In [0]:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, BatchNormalization
from keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
import numpy as np
from collections import Counter
from collections import defaultdict
import matplotlib.pyplot as plt

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

Initial analysis of MNIST dataset

In our initial data analysis, we will try to answer to the following questions or properties :

- How many examples do we have in MNIST dataset?
- What is the shape of my input images?
- How many classes do I have?
- Is the testing set similar to the training set?
- What is the class distribution in my dataset?
- What does an average of each digit look like?
- What are the images which are the least similar to the average, and nearest?
- How much gray is there in the set of images?

From all answers we will obtain a deeper understanding on what our dataset is composed of.

Basic dataset analysis

Here we can answer to our first questions :

- How many examples do we have in MNIST dataset?
  - Answer : we have in total 70000 examples in our dataset, divided as following : 60000 traning examples and 10000 testing examples.
- Is the testing set similar to the training set?
  - Answer : we have similar testing set and training set with respect to the size of the images, their grayscale, but the number of testing examples and traning examples are different.
- How many classes do I have?
  - Answer : we have 10 classes which are the numbers between 0 and 9.
- What is the class distribution in my dataset?
  - Answer : the classe distribution is the number of occurences of each class value, (see below code for the exact answer).
- What is the shape of my input images?
  - Answer : the shape of the images is 28 by 28 pixels.

In [0]:

```
num_classes = np.unique(Y_train).shape[0]
print("Number of training examples:", X_train.shape[0])
print("Number of testing examples:", X_test.shape[0])
print("Number of classes:", num_classes)
print("Which are: ", np.unique(Y_train))
print('Class distribution: ', np.bincount(Y_train))
print("Image shape:", X_train[0].shape)
print("Image data type:", X_train.dtype)
```

```
Number of training examples: 60000
Number of testing examples: 10000
Number of classes: 10
Which are:  [0 1 2 3 4 5 6 7 8 9]
```

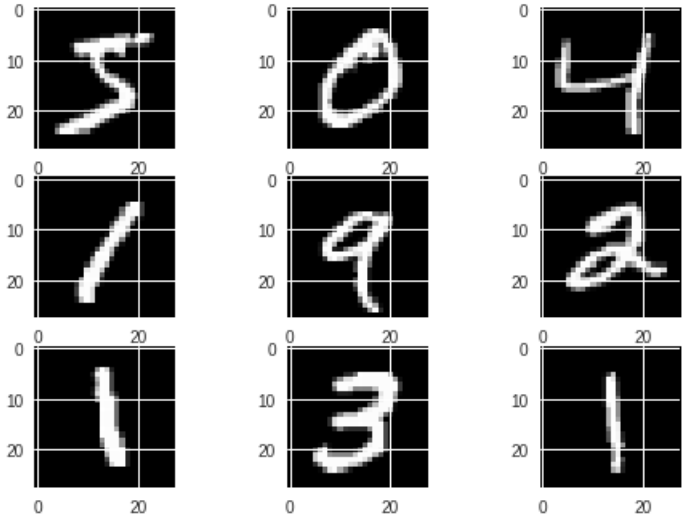
```
Class distribution:  [5923 6742 5958 6131 5842 5421 5918 6265 5851 5949]
Image shape: (28, 28)
Image data type: uint8
```

The first 9 numbers of MNIST dataset

First of all, it can be interesting to have a look at our first 9 images. Thanks to that we can better imagine and visualize our dataset.

In [0]:

```
for row in range(3):
    for col in range(3):
        idx = row*3 + col + 1
        plt.subplot(3,3, idx)
        plt.imshow(X_train[idx-1], cmap="gray")
```



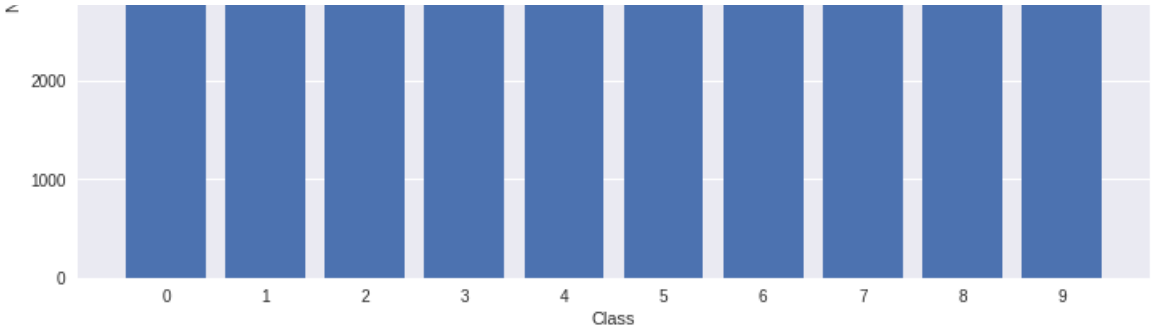
To complete this first approach, we can even plot our class distribution as following.

In [0]:

```
class_distribution = Counter(Y_train)
x = range(10)
y = [class_distribution[cls] for cls in x]
plt.figure(figsize=(12,8))
plt.xticks(x)
plt.title("Number of training examples in each class")
plt.xlabel("Class")
plt.ylabel("Number of examples")
plt.bar(x, y)
```

Out[0]:

```
<Container object of 10 artists>
```



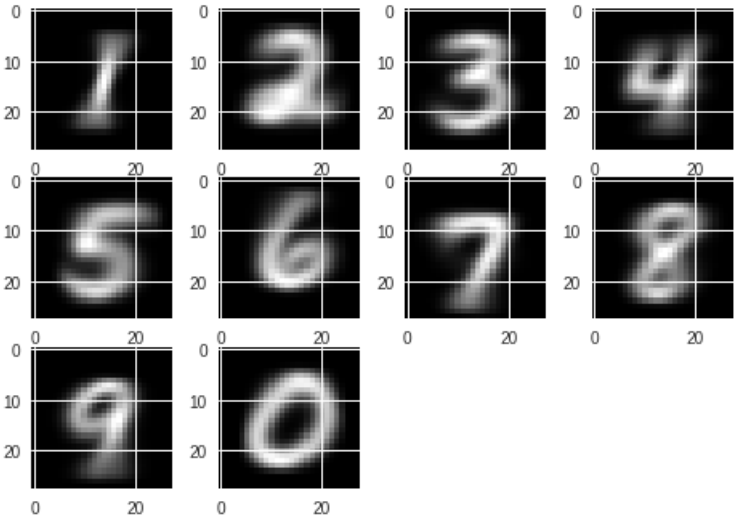
What does an average of each digit look like?

Let's take a look at the averages of each digit. On the images, the lighter the pixel is, the more people has used this space to draw the given digit. Conversely, the darker parts have been less used to drawing the digit.

