Introduction to CNN Keras - Acc 0.997 (top 8%)

**Yassine Ghouzam, PhD**

18/07/2017

* **1. Introduction**
* **2. Data preparation**
  + 2.1 Load data
  + 2.2 Check for null and missing values
  + 2.3 Normalization
  + 2.4 Reshape
  + 2.5 Label encoding
  + 2.6 Split training and valdiation set
* **3. CNN**
  + 3.1 Define the model
  + 3.2 Set the optimizer and annealer
  + 3.3 Data augmentation
* **4. Evaluate the model**
  + 4.1 Training and validation curves
  + 4.2 Confusion matrix
* **5. Prediction and submition**
  + 5.1 Predict and Submit results

1. Introduction

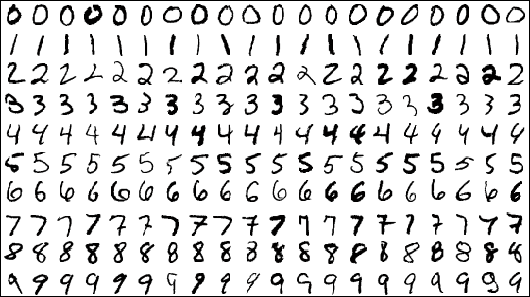
This is a 5 layers Sequential Convolutional Neural Network for digits recognition trained on MNIST dataset. I choosed to build it with keras API (Tensorflow backend) which is very intuitive. Firstly, I will prepare the data (handwritten digits images) then i will focus on the CNN modeling and evaluation.

I achieved 99.671% of accuracy with this CNN trained in 2h30 on a single CPU (i5 2500k). For those who have a >= 3.0 GPU capabilites (from GTX 650 - to recent GPUs), you can use tensorflow-gpu with keras. Computation will be much much faster !!!

**For computational reasons, i set the number of steps (epochs) to 2, if you want to achieve 99+% of accuracy set it to 30.**

This Notebook follows three main parts:

* The data preparation
* The CNN modeling and evaluation
* The results prediction and submission



In [1]:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import seaborn as sns

%matplotlib inline

np.random.seed(2)

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

import itertools

from keras.utils.np\_utils import to\_categorical *# convert to one-hot-encoding*

from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D

from keras.optimizers import RMSprop

from keras.preprocessing.image import ImageDataGenerator

from keras.callbacks import ReduceLROnPlateau

sns.set(style='white', context='notebook', palette='deep')

Using TensorFlow backend.

2. Data preparation

2.1 Load data

In [2]:

*# Load the data*

train = pd.read\_csv("../input/train.csv")

test = pd.read\_csv("../input/test.csv")

In [3]:

Y\_train = train["label"]

*# Drop 'label' column*

X\_train = train.drop(labels = ["label"],axis = 1)

*# free some space*

del train

g = sns.countplot(Y\_train)

Y\_train.value\_counts()

Out[3]:

1 4684

7 4401

3 4351

9 4188

2 4177

6 4137

0 4132

4 4072

8 4063

5 3795

Name: label, dtype: int64

We have similar counts for the 10 digits.

2.2 Check for null and missing values

In [4]:

*# Check the data*

X\_train.isnull().any().describe()

Out[4]:

count 784

unique 1

top False

freq 784

dtype: object

In [5]:

test.isnull().any().describe()

Out[5]:

count 784

unique 1

top False

freq 784

dtype: object

I check for corrupted images (missing values inside).

There is no missing values in the train and test dataset. So we can safely go ahead.

2.3 Normalization

We perform a grayscale normalization to reduce the effect of illumination's differences.

Moreover the CNN converg faster on [0..1] data than on [0..255].

In [6]:

*# Normalize the data*

X\_train = X\_train / 255.0

test = test / 255.0

2.3 Reshape

In [7]:

*# Reshape image in 3 dimensions (height = 28px, width = 28px , canal = 1)*

X\_train = X\_train.values.reshape(-1,28,28,1)

test = test.values.reshape(-1,28,28,1)

Train and test images (28px x 28px) has been stock into pandas.Dataframe as 1D vectors of 784 values. We reshape all data to 28x28x1 3D matrices.

