takes in the following arguments :

- `data_X_1d` : the feature data that we want to split.
- `data_Y` : the target labels associated with the feature data.
- `test_size` : the proportion of the dataset that we want to use for testing. The default value is 0.25.
- `random_state` : a seed for the random number generator. This is used to ensure that the results are reproducible.

The function returns four arrays :

— `X_train` : the feature data for the training set.
— `X_test` : the feature data for the test set.
— `y_train` : the target labels for the training set.
— `y_test` : the target labels for the test set.

1.4 Importance of the test/train split

The test and train sets are used to evaluate the performance of a machine learning model. The idea is to use the training set to train the model and the test set to evaluate its performance on unseen data.

It is important to split the data into a training and test set because it allows to evaluate the performance of the model on unseen data. If we only trained the model on the same data that we test it on, it is likely to perform very well on that data, but may not generalize well to new, unseen data. This is known as **overfitting**.

By evaluating the model on the test set, we can get a better idea of how well the model will perform on new, unseen data. This is especially important when we want to use the model to make predictions on new data in the future.

1.5 Balance of train and test sets and its importance

It is important to ensure that the training and test sets are well balanced, meaning that they have a similar distribution of labels. If the sets are not well balanced, then the classifier may not perform well because it is unable to learn from a representative sample of the data. In other words, if the training set is heavily skewed towards a certain class, the classifier may be biased towards that class and not perform well on other classes.

It is important to ensure that the training and test sets are well balanced in machine learning for a few reasons :

1. **Bias :** As mentioned earlier, if the training set is heavily skewed towards a certain class, the classifier may be biased towards that class and not perform well on other classes. This can lead to poor generalization to new, unseen data.
2. **Overfitting :** If the training set is not well balanced, the classifier may overfit to the dominant class and not learn enough about the minority class. This can also lead to poor generalization.
3. **Evaluation :** If the test set is not well balanced, it may be difficult to accurately evaluate the performance of the classifier. For example, if the test set only contains examples from the dominant class, the classifier may achieve a high accuracy simply by predicting the dominant class all the time. This would not be a fair evaluation of the classifier's true performance.

As our experiments show, there is no one-size-fits-all answer to the question of what the perfect balance is for the training and test sets in machine learning. The appropriate balance will depend on the specific problem we try to solve and the size of the dataset. By reducing the size of the dataset, we need to use a smaller test set to ensure that we have enough data for training. On the other hand, with a large dataset, we may be able to use a larger test set without sacrificing the size of the training set.

In general, it appears to be a good idea to use a large enough training set to allow the machine learning model to learn effectively, but also to reserve enough data for the test

set to accurately evaluate the model's performance. Using cross-validation, we observed that a split of 70/30 or 80/20 (training set/test set) offers a better performance of the model.

2 Unsupervised Machine Learning

2.1 Principal Component Analysis (PCA)

Among all the features a picture possess, not all are useful for the machine. Thus, instead of computing all the features, we will choose only the best features (the ones with the most of information), and then use them to make the computing faster. The following code performs a PCA test :

```
from sklearn.decomposition import PCA

# create a PCA object
pca = PCA(n_components=2)

# fit the PCA object to the train set
pca.fit(X_train)

# transform the train set and the test set
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

It is up to us to choose the number of features to work with, the more we have the best (in term of prediction) the machine will be, but the slowest it will become.

2.2 Explained Variance Ratio

`explained_variance_ratio_` is an attribute of the PCA object that gives the percentage of variance explained by each of the selected components. It is a measure of the importance of each principal component.

For example, if we fit a PCA model on a dataset with 10 features and set the number of components to 2, `explained_variance_ratio_` will be an array of length 2, where each element represents the percentage of variance explained by each of the selected components. The sum of all the elements in the array will be equal to 1.

The `explained_variance_ratio_` can be used to determine the number of components to keep in the model.

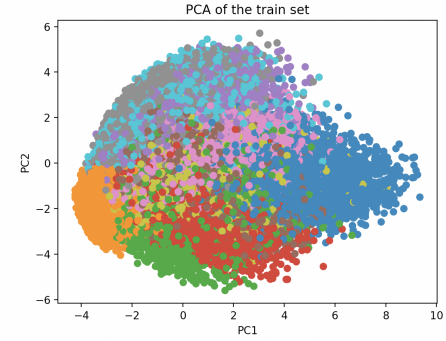
The following code displays pertinent graphs :

