

Machine Learning

Practical Session Report

Table des matières

.....	1
Introduction	4
The MNIST Dataset	5
Unsupervised Learning	7
Dimensionality Reduction	7
Data Clustering.....	9
K-Means	10
Expectation Maximization, Gaussian Mixture	12
Supervised Learning.....	14
Decision Tree.....	14
Support Vector Machines	15
Gaussian Naïve Bayes	18
Deep Learning	19
Multilayer Perceptron (MLP)	19
A simple architecture	19
Exploring the parameters and their importance	20
Overfitting and regularization methods	20
The batch size's influence	23
Convolutional Neural Networks (CNN)	24
Hyperparameters Discussion	26
Training Visualization	26
Looking for the best architecture	28
Conclusion.....	31

Table des Figures

Figure 1 : plot_samples - code	5
Figure 2 : Data Samples - view	6
Figure 3 : Digits distribution in the dataset - view	6
Figure 4 : Pixel distribution according to their value - view	7
Figure 5 : Dimensionality Reduction - code	8
Figure 6 : Raw data VS reshaped data	8
Figure 7 : Cumulative Variance according to the dimension of the dataset	9
Figure 8 : Raw data VS reshaped data with optimum value	9
Figure 9 : K-Means - code	10
Figure 10 : K-Means Clusters representative samples.....	10
Figure 11 : K-means – Confusion matrix.....	11
Figure 12 : K-means Cluster Centers - view	11
Figure 13 : K-Means - code.....	12
Figure 14 : K-means Clusters Visualization - view.....	12
Figure 15 : Gaussian Mixture – confusion matrix	13
Figure 16 : Gaussian Mixture Clusters Visualization - view	13
Figure 17 : Decision tree - code	14
Figure 18 : Decision Tree - view	14
Figure 19 : Performance measure of decision tree - code.....	15
Figure 20 : Performance measure of decision tree - view	15
Figure 21 : SVC with C=1 - code	16
Figure 22 : SVC with C=1 – Confusion matrix.....	16
Figure 23 : Grid search applied to our SVC model - code	16
Figure 24 : Accuracy according to C & gamma's values.....	17
Figure 25 : Accuracy of our SVC model according to the number of components.....	17
Figure 26 : Gaussian Naïve Bayes - code.....	18
Figure 27 : Neural Network - code.....	19
Figure 28 : Neural Network training - code	19
Figure 29 : Neural Network test - code.....	20
Figure 30 : Neural Network test results - view	21
Figure 31 : Neural Network test results after epoch reduction - view	21
Figure 32 : Second Neural Network - code	22
Figure 33 : Second Neural Network test results – Learning Curves.....	22
Figure 34 : Batch size - code.....	23
Figure 35 : Batch size – Learning curves	24
Figure 36 : Making of CNN - code	25
Figure 37 : Basic CNN modification - code.....	25
Figure 38 : Basic CNN Learning Curves - view	25
Figure 39 : CONV layer filters visualization - view	27
Figure 40 : Activation Maps Visualization – view (left: original digit / right: activation maps)	27
Figure 41 : Model Building workflow - code	28
Figure 42 : CNN convolutional blocks benchmark.....	28
Figure 43 : CNN convolutional blocks benchmark - view	29
Figure 44 : CNN filters benchmark.....	29
Figure 45 : CNN filters benchmark - view	30
Figure 46 : Best CNN model - code	Error! Bookmark not defined.

Introduction

During the first semester of our engineering studies in the “Mines de Saint-Etienne” School. During the first semester of our engineering studies in the “Mines de Saint-Etienne” School, we had our first classes on the topic of Machine Learning. After learning the theory, we had to put in practice all we had learnt so far by doing a project.


This project can be seen as the “hello world” of Machine Learning but we will try to go further in the details to fully understand every part of it. In fact, it is a classification project using the well-known MNIST dataset.

Throughout this report, we will not explain in detail the python code, but we will try to explain our reasoning behind each method and algorithm employed to highlight the theory of machine learning we were taught in class.

The MNIST Dataset

The MNIST Dataset is a very popular dataset for training classification models. It is made of 70000 binary 28x28 images of handwritten numbers. This dataset has 10 labels which are basically the digits from '0' to '9'. Therefore, the dataset has a shape of (7000, 28, 28).

Then, we can easily plot a few samples of the dataset after loading them into NumPy arrays:



```
def plot_samples(rows=4, cols=5):  
    """  
    Plot a few image samples of the dataset  
    """  
    fig = plt.figure(figsize=(8, 8))  
  
    for i in range(1, cols * rows + 1):  
        img_index = np.random.randint(len(X))  
        fig.add_subplot(rows, cols, i)  
        plt.imshow(X[img_index])  
        plt.title(f"Classe {str(Y[img_index])}")  
  
    plt.show()  
    plt.clf()
```

Figure 1 : plot_samples - code

Thus, leading to the following plot:

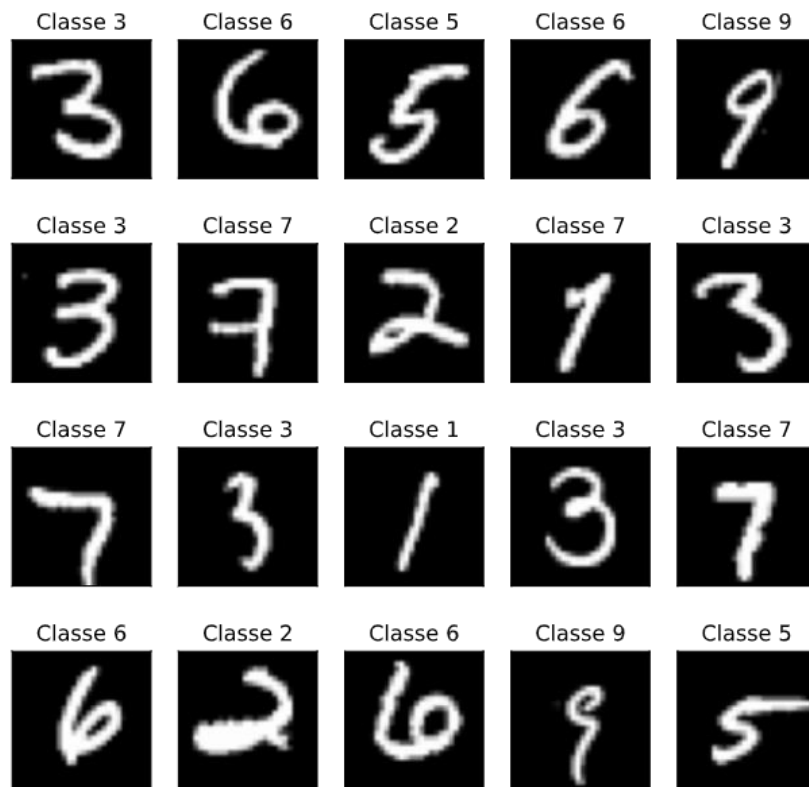


Figure 2 : Data Samples - view

We then had to split our data into training and test sets using the `train_test_split` method from the sklearn library. We chose to pick a common splitting ratio of 80/20.

Then, we can easily make sure our data is well distributed, with around 5000/6000 thousand samples for each digit, using a simple histogram plot as followed:

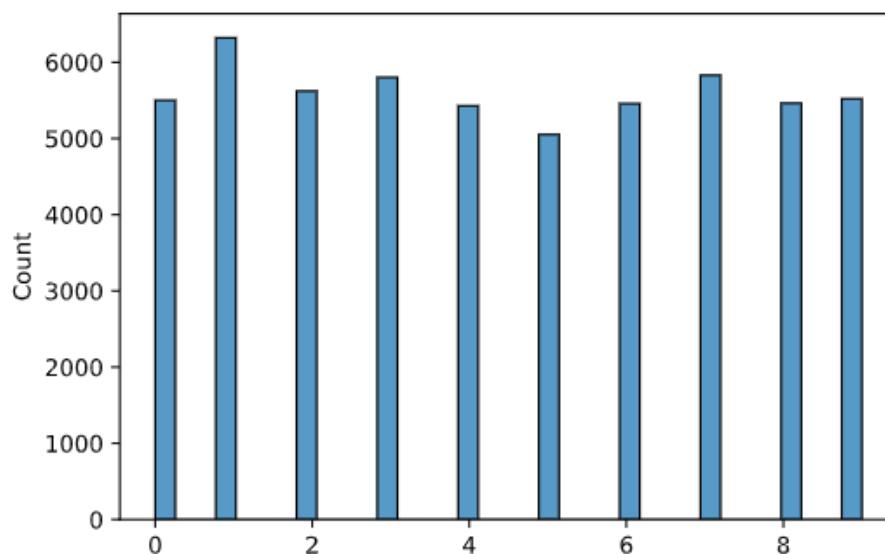


Figure 3 : Digits distribution in the dataset - view

We also took a quick look at how the pixels' values were distributed for a random sample. As expected, most of the pixels are completely white or completely black but there are a few gray pixels between them that we could totally get rid of without losing that much of information.