In [0]:

```
groupped_labels = [np.where(Y_train == digit) for digit in range(10)]
groupped_digits = [X_train[groupped_labels[digit]] for digit in range(len(groupped_labels))]
groupped_digits_avg = [np.mean(groupped_digits[digit], axis=0) for digit in range(len(groupped_labe
ls))]
for row in range(3):
    for col in range(3):
        digit = row*3 + col + 1
        plt.subplot(3,4, digit)
        plt.imshow(groupped_digits_avg[digit], cmap="gray")

plt.subplot(3,4, 10)
plt.imshow(groupped_digits_avg[0], cmap="gray")
print()
```



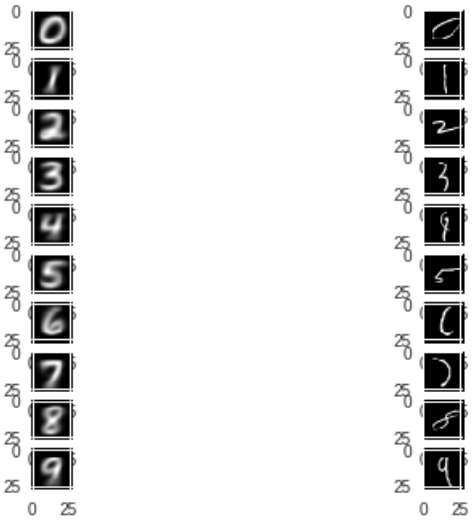
What are the images which are the least similar to the average, and nearest?

Least similar

In [0]:

```
for row in range(10):
  plt.subplot(10,2, row*2+1)
  plt.imshow(groupped_digits_avg[row], cmap="gray")
  worst = groupped_digits[row][0]
  best = groupped_digits[row][0]
  for img in groupped_digits[row]:
    if (worst - groupped_digits_avg[row]).mean() > (img - groupped_digits_avg[row]).mean():
      worst = img
```

```
  plt.subplot(10,2,row*2+2)
  plt.imshow(worst, cmap="gray")
```



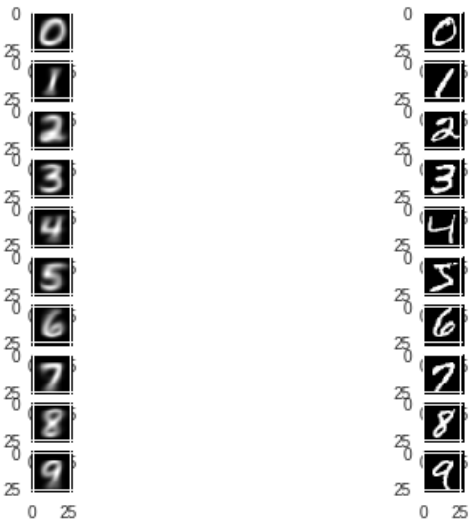
Here we can see which images are the least similar to our average images. We can point out that the six and the seven are far from the shape they should have.

Nearest to average

In [0]:

```
for row in range(10):
  plt.subplot(10,2, row*2+1)
  plt.imshow(groupped_digits_avg[row], cmap="gray")
  worst = groupped_digits[row][0]
  best = groupped_digits[row][0]
  for img in groupped_digits[row]:
    if (best - img).mean() < 10^-5:
      best = img

  plt.subplot(10,2,row*2+2)
  plt.imshow(best, cmap="gray")
```



How much gray is there in the set of images?

Making an average of the matrix corresponding to each image allows us to see if the input images are mostly black or white.

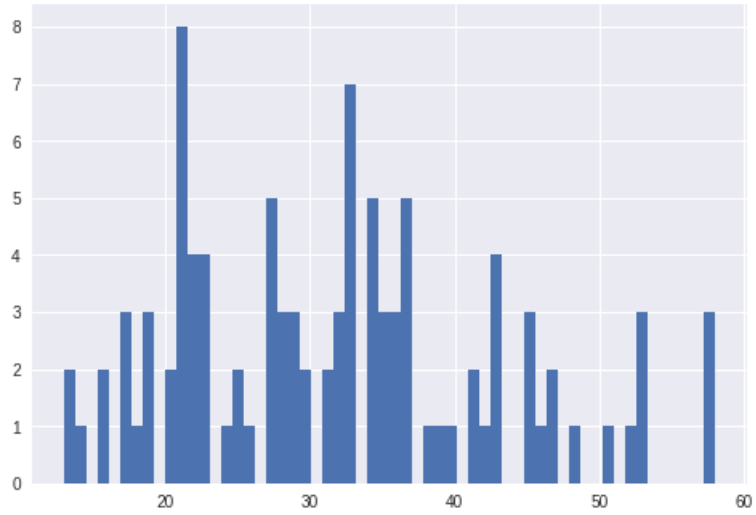
In [0]:

```
my_means =[]
for index in range(0,100):
    my_means = my_means + [int(X_train[index].mean())] #cast to int to group images by integer aver
```

```
ages

print(my_means)
plt.hist(my_means, bins=np.amax(my_means)) #bins equals to the maximum average we obtain
print() #to remove the print of the array from plt.hist
```

```
[35, 39, 24, 21, 29, 37, 22, 45, 13, 27, 36, 18, 45, 36, 14, 32, 31, 34, 17, 22, 43, 45, 19, 21,
28, 52, 17, 58, 53, 20, 32, 41, 22, 27, 46, 20, 40, 47, 27, 34, 17, 36, 16, 22, 21, 30, 29, 28, 2
1, 42, 23, 58, 35, 21, 30, 41, 53, 25, 43, 21, 37, 23, 38, 58, 33, 21, 34, 19, 31, 53, 28, 26, 13
, 33, 33, 43, 33, 21, 25, 29, 33, 47, 51, 34, 23, 33, 27, 35, 43, 32, 37, 33, 19, 37, 34, 48, 23,
37, 27, 16]
```



On this histogram, the image averages seem to be pretty well distributed. We can interprete this as the fact that our images have random gray scale values. What we mean is that there are no very dark or light images.