```
# plot the first 2 principal components of the train set
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train,
↪  cmap='tab10')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('PCA of the train set')
plt.show()
```

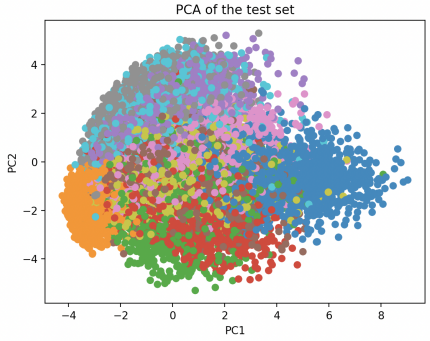
```
7
8  # plot the first 2 principal components of the test set
9  plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test,
   ↪  cmap='tab10')
10 plt.xlabel('PC1')
11 plt.ylabel('PC2')
12 plt.title('PCA of the test set')
13 plt.show()
14
15 # plot the explained variance ratio of the PCA
16 plt.plot(pca.explained_variance_ratio_)
17 plt.xlabel('Principal Component')
18 plt.ylabel('Explained Variance Ratio')
19 plt.title('Explained Variance Ratio of the PCA')
20 plt.show()
21
22 # plot the cumulative explained variance ratio of the PCA
23 plt.plot(np.cumsum(pca.explained_variance_ratio_))
24 plt.xlabel('Principal Component')
25 plt.ylabel('Cumulative Explained Variance Ratio')
26 plt.title('Cumulative Explained Variance Ratio of the PCA')
27 plt.show()
```

The output is shown in figure 2.

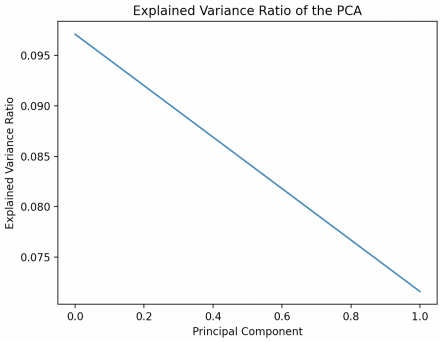
In order to know how many features to work with, figure 3 shows the value of the variance (how many information) as a function of the number of components.



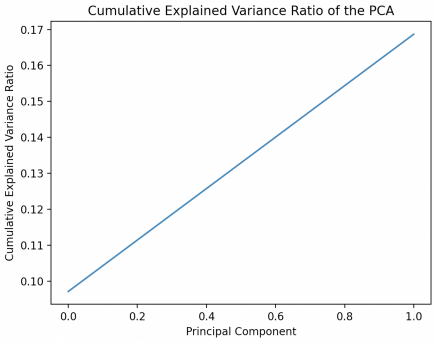
(a) PCA of the train set



(b) PCA of the test set



(c) Explained Variance Ratio



(d) Cumulated Explained Variance Ratio

FIGURE 2 – Output of the PCA code

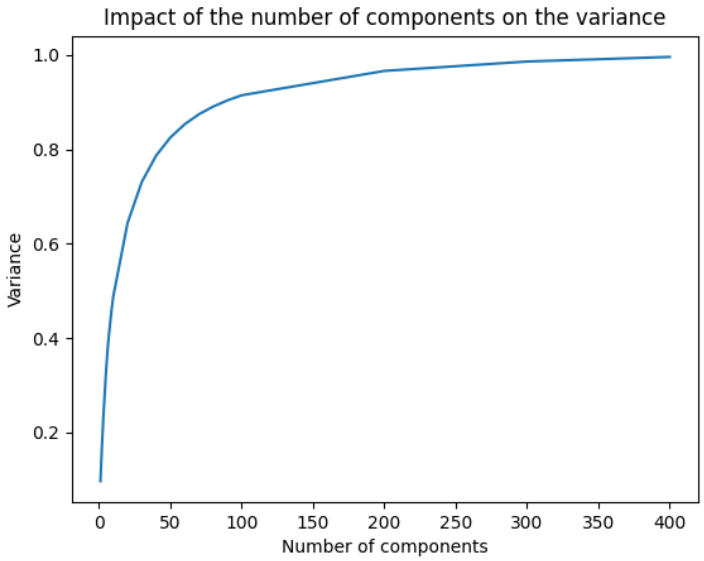


FIGURE 3 – Variance following n_components

2.3 Display of some MNIST pictures with different values of n_components

Because we reduce the number of information, we need to make sure that the picture is still recognizable. Thus, we have to find a limit of the minimum of information to keep.

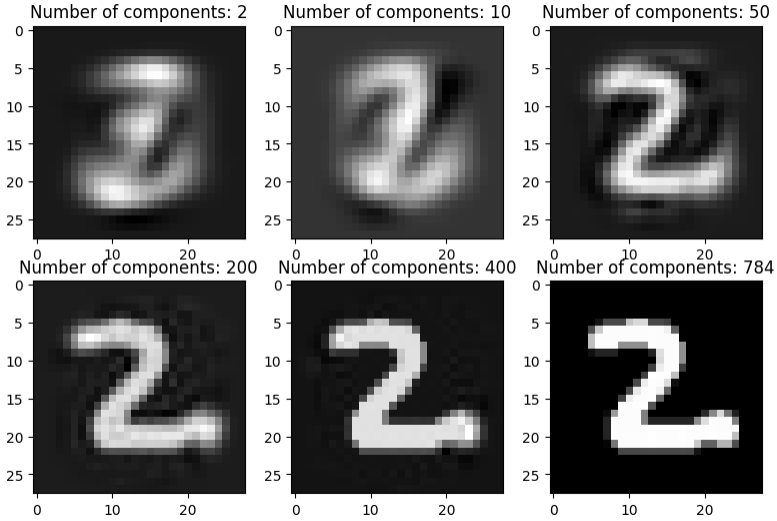


FIGURE 4 – Comparison of a sample with different n_components

The figure 4 is an MNIST sample with different values of `n_components`. We can see that the more components, the more "realistic" the picture. Thus, it is impossible to work with a 2 or 10 dimensions data for MNIST pictures, however, it may be a good idea to work with a 50 or 200 dimensions data.

2.4 Data clustering

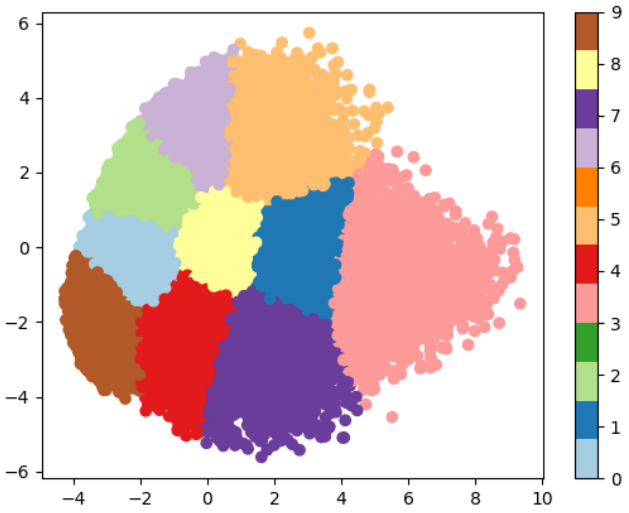


FIGURE 5 – Partition with K = 10 and n_components = 2

The figure 5 shows that after a PCA with `n_components` = 2 (working in $\mathbb{R}^2$), all the clusters are clearly differentiated. However, because there is no information, this is not relevant (the variance sum is near 0).

The following code cluster the data X with K-MEANS and EM with Gaussian mixture. It displays pertinent graphs as well.