Keras requires an extra dimension in the end which correspond to channels. MNIST images are gray scaled so it use only one channel. For RGB images, there is 3 channels, we would have reshaped 784px vectors to 28x28x3 3D matrices.

2.5 Label encoding

In [8]:

*# Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])*

Y\_train = to\_categorical(Y\_train, num\_classes = 10)

Labels are 10 digits numbers from 0 to 9. We need to encode these lables to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0]).

2.6 Split training and valdiation set

In [9]:

*# Set the random seed*

random\_seed = 2

In [10]:

*# Split the train and the validation set for the fitting*

X\_train, X\_val, Y\_train, Y\_val = train\_test\_split(X\_train, Y\_train, test\_size = 0.1, random\_state=random\_seed)

I choosed to split the train set in two parts : a small fraction (10%) became the validation set which the model is evaluated and the rest (90%) is used to train the model.

Since we have 42 000 training images of balanced labels (see 2.1 Load data), a random split of the train set doesn't cause some labels to be over represented in the validation set. Be carefull with some unbalanced dataset a simple random split could cause inaccurate evaluation during the validation.

To avoid that, you could use stratify = True option in train\_test\_split function (**Only for >=0.17 sklearn versions**).

We can get a better sense for one of these examples by visualising the image and looking at the label.

In [11]:

*# Some examples*

g = plt.imshow(X\_train[0][:,:,0])

3. CNN

3.1 Define the model

I used the Keras Sequential API, where you have just to add one layer at a time, starting from the input.

The first is the convolutional (Conv2D) layer. It is like a set of learnable filters. I choosed to set 32 filters for the two firsts conv2D layers and 64 filters for the two last ones. Each filter transforms a part of the image (defined by the kernel size) using the kernel filter. The kernel filter matrix is applied on the whole image. Filters can be seen as a transformation of the image.

The CNN can isolate features that are useful everywhere from these transformed images (feature maps).

The second important layer in CNN is the pooling (MaxPool2D) layer. This layer simply acts as a downsampling filter. It looks at the 2 neighboring pixels and picks the maximal value. These are used to reduce computational cost, and to some extent also reduce overfitting. We have to choose the pooling size (i.e the area size pooled each time) more the pooling dimension is high, more the downsampling is important.

Combining convolutional and pooling layers, CNN are able to combine local features and learn more global features of the image.

Dropout is a regularization method, where a proportion of nodes in the layer are randomly ignored (setting their wieghts to zero) for each training sample. This drops randomly a propotion of the network and forces the network to learn features in a distributed way. This technique also improves generalization and reduces the overfitting.

'relu' is the rectifier (activation function max(0,x). The rectifier activation function is used to add non linearity to the network.

The Flatten layer is use to convert the final feature maps into a one single 1D vector. This flattening step is needed so that you can make use of fully connected layers after some convolutional/maxpool layers. It combines all the found local features of the previous convolutional layers.

In the end i used the features in two fully-connected (Dense) layers which is just artificial an neural networks (ANN) classifier. In the last layer(Dense(10,activation="softmax")) the net outputs distribution of probability of each class.

In [12]:

*# Set the CNN model*

*# my CNN architechture is In -> [[Conv2D->relu]\*2 -> MaxPool2D -> Dropout]\*2 -> Flatten -> Dense -> Dropout -> Out*

model = Sequential()

model.add(Conv2D(filters = 32, kernel\_size = (5,5),padding = 'Same',

activation ='relu', input\_shape = (28,28,1)))

model.add(Conv2D(filters = 32, kernel\_size = (5,5),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2)))

model.add(Dropout(0.25))

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(Conv2D(filters = 64, kernel\_size = (3,3),padding = 'Same',

activation ='relu'))

model.add(MaxPool2D(pool\_size=(2,2), strides=(2,2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(256, activation = "relu"))

model.add(Dropout(0.5))

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

3.2 Set the optimizer and annealer

Once our layers are added to the model, we need to set up a score function, a loss function and an optimisation algorithm.