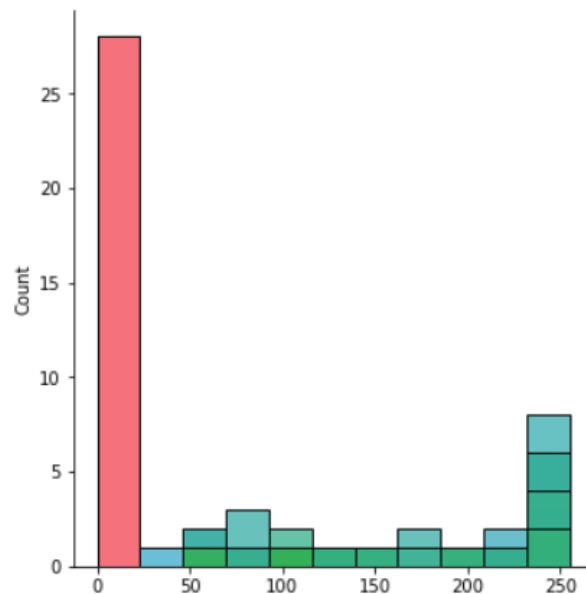


Figure 4 : Pixel distribution according to their value - view

Unsupervised Learning

Dimensionality Reduction

Our MNIST dataset has a lot of dimensions as we have seen earlier, in fact each sample has its own 28x28 or 784 unique features as a gray value between 0 and 255. This is a lot of features and we could figure out a way to reduce this number of features without losing any information or said in another manner, maximizing the variance of the data. What sounds like a difficult task can be quite easily solved with the Principal Component Analysis (PCA) method.

The PCA method is a Singular Value Decomposition (SVD) method widely used in the field of linear algebra. Let us see it in action on our dataset with scikit-learn.

However, before applying the PCA to our data a good idea would be to normalize our data by converting our data that currently ranges from 0 to 255, to a range from 0 to 1. For convenience purposes, each image array has been unraveled as a flat 784 pixels array.

```
scaler = MinMaxScaler()  
X_norm = scaler.fit_transform(X)  
  
pca = PCA(n_components=10)  
X_pca = pca.fit_transform(X_norm)  
  
X_pca_proj = pca.inverse_transform(X_pca)
```

Figure 5 : Dimensionality Reduction - code

With an arbitrary chosen number of ten components, we get the following result:

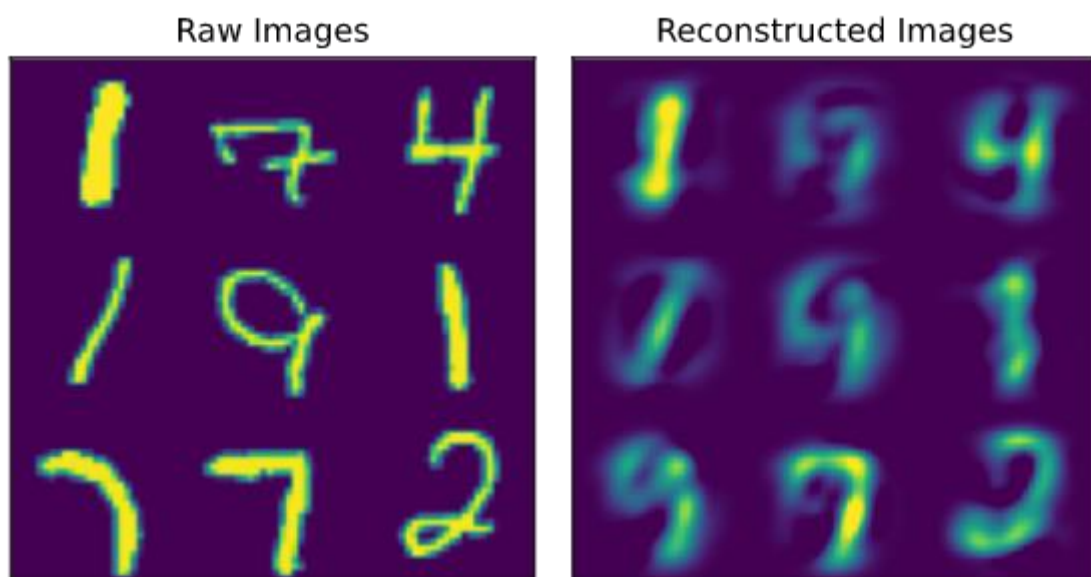


Figure 6 : Raw data VS reshaped data

We can kind of recognize the digits and we have magically reduced the number of features to 10 which dramatically reduces the dimension of our data. In fact, we had $70000 * 784$ data points and now we only have $70000 * 10$ subsequently dividing the dimension by a factor of almost 80.

Now that we saw how powerful PCA is, how can we choose the optimum number of components? We want to maximize the variance when reducing the dimension, therefore if we can get this variance ratio, we could set a threshold of about 90% and only keep the minimum number of components that encapsulate enough of information.

Thankfully, scikit-learn gives us right immediately the *explained_variance_ratio*, which appears as following:

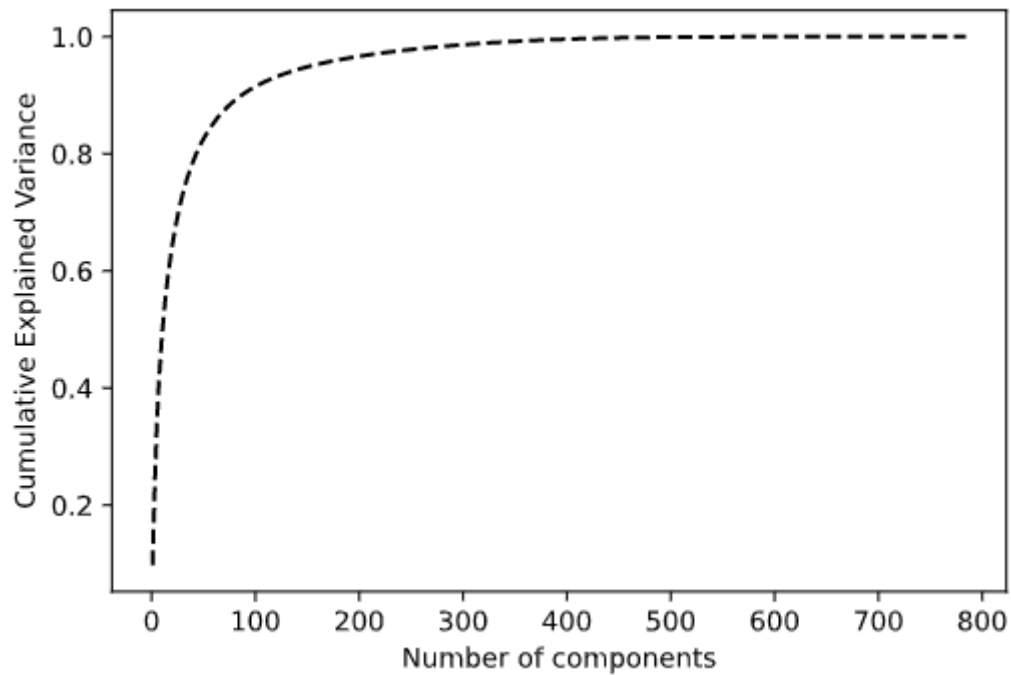


Figure 7 : Cumulative Variance according to the dimension of the dataset

The above graph demonstrates that only a few components are needed to capture most of the explained variance. In this case for instance, we set a threshold to 90% which gives us an optimum value of **86**.

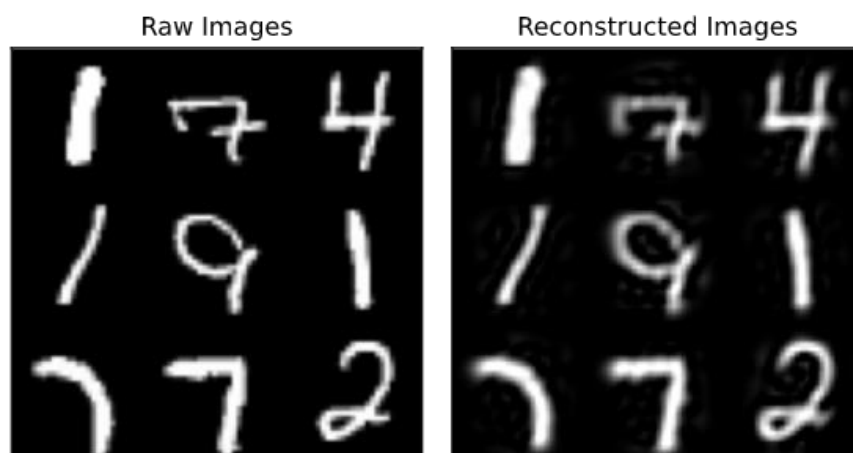


Figure 8 : Raw data VS reshaped data with optimum value

Data Clustering

In this project, we will explore 2 other methods which are widely used for data clustering.

K-Means

The first one is the K-Means algorithm. It basically divides the data into K clusters by minimizing the distance between the clusters' centers, called centroids, and the data points. However, the parameter K and the initial clusters' positions must be chosen beforehand. We will briefly study with the help of our MNIST dataset how both parameters impact the accuracy of the model.