```
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture

# perform PCA on the data with 300 components
pca = PCA(n_components=300)
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)

# perform a clustering on the data using KMeans
kmeans = KMeans(n_clusters=100, random_state=0).fit(X_train)

# perform a clustering on the data using GaussianMixture
gmm = GaussianMixture(n_components=100, random_state=0).fit(X_train)

# predict the labels of the test data
y_pred_kmeans = kmeans.predict(X_test)
y_pred_gmm = gmm.predict(X_test)

# calculate the homogeinity score, completeness score and v-measure
    ↪  score
```

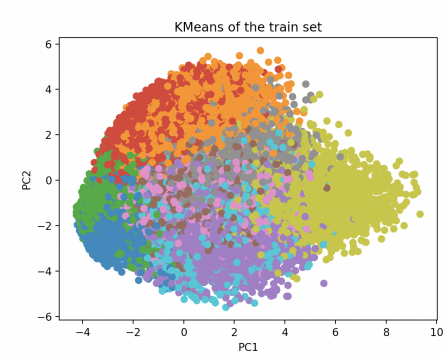
```
21 from sklearn import metrics
22 print("KMeans")
23 print("Homogeneity: %0.3f" % metrics.homogeneity_score(y_test,
    ↪ y_pred_kmeans))
24 print("Completeness: %0.3f" % metrics.completeness_score(y_test,
    ↪ y_pred_kmeans))
25 print("V-measure: %0.3f" % metrics.v_measure_score(y_test,
    ↪ y_pred_kmeans))
26
27 print("GaussianMixture")
28 print("Homogeneity: %0.3f" % metrics.homogeneity_score(y_test,
    ↪ y_pred_gmm))
29 print("Completeness: %0.3f" % metrics.completeness_score(y_test,
    ↪ y_pred_gmm))
30 print("V-measure: %0.3f" % metrics.v_measure_score(y_test,
    ↪ y_pred_gmm))
```

A more visual output of the algorithm is shown by figure 6 for the test and train set. The results for homogeneity, completeness and v-measure are given in the following table :

	Homogeneity	Completeness	V-Measure
KMeans	0.812	0.409	0.544
EM with Gaussian Mixture	0.756	0.414	0.535

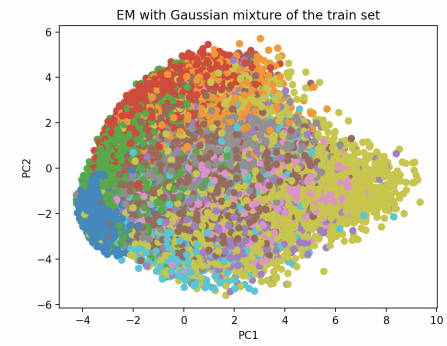
Also, we get a computation time of 30 minutes, which is, in the frame of this practical session, quite high. However, as KMeans is very greedy as it computes the distance of each point from one another, it is not surprising. Concerning the interpretation of the scores, a score of 1.0 for the homogeneity means that all of the clusters contain only data points which are members of a single class. A completeness score of 1.0 means that all the data points of a given class are assigned to the same cluster. The V-measure score is an harmonic mean of homogeneity and completeness. It's a compromise between homogeneity and completeness. In our case, a value under 0.5 indicates that the clustering is not good, and the results are just better than random assignments. The clustering algorithm is not able to identify the underlying patterns in the data and is not grouping similar instances together effectively.

We can see that the results are better with a KMeans model than with a GMM model. This may due to the fact that we have a neat and defined dataset, with clearly defined clusters.

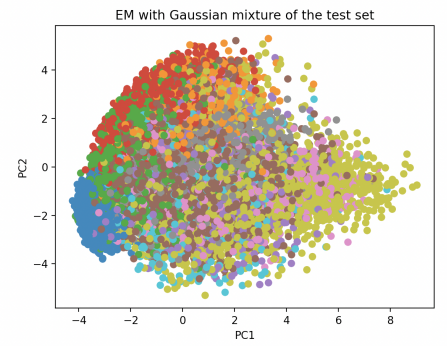


(a) KMeans of the train set

(b) KMeans of the test set



(c) EM with Gaussian mixture of the train set



(d) EM with Gaussian mixture of the test set

FIGURE 6 – Output of the KMeans and EM code

3 Supervised Machine Learning

3.1 Difference between Naïve Bayes Classifier and Support Vector Machine

Naive Bayes is a probabilistic classifier that makes classifications based on the probability of a given data point belonging to a particular class, based on the features that it has. It makes the assumption that the features are independent, which means that the presence or absence of one feature does not affect the presence or absence of any other feature. This is often not the case in real-world data, but the assumption simplifies the calculations and can still lead to good results.

Support Vector Machines (SVMs) and logistic regression are both supervised learning algorithms that can be used for classification. They both seek to find the hyperplane in a high-dimensional feature space that maximally separates the different classes. However, there are some key differences between them :

SVMs try to find the hyperplane that has the largest margin, or distance, from the nearest data points of any class. This makes them more resistant to overfitting than logistic regression. SVMs can be used for both linear and non-linear classification, whereas logistic regression is only suitable for linear classification. SVMs can handle non-linear classification by using the kernel trick, which maps the data into a higher-dimensional space where it becomes linearly separable. SVMs are more memory-intensive and computationally expensive than logistic regression, so they are not suitable for very large datasets.

3.2 Classification using SVM

The following code performs a classification with SVMs.

```
import numpy as np
from sklearn.model_selection import train_test_split

data_X = np.load('MNIST_X_28x28.npy')
data_Y = np.load('MNIST_y.npy')
data_X_1d = data_X.reshape(len(data_X), -1)

data_X_1d = data_X_1d/255.0

# Split the data into training/testing sets with 80% training and 20%
↪  test
X_train, X_test, Y_train, Y_test = train_test_split(data_X_1d, data_Y,
↪  test_size=0.2)

# perform a classification
from sklearn import svm
clf = svm.SVC(kernel='linear', C=0.05)
clf.fit(X_train, Y_train)

# predict the test set
```

```
Y_pred = clf.predict(X_test)

# evaluate the performance of the train and test set
from sklearn.metrics import accuracy_score
print("Accuracy of the training set: ", accuracy_score(Y_train,
    clf.predict(X_train)))
print("Accuracy of the testing set: ", accuracy_score(Y_test, Y_pred))
```

Let's take a closer look to the following code line :