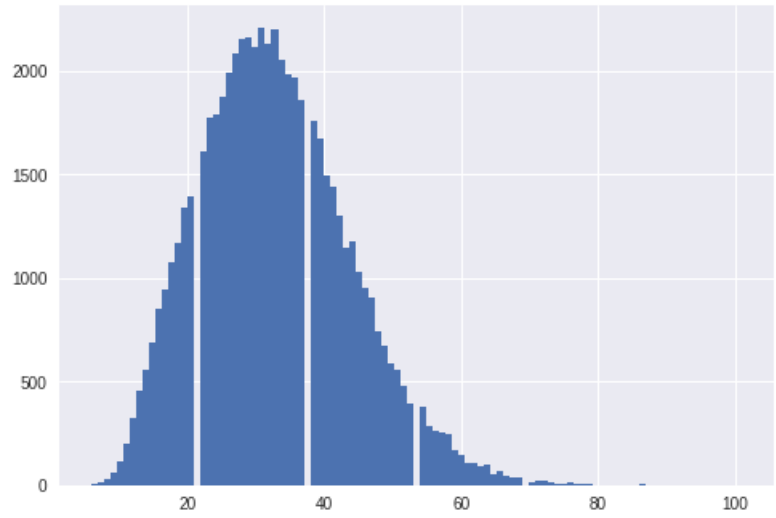
Also, it is true that we can't be sure of such deduced things from only 100 images over the 60000 training images we have. Let's try the same with all the images.

In [0]:

```
my_means =[]

for index in range(0, len(X_train)):
    my_means = my_means + [int(X_train[index].mean())] #cast to int to group images by integer aver
ages

print(my_means)
plt.hist(my_means, bins=np.amax(my_means)) #bins equals to the maximum average we obtain
print() #to remove the print of the array from plt.hist
```



First of all, from this histogram we can point out that making interpretation on few images is not significant. With all the images we can

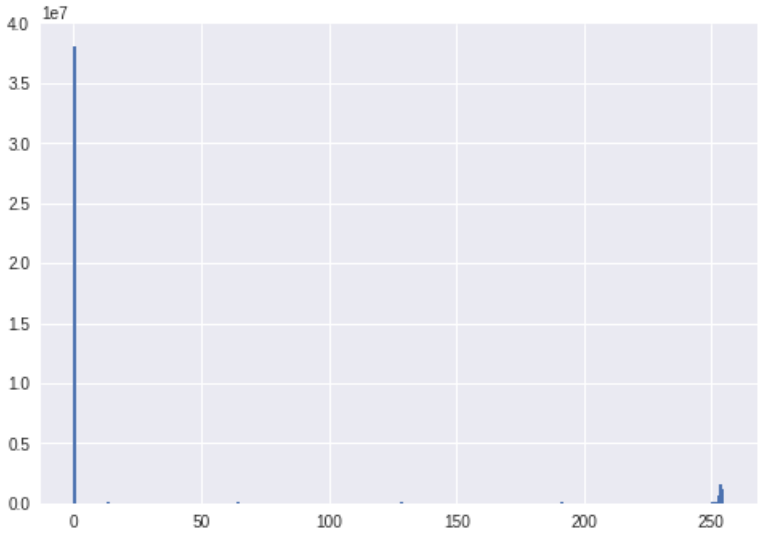
notice that the average values seem to take the form of a normal distribution, centered around 28. This is a good property for our input images as this means that we have few outliers. Then we could expect less good results for images which average are smaller than 10 or larger than 75, since our model will train of images with averages mostly between 20 and 40.

An other way to see how much gray there is in the dataset could be to plot the histogram of all pixels' values.

In [0]:

```
pixels =[]
pixels = X_train.flatten()

plt.hist(pixels, bins=255)
print()
```



From this histogram, we point out that the images in dataset are mostly black : approximately 3,8.10e7 pixel's value equals to 0, whereas there are "only" approximately 0,2.10e7 pixel's value equals to 255 (or almost) which is white.

Data pre-processing

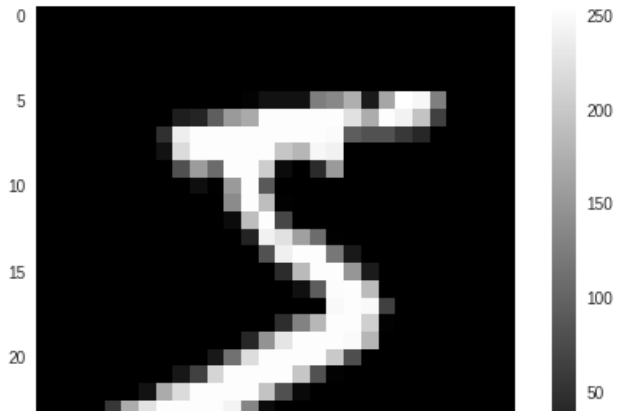
As said in the project working instructions, we have to split our training set into a training and a validation set. The latter will enable us to tune the hyperparameters of our model. Also, data pre-processing consists of normalizing the data.

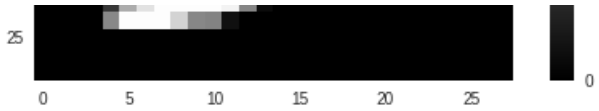
Normalization of the data

Let's remind first the range of pixels' values by looking at our first image.

In [0]:

```
plt.figure()
plt.imshow(X_train[0], cmap="gray")
plt.colorbar()
plt.grid(False)
```





By observing the first image we can see that the pixels' values range between 0 and 255. We want to have values between 0 and 1. That is why we have to normalize them. More exactly, by substracting the mean and then dividing by the standard deviation we standarize the data.

We use pixel-wise normalization here. This normalization aims to help the convergence rate of the training process and also zero-centers our input data.

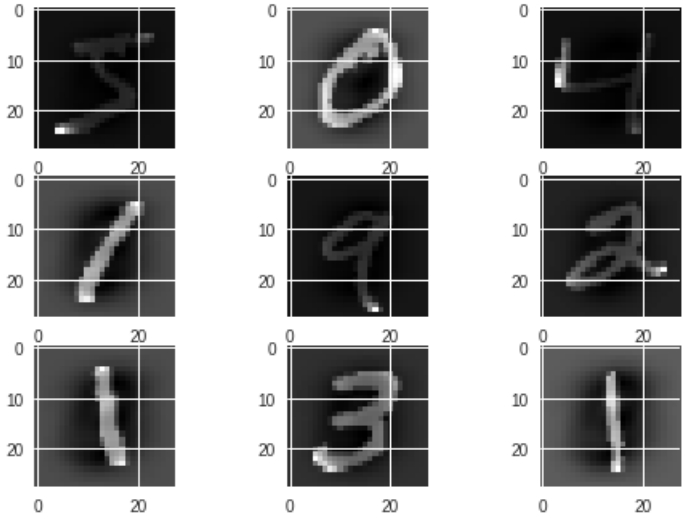
In [0]:

```
#mean
pixel_mean = X_train.mean(axis=0)
#standard deviation
pixel_std = X_train.std(axis=0) + 1e-10 # Prevent division-by-zero errors
# Normalize the train and test set
#we want the input to have values between 0 and 1
X_train = (X_train - pixel_mean) / pixel_std
X_test = (X_test - pixel_mean) / pixel_std
```

Then, we can even visualize the normalized data.