We define the loss function to measure how poorly our model performs on images with known labels. It is the error rate between the oberved labels and the predicted ones. We use a specific form for categorical classifications (>2 classes) called the "categorical\_crossentropy".

The most important function is the optimizer. This function will iteratively improve parameters (filters kernel values, weights and bias of neurons ...) in order to minimise the loss.

I choosed RMSprop (with default values), it is a very effective optimizer. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. We could also have used Stochastic Gradient Descent ('sgd') optimizer, but it is slower than RMSprop.

The metric function "accuracy" is used is to evaluate the performance our model. This metric function is similar to the loss function, except that the results from the metric evaluation are not used when training the model (only for evaluation).

In [13]:

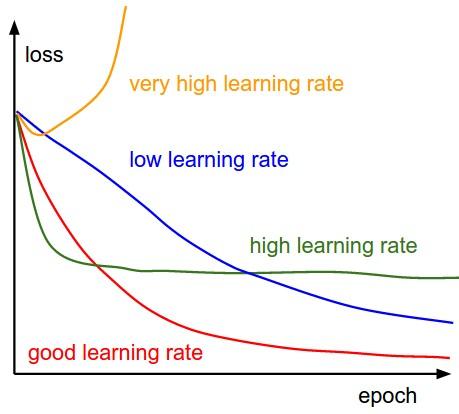
*# Define the optimizer*

optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)

In [14]:

*# Compile the model*

model.compile(optimizer = optimizer , loss = "categorical\_crossentropy", metrics=["accuracy"])



In order to make the optimizer converge faster and closest to the global minimum of the loss function, i used an annealing method of the learning rate (LR).

The LR is the step by which the optimizer walks through the 'loss landscape'. The higher LR, the bigger are the steps and the quicker is the convergence. However the sampling is very poor with an high LR and the optimizer could probably fall into a local minima.

Its better to have a decreasing learning rate during the training to reach efficiently the global minimum of the loss function.

To keep the advantage of the fast computation time with a high LR, i decreased the LR dynamically every X steps (epochs) depending if it is necessary (when accuracy is not improved).

With the ReduceLROnPlateau function from Keras.callbacks, i choose to reduce the LR by half if the accuracy is not improved after 3 epochs.

In [15]:

*# Set a learning rate annealer*

learning\_rate\_reduction = ReduceLROnPlateau(monitor='val\_acc',

patience=3,

verbose=1,

factor=0.5,

min\_lr=0.00001)

In [16]:

epochs = 1 *# Turn epochs to 30 to get 0.9967 accuracy*

batch\_size = 86

3.3 Data augmentation

In order to avoid overfitting problem, we need to expand artificially our handwritten digit dataset. We can make your existing dataset even larger. The idea is to alter the training data with small transformations to reproduce the variations occuring when someone is writing a digit.

For example, the number is not centered The scale is not the same (some who write with big/small numbers) The image is rotated...

Approaches that alter the training data in ways that change the array representation while keeping the label the same are known as data augmentation techniques. Some popular augmentations people use are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more.

By applying just a couple of these transformations to our training data, we can easily double or triple the number of training examples and create a very robust model.

The improvement is important :

* Without data augmentation i obtained an accuracy of 98.114%
* With data augmentation i achieved 99.67% of accuracy

In [17]:

*# Without data augmentation i obtained an accuracy of 0.98114*

*#history = model.fit(X\_train, Y\_train, batch\_size = batch\_size, epochs = epochs,*

*# validation\_data = (X\_val, Y\_val), verbose = 2)*

In [18]:

*# With data augmentation to prevent overfitting (accuracy 0.99286)*

datagen = ImageDataGenerator(

featurewise\_center=False, *# set input mean to 0 over the dataset*

samplewise\_center=False, *# set each sample mean to 0*

featurewise\_std\_normalization=False, *# divide inputs by std of the dataset*

samplewise\_std\_normalization=False, *# divide each input by its std*

zca\_whitening=False, *# apply ZCA whitening*

rotation\_range=10, *# randomly rotate images in the range (degrees, 0 to 180)*

zoom\_range = 0.1, *# Randomly zoom image*

width\_shift\_range=0.1, *# randomly shift images horizontally (fraction of total width)*

height\_shift\_range=0.1, *# randomly shift images vertically (fraction of total height)*

horizontal\_flip=False, *# randomly flip images*

vertical\_flip=False) *# randomly flip images*

datagen.fit(X\_train)

For the data augmentation, i choosed to :

* Randomly rotate some training images by 10 degrees
* Randomly Zoom by 10% some training images
* Randomly shift images horizontally by 10% of the width
* Randomly shift images vertically by 10% of the height

I did not apply a vertical\_flip nor horizontal\_flip since it could have lead to misclassify symetrical numbers such as 6 and 9.

Once our model is ready, we fit the training dataset .

In [19]:

*# Fit the model*

history = model.fit\_generator(datagen.flow(X\_train,Y\_train, batch\_size=batch\_size),

epochs = epochs, validation\_data = (X\_val,Y\_val),

verbose = 2, steps\_per\_epoch=X\_train.shape[0] // batch\_size

, callbacks=[learning\_rate\_reduction])

Epoch 1/1

644s - loss: 0.4219 - acc: 0.8644 - val\_loss: 0.0740 - val\_acc: 0.9781

4. Evaluate the model

4.1 Training and validation curves

In [20]:

*# Plot the loss and accuracy curves for training and validation*

fig, ax = plt.subplots(2,1)

ax[0].plot(history.history['loss'], color='b', label="Training loss")

ax[0].plot(history.history['val\_loss'], color='r', label="validation loss",axes =ax[0])

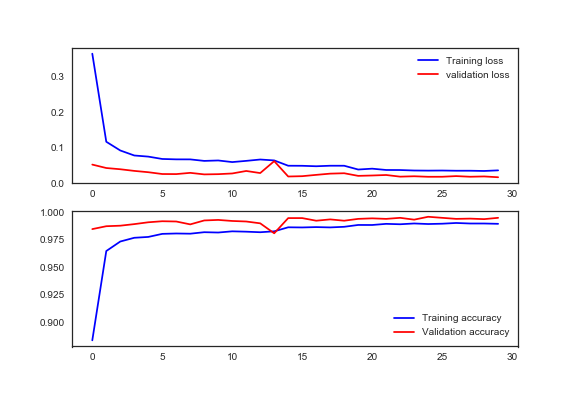
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['acc'], color='b', label="Training accuracy")

ax[1].plot(history.history['val\_acc'], color='r',label="Validation accuracy")

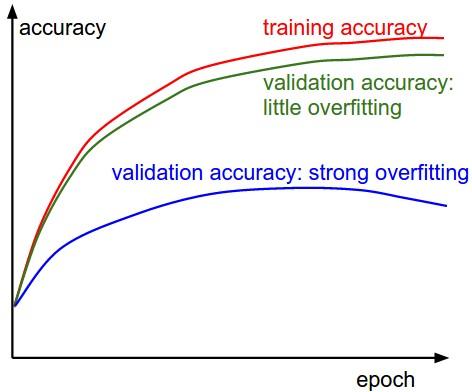
legend = ax[1].legend(loc='best', shadow=True)

The code below is for plotting loss and accuracy curves for training and validation. Since, i set epochs = 2 on this notebook . I'll show you the training and validation curves i obtained from the model i build with 30 epochs (2h30)



The model reaches almost 99% (98.7+%) accuracy on the validation dataset after 2 epochs. The validation accuracy is greater than the training accuracy almost evry time during the training. That means that our model dosen't not overfit the training set.

Our model is very well trained !!!



4.2 Confusion matrix

Confusion matrix can be very helpfull to see your model drawbacks.

I plot the confusion matrix of the validation results.