In the case of the MNIST dataset, it is obvious that we are looking for 10 clusters, one for every digit but sometimes we might be looking for another number of clusters. We can easily train our algorithm and plot the clusters with the usual libraries.

```
kmeans = KMeans(n_clusters=10, random_state=42).fit(X_train)
predict = kmeans.predict(X_test)
```

Figure 9 : K-Means - code

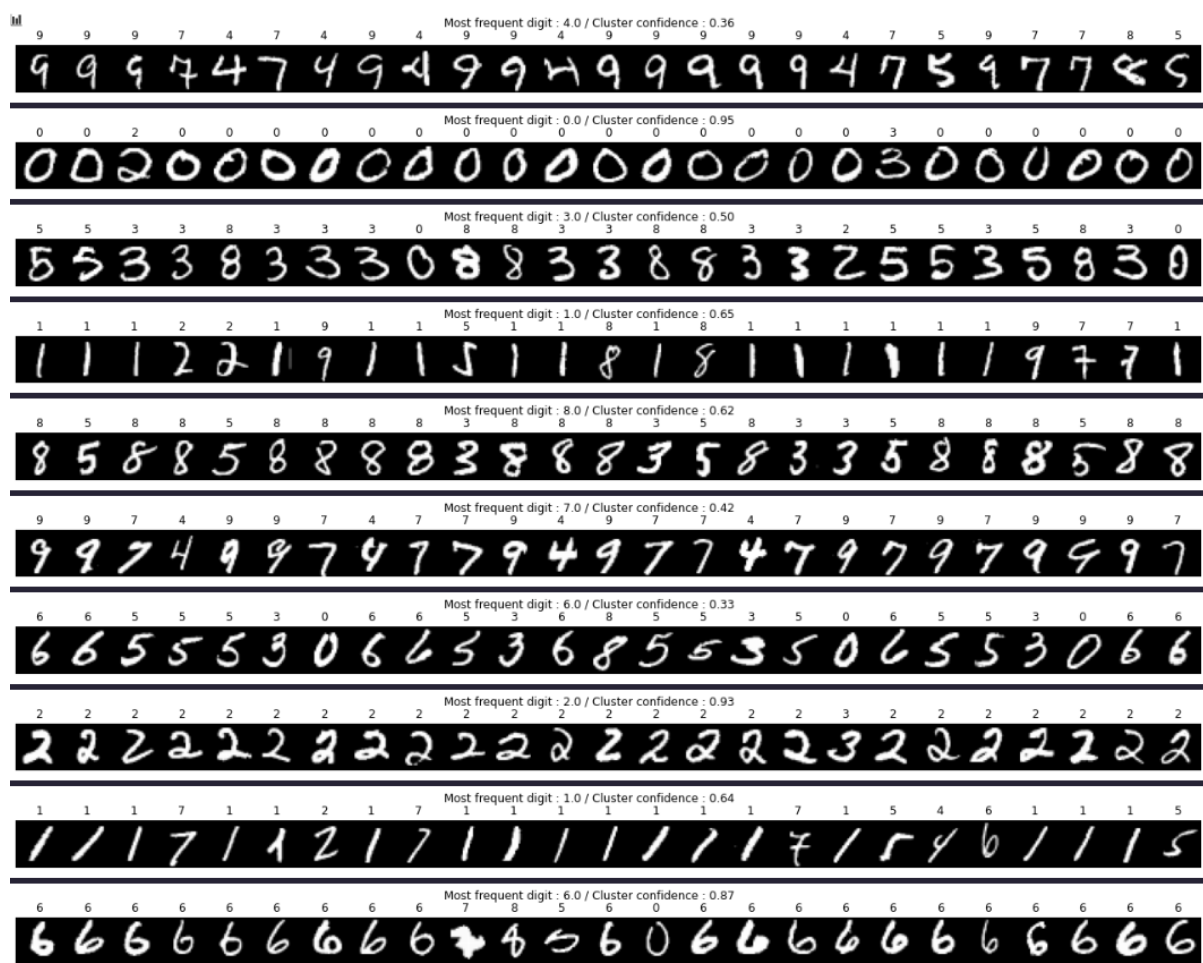


Figure 10 : K-Means Clusters representative samples

From now on, we will use the confusion matrix metric to understand how our clustering algorithm is performing and where it is failing. The confidence metric we will use is quite similar to what would be the accuracy of a supervised model.

The K-Means model led to the following confusion matrix:

	0	1	2	3	4	5	6	7	8	9
0	1022	0	9	3	2	16	10	4	3	12
1	3	1548	166	88	95	99	48	148	130	78
2	3	1	972	40	5	0	12	6	7	3
3	71	0	67	834	0	426	10	1	228	18
4	8	1	28	47	748	81	12	427	37	685
5	0	0	0	0	0	0	0	0	0	0
6	282	5	81	101	80	359	1322	9	65	11
7	1	1	13	15	462	44	2	862	40	610
8	12	0	32	211	3	236	2	6	852	19
9	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Figure 11 : K-means – Confusion matrix

Thus, the K-Means algorithm with the K-Means++ initialization method has an overall confidence of 63% which is not bad, but we can clearly see that the model is struggling to divide digits such as 9 and 4 or 6 and 8, which are calligraphically similar, in separate clusters and often tend to mix them up. In fact, we can visualize the typical digit for each cluster computed by the model:

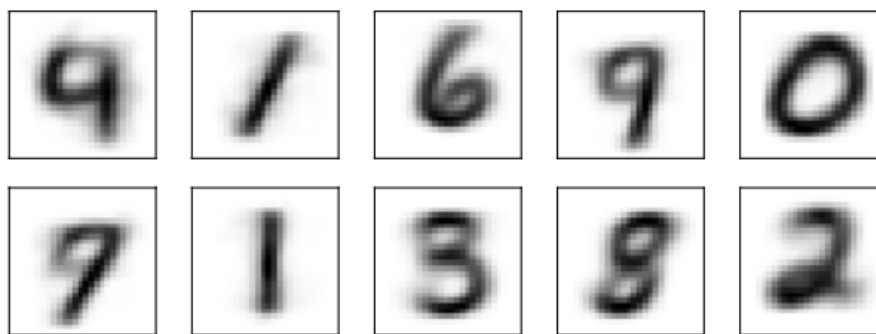


Figure 12 : K-means Cluster Centers - view

For comparison sake, we tried to run the model with the random initialization method which gave us an overall confidence of 62%. A good approach would be to initialize our K-Means model with the 10 components given by a PCA model.

Finally, we can drastically speed up the computation by applying PCA right before running our K-Means model as shown below:

```
steps = Pipeline([('pca', PCA(n_components=2)) , ('kmeans', KMeans(n_clusters=10, random_state=42))])  
predictor = steps.fit(X_train)  
predict = predictor.predict(X_test)
```

Figure 13 : K-Means - code

With 2 components, the overall confidence of our model is 42% which is slightly lower than expected. But we can easily visualize the clusters on a 2D plot as shown right below, with our 10 clusters which each appear distinctly.

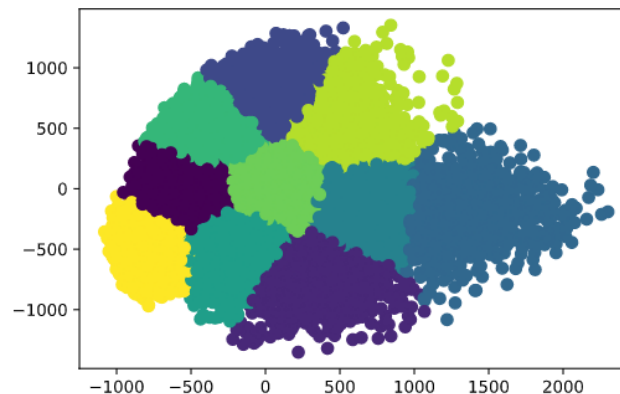


Figure 14 : K-means Clusters Visualization - view

Expectation Maximization, Gaussian Mixture

Another approach towards clustering is a probabilistic model called a Gaussian Mixture, it basically implements the expectation maximization algorithm with a Gaussian distribution.

The main issue with this approach is the lack of scalability, K-Means can easily scale up with methods such as MiniBatch K-Means. However, for density estimation and for data that appears to follow a Gaussian distribution it is a simple and truly efficient solution. In a technical point of view, K-Means uses distances between points to optimize the centroids' position whereas the Gaussian Mixture computes Mahalanobis (correlation factor) distances to centers.

It is important to note that GMM (Gaussian Mixture Models) are often used with another algorithm such as K-Means for initialization. In fact, GMM tend to converge to local non-optimum solutions and initializing them well is the key to performance.

We simply applied the GMM in the same fashion as we did for the K-Means model and obtained the following results:

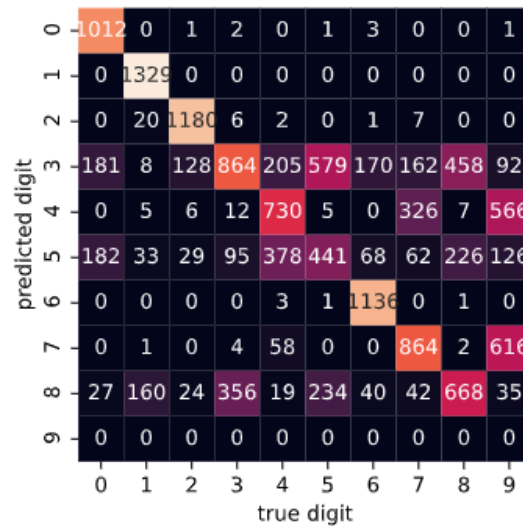


Figure 15 : Gaussian Mixture – confusion matrix

Our GMM is confident at 100% on some clusters as the '6' digit or the '1' digit, but still struggles with some. It succeeds to reach a 69% overall confidence which is slightly better than K-Means. Again, if we run the GMM while applying PCA with 2 components beforehand, we can easily distinct the clusters:

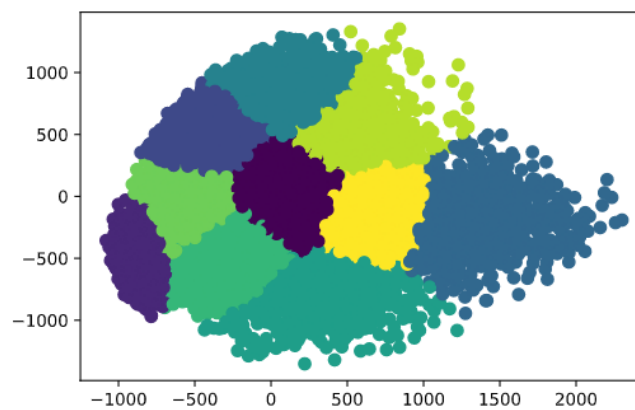


Figure 16 : Gaussian Mixture Clusters Visualization - view

Supervised Learning

We will keep using our training and test sets to prevent overfitting. In fact, we might use cross validation to prevent overfitting further on as the model can overfit on the test set if we run multiple tests. By overfitting we mean the fact that our model learns by heart to recognize the digits, it is too specific and does not generalize well on unseen data.