In [0]:

```
for row in range(3):
    for col in range(3):
        idx = row*3 + col + 1
        plt.subplot(3,3, idx)
        plt.imshow(X_train[idx-1], cmap="gray")
```



One-hot encoding

A one hot encoding allows the representation of categorical data to be more expressive. Indeed, a categorical data representation could raise some issues machine learning algorithms. Although some algorithms can work with categorical data directly, it is better to convert them into a numerical form. This will enable efficient implementation of machine learning algorithms.

$to__categorical()$ returns a binary matrix of its input Y_train and Y_test which have integers between 0 and 9.

In [0]:

```
#1-dimensional class arrays are converted to 10-dimensional class matrices because the ouput is a
number between 0 and 9
print(Y_train.shape)
Y_train = keras.utils.to_categorical(Y_train, num_classes)
Y_test = keras.utils.to_categorical(Y_test, num_classes)
print(Y_train.shape)
```

```
(60000,)
(60000, 10)
```

Change data shape

We have to convert our data to the required shape of tensor, which is (28, 28, 1). This also means that MNIST images have a depth of 1.

Also, we convert here our data type to float32 to reduce memory requirements.

In [0]:

```
print("Old shape:", X_train.shape)
X_train = X_train[:,:,:, np.newaxis].astype(np.float32)
X_test = X_test[:,:,:, np.newaxis].astype(np.float32)
print("New shape:", X_train.shape)
```

```
Old shape: (60000, 28, 28)
New shape: (60000, 28, 28, 1)
```

Split training set into training and validation

The test set will be used for testing purpose only, for our final evaluation.The validation set will be used to validate our model and tune different hyperparameters in our model.

We will discuss hyperparameter tuning during the training and the evaluation of our model later on.

In [0]:

```
train_val_split = 0.9 # Percentage of data to use in training set
indexes = np.arange(X_train.shape[0])
np.random.shuffle(indexes)
# Select random indexes for train/val set
idx_train = indexes[:int(train_val_split*X_train.shape[0])]
idx_val = indexes[int(train_val_split*X_train.shape[0]):]

X_val = X_train[idx_val]
Y_val = Y_train[idx_val]

X_train = X_train[idx_train]
Y_train = Y_train[idx_train]

print("Training set shape:", X_train.shape)
print("Validation set shape:", X_val.shape)
print("Testing set shape:", X_test.shape)
```

```
Training set shape: (54000, 28, 28, 1)
Validation set shape: (6000, 28, 28, 1)
Testing set shape: (10000, 28, 28, 1)
```

Implementation of the given network

Model construction

The given network in the project instruction is composed of two dense layers. The first one is a hidden layer with 128 neurons, and the second one is the output layer with 10 neurons (the number of classes). Relu is used as an activation function for the hidden layer, whereas we will us the softmax activation function for the output layer.

In [0]:

```
model = Sequential()
input_shape = X_train.shape[1:] # (28, 28, 1)
model.add(Flatten(input_shape=input_shape)) #transforms the format of the images from the input sh
ape 28*28 to a 1d-array
model.add(Dense(128, activation = "relu"))
```

```
model.add(Dense(num_classes, activation="softmax")) #num_classes = 10
#the layer is a 10-node softmax layer. It returns an array of 10 probabilities that sum to 1
#Each node contains a score that indicates the probability that the current image belongs to one o
f the 10 classes
```

Compilation, optimization function and loss metric

We then compile our model where the parameters are the loss, the optimizer and the metric.

Loss function

To compile a model we have to use a loss function that returns a scalar for each data-point. This loss function takes two arguments : true labels and predictions. It measures the performance of a classification model whose output is a probability value between 0 and 1.

Cross entropy

Cross entropy loss could also be called log loss. If the predicted probability diverges from the actual observation label, cross entropy increases. Then a perfect model, which means that the predicted probability is 1, would have a cross entropy loss of 0 (log loss : log(1) = 0).

Therefore, probability decreases as log loss increases rapidly, and as predicted probability increases, log loss decreases slowly.

Different optimization algorithm

Optimization algorithms help us to minimize (or maximize) the error function, which is our most important objective while doing back propagation. This function depends on the learnable parameters of the network : the weights and the biases.

Standard stochastic Gradient Descent (SGD)

In SGD we perform a paramer update for each training example and label. Because of the more frequent updates, the objective function will fluctuate much more, which will help to find a possibly better local minima. Though it complicates to converge to the exact minimum.

Adam optimizer

Adam or ADAptive Momentum estimation builds on previous optimization techniques, Momentum and ADAptive Gradient (ADAGrad). Momentum helps to navigate along the relevant directions, and softens the fluctuations along the irrelevant directions, by adding a fraction of the direction of the previous step to the current step. It results in faster convergence and reduced oscillations. ADAGrad will change the learning rate for every parameter at a given time step based on the previous gradients for that given parameter that were computed earlier. So it makes big updates for infrequent parameters and smaller ones for frequent parameters. This means that we don't have to manually tune the learning rate. (Though ADAGrad leads to a constantly decreasing learning rate which can result in no updates after a while. ADADelta fixes that) So what ADAM does, is that it stores momentum AND learning rates for each parameter separatly.

Nesterov Adam optimizer

Nesterov saw a problem with Momentum. When getting near to the minimum, the Momentum becomes very high. It can lead to that the estimation will skip the minimum entirely. Nesterov solved it by changing the order how Momentum works. First it takes a jump based on the previous Momentum, calculetes the gradient and then makes a correction, which results in an update. The rest of the optimizer works just like ADAM.

In all our networks, we tried to use Nadam instead of Adam sometimes. However, it seems, according to some litterature, that Adam is the best optimizer to use. That is why we are going to use Adam in each of our models.

Metrics

When a model is compiled, we can put a special metric function which will judge the performance of the model.

In all our models we will use categorical accuracy.

In [0]:

```
#before trainning the model we have a few more compiling steps
#the loss function measures how accurate the model is
#optimizer : measures how the model is updated
```

```
learning_rate = 0.05

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=learning_rate),
                metrics=['accuracy'])

#accuracy : the fraction of images that are correctly classified

#other options :
#model.compile(loss=keras.losses.categorical_crossentropy,
#              optimizer=keras.optimizers.Adam(learning_rate),
#                metrics=['accuracy'])


#model.compile(loss=keras.losses.categorical_crossentropy,
#               optimizer=keras.optimizers.RMSprop(learning_rate),
 #               metrics=['accuracy'])

#model.compile(loss=keras.losses.sparse_categorical_crossentropy,
 #              optimizer=keras.optimizers.Adam(learning_rate),
  #               metrics=['sparse_categorical_accuracy'])
```

Model summary

In [0]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_50 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_134 (Dense)            (None, 128)               100480
_________________________________________________________________
dense_135 (Dense)            (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

Model training

Before training our model, we have to choose two other hyperparameters which are the number of epochs and the batch size. The number of epochs is the number of times we choose to train our model. The batch size represents the number of samples which will propagate through the network.

