script

May 17, 2018

1 MNIST using CNN Keras - Acc 0.997

1.0.1 Sudheer Achary

2 1. Introduction

This is a 9 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 99.70% of accuracy with this CNN trained on GTX 1080Ti for 50 epochs with batch size of 64 using tensorflow-gpu with keras.

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, BatchNormalizatio
from keras.optimizers import Adagrad
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau


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

```
/home/sudheer.achary/.local/lib/python2.7/site-packages/h5py/__init__.py:36: FutureWarning: Co
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

3 2. Data preparation

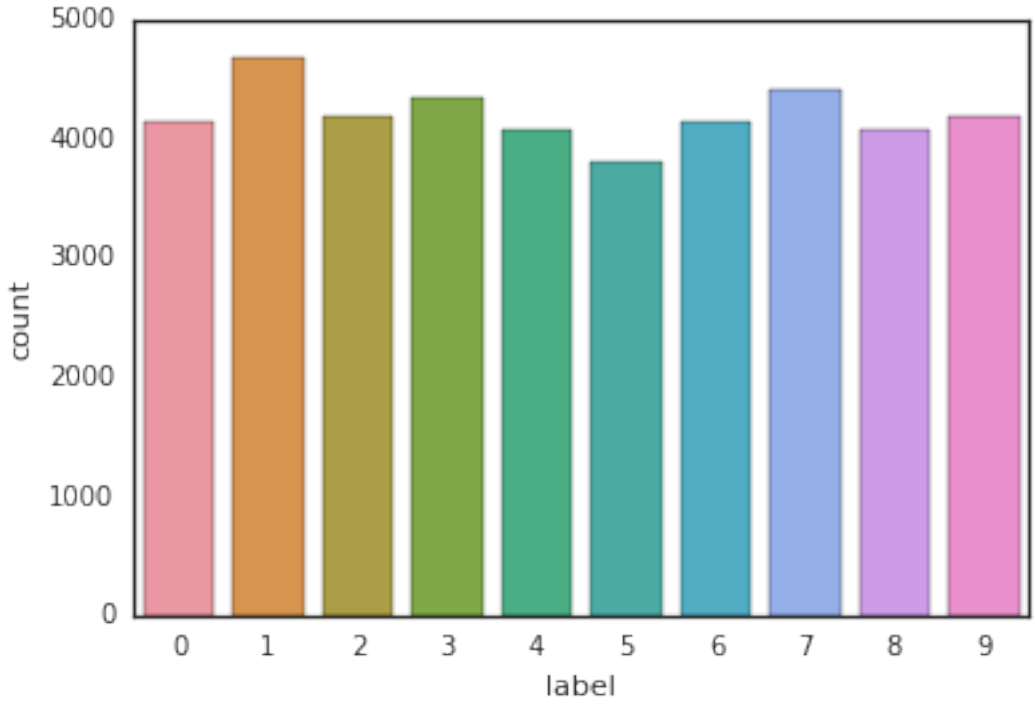
3.1 2.1 Load data

In [2]:
```
# Load the data
train = pd.read_csv("train.csv")
test = pd.read_csv("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.

3.2 2.2 Check for null and missing values

```
In [4]: # Check the data
        X_train.isnull().any().describe()
```

```
Out[4]: count      784
        mean         0
        std          0
```

```
min      False
25%          0
50%          0
75%          0
max      False
dtype: object
```

```
In [5]: test.isnull().any().describe()
```

```
Out[5]: count      784
        mean         0
        std          0
        min      False
        25%          0
        50%          0
        75%          0
        max      False
        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.

3.3 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
```

3.4 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.

3.5 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]).