A few standard models will be briefly explored and compared on our MNIST dataset before diving into the Deep Learning models. One should remember that, in the real world, it is not necessary to do complex stuff such as a Deep Convolutional Network when simple models such as Decision Trees can perform just as good while being easily interpretable.

The main metric that we will be using in this section to evaluate our model is the accuracy as it is simple but other metrics like ROC might be more relevant in some cases.

Decision Tree

The first model we tried is a Decision Tree, it is the most simple and interpretable model of all. It is quite easy to train using scikit-learn as always.

```
X_norm = getNormalizedProjectedData(components=2)
X_train, X_test, y_train, y_test = get_split_data(features=X_norm, test_size=0.2)

tree_clf = DecisionTreeClassifier(max_depth=3).fit(X_train, y_train)
plot_tree(tree_clf)
```

Figure 17 : Decision tree - code

And here is the tree visualization which in our case does not really make any sense as the features are just pixels' values but still it is a powerful tool.

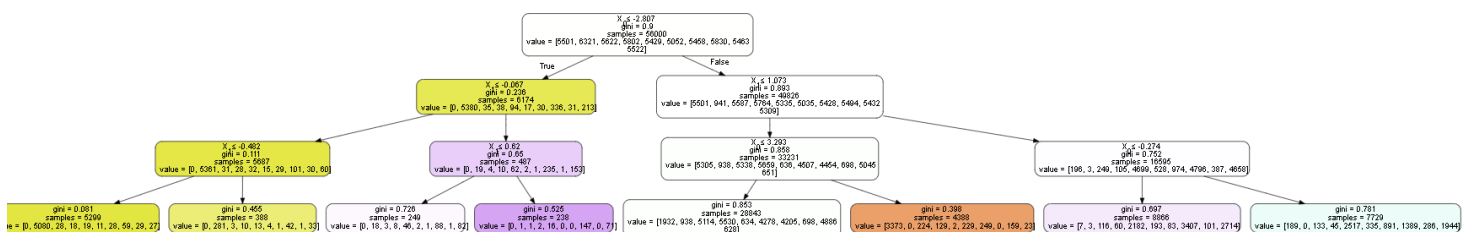


Figure 18 : Decision Tree - view

Finally, we can evaluate our model using the accuracy score:

```
train_preds = tree_clf.predict(X_train)
val_preds = tree_clf.predict(X_test)

print(f'Training accuracy : {accuracy_score(train_preds, y_train):.2f}')
print(f'Test accuracy : {accuracy_score(val_preds, y_test):.2f}')
```

Figure 19 : Performance measure of decision tree - code

Which gives us a training accuracy of 90% and a test accuracy of 85%, it is pretty good for such a simple model as it was blazingly fast to fit our 56000 samples dataset.

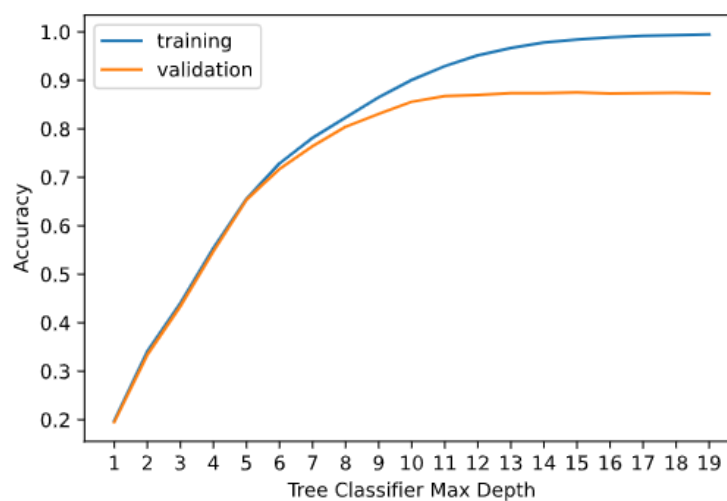


Figure 20 : Performance measure of decision tree - view

We tried to study the influence of the max depth parameter on the classifier's accuracy and as shown on the graph above, the validation accuracy does not seem to progress significantly above 10 maximum depth. The gap between the two curves shows the model starts overfitting when the max depth exceeds 10.

Support Vector Machines

Another model we explored was the Support Vector Machines, it is an effective and efficient model for data in high dimensional spaces. Basically, it works by maximizing the margin between what we call "support vectors" and thus tries, if possible, to separate the data linearly. A nice incentive of this approach is the possibility of using other kernels such as the RBF kernel that makes the SVM able to create nonlinear decision boundaries.

We first train and evaluate our linear SVM with different regularization parameters before trying out the RBF and polynomial kernels. We were already surprised by the great results of the linear SVM with default parameters ($C = 1$):

```
# Normalization and splitting
X_norm = Normalizer().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_norm, Y, test_size = 0.3, train_size = 0.2, random_state = 42)
# Model Training
linear_svc = SVC(kernel='linear').fit(X_train, y_train)
# Model Evaluation
y_pred = linear_svc.predict(X_test)

print(f'Test accuracy : {accuracy_score(y_pred, y_test):.2f}') # 93 % ! Actually pretty good !
```

Figure 21 : SVC with $C=1$ - code

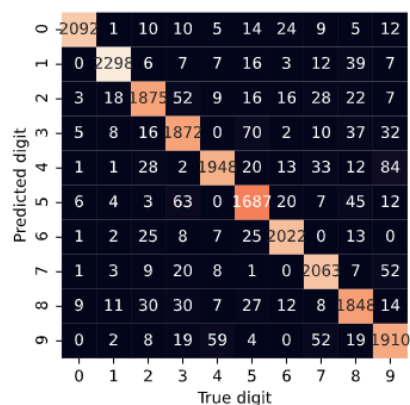


Figure 22 : SVC with $C=1$ – Confusion matrix

Of course, we already know that a nonlinear kernel might be a more suitable approach to this problem. So, we decided to apply Grid Search which tries out a few parameters on the RBF kernel model. Cross Validation came in handy at that point, we can split our dataset randomly into 10 folds each time we tune the hyperparameters to obtain our results even faster. Hopefully, all this job is done automatically by our library.

```
# Hyperparameter tuning with grid search cross validation
Cs = [0.01, 0.1, 1, 10]
gammas = [0.01, 0.1, 1]

hyperparameters = dict(C=Cs, gamma=gammas)

rbf_svc = SVC(kernel='rbf')

rbf_svc_cv = GridSearchCV(estimator=rbf_svc, param_grid=hyperparameters, scoring='accuracy', cv=3,
verbose=3, return_train_score=True)

rbf_svc_cv.fit(X_train, y_train)

# Best parameters {'C': 10, 'gamma': 0.01} give a score of 0.97
```

Figure 23 : Grid search applied to our SVC model - code

And then we can visualize the influence of the hyperparameters quite easily:

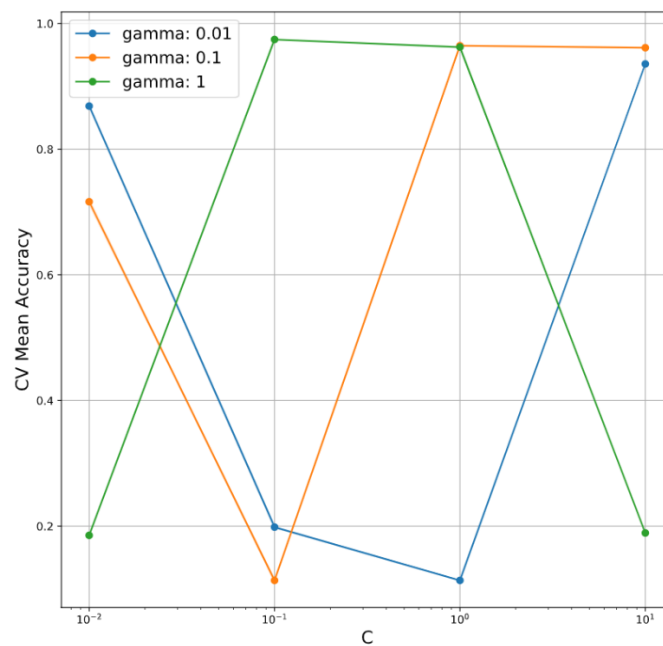


Figure 24 : Accuracy according to C & gamma's values

Finally, we chose the SVM model to study the influence of Principal Component Analysis on its performance. We picked 4 different number of components (2, 5, 10 and 87) which we thought would cover the range of cumulative explained variance of our data.