It is important to point out that the batch size needs to be determined depending on the network (as for the learning rate). Indeed a batch size smaller than the number of samples requires less memory and enables the network to train faster. However, in this case the gradient will also be less accurate (more oscillations). That is why, we will have to update the batch size in order to find one in between.

In [0]:

```
number_of_epochs = 10
batch_size=128

model.fit(X_train, Y_train,
          batch_size=batch_size,
          epochs=number_of_epochs,
          verbose=1,
          validation_data=(X_val, Y_val))
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 4s 81us/step - loss: 0.4126 - acc: 0.8858 -
val_loss: 0.2422 - val_acc: 0.9310
Epoch 2/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.2067 - acc: 0.9409 -
```

```
val_loss: 0.1937 - val_acc: 0.9437
Epoch 3/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.1617 - acc: 0.9539 -
val_loss: 0.1722 - val_acc: 0.9517
Epoch 4/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.1354 - acc: 0.9616 -
val_loss: 0.1579 - val_acc: 0.9550
Epoch 5/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.1171 - acc: 0.9674 -
val_loss: 0.1479 - val_acc: 0.9577
Epoch 6/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.1034 - acc: 0.9713 -
val_loss: 0.1416 - val_acc: 0.9603
Epoch 7/10
54000/54000 [==============================] - 2s 43us/step - loss: 0.0921 - acc: 0.9751 -
val_loss: 0.1342 - val_acc: 0.9635
Epoch 8/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.0825 - acc: 0.9778 -
val_loss: 0.1354 - val_acc: 0.9635
Epoch 9/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.0753 - acc: 0.9798 -
val_loss: 0.1244 - val_acc: 0.9655
Epoch 10/10
54000/54000 [==============================] - 2s 42us/step - loss: 0.0688 - acc: 0.9822 -
val_loss: 0.1247 - val_acc: 0.9667
```

Out[0]:

```
<keras.callbacks.History at 0x7fa9d64cc438>
```

The code above implements the given network from the project instruction. We chose to let the number of epochs and the batch size as it was provided in $Keras__example$ folder, as well as the loss function, the optimizer and the metric which have been used. However, we changed the learning rate hyperparameter to 0.05. This change enable us to obtain an accuracy of approximately 96% (depending on each running) after 10 epochs. This code is then better than the one provided in $Keras__example$, with only 91%.

To get a higheer accuracyl we only had to modify the learning rate. This shows us that it is a very significant hyperparameter which should be very well chosen.

This result is better than before, but the model still has one major issue: it has too many parameters: 101.770! The following part of our project consists of obtaining a good accuracy - over 95% - while taking into account the number of parameters. Indeed, the more parameters we have, the more complicated the network becomes.

Our network (simplest fully-connected network with accuracy >95%)

To find the simplest fully-connected network, our goal is to reduce the number of parameters in our network. To do so, we chose to add hidden layers to our network but containing less neurons.

Model construction

In [0]:

```
model = Sequential()
input_shape = X_train.shape[1:] # (28, 28, 1)
model.add(Flatten(input_shape=X_train.shape[1:])) #transforms the format of the images from the in
put shape 28*28 to a 1d-array

model.add(Dense(13, activation = "relu"))
model.add(Dense(128, activation = "relu"))


model.add(Dense(num_classes, activation="softmax")) #num_classes = 10
```

Compilation, optimization function and loss metric

We chose here to modify our optimizer to use a supposed better one: Adam optimizer. At the same time, we lower our learning rate.

In [0]:

```
learning_rate = 0.00045

model.compile(loss=keras.losses.categorical_crossentropy,
            optimizer=keras.optimizers.Adam(lr=learning_rate),
              metrics=['accuracy'])
```

Model summary

In [0]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_51 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_136 (Dense)            (None, 13)                10205
_________________________________________________________________
dense_137 (Dense)            (None, 128)               1792
_________________________________________________________________
dense_138 (Dense)            (None, 10)                1290
=================================================================
Total params: 13,287
Trainable params: 13,287
Non-trainable params: 0
_________________________________________________________________
```

Model training

In first thought, we chose to reduce the batch size in order to train faster. Also, by doing this modification we pointed out that we got better results. The risk of reducing this number is to create changes in the gradient direction. But since this choice seemed to produce a good training we chose to keep it at 32.

In [0]:

```
number_of_epochs = 10
batch_size=32

model.fit(X_train, Y_train,
          batch_size=batch_size,
          epochs=number_of_epochs,
          verbose=1,
          validation_data=(X_val, Y_val))
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/13
54000/54000 [==============================] - 14s 253us/step - loss: 0.4629 - acc: 0.8600 - val_l
oss: 0.2470 - val_acc: 0.9250
Epoch 2/13
54000/54000 [==============================] - 11s 209us/step - loss: 0.2278 - acc: 0.9322 - val_l
oss: 0.2130 - val_acc: 0.9345
Epoch 3/13
54000/54000 [==============================] - 11s 210us/step - loss: 0.1914 - acc: 0.9431 - val_l
oss: 0.2046 - val_acc: 0.9382
Epoch 4/13
54000/54000 [==============================] - 11s 210us/step - loss: 0.1703 - acc: 0.9485 - val_l
oss: 0.1882 - val_acc: 0.9412
Epoch 5/13
54000/54000 [==============================] - 11s 211us/step - loss: 0.1559 - acc: 0.9525 - val_l
oss: 0.1827 - val_acc: 0.9453
Epoch 6/13
54000/54000 [==============================] - 11s 209us/step - loss: 0.1439 - acc: 0.9566 - val_l
oss: 0.1689 - val_acc: 0.9495
Epoch 7/13
54000/54000 [==============================] - 11s 209us/step - loss: 0.1338 - acc: 0.9585 - val_l
oss: 0.1693 - val_acc: 0.9495
Epoch 8/13
54000/54000 [==============================] - 11s 213us/step - loss: 0.1267 - acc: 0.9614 - val_l
```

```
oss: 0.1742 - val_acc: 0.9497
Epoch 9/13
54000/54000 [==============================] - 11s 212us/step - loss: 0.1202 - acc: 0.9627 - val_l
oss: 0.1649 - val_acc: 0.9537
Epoch 10/13
54000/54000 [==============================] - 11s 211us/step - loss: 0.1138 - acc: 0.9648 - val_l
oss: 0.1652 - val_acc: 0.9518
Epoch 11/13
54000/54000 [==============================] - 11s 209us/step - loss: 0.1090 - acc: 0.9666 - val_l
oss: 0.1642 - val_acc: 0.9517
Epoch 12/13
54000/54000 [==============================] - 11s 209us/step - loss: 0.1048 - acc: 0.9674 - val_l
oss: 0.1666 - val_acc: 0.9515
Epoch 13/13
54000/54000 [==============================] - 11s 207us/step - loss: 0.1004 - acc: 0.9688 - val_l
oss: 0.1717 - val_acc: 0.9532
```

Out[0]:

```
<keras.callbacks.History at 0x7fa9d3151be0>
```

Now this model is able to reach 95%, but with about 1/10 of the parameters of the original model! That is some big performance improvement.

Model with highest test accuracy

Model construction

In [0]:

```
model = Sequential()


model.add(Flatten(input_shape=input_shape)) #transforms the format of the images from the input sh
ape 28*28 to a 1d-array
model.add(Dense(256, activation = "relu"))
model.add(BatchNormalization())
model.add(Dense(128, activation = "relu"))
model.add(BatchNormalization())
model.add(Dense(64, activation = "relu"))
model.add(BatchNormalization())
model.add(Dense(10, activation='softmax'))
```

Compilation, optimization function and loss metric

In [0]:

```
learning_rate = 0.00035

model.compile(loss=keras.losses.categorical_crossentropy,
            optimizer=keras.optimizers.Adam(learning_rate),
              metrics=['accuracy'])
```

Model summary

In [0]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_52 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_139 (Dense)            (None, 256)               200960
```

```
_________________________________________________________________
batch_normalization_4 (Batch (None, 256)               1024
_________________________________________________________________
dense_140 (Dense)            (None, 128)               32896
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
dense_141 (Dense)            (None, 64)                8256
_________________________________________________________________
batch_normalization_6 (Batch (None, 64)                256
_________________________________________________________________
dense_142 (Dense)            (None, 10)                650
=================================================================
Total params: 244,554
Trainable params: 243,658
Non-trainable params: 896
_________________________________________________________________
```

Model training

In [0]:

```
number_of_epochs = 10
batch_size=256

model.fit(X_train, Y_train,
          batch_size=batch_size,
          epochs=number_of_epochs,
          verbose=1,
          validation_data=(X_val, Y_val),
          )
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 6s 112us/step - loss: 0.5005 - acc: 0.8575 -
val_loss: 0.2148 - val_acc: 0.9387
Epoch 2/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.1695 - acc: 0.9521 -
val_loss: 0.1577 - val_acc: 0.9552
Epoch 3/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.1076 - acc: 0.9704 -
val_loss: 0.1273 - val_acc: 0.9637
Epoch 4/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.0736 - acc: 0.9807 -
val_loss: 0.1157 - val_acc: 0.9652
Epoch 5/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.0520 - acc: 0.9873 -
val_loss: 0.1057 - val_acc: 0.9675
Epoch 6/10
54000/54000 [==============================] - 3s 57us/step - loss: 0.0370 - acc: 0.9916 -
val_loss: 0.1038 - val_acc: 0.9697
Epoch 7/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.0259 - acc: 0.9948 -
val_loss: 0.0997 - val_acc: 0.9712
Epoch 8/10
54000/54000 [==============================] - 3s 58us/step - loss: 0.0194 - acc: 0.9968 -
val_loss: 0.1006 - val_acc: 0.9700
Epoch 9/10
54000/54000 [==============================] - 3s 57us/step - loss: 0.0138 - acc: 0.9979 -
val_loss: 0.0942 - val_acc: 0.9725
Epoch 10/10
54000/54000 [==============================] - 3s 59us/step - loss: 0.0112 - acc: 0.9985 -
val_loss: 0.0939 - val_acc: 0.9732
```

Out[0]:

```
<keras.callbacks.History at 0x7fa9d4770518>
```

Final analysis and evaluation of our best model

Hyperparameters tuning

Number of layers

We set the number of layers by trying. However, the number of layers should be a reasonable number as the larger it is the higher we could loose performance in our model.

Learning rate

We adapted the learning rate while trying training our network.

Number of epochs

The number of epochs is a hyperparameter determining the number of complete passes through all the training data. After some iterations, we should be able to observe some kind of converging. If we seem to be "locked" to an accuracy which we start to only "bounce around", probably other hyperparameters need furhter tuning to get a different result.

Batch-size

The batch size is the number of samples propagated through the network during one iteration. That means that we have to work on this number of training samples before updating the parameters. Then, it affects the number of parameters updates.

On one hand, the smaller the batch size is, the less accurate gradient we can get. Indeed, the gradient will less change its direction by considering a high batch size. On the other hand, the smaller the batch is, the more we update parameters and then adapt our network to the samples going trough it. A network is training faster when the batch size is small (due to a smaller memory footprint).

We have chosen all our batch sizes regarding those effects and we tried to find a good value according to our results' observations.

Tracking validation loss

When our model has been trained, we should always compare the difference between the validation and training loss/accuracy. (Training loss/accuracy comes from the actual data we have been training the model on, while validation loss/accuracy is the part of the data the network has not seen before). This part is crucial to be able to determine if our model has been generalized enough. What we want to see is that when the model has been finished training, ideally the validation and training results has the smallest possible gap. In other words, if the training result has a much better outcome, that means that we have got an overfit model for the training data. This network could predict the output of data that is similar to the training data very well, but would perform much worse on never-seen datasets. Though, usually the goal of a model is to perform best on new data. On the figures below, we can see that in both cases the validation gets a better score. Though it is not favorable, we can still confidently say that our network has done a good job, as the gap is pretty small between the training and validation results. An ideal model should have no gap between the two results. Though we think, it would be hard to achieve. As an example, you will always be more confident to predict outcomes of situations you have been in, rather than those you have not been in before.