```
clf = svm.SVC(kernel='linear', C=0.05)
```

This line of code is creating an instance of the SVC class. The SVC class takes several hyperparameters as arguments to customize its behavior, such as "kernel" and `C`. The "kernel" argument specifies the kernel function to be used by the classifier. The "linear" value for this argument specifies that the classifier should use a linear kernel. The `C` argument is a regularization parameter that controls the tradeoff between the goal of maximizing the margin between the two classes and minimizing the misclassification error. The value 0.05 for `C` specifies that the classifier should try to minimize the misclassification error, potentially at the expense of a smaller margin.

Let's change the kernel parameter from `linear` to `poly` and `rbf`. The following table sums up the results :

Kernel function	linear	polynomial	radial basis function
Train set score (%)	95.51	94.59	95.19
Test set score (%)	94.46	93.89	94.56
Computation time (min)	15	20	40

Thus, we can assume that the linear decision boundary's main advantage is to hold a good complexity/accuracy ratio. In the other hand, the rbf (Radial Basis Function) considerably increases the computation time, and this function may not seem the best option in our case. Furthermore, the "linear" kernel function may be particularly suitable for data that is linearly separable, or can be made linearly separable through a simple transformation of the input features. The "poly" kernel function may prove relevant when the data is not linearly separable, but it exhibits some degree of polynomial structure. However, this kernel function must be sensitive to the choice of the polynomial degree, and the model may be prone to overfitting if the degree is too high. Lastly, the "rbf" kernel function can be useful when the data is highly non-linear and exhibits complex patterns that cannot be captured by a linear or polynomial decision boundary.

Concerning the `C` hyperparameter, a larger value of `C` corresponds to a higher weight on the misclassification error term and a lower weight on the margin term, and vice versa for a smaller value of `C`. The following table shows the accuracy on the test set for several values of `C`.

`C` value	0.000001	0.01	0.05	10	100
Accuracy (%)	94.11	93.85	94.35	92.63	92.16

Our experiments show that increasing the value of `C` leads to a model that has a higher bias and a lower variance. This means that the model will be less flexible and more prone to underfitting. Conversely, decreasing the value of `C` can lead to a model that has a lower bias and a higher variance. This means that the model will be more flexible and more prone to overfitting, but it may also be more sensitive to the specific details of the training data.

3.3 SVM and Gaussian Naïve Bayes comparison

The following code trains and tests a Gaussian Naive Bayes classifier :

```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
gnb.fit(X_train, Y_train)

Y_pred = gnb.predict(X_test)

gnb_score_tr = gnb.score(X_train, Y_train)
print("train score: ",gnb_score_tr)

gnb_score_te = gnb.score(X_test, Y_test)
print("test score: ",gnb_score_te)
```

The following observations can be made :

1. The computation time is extremely low (less than 10 seconds) in comparison with the SVM method (more than 10 minutes with a linear kernel)
2. The train set score is : 55.61%
3. The test set score is : 55.92%
4. Regarding the scores, the model is clearly underfitting.

Thus, the Gaussian Naive Bayes classifier is a relatively too simple model and may not be powerful enough to achieve high accuracy on the MNIST dataset. More precisely, its hypothesis are too strong for MNIST ; as we work in a $\mathbb{R}^{784}$ space, it is too difficult to properly cluster the data with a gaussian. Furthermore, we can assume that a linear classifier will give even less accuracy.

3.4 Logistic Regression

The following code performs a clissification using a logistic regression :

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split

data_X = np.load('MNIST_X_28x28.npy')
```

```
data_Y = np.load('MNIST_y.npy')
data_X_1d = data_X.reshape(len(data_X), -1)

data_X_1d = data_X_1d/255.0

# Split the data into training/testing sets with 80% training and 20% test
X_train, X_test, Y_train, Y_test = train_test_split(data_X_1d, data_Y, test_size=0.2)

# perform a classification using logistic regression
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0, solver='lbfgs', multi_class='multinomial')
clf.fit(X_train, Y_train)

# predict the test set
Y_pred = clf.predict(X_test)

# evaluate the performance of the train and test set
from sklearn.metrics import accuracy_score
print("Accuracy of the training set: ", accuracy_score(Y_train, clf.predict(X_train)))
print("Accuracy of the testing set: ", accuracy_score(Y_test, Y_pred))
```

The output is an accuracy of 93.64% for the training set, and 92.41% for the testing set. The parameters for the logistic regression classifier are :

- `random_state` : optional seed for the random number generator that is used when shuffling the data. Setting this value to a fixed number ensures that the same shuffled data is used every time the model is trained, which can be useful for debugging or reproducibility.
- `solver` : specifies the algorithm to be used to find the optimal solution to the logistic regression problem. The value of `lbfgs` specifies that the limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) algorithm is used.
- `multi_class` : specifies the type of classification to be used. The value of `multinomial` specifies that the model should be used for multi-class classification.

As usual, let us play with the parameters. With `multi_class ='multinomial'` (except for `liblinear`, which does not support a multinomial backend), we obtain the following table :

solver	newton-cg	lbfgs	liblinear	sag	saga
Train set score (%)	93.92	93.64	92.88	93.88	93.79
Test set score (%)	92.20	92.41	91.92	92.42	92.55
Computation time (min)	2	1	1	2	2

As the `newton-cg` uses a Newton-CG algorithm to find the optimal solution, it is suitable for small to medium-sized datasets. The speed and results are comparable with the `lbfgs`

method. Concerning the linear method, coordinate descent algorithm is used to find the optimal solution. It is probably more suitable for large datasets, and does not prove more useful than the other solvers is not our case. The two other solvers use a coordinate descent algorithm to find the optimal solution, which seems to be a viable option in our case.

The other important parameter is `multi_class`, which can be set to `ovr`, `multiclass`, ou `auto`. The `ovr` specifies that the model should use one-vs-rest (OvR) classification, which is a method of multiclass classification where a separate binary classifier is trained for each class. The classifier that returns the highest score is used to predict the class. `multinominal` specifies that the model should use a multinomial loss function and a one-vs-rest classification strategy. The `auto` parameter uses `ovr` if the data is binary, and `multinominal` if the data is multiclass. In our case, the `ovr` mode may not be relevant, so the data gained earlier with a linear solver may not be as well.