In [21]:

*# Look at confusion matrix*

def plot\_confusion\_matrix(cm, classes,

normalize=False,

title='Confusion matrix',

cmap=plt.cm.Blues):

*"""*

*This function prints and plots the confusion matrix.*

*Normalization can be applied by setting `normalize=True`.*

*"""*

plt.imshow(cm, interpolation='nearest', cmap=cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(classes))

plt.xticks(tick\_marks, classes, rotation=45)

plt.yticks(tick\_marks, classes)

if normalize:

cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

thresh = cm.max() / 2.

for i, j **in** itertools.product(range(cm.shape[0]), range(cm.shape[1])):

plt.text(j, i, cm[i, j],

horizontalalignment="center",

color="white" if cm[i, j] > thresh else "black")

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

*# Predict the values from the validation dataset*

Y\_pred = model.predict(X\_val)

*# Convert predictions classes to one hot vectors*

Y\_pred\_classes = np.argmax(Y\_pred,axis = 1)

*# Convert validation observations to one hot vectors*

Y\_true = np.argmax(Y\_val,axis = 1)

*# compute the confusion matrix*

confusion\_mtx = confusion\_matrix(Y\_true, Y\_pred\_classes)

*# plot the confusion matrix*

plot\_confusion\_matrix(confusion\_mtx, classes = range(10))

Here we can see that our CNN performs very well on all digits with few errors considering the size of the validation set (4 200 images).

However, it seems that our CNN has some little troubles with the 4 digits, hey are misclassified as 9. Sometime it is very difficult to catch the difference between 4 and 9 when curves are smooth.

Let's investigate for errors.

I want to see the most important errors . For that purpose i need to get the difference between the probabilities of real value and the predicted ones in the results.

In [22]:

*# Display some error results*

*# Errors are difference between predicted labels and true labels*

errors = (Y\_pred\_classes - Y\_true != 0)

Y\_pred\_classes\_errors = Y\_pred\_classes[errors]

Y\_pred\_errors = Y\_pred[errors]

Y\_true\_errors = Y\_true[errors]

X\_val\_errors = X\_val[errors]

def display\_errors(errors\_index,img\_errors,pred\_errors, obs\_errors):

*""" This function shows 6 images with their predicted and real labels"""*

n = 0

nrows = 2

ncols = 3

fig, ax = plt.subplots(nrows,ncols,sharex=True,sharey=True)

for row **in** range(nrows):

for col **in** range(ncols):

error = errors\_index[n]

ax[row,col].imshow((img\_errors[error]).reshape((28,28)))

ax[row,col].set\_title("Predicted label :**{}\n**True label :**{}**".format(pred\_errors[error],obs\_errors[error]))

n += 1

*# Probabilities of the wrong predicted numbers*

Y\_pred\_errors\_prob = np.max(Y\_pred\_errors,axis = 1)

*# Predicted probabilities of the true values in the error set*

true\_prob\_errors = np.diagonal(np.take(Y\_pred\_errors, Y\_true\_errors, axis=1))

*# Difference between the probability of the predicted label and the true label*

delta\_pred\_true\_errors = Y\_pred\_errors\_prob - true\_prob\_errors

*# Sorted list of the delta prob errors*

sorted\_dela\_errors = np.argsort(delta\_pred\_true\_errors)

*# Top 6 errors*

most\_important\_errors = sorted\_dela\_errors[-6:]

*# Show the top 6 errors*

display\_errors(most\_important\_errors, X\_val\_errors, Y\_pred\_classes\_errors, Y\_true\_errors)

The most important errors are also the most intrigous.

For those six case, the model is not ridiculous. Some of these errors can also be made by humans, especially for one the 9 that is very close to a 4. The last 9 is also very misleading, it seems for me that is a 0.

In [23]:

*# predict results*

results = model.predict(test)

*# select the indix with the maximum probability*

results = np.argmax(results,axis = 1)

results = pd.Series(results,name="Label")

In [24]:

submission = pd.concat([pd.Series(range(1,28001),name = "ImageId"),results],axis = 1)

submission.to\_csv("cnn\_mnist\_datagen.csv",index=False)

**you found this notebook helpful or you just liked it , some upvotes would be very much appreciated - That will keep me motivated :)**