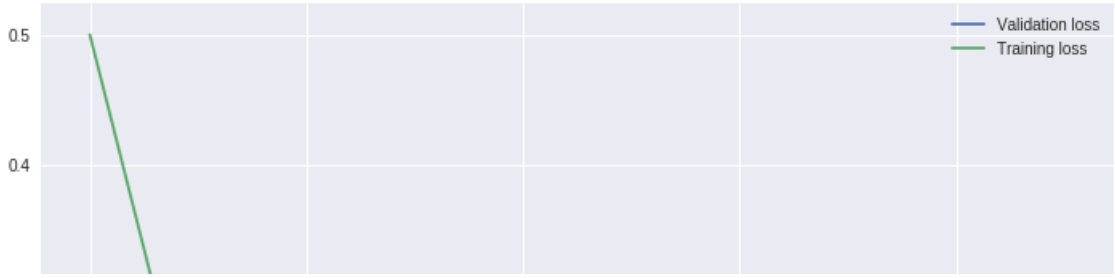
Loss history

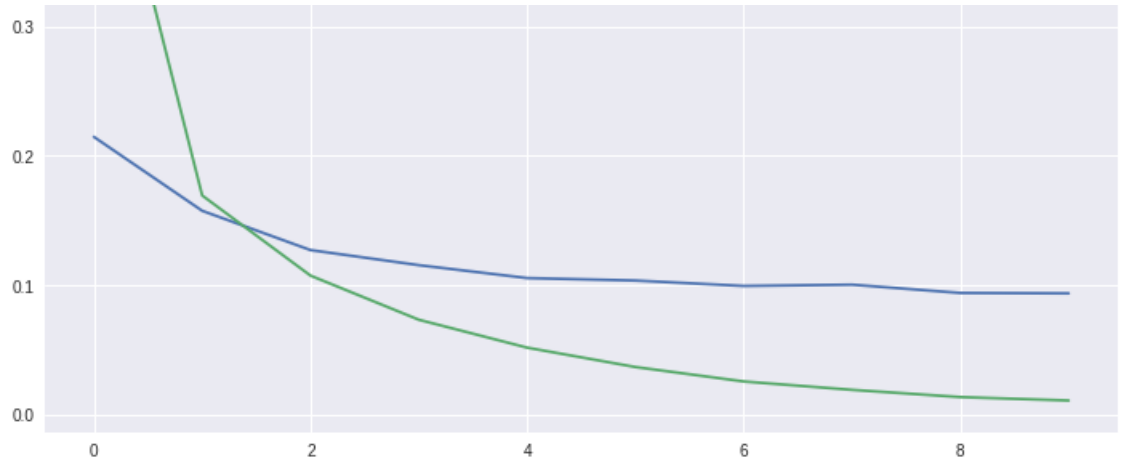
In [0]:

```
history = model.history.history
plt.figure(figsize=(12, 8))
plt.plot(history["val_loss"], label="Validation loss")
plt.plot(history["loss"], label="Training loss")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7fa9d2f07d30>
```





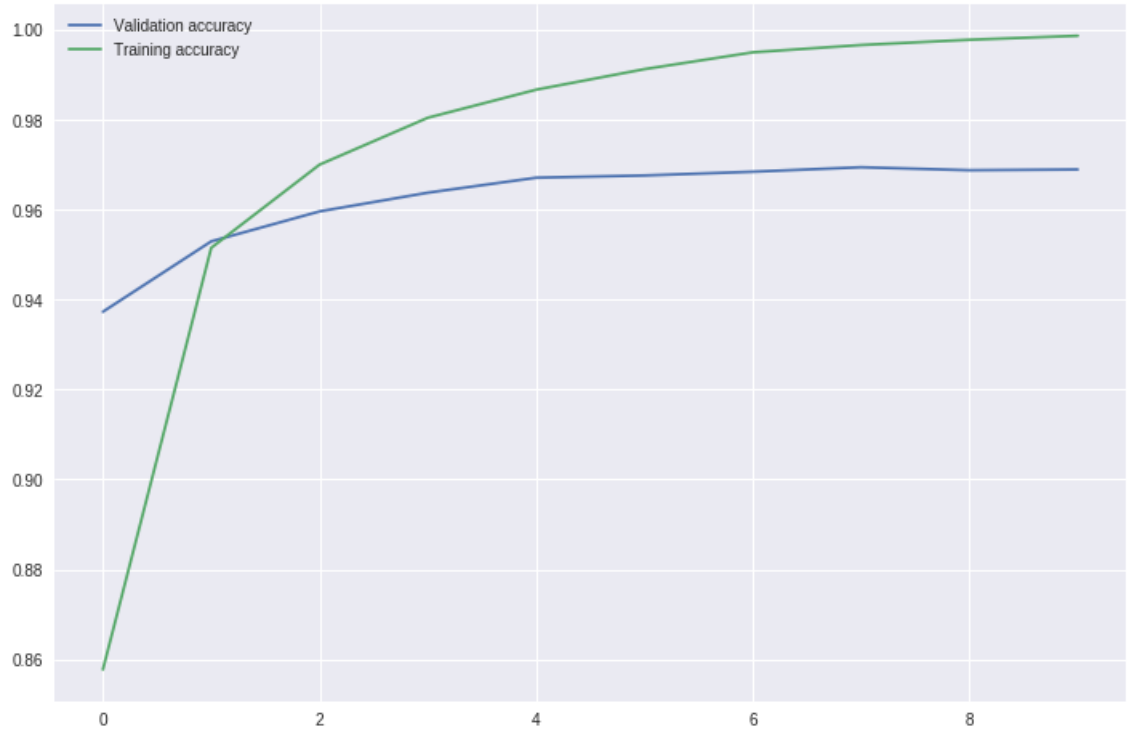
Accuracy history

In [0]:

```
plt.figure(figsize=(12, 8))
plt.plot(history["val_acc"], label="Validation accuracy")
plt.plot(history["acc"], label="Training accuracy")
plt.legend()
```

Out[0]:

```
<matplotlib.legend.Legend at 0x7fa9d8a338d0>
```



Analysis of performance

Confusion matrix

The confusion matrix allows us to visualize the performance of the algorithm. This matrix can be seen as a summary of the number of correct and incorrect predictions. Then we will know for which class our model is confused.

In [0]:

```
expected = np.argmax(Y_val, axis=1)
predicted = np.argmax(model.predict(X_val), axis=1)
```

```
print ("Confusion matrix")
print()
conf_m = confusion_matrix(expected,predicted)
print(conf_m)
print()
print("Total of the digits estimated by the model : ")
print("[0   1   2   3   4   5   6   7   8   9]")
print(sum(conf_m.transpose()))

print()
print("Report of classification :")
print()
report = classification_report(expected, predicted)

print(report)
```

```
Confusion matrix

[[599   0   1   0   1   0   2   2   1   1]
 [  0 659   4   1   0   0   1   1   1   1]
 [  0   2 562   4   5   0   0   3   1   1]
 [  2   1   5 590   0   6   0   2   6   4]
 [  0   0   1   0 561   0   3   3   4   8]
 [  0   1   0   3   0 530   7   0   5   4]
 [  4   0   2   0   2   3 610   1   0   0]
 [  0   2   3   1   3   0   0 598   2   4]
 [  1   5   0   2   0   1   2   0 561   0]
 [  1   0   0   2   7   3   0   6   6 569]]

Total of the digits estimated by the model :
[0   1   2   3   4   5   6   7   8   9]
[607 668 578 616 580 550 622 613 572 594]

Report of classification :

             precision    recall  f1-score   support

          0       0.99      0.99      0.99       607
          1       0.98      0.99      0.99       668
          2       0.97      0.97      0.97       578
          3       0.98      0.96      0.97       616
          4       0.97      0.97      0.97       580
          5       0.98      0.96      0.97       550
          6       0.98      0.98      0.98       622
          7       0.97      0.98      0.97       613
          8       0.96      0.98      0.97       572
          9       0.96      0.96      0.96       594

avg / total       0.97      0.97      0.97      6000
```

Computations from the confusion matrix

The confusion matrix allows us to make some percentage computations which enable us to interpret it. In this matrix, we can find the so called: "false positives" (FP), "true positives" (TP), "false negatives" (FN) and "true negatives" (TN). Each of them stands for, respectively :

- FP : when we predicted as a "yes, this corresponds to this class", but it is actually not.
- TP : when we predicted as a "yes, this corresponds to this class", and it is.
- FN : when we predicted as a "no, this doesn't correspond to this class", but it is this class in reality.
- TN : when we predicted as a "no, this doesn't correspond to this class", and it is not this class.