3.5 Decision Tree

The following code performs a classification using decision tree and plot train and test sets accuracies as a function of the depth of the tree. It also prints the best accuracies obtained, and the corresponding depth of the tree.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# create a list of depths
depths = np.arange(1, 20)

# create a list to store the train and test accuracies
train_acc = []
test_acc = []

# loop over the depths
for depth in depths:
    # create a decision tree classifier
    clf = DecisionTreeClassifier(max_depth=depth)

    # fit the classifier to the train set
    clf.fit(X_train, Y_train)

    # predict the train set
    Y_pred_train = clf.predict(X_train)

    # predict the test set
    Y_pred_test = clf.predict(X_test)

    # evaluate the performance of the train and test set
    train_acc.append(accuracy_score(Y_train, Y_pred_train))
    test_acc.append(accuracy_score(Y_test, Y_pred_test))

```

```
30  # plot the train and test accuracies
31  plt.plot(depths, train_acc, label='train accuracy')
32  plt.plot(depths, test_acc, label='test accuracy')
33  plt.xlabel('depth')
34  plt.ylabel('accuracy')
35  plt.legend()
36  plt.show()
37
38  # plot the best accuracy for the train and the test set
39  print("Best accuracy for the train set: ", max(train_acc))
40  print("Best accuracy for the test set: ", max(test_acc))
41  # print the corresponding depth of the tree
42  print("Depth of the tree: ", depths[test_acc.index(max(test_acc))])
```

The output is given in figure 7. It was obtained in 4 minutes, which is a significantly small amount of time. The best accuracy for the train set is 99.25% and 87.72% for the test set. It was obtained for a depth of the tree equals to 15.

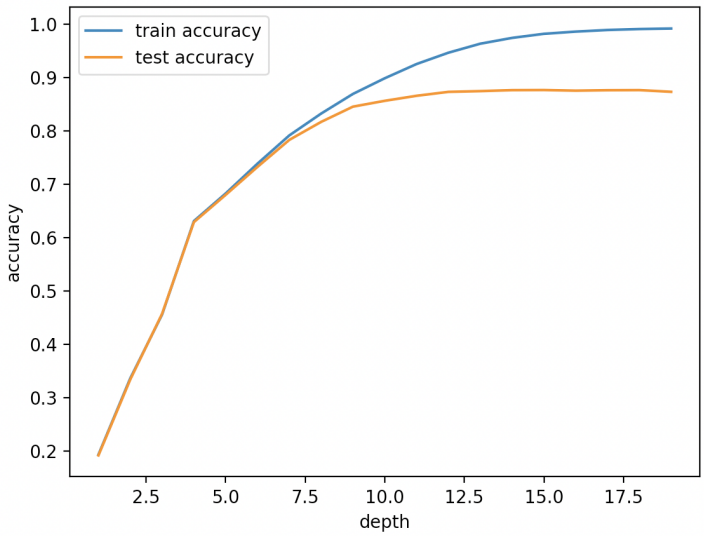


FIGURE 7 – Accuracies as a function of the tree depth

Those are very interesting results, as the model is clearly overfitting for tree depth values superior to 10. Our interpretation is that giving the model a tree depth of 12, for example, is the same as assuming there are 12 different digits in the dataset. As a consequence the model could be able to detect the three different ways of writing the digit '1', and count them as different digits. In the same way, a tree depth of 20 would be able to tell the difference between the different ways of writing '1', '4', '7', etc. However, as we only have 10 different digits, we can not go with a tree depth exceeding this number as it would be a synonym of overfitting.

To avoid this overfitting, a solution could be to manually tell the model that the different ways of writing a digit still belong to the cluster of this digit. In other terms, we would merge the clusters formed by an excessive tree depth, so we can gain achieve high

precision without overfitting. However, this is just a though and may not prove feasible, as we did not put that in practice.

3.6 Need of analyzing the performance of the test and train sets

There are three main reasons for evaluating a model's performance at both training and testing time :

1. Avoiding overfitting. Indeed, computing the accuracy of the train set can give a clue on whether the accuracy of the test set is reliable. For example, having a great accuracy on the test set can be a sign of overfitting : a simple way to verify it is to display the accuracy of the train set. On the other hand, if the training and test accuracies are both low, it could indicate that the model is underfitting and not learning from the data effectively.
2. Debugging. If a model is not performing as well as expected, evaluating its performance on the training and test sets can help identify the cause of the problem.
3. Choosing the right model. If we use multiple models, it can be useful to compare their performance on the training and test sets.

In all the cases, the test set accuracy is as important as the train set accuracy because it gives precious information on whether the model is overfitting, underfitting, or is well trained.

3.7 VSM classification after PCA

The following code performs a VSM classification after a PCA. The parameters are :
— 80% training and 20% test
— `n_components` = 0.95
— kernel function : linear
— C = 0.05

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split

data_X = np.load('MNIST_X_28x28.npy')
data_Y = np.load('MNIST_y.npy')
data_X_1d = data_X.reshape(len(data_X), -1)

data_X_1d = data_X_1d/255.0

# Split the data into training/testing sets with 80% training and 20%
↪  test
X_train, X_test, Y_train, Y_test = train_test_split(data_X_1d, data_Y,
↪  test_size=0.2)

from sklearn.decomposition import PCA
```

```
15 from sklearn import svm
16 from sklearn.metrics import accuracy_score
17
18 # perform a PCA
19 pca = PCA(n_components=0.95)
20 pca.fit(X_train)
21 X_train_pca = pca.transform(X_train)
22
23
24 # perform a classification
25 clf = svm.SVC(kernel='linear', C=0.05)
26 clf.fit(X_train_pca, Y_train)
27
28 # predict the test set
29 X_test_pca = pca.transform(X_test)
30 Y_pred = clf.predict(X_test_pca)
31
32 # evaluate the performance of the train and test set
33 print("Accuracy of the training set: ", accuracy_score(Y_train,
   ↪  clf.predict(X_train_pca)))
34 print("Accuracy of the testing set: ", accuracy_score(Y_test, Y_pred))
```

The result is an accuracy of 95.04% for the training set, and 94.14% for the testing set. Let us note that that computation time is much lower with a PCA, which is logic as we work on a reduced space. Now, we can make a few test by varying the value of `n_components`. The results are :

`n_components`	0.95	10	200	784
Train set score (%)	95.04	83.94	95.24	95.51
Test set score (%)	94.14	84.23	94.24	94.37
Computation time (min)	<1	<1	4	15

We notice that for `n_component` $= 784$, tha accuracy is approximately the same that was previously found in the section 3.2. We can also notice that the computation time seems to grow in an exponential way with `n_components`.