3.6 2.6 Split training and valdiation set

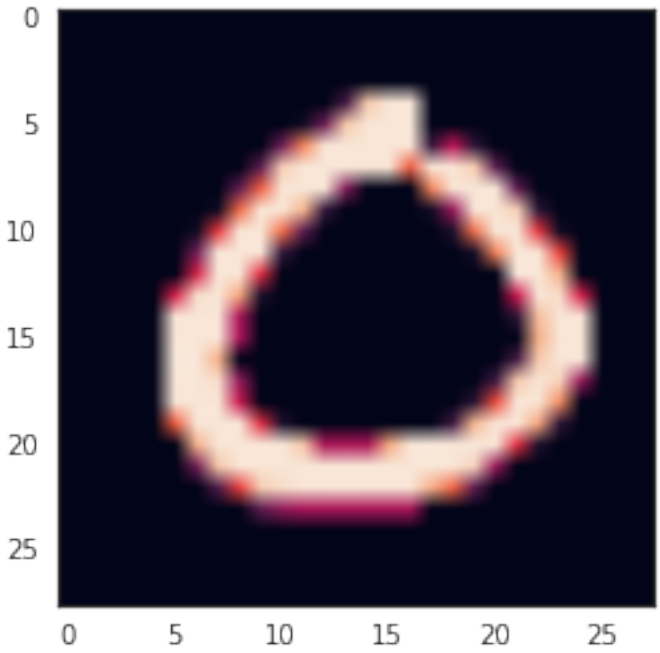
```
In [9]: # Set the random seed
        random_seed = 894

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 = 0.001
```

I choosed to split the train set in two parts : a small fraction (0.01%) became the validation set which the model is evaluated and the rest (99.99%) 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.

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])
```



4 3. CNN

4.1 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 64 filters for the first conv2D layer with kernel of size 7x7 and 128 filters for the next two conv2D

layers with 5x5, 33x3 kernels respectively. 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.

'elu' is the exponential linear unit (activation function. The **elu** 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 Dense 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
         model = Sequential()

         model.add(Conv2D(filters = 64, kernel_size = (7,7),padding = 'Same', activation ='elu
         model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
         model.add(BatchNormalization())

         model.add(Conv2D(filters = 128, kernel_size = (5,5),padding = 'Same', activation ='elu
         model.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same', activation ='elu
         model.add(MaxPool2D(pool_size=(2,2)))
         model.add(BatchNormalization())

         model.add(Conv2D(filters = 256, kernel_size = (3,3),padding = 'Same', activation ='elu
         model.add(MaxPool2D(pool_size=(2,2)))
         model.add(BatchNormalization())

         model.add(Flatten())
         model.add(Dense(256, activation = "elu"))
         model.add(Dense(10, activation = "softmax"))
         model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 64)        3200
_________________________________________________________________
```

```
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 128)       204928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 128)       147584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
batch_normalization_2 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 7, 7, 256)         295168
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 3, 3, 256)         0
_________________________________________________________________
batch_normalization_3 (Batch (None, 3, 3, 256)         1024
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               590080
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 1,245,322
Trainable params: 1,244,426
Non-trainable params: 896
_________________________________________________________________
```

4.2 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 Adam (with default values), it is a very effective optimizer. The Adam 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 Adam.

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 = Adagrad(lr=0.01, epsilon=1e-4, decay=0.0)
```

```
In [14]: # Compile the model
         model.compile(optimizer = 'Adam' , loss = "categorical_crossentropy", metrics=["accura
```

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 = 30
         batch_size = 64
```