Thanks to those values, we then can compute some ratios which can answer some of the questions we might ask ourselves after training a network. These are the following :

- How much is our model correct in overall ? (accuracy)
- By regarding all the results, how often our model is wrong ? (missclassification rate)
- How often does it predict "this is this class" when it is actually this one ? (sensitivity or recall)
- How often does it predict "this is this class" when it is not this one ?
- How often does it predict "this is not this class" when it is actually not ? (specificity)
- How often does it predict "this is not this class" when it is ?
- When our model is predicting "yes this is this class", how often is it correct ? (precision)

In [0]:

```
TP = np.diag(conf_m)
FP = np.sum(conf_m, axis=0) - TP
FN = np.sum(conf_m, axis=1) - TP

num_classes = 10
TN = []
for i in range(num_classes):
    temp = np.delete(conf_m, i, 0)    # delete row i
    temp = np.delete(temp, i, 1)  # delete column i
    TN.append(sum(sum(temp)))

print(TP)
print()
print(FP)
print()
print(FN)
print()
print(TN)
```

```
[599 659 562 590 561 530 610 598 561 569]

[ 8 11 16 13 18 13 15 18 26 23]

[ 8  9 16 26 19 20 12 15 11 25]

[5385, 5321, 5406, 5371, 5402, 5437, 5363, 5369, 5402, 5383]
```

In [0]:

```
acc = (TN+TP)/(TP+TN+FP+FN)  #true over total
miss_class = (FP+FN)/(TP+TN+FP+FN) #false over total
precision = TP/(TP+FP)
sensitivity = TP/(TP + FN)
specificity = TN/(FP + TN)
FP_rate = FP/TN #how often does it predict "yes this is this class" but it is not
FN_rate = FN/TP #how often does it predict "no it's not this class" but it actually is

for number_class in range(num_classes):
  print()
  print("Class " + str(number_class + 1) + " (digit " + str(number_class) +") :")
  print("accuracy: " + str(acc[number_class]))
  print("missclassification : " + str(miss_class[number_class]))
  print("precision : " + str(precision[number_class]))
  print("sensitivity : " + str(sensitivity[number_class]))
  print("specificity : " + str(specificity[number_class]))
  #print("false positive : " + str(FP_rate[number_class]))
  #print("false negative : " + str(FN_rate[number_class]))
  #print(str( FN_rate[number_class] + sensitivity[number_class]))
  #print(FP_rate[number_class] + specificity[number_class])
```

```
Class 1 (digit 0) :
accuracy: 0.9973333333333333
missclassification : 0.0026666666666666666
precision : 0.9868204283360791
sensitivity : 0.9868204283360791
specificity : 0.9985165955868719

Class 2 (digit 1) :
accuracy: 0.9966666666666667
missclassification : 0.0033333333333333335
precision : 0.9835820895522388
sensitivity : 0.9865269461077845
specificity : 0.9979369842460615

Class 3 (digit 2) :
accuracy: 0.9946666666666667
missclassification : 0.005333333333333333
precision : 0.972318339100346
sensitivity : 0.972318339100346
specificity : 0.9970490593876798

Class 4 (digit 3) :
```

```
accuracy: 0.9935
missclassification : 0.0065
precision : 0.978441127694859
sensitivity : 0.9577922077922078
specificity : 0.9975854383358098

Class 5 (digit 4) :
accuracy: 0.9938333333333333
missclassification : 0.006166666666666667
precision : 0.9689119170984456
sensitivity : 0.9672413793103448
specificity : 0.9966789667896679

Class 6 (digit 5) :
accuracy: 0.9945
missclassification : 0.0055
precision : 0.9760589318600368
sensitivity : 0.9636363636363636
specificity : 0.9976146788990826

Class 7 (digit 6) :
accuracy: 0.9955
missclassification : 0.0045
precision : 0.976
sensitivity : 0.9807073954983923
specificity : 0.997210859055411

Class 8 (digit 7) :
accuracy: 0.9945
missclassification : 0.0055
precision : 0.9707792207792207
sensitivity : 0.9755301794453507
specificity : 0.996658622609987

Class 9 (digit 8) :
accuracy: 0.9938333333333333
missclassification : 0.006166666666666667
precision : 0.9557069846678024
sensitivity : 0.9807692307692307
specificity : 0.9952100221075902

Class 10 (digit 9) :
accuracy: 0.992
missclassification : 0.008
precision : 0.9611486486486487
sensitivity : 0.9579124579124579
specificity : 0.9957454679985202
```

In [0]:

```
min_prec = min(precision)
max_prec = max(precision)
print("Highest precision digit %d: %f" % (np.where(precision==max_prec)[0][0], max_prec))
print("Lowest precision digit %d: %f" % (np.where(precision==min_prec)[0][0], min_prec))
print("Difference: %f" % (max_prec - min_prec))
```

```
Highest precision digit 0: 0.986820
Lowest precision digit 8: 0.955707
Difference: 0.031113
```

Interpretation

Regarding precision, the results for different digits vary ~3%, 1 being the most precisly predicted digit, and 9 the least. The digit 1 is represented possibly the easiest way, with virtually no variation how people are drawing it (considering the data was only collected in US, where people do not draw a little tail on the top). 9 though, has a shape that can be easily confused with other digits, like 3, 6 or 8 because of their curves which they all share.

Regarding missunderstanding rate, we can see that its order is of 10^-2. That means that our network is not oftenly wrong, which is a good thing to point out.

Also, specificity and accuracy are both 0.99 for all the digits which indicates the good correctness of our classifier: in overall, it is correct (accuracy), and it is able to say wich a high confidence percentage that it is not a given digit if it's actually not.

F1-score is a mix between precision and recall (harmonic average). The closer is f1-score to 1, the better it is. Most of the times, the lowest one can't be under 0.95 (oftenly 0.96). This rate is taking into account how much precision and recall are similar. That is why this rate can tell us if the predictions from our classifier are good. We can say that 0.96 as a minimum is a good f1-score and confirm that our classifier is quite correct.