3.8 MultiLayer Perceptron

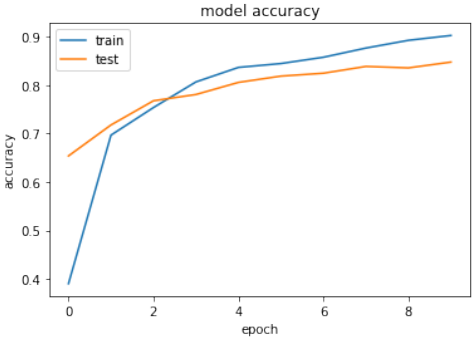
We will now try to create a model that will classifies somes images. The main steps to create a multilayer perceptron are :

1. Create our model
2. Compile it
3. Fit it to our dataset
4. Evaluate it

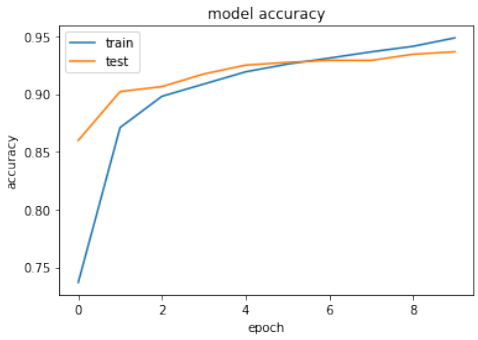
This kind of algorithm are pretty capable, as just few hidden layers may lead to a good accuracy. We will use our model to classified MNIST black and white pictures, reshaped in 28*28 pixels, so the input tensor is (batch_size,784). The batch_size is the size of the **batch**, which is a group of data : of passing only a single picture in your model and then adjust the weight and bias according to the propagation, a batch will allow you to adjust the weight of your model according the porccessing of all the image of the batch. Here, we fixed the batch_size to 128, which means the model you process 128 pictures before adjusting the weights.

Using this batch_size, we will divide all the dataset in groups of `batch_size` size, and then use each of the new batches acquired to pass it trough our model a certain amount of time. This amount is specified by the number of **epoch** we chose. The higher the epoch is, the more our model will be confronted with the same data. During the processing, we can use validation data, which are new data that the model have never seen, in order to estimate the accuracy and the loss of our model on new data, and then correct it.

Once we set these two parameters, we can start to set the most important parameters. Indeed, there are some parameters we have to set correctly in order to avoid overfitting or underfitting. The first one is the size of our dataset. Indeed, the less our model as data to train, the more likely it will learn them by heart, and thus overfit. Here is an example of what the same basic model (a single hidden layer) can do on a dataset of the firt 10,000 pictures of MNIST, and what it can do with the same 1,000 first :



(a) Accuracy of the model on 1,000 training data



(b) Accuracy of the model on 10,000 training data

We can see that the accuracy with only 1,000 training data is way worse than with 10,000 training data, while the training accuracy is roughly the same : it overfitted.

Another important parameter is the complexity of our model : the more hidden layer there are, the more likely it will overfit. It will have to much capacity against what is required, so it will only learn by heart the training data, and will be useless when it

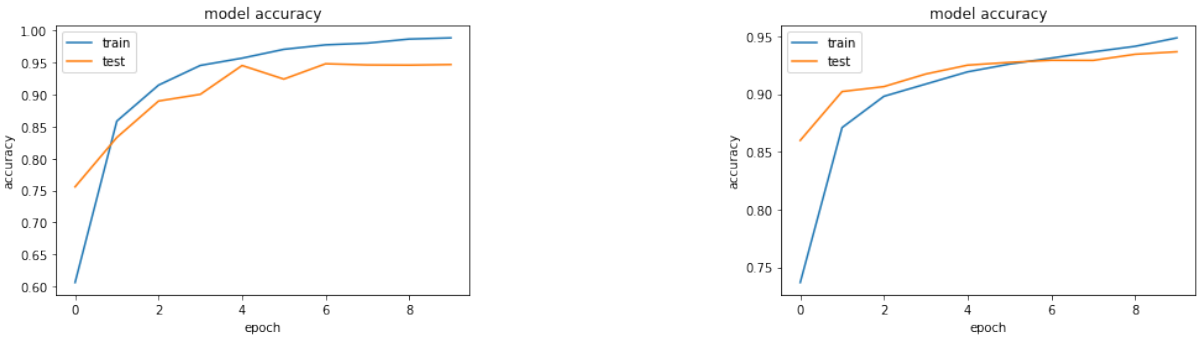
comes to new data. Let us take back our previous model with 10,000 training data. Here how we programmed it :

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
  ])
```

To make this more challenging, let us remove the dropout (its impact on overfitting will be tested after), and make a way more complex model :

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(528, activation='relu'),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(528, activation='relu'),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(528, activation='relu'),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(528, activation='relu'),
    tf.keras.layers.Dense(10)
  ])
```

The resulting accuracy and loss for this 2 different kind of models is shown in figure 9.



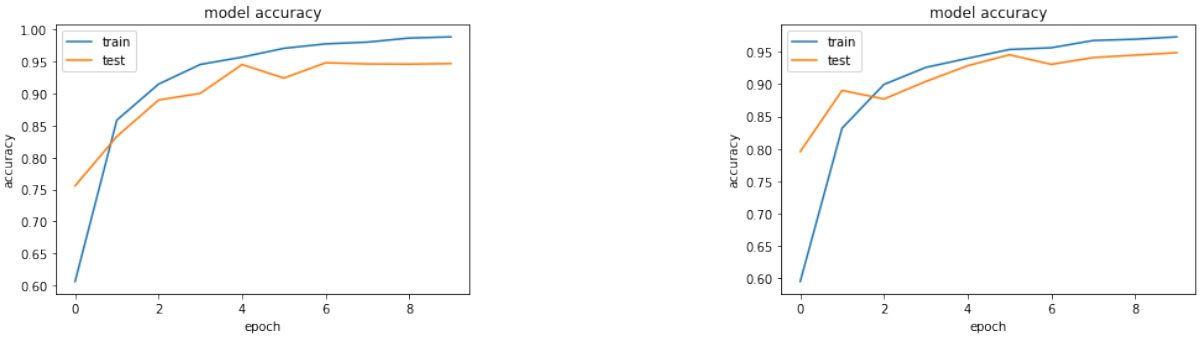
(a) Accuracy of a complex model

(b) Accuracy of a simple model

FIGURE 9 – Complex and simple models comparison

The ovefitting is quite hard to detect using MNIST dataset, because it is an ideal dataset for training, so it is meant to be difficult to overfit. However, the difference between the train accuracy and the test accuracy is what telling us that the model is overfitting. The bigger the difference between these two values, the more the model overfits.