It led us to the following results:

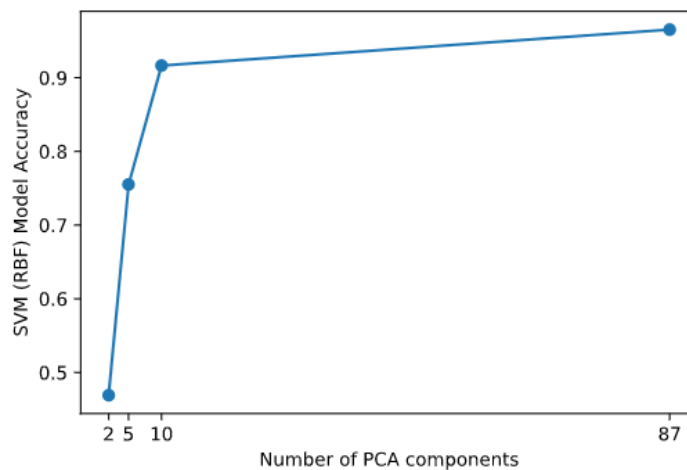


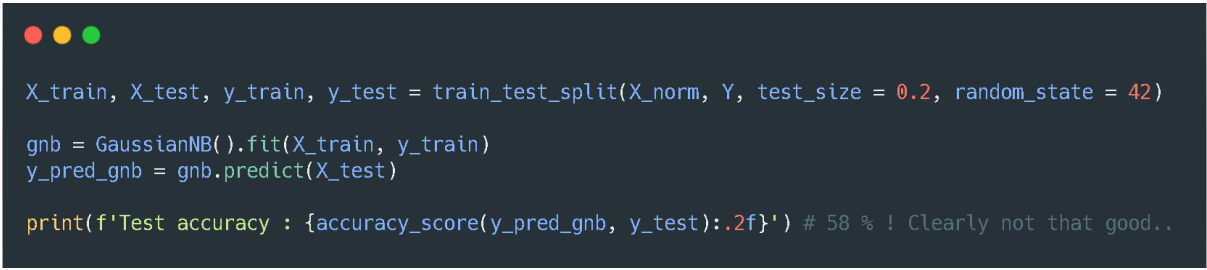
Figure 25 : Accuracy of our SVC model according to the number of components

The evolution is quite exponential from 1 to 10 components and above, the accuracy does not change much as we get more features. As we have talked about in the PCA section, having 10 components sounds like a good idea to represent the 10 digits. Reducing the dimension of our huge MNIST dataset is a nice computation speed up and will not impact the model's performance as long as we do not reduce it too much of course.

Gaussian Naïve Bayes

The Naïve Bayes Classifier is a little different from the last two models we trained. Here, we want to predict to probability of an image being a digit from 0 to 9. The Naïve Bayes algorithm does not minimize any error or maximize any loss function like in Support Vector Machines but tries to estimate a joint probability from the training data. This shows the difference between generative and discriminative models. The main issue with the Naïve Bayes classifier is that it is “naïve” and thinks all the features are independent which might or not be true.

In the case of digit recognition, the Gaussian Naïve Bayes model is not known to be really performant. We can still try it out to assess this:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains Python code for training and testing a Gaussian Naïve Bayes model. The code splits the data, fits the model, predicts on the test set, and prints the accuracy, which is 58%.

```
X_train, X_test, y_train, y_test = train_test_split(X_norm, Y, test_size = 0.2, random_state = 42)

gnb = GaussianNB().fit(X_train, y_train)
y_pred_gnb = gnb.predict(X_test)

print(f'Test accuracy : {accuracy_score(y_pred_gnb, y_test):.2f}') # 58 % ! Clearly not that good..
```

Figure 26 : Gaussian Naïve Bayes - code

Deep Learning

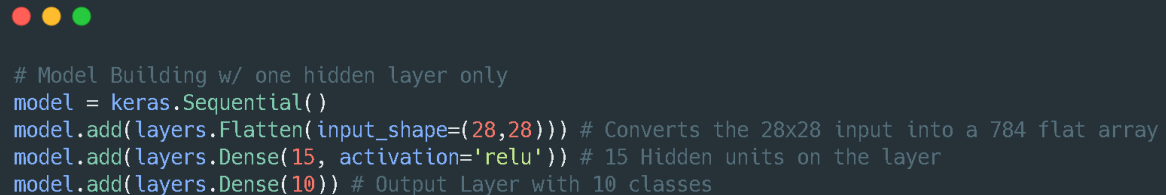
Throughout this section, we used the Tensorflow/Keras library to create an artificial neural network with ease. Those libraries allow us to manipulate the layers like Legos to build our own neural net. We chose to use the sequential API since we understood it better, despite the advice of using the functional API.

Multilayer Perceptron (MLP)

A simple architecture

At first, we will create a basic feedforward neural network only made of fully connected layers. Our neural nets take as input a tensor of size (28, 28) and return as output a tensor of size 10 corresponding to a probability vector. Each value representing the probability of the image belonging to one of our ten classes.

We build the model using the Keras Sequential API as followed:



```
# Model Building w/ one hidden layer only
model = keras.Sequential()
model.add(layers.Flatten(input_shape=(28,28))) # Converts the 28x28 input into a 784 flat array
model.add(layers.Dense(15, activation='relu')) # 15 Hidden units on the layer
model.add(layers.Dense(10)) # Output Layer with 10 classes
```

Figure 27 : Neural Network - code

We started with a basic neural network with only one hidden layer made of 15 hidden units. We now have to compile the model to feed him with some parameters that it needs while learning the weights. Finally, we train our neural network on our training data.



```
model.compile(optimizer='adam', loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

Figure 28 : Neural Network training - code

Let us break into pieces the above code, it is very important to understand how every parameter we set influence our model's training and therefore its performance.

Exploring the parameters and their importance

- Optimizer: defines how the weights are updated during the backpropagation algorithm. The most well-known method is SGD (Stochastic Gradient Descent) but most of the time the ADAM optimizer tends to be more efficient.
- Loss: defines how the model measures his precision, it is used by the optimizer to optimize the weights towards the global minimum. Again, the most well-known one is the MSE (Mean Squared Error) but for multiclass classification models a cross-entropy loss is way more suitable especially using the 'from_logits' parameter set to True (A logits vector is basically the probability vector that we get as an output).
- Metrics: A list of metrics used to track the model's performance, quite useful for analyzing our model afterwards.

The fit function also has a few parameters that are to be mentioned:

- Epochs: the number of times we iterate through our data to update the weights using the optimizer algorithm.
- Batch size: the number of samples from the training set we use to update the weights. There are a few advantages to use smaller batch sizes, it reduces the impact of the training on the GPU (Graphics Processing Unit). It adds some noise to our training and therefore kind of acts as a regularization process, in the long term it will prevent overfitting by making the gradient computation less precise.
- Validation Split / Validation Data: similar to the train_test_split method from scikit-learn but makes it easier to visualize the training process afterwards. Also, it does not shuffle the data while splitting. A good practice is to use the train_test_split method beforehand and then feed the validation set into the validation_data parameter.

Overfitting and regularization methods

Ok, now that we have a complete understanding of how every parameter can influence our neural network, we can evaluate it and see how it performed on our MNIST dataset.

```
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print(f'Test accuracy : {test_acc:.2f}') # 95%, Awesome result for such a simple architecture !
```

Figure 29 : Neural Network test - code

The Keras API allows us to easily plot the accuracy as well as the loss function throughout the training, we are looking for a decreasing loss function going towards zero in a steadily pace and an accuracy increasing towards 1.

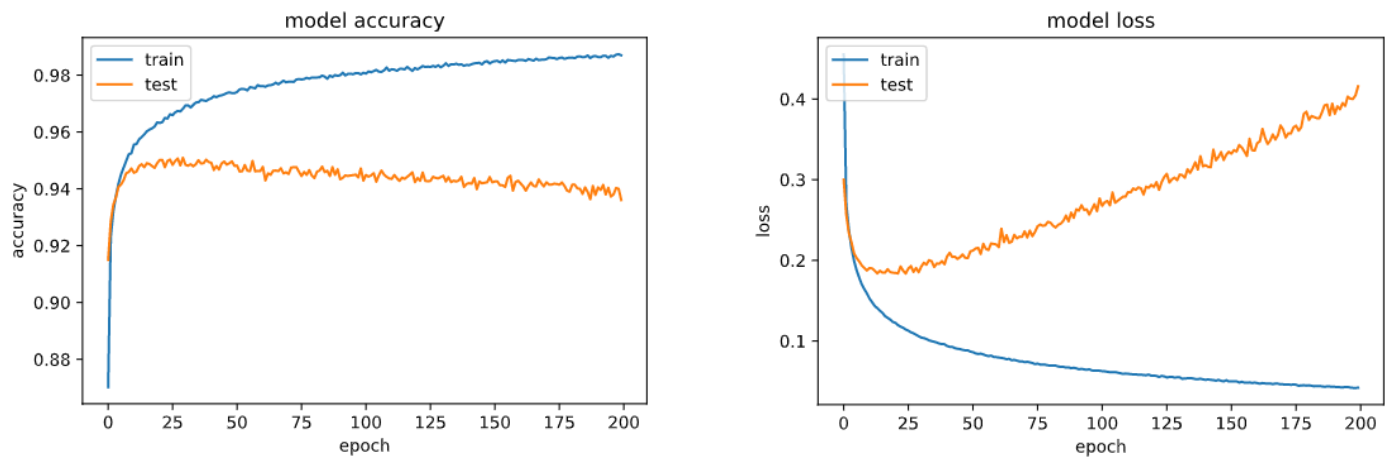


Figure 30 : Neural Network test results - view

However, we get the above results. It paints what we call overfitting. While the training accuracy goes up towards 1, our test accuracy starts decreasing and the gap between the two curves, referring to the amount of overfitting of the model, keeps increasing. It is even more obvious on the loss curves, as the training loss goes down as expected, the test loss does not go in the right way at all. Therefore our model is learning to recognize the digits by heart and will not generalize well on unseen data.

Thus, we do not need to train our model for so long. In fact, we can drastically reduce the number of epochs to 20, this technique is often referred as early stopping because we just stop the model earlier before it overfits even more.

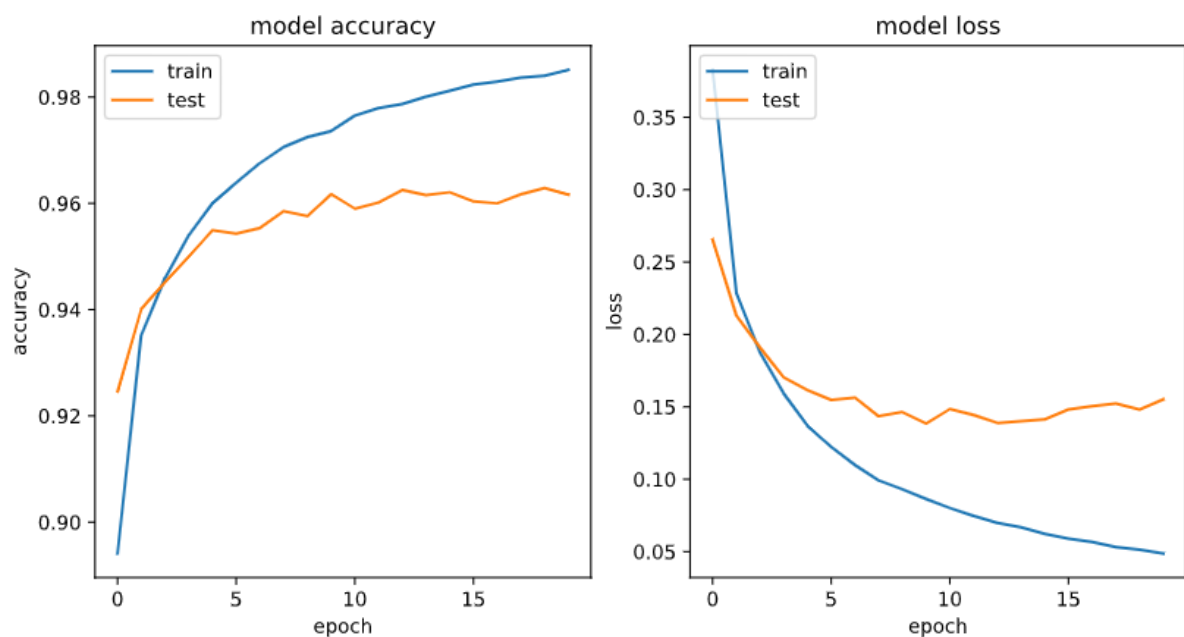


Figure 31 : Neural Network test results after epoch reduction - view

Before diving into Convolutional Neural Networks, we wanted to try out a deeper and more complex feedforward neural network with 3 layers of 512 hidden units and dropout layers that help preventing overfitting by simply dropping hidden units randomly throughout the training process.