4.3 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 99.7% of accuracy

```
In [17]: # 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 tot
        height_shift_range=0.1,  # randomly shift images vertically (fraction of tota
        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 [18]: # 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 = 1, steps_per_epoch=X_train.shape[0] // batch_s
                                       , callbacks=[learning_rate_reduction])

Epoch 1/30
655/655 [==============================] - 11s 17ms/step - loss: 0.2009 - acc: 0.9419 - val_los
Epoch 2/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0785 - acc: 0.9767 - val_loss
Epoch 3/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0621 - acc: 0.9818 - val_loss
Epoch 4/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0539 - acc: 0.9843 - val_loss

Epoch 00004: ReduceLROnPlateau reducing learning rate to 0.000500000023749.
Epoch 5/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0295 - acc: 0.9907 - val_loss
Epoch 6/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0291 - acc: 0.9912 - val_loss
Epoch 7/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0249 - acc: 0.9920 - val_loss

Epoch 00007: ReduceLROnPlateau reducing learning rate to 0.000250000011874.
Epoch 8/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0180 - acc: 0.9940 - val_loss
Epoch 9/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0147 - acc: 0.9956 - val_loss
Epoch 10/30
```

```
655/655 [==============================] - 10s 15ms/step - loss: 0.0157 - acc: 0.9949 - val_los

Epoch 00010: ReduceLROnPlateau reducing learning rate to 0.000125000005937.
Epoch 11/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0110 - acc: 0.9966 - val_loss
Epoch 12/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0100 - acc: 0.9966 - val_loss
Epoch 13/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0089 - acc: 0.9971 - val_loss

Epoch 00013: ReduceLROnPlateau reducing learning rate to 6.25000029686e-05.
Epoch 14/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0069 - acc: 0.9977 - val_loss
Epoch 15/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0063 - acc: 0.9981 - val_loss
Epoch 16/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0068 - acc: 0.9981 - val_loss

Epoch 00016: ReduceLROnPlateau reducing learning rate to 3.12500014843e-05.
Epoch 17/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0056 - acc: 0.9983 - val_loss
Epoch 18/30
655/655 [==============================] - 10s 15ms/step - loss: 0.0048 - acc: 0.9985 - val_los
Epoch 19/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0046 - acc: 0.9986 - val_loss

Epoch 00019: ReduceLROnPlateau reducing learning rate to 1.56250007421e-05.
Epoch 20/30
655/655 [==============================] - 10s 15ms/step - loss: 0.0041 - acc: 0.9988 - val_los
Epoch 21/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0039 - acc: 0.9987 - val_loss
Epoch 22/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0040 - acc: 0.9989 - val_loss

Epoch 00022: ReduceLROnPlateau reducing learning rate to 1e-05.
Epoch 23/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0043 - acc: 0.9988 - val_loss
Epoch 24/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0040 - acc: 0.9987 - val_loss
Epoch 25/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0039 - acc: 0.9989 - val_loss
Epoch 26/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0039 - acc: 0.9990 - val_loss
Epoch 27/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0043 - acc: 0.9990 - val_loss
Epoch 28/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0033 - acc: 0.9990 - val_loss
Epoch 29/30
```

```
655/655 [==============================] - 9s 14ms/step - loss: 0.0033 - acc: 0.9991 - val_loss
Epoch 30/30
655/655 [==============================] - 9s 14ms/step - loss: 0.0037 - acc: 0.9990 - val_loss
```