To reduce the overfitting due to a too complex model, we can use the **Dropout** function, which will randomly freeze a weight during an epoch, thus not adjusting it. The figure 10 shows the difference between the overfitting model from before without Droupout and with Dropout :



(a) Accuracy of a complex model without Dropout

(b) Accuracy of a a complex model with dropout

FIGURE 10 – Dropout method efficiency comparison

Finally, we can plot a picture with the predicted label returned by the model, and see if there is any mistake. This should not happen, as the accuracy of the model is quite high.

```
import random as rand

probability_model = tf.keras.Sequential([
  model,
  tf.keras.layers.Softmax()
])

L = rand.sample(range(0,100), 5)

plt.image = probability_model(X_test[:5])
for i in range(5):
    plt.subplot(1,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_test[L[i]], cmap=plt.cm.binary)
    plt.xlabel(y_test[L[i]])
plt.show()
```

3.9 Convolutional Neural Network

Using a CNN, the input tensor is now a 4D tensor, because a CNN is designed to understand pictures and does not need to flatten the input data to a 1D vector anymore, of dimension `(batch\_size,28,28,1)`.

FIGURE 11 – Some MNIST pictures with their predicted label

The main difference between MLP and CNN is the layers that we can include in the model. For instance, our model could be more complex :

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
    ↪  input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

The figure 12 shows the accuracy we get from this model :

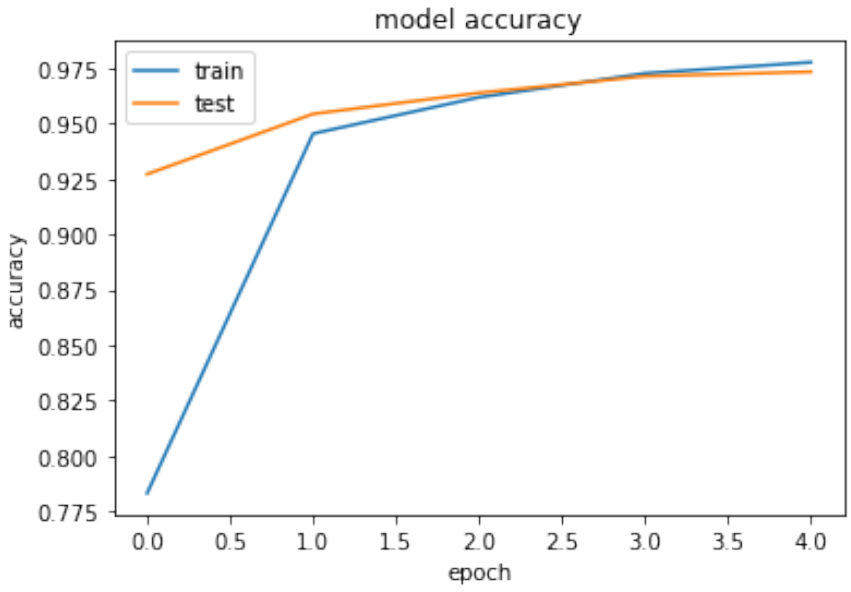


FIGURE 12 – Accuracy of a classic CNN on MNIST dataset

This model does not overfit, so to make it overfit, we can us the same parameters as seen for MLP, such as the number of training data. The figure 13 shows what happened when there were not enough training data.

In this example, the number of epoch and the number of training data has been, in an extreme way, pushed to the maximum to exemplify the overfitting of the model (epoch = 40 and 10 training data).

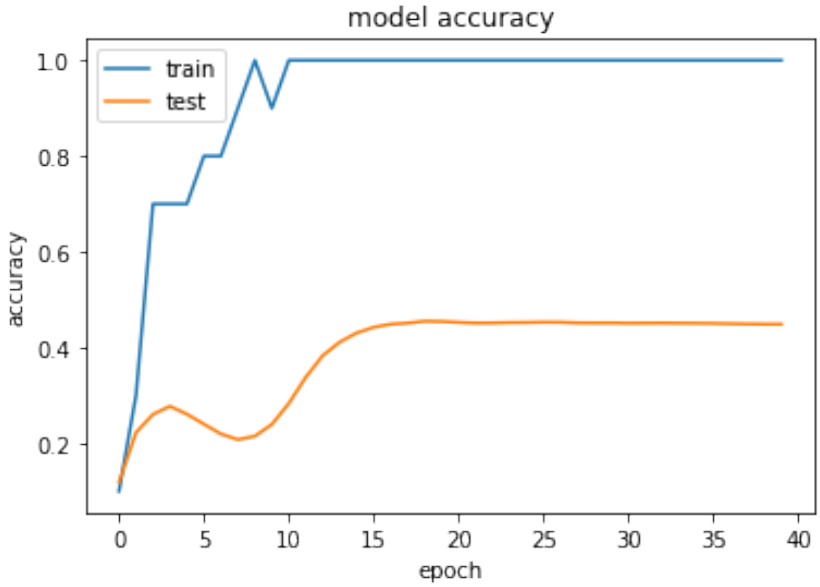
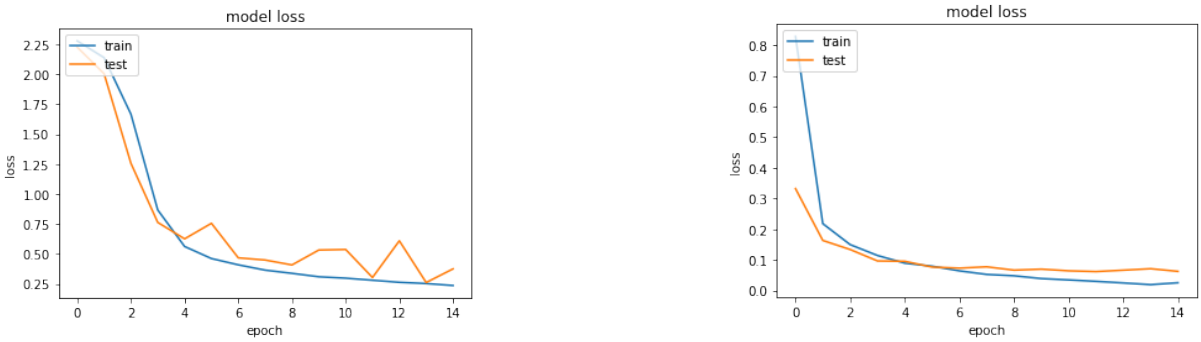