```
model = keras.Sequential()  
model.add(layers.Flatten(input_shape=(28,28)))  
model.add(layers.Dense(512, activation='relu'))  
model.add(layers.Dropout(0.5)) # 50% chance of dropping the hidden unit  
model.add(layers.Dense(512, activation='relu'))  
model.add(layers.Dropout(0.5))  
model.add(layers.Dense(512, activation='relu'))  
model.add(layers.Dropout(0.5))  
model.add(layers.Dense(10))
```

Figure 32 : Second Neural Network - code

Of course, the model being way more complex we would expect it to overfit even more so that's part of the reason why we used a Dropout layer after each Dense layer with a 50% chance of being dropped for every hidden unit.

It led us to the following results with a final test accuracy of 98%:

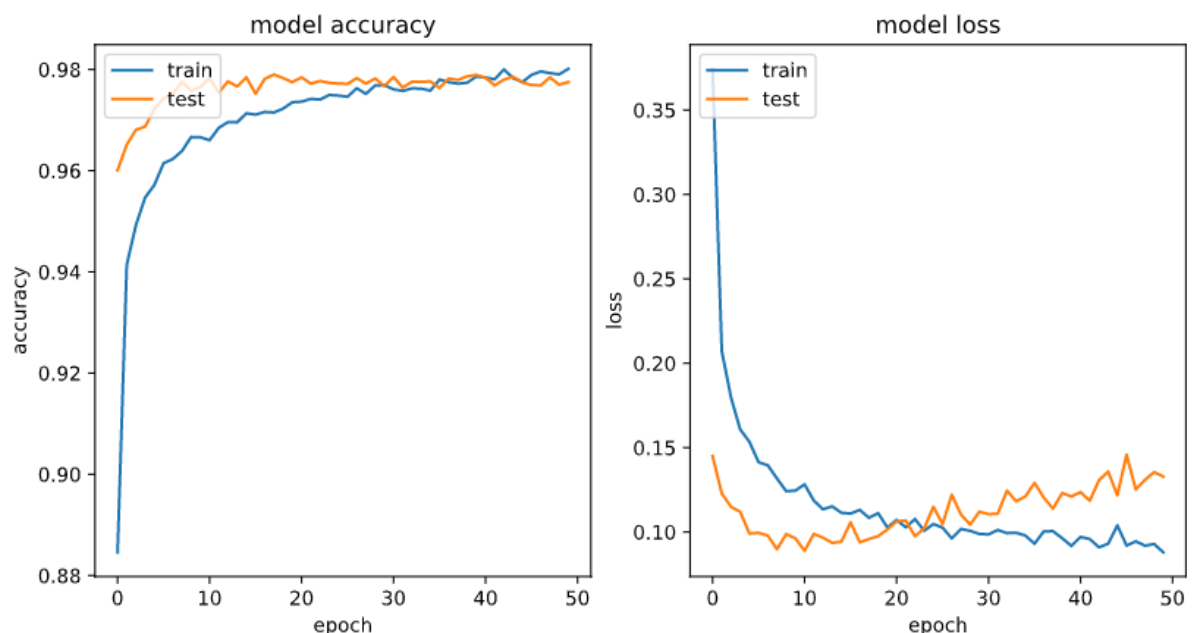


Figure 33 : Second Neural Network test results – Learning Curves

Our neural network with an architecture way more complex performs quite impressively, the dropout layers do a really good job at preventing overfitting. As we can see, the test accuracy is

slightly higher than the training accuracy for the first 30 epochs mainly because of the dropout regularization. Indeed, the dropout layers are applied during the training but not while computing the validation accuracy therefore resulting in this unusual behavior.

Further improvements could have been made to prevent overfitting and some tweaking around the neural network architecture itself could also have been made. In fact, L1 or L2 regularization methods could have easily been applied here and might have helped. However, we thought that it was not worth the effort as the CNN architecture that we will explore later tackles this issue quite well already.

The batch size's influence

Finally, we went back to a simple architecture to study the impact of the batch size hyperparameter. The batch size has an influence on how fast a model learns along with the stability of the learning process.

We picked six different batch sizes and trained a simple neural network with only one hidden layer made of 30 hidden units to get a grasp on how the batch size truly impacts the model's behavior. Having a batch size of 1 is often referred to as a stochastic method, the model will learn way faster as it will update the weights for each sample in our data. Nevertheless, it dramatically slows down the learning process, thus it might not be the best model.

For this brief study we used the following simple workflow:

```
batch_sizes = [4, 16, 32, 128, 256, 512]

for i, batch_size in enumerate(batch_sizes):
    plt.subplot(230 + (i+1)) # Create the subplot for the current batch_size
    fit_model(n_batch=batch_size) # Train the model and plot the associated learning curve
```

Figure 34 : Batch size - code

Thus, leading to the following six learning curves:

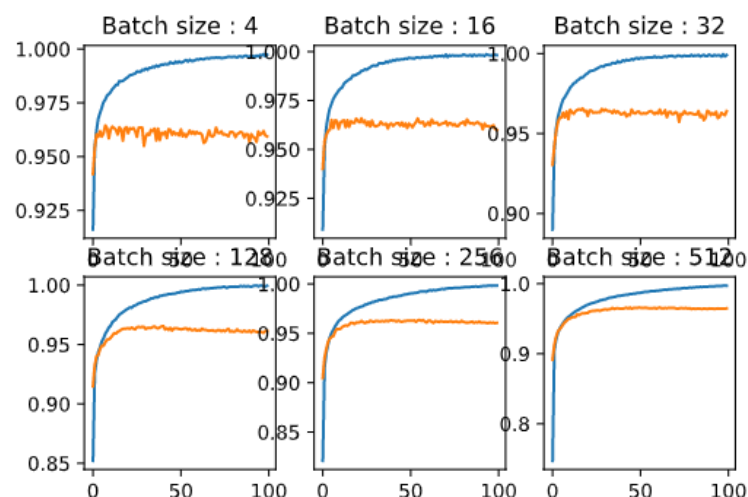


Figure 35 : Batch size – Learning curves

Apart from the overfitting issue that we have already tackled previously, the above plot pictures show that small batch sizes often result in a fast but noisy learning curve leading to higher variance in the validation accuracy. On the other hand, larger batch sizes slow down the learning process while stabilizing it, resulting in a lower variance for the validation accuracy.

Convolutional Neural Networks (CNN)

In the previous section, we covered the classic feedforward neural network model. We observed how it could overfit quite fast as the number of parameters to learn can substantially grow, thus leading to huge training time lengths. The MLP architecture is not well suited to images, it does not consider the fact that the inputs are images and treat them like regular inputs. That is part of the reason why Convolutional Neural Nets have been created, they assume that the inputs are images and use this assumption in a clever way. In fact, a CNN does not treat an image as a flat unordered vector of pixels but consider the location of every pixel with its surroundings, this is what makes it so powerful.

A CNN sees things differently, it considers volumes of neurons, every layer has a width, a height, and a depth. This third dimension added by the CNN refers to the color channels of the image. For instance, a colored image of size 28x28 will be treated as a volume of size 28x28x3 as an input of the CNN. The MNIST dataset only has one color channel and therefore our input tensor will have dimensions 28x28x1.

A CNN has different types of layers:

- Convolutional Layers
- RELU Layers
- Pooling Layers

When assembled, these layers form a convolutional block, and we can stack convolutional blocks to form a full convolutional neural network.

On top of the convolutional blocks, the CNN architecture adds Fully Connected blocks that do the classification task. These layers are exactly the one that we were using in the MLP architecture.

At first, to get a good grasp and understanding on the CNN architecture we will work with the most basic CNN made of only one convolutional block.