5 4. Evaluate the model

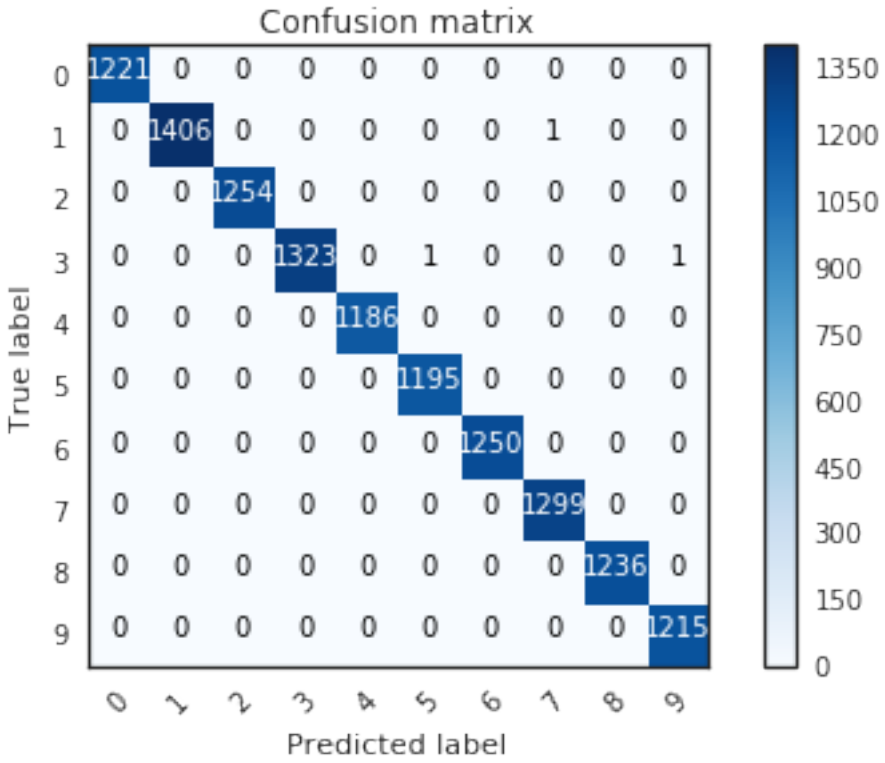
5.1 4.1 Confusion matrix

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
         X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.3, r

         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 images).

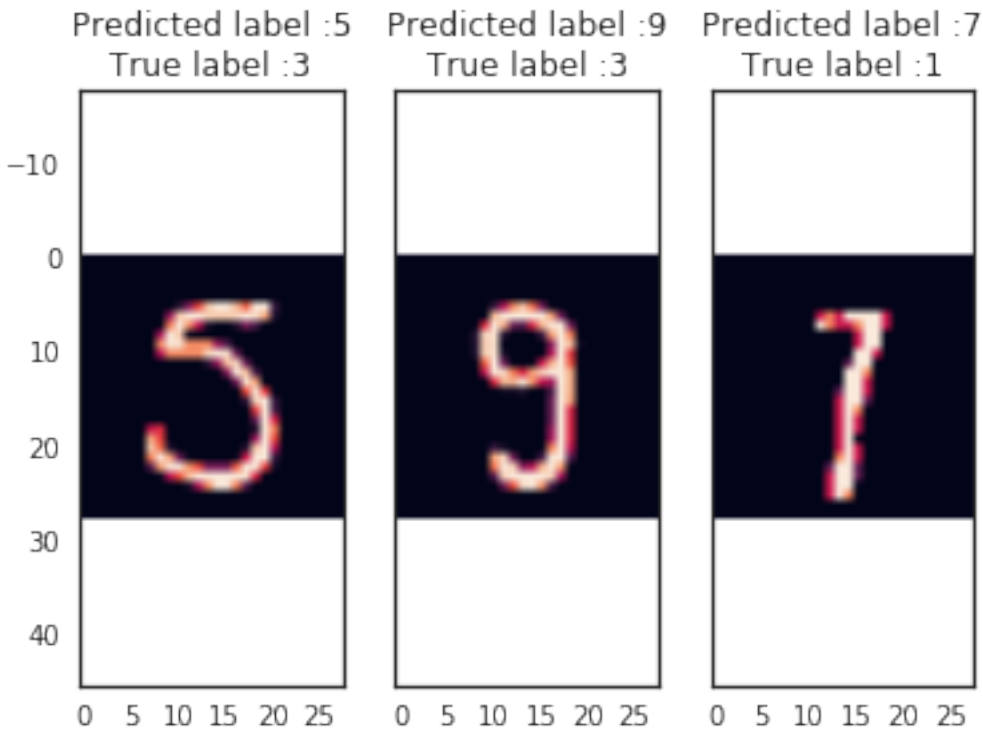
However, it seems that our CNN has some little troubles with the 4 digit, they are misclassified as 9. Sometime it is very difficult to catch the difference between 4 and 9 when curves are smooth.

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 [23]:
```
# 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 = 1
    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[col].imshow((img_errors[error]).reshape((28,28)))
            ax[col].set_title("Predicted label :{}\nTrue label :{}".format(pred_errors
            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_erro
```



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 [24]: # 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 [25]: submission = pd.concat([pd.Series(range(1,28001),name = "ImageId"),results],axis = 1)

         submission.to_csv("cnn_mnist_datagen.csv",index=False)
```