FIGURE 13 – Overfitting of the model with insufficient training data

Now, let us modifiy the parameters. There is a solid amount of parameters that are possible to change and create a model that fits to our will. By changing the layers and the optimizer of the compile function, we can create whatever we want, and we want the best model possible, without overfitting. To get it, we can use the optimizer **Adam**, which has a learning rate of 0.001, which ensures the convergence of the solution, in a moderately short-time period. It also has the possibility to "jump" a level where the loss is constant. In the figure 14, we see the difference in loss using the optimizer Adam and SGD.



(a) Loss using SGD

(b) Loss using adam

FIGURE 14 – Difference in loss using Adam or SGD

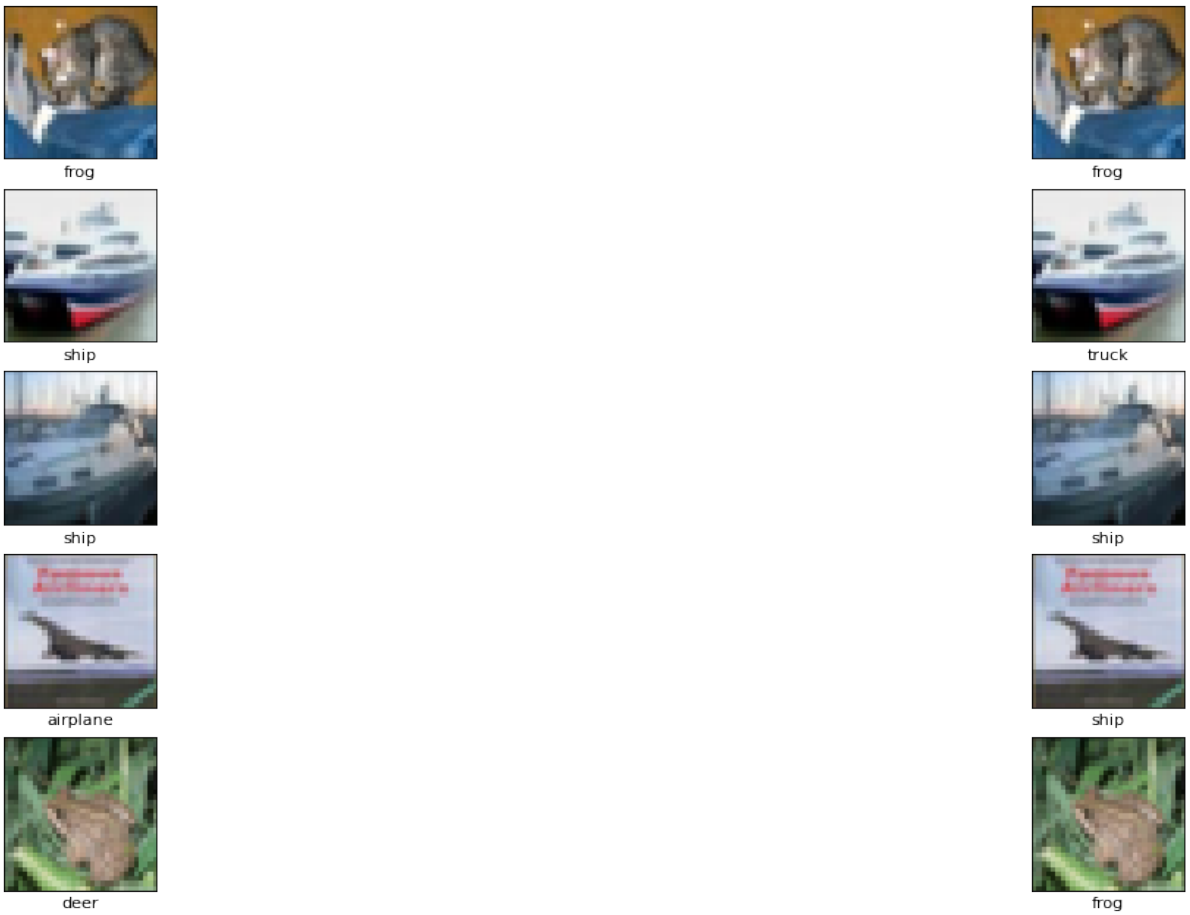
Finally, using the previous model, we can get a pretty decent loss, and plot some pictures with their predicted labels :

Regarding CNN's loss and MLP's, CNN seems to be a little bit better than MLP, but it could be because the model of the CNN matches better than the MLP model. We also don't think that MNIST is the best dataset to see a difference between MLP and CNN, because MNIST is a kind of sandbox for machine learning, so everything should work with it.

FIGURE 15 – Predicted label using CNN

However, testing both a CNN and a MLP on the cifar10 data set, we've seen that MLP are way faster, but is also less accurate than a CNN. Indeed, this is the information of the model after compiling :
`loss: 1.8318 - accuracy: 0.3350 - 653ms/epoch - 20ms/step` for the CNN
`loss: 1.9666 - accuracy: 0.2720 - 224ms/epoch - 7ms/step` for the MLP

This is not abnormal, as CNN are design to classified images. The figure 16 shows the prediction made by both the CNN and MLP for the same pictures :



(a) Prediction with CNN

(b) Prediction with MLP

FIGURE 16 – Difference of prediction between MLP and CNN