```

model = models.Sequential()

# CONV layer w/ 8 Filters of size (2,2) / Stride of 1 / Padding that keeps input size / Activation RELU -> Common values
model.add(layers.Conv2D(8, (2, 2), input_shape=(28, 28, 1), activation='relu', padding='same'))
# POOL layer w/ Spatial Extent of 2 / Strides of 2 -> Common values
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Flatten the tensor to feed the FC Layers
model.add(layers.Flatten())
# FC layer w/
model.add(layers.Dense(15))
# Output layer w/ 10 classes
model.add(layers.Dense(10))

```

Figure 36 : Making of CNN - code

The most basic CNN network that we could think about already gives shining results with a validation accuracy of 97.5%. However, we can see it is overfitting quite a lot which the FC layer might be responsible of, especially considering the fact that we only have 8 filters of size (2, 2) in our sole convolutional block. Looking at these learning curves, some modifications come naturally:

```

model = models.Sequential()

# CONV layer w/ 24 Filters of size (5,5) / Stride of 1 / Padding that keeps input size / Activation RELU
model.add(layers.Conv2D(24, (5, 5), input_shape=(28, 28, 1), activation='relu', padding='same'))
# POOL layer w/ Spatial Extent of 2 / Strides of 2 -> Common values
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Flatten the tensor to feed the FC Layers
model.add(layers.Flatten())
# FC layer w/ 30 hidden units
model.add(layers.Dense(30, activation='relu'))
# Output layer w/ 10 classes and softmax activation
model.add(layers.Dense(10, activation='softmax'))

```

Figure 37 : Basic CNN modification - code

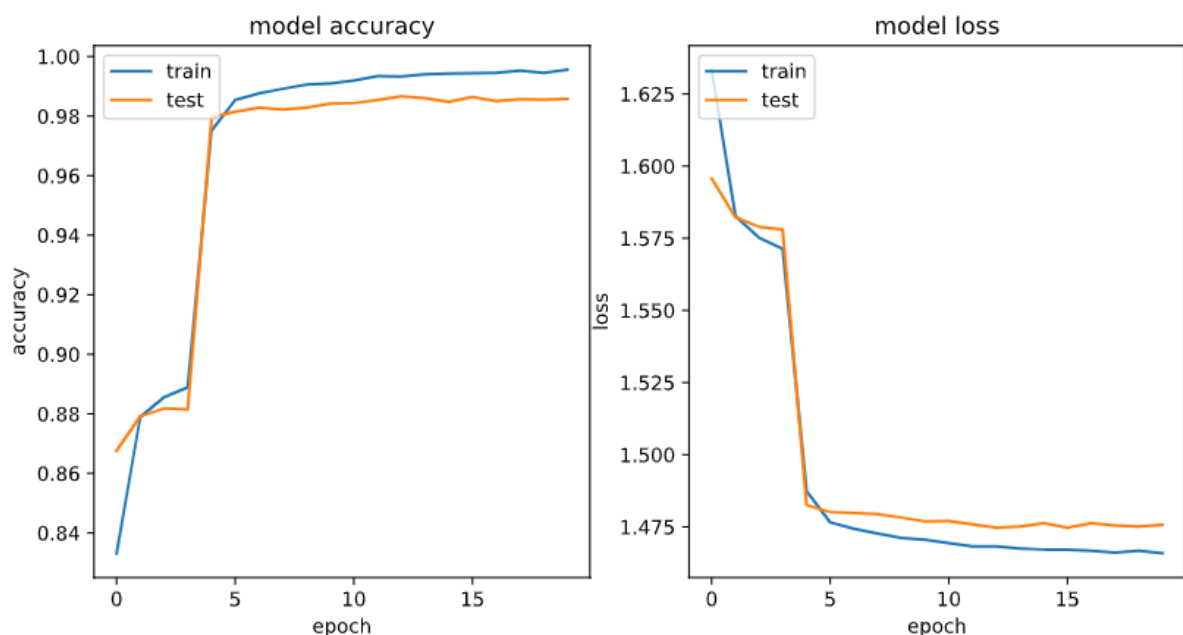


Figure 38 : Basic CNN Learning Curves - view

The learning curves already look way better with a much smaller gap between the training and test accuracies. This fine-tuned basic CNN gives a 98.7% validation accuracy which is the best we have done (yet).

However, in this case the pooling layer is usually not useful as we do not apply any other convolution on the reduced images. The above tests were done without this pooling layer and the results do not change that much from one architecture to the other.

Hyperparameters Discussion

The choice of the model's hyperparameters are not random at all, in fact let us discuss these choices:

- The number of filters used for each CONV layer is quite arbitrary, the most important thing to keep in mind is to add more filters as we add layers.
- The stride is kept at 1 as we do not want to change the size of our inputs within the CONV layer, it should only be the job of the POOL layer.
- The padding is set to 'same' meaning that it is adjusted accordingly to keep the input's size unchanged. In fact, it depends on the kernel size we decide to choose. We would lose information at the borders if only the 'valid' padding is used.
- The POOL layer has most of the time 2x2 kernel size with a stride of 2. Those settings allow the pooling layer to divide the width and height of our image by 2. By taking the maximum of every 2x2 pixel squares, we drastically reduce the dimension of our images and the next CONV layers will be able to focus on more specific details of the images.

Training Visualization

To fully understand how our CNN trains and learns, we can visualize the filters as well as the activation maps (applying the filters to the inputs) that come as an output of the CONV layers. Visualizing those filters and activation maps will help understanding which features our convolutional blocks managed to extract from our images. In our case, we only have one CONV layer with a set of 24 filters of size 5x5.

Here is how they look:

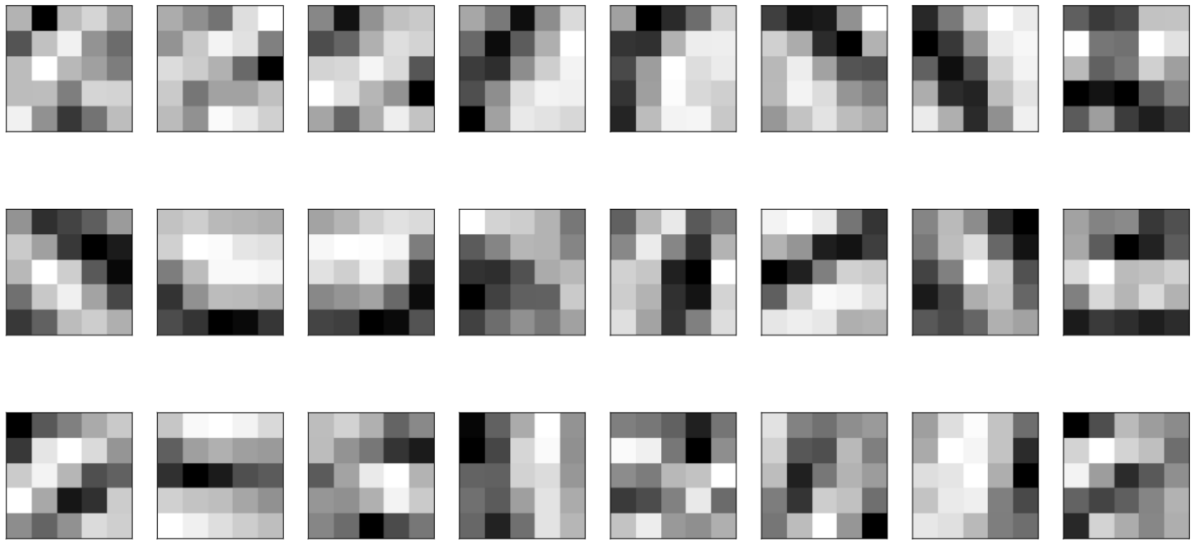


Figure 39 : CONV layer filters visualization - view

The filters themselves are a hard to interpret but we can kind of recognize the digits' borders on some filters or horizontal and vertical bars that are often present on many images.

If we apply them to an example input, we get the following activation maps:

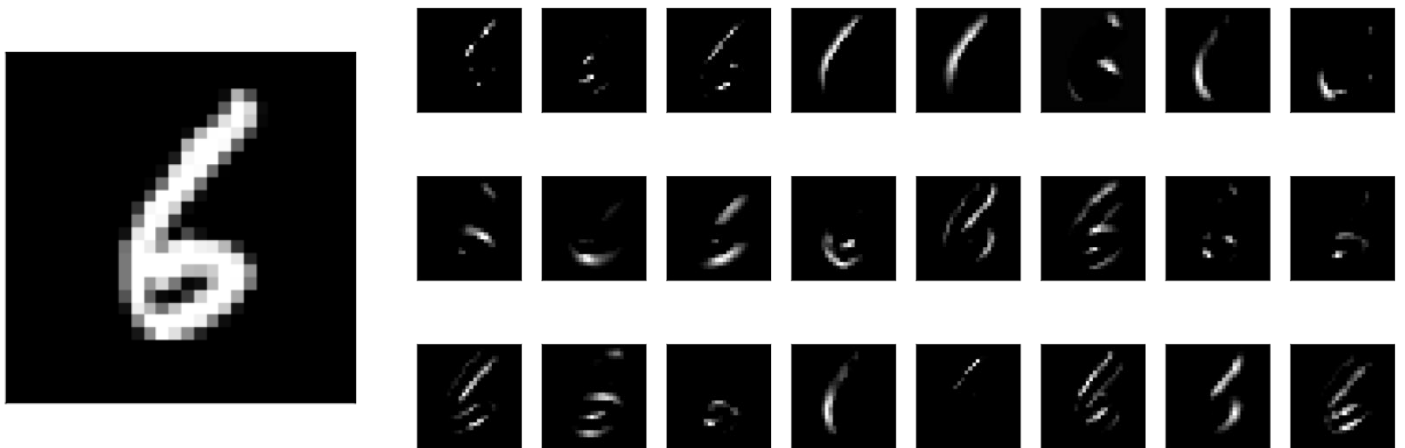


Figure 40 : Activation Maps Visualization – view (left: original digit / right: activation maps)

Those activation maps are way easier to interpret, our convolutional layer perfectly manages to recognize the different patterns in our '6' digit. Some filters help to recognize the outline of the digit while other filters are more specific and focusing on small parts like the curve or the small circle inside the 6.

Looking for the best architecture

The next logical steps would be to add more convolutional blocks. But how do we know how many we want to add with how many filters and what kernel sizes to choose? A few methods exist to make these choices, but we will try out different options and look by ourselves what seem to be the best. We will use functions like the one below to create multiple models at once with different settings.

