Import Libraries

In [1]:

```
import warnings
warnings.filterwarnings("ignore")

# Import the necessary libs
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, BatchNormalization
from tensorflow.keras.optimizers import RMSprop
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
from keras.datasets import mnist
import tensorflow as tf

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

Load Data

In [2]:

```
train = pd.read_csv('data/train.csv')
test = pd.read_csv('data/test.csv')
sub = pd.read_csv('data/sample_submission.csv')

print("Data are Ready!!")
```

```
Data are Ready!!
```

In [3]:

```
print(f"Training data size is {train.shape}\nTesting data size is {test.shape}")
```

```
Training data size is (42000, 785)
Testing data size is (28000, 784)
```

Set data features and Target labels

In [4]:

```
Y_train = train["label"]
X_train = train.drop(labels = ["label"], axis = 1)
```

Load more data sets

In [5]:

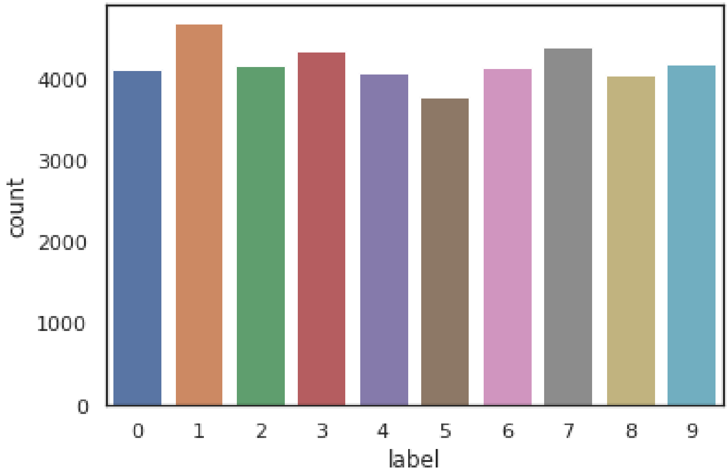
```
(x_train1, y_train1), (x_test1, y_test1) = mnist.load_data()

train1 = np.concatenate([x_train1, x_test1], axis=0)
```

```
y_train1 = np.concatenate([y_train1, y_test1], axis=0)

Y_train1 = y_train1
X_train1 = train1.reshape(-1, 28*28)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.n
pz
11493376/11490434 [==============================] - 0s 0us/step
```

In [6]:

```
# Print data histogram
sns.countplot(Y_train);
```



Normalization

Grayscale normalization to reduce the effect of illumination's differences.

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

In [7]:

```
# Normalize data to make CNN faster
X_train = X_train / 255.0
test = test / 255.0

X_train1 = X_train1 / 255.0
```

Merging all the data we got

In [8]:

```
# Reshape Picture is 3D array (height = 28px, width = 28px , canal = 1)
X_train = np.concatenate((X_train.values, X_train1))
Y_train = np.concatenate((Y_train, Y_train1))
```

Reshape

In [9]:

```
# Reshape image in 3 dimensions (height = 28px, width = 28px , canal = 1)
# canal = 1 => For gray scale
X_train = X_train.reshape(-1,28,28,1)
test = test.values.reshape(-1,28,28,1)
```

One-Hot Encoding

In [10]:

```
# Convert label 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)
```

Split training and valdiation set

In [11]:

```
# 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, ran
dom_state=2)
```

In [12]:

```
X_train.shape, X_val.shape, Y_train.shape, Y_val.shape
```

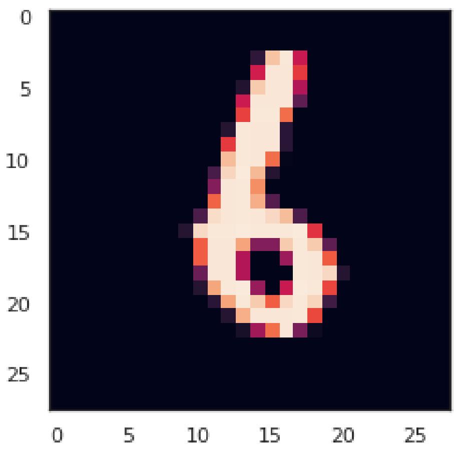
Out[12]:

```
((100800, 28, 28, 1), (11200, 28, 28, 1), (100800, 10), (11200, 10))
```

Data Visualization

In [13]:

```
# Draw an example of a data set to see
g = plt.imshow(X_train[189][:,:,0])
```



Model Definition

In [14]:

```
model = Sequential()

model.add(Conv2D(filters = 64, kernel_