```
def build_model(n_conv_blocks, kernel_size=5, n_hidden_units=64):
    """
    Function that builds a CNN with a fixed amount of convolutional blocks
    The number of filters increases at each activation map by a power of 2 starting at 2^4=16
    The kernel size is set to 5x5 but can be changed
    The final FC layer has 64 hidden units but this can also be modified
    args:
        {n_conv_blocks} : number of convolutional blocks we want the CNN to have
        {kernel_size} : the kernel size used in each CONV layer
        {n_hidden_units} : number of hidden units of the unique FC layer
    """

    model = models.Sequential()

    n_filters = 16

    for idx_conv_block in range(n_conv_blocks):
        # Add a CONV layer
        model.add(layers.Conv2D(n_filters, kernel_size=kernel_size, padding='same', activation='relu', input_shape=(28, 28, 1)))
        # Add the associated POOL layer
        model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
        # Update the number of filters
        n_filters *= 2

    # Flatten the data
    model.add(layers.Flatten())
    # FC layer w/ {n_hidden_units} hidden units
    model.add(layers.Dense(n_hidden_units, activation='relu'))
    # Output layer w/ 10 classes and softmax activation
    model.add(layers.Dense(10, activation='softmax'))

    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

    return model
```

Figure 41 : Model Building workflow - code

We started by figuring out what would be the best between one, two or three convolutional blocks.

```
CNN 1 Conv Block: Epochs=30, Train accuracy=0.99911, Validation accuracy=0.98629
CNN 2 Conv Block: Epochs=30, Train accuracy=0.99921, Validation accuracy=0.98950
CNN 3 Conv Block: Epochs=30, Train accuracy=0.99846, Validation accuracy=0.99093
```

Figure 42 : CNN convolutional blocks benchmark

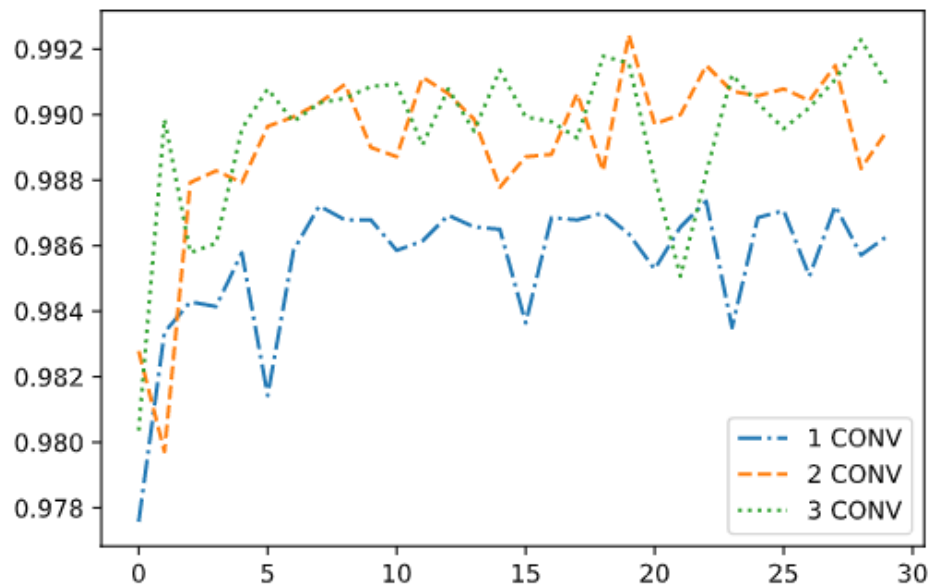


Figure 43 : CNN convolutional blocks benchmark - view

The above results highlight the power of having multiple convolutional blocks being more and more specific about the features they extract from the images. However, we are limited in the number of convolutional blocks because of the pooling layer which decreases our input size after each convolution. In our case, having more than 3 convolutional blocks would not make any sense as we do not expect our network to find any relevant patterns in 3x3 images.

We decided to stick to 2 convolutional blocks for the rest of the study as it leads to a great ratio between performance and time.

Then, using a similar workflow, we tried to figure out the optimum number of filters for our 2 convolutional layers.

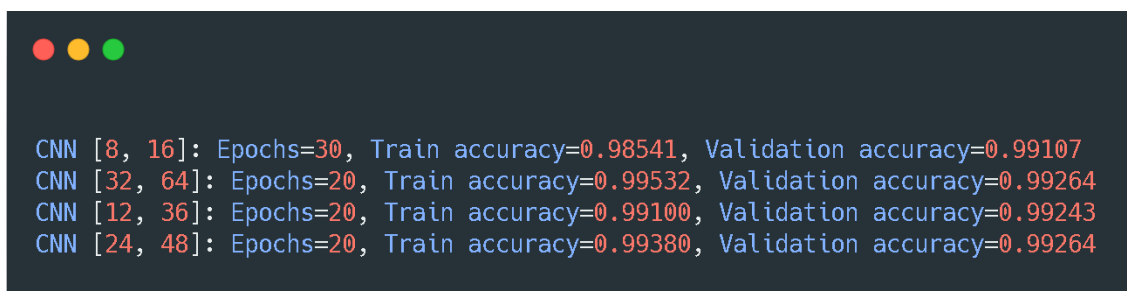


Figure 44 : CNN filters benchmark

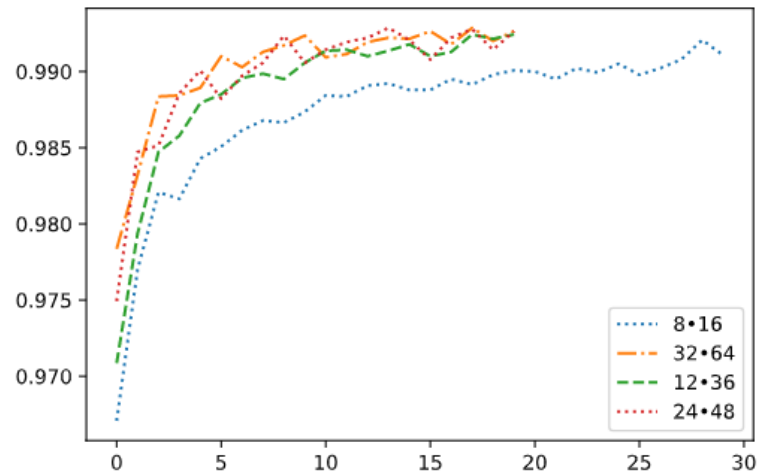


Figure 45 : CNN filters benchmark - view

Looking at the above results, we decided that the best architecture with our little tweaking, which is not complete at all, is the following one. We notice that it is also the best model in terms of performance that we managed to train with a validation accuracy of 99.264%.

```

model = models.Sequential()

# CONV layer w/ 32 Filters of size (5 ,5) / Stride of 1 / Padding that keeps input size / Activation RELU
model.add(layers.Conv2D(32, (5, 5), input_shape=(28, 28, 1), activation='relu', padding='same'))
# POOL layer w/ Spatial Extent of 2 / Strides of 2 -> Common values
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Dropout layer w/ 40% factor
model.add(layers.Dropout(dropout))
# CONV layer w/ 64 Filters of size (5 ,5) / Stride of 1 / Padding that keeps input size / Activation RELU
model.add(layers.Conv2D(64, (5, 5), activation='relu', padding='same'))
# POOL layer w/ Spatial Extent of 2 / Strides of 2 -> Common values
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Dropout layer w/ 40% factor
model.add(layers.Dropout(dropout))
# Flatten the tensor to feed the FC Layers
model.add(layers.Flatten())
# FC layer w/ 30 hidden units
model.add(layers.Dense(30, activation='relu'))
# Output layer w/ 10 classes and softmax activation
model.add(layers.Dense(10, activation='softmax'))

# Best model we managed to put together with a 99.3% validation accuracy !

```

Figure 46 : Best CNN model - code

At that point, a lot more improvements and research could have been made to find the perfect CNN architecture for the MNIST dataset. We decided to stop there and sum up what we have learnt so far.

Conclusion

This project has been quite massive as we have explored many technics widely used in the field of Machine Learning. First things first, exploring the dataset and understanding its formation is the cornerstone of a successful ML project. We then had a lot of fun playing with the unsupervised learning methods as they can lead to unexpected results especially when using dimensionality reduction. Experimenting was a fun part of the project, and there were many more experimentations available as we were exploring more complex models.

If we had to pick our two favorite models, we would definitely say Support Vector Machines and, of course, the Convolutional Neural Network.

Apart from experimenting, we tried on our side to understand what was happening behind the scenes as we were doubtful while playing with the hyper-parameters without knowing exactly how they influence the computations mathematically.

The most difficult part of this project was understanding precisely how a CNN works for being able to understand what each layer does and what each hyper-parameter impacts. However, once understood it was again very fun (despite the endless training times).

For one of us it was not his first Machine Learning project as he had done a few Kaggle projects before and knew his methods and strategies. For the other, it was the perfect occasion to assimilate the new concepts and to practice quickly and efficiently. We both have good skills in mathematics, and we have a critical mind, which came in handy during this project. Thus, there was a lot of knowledge sharing and it was a truly interesting and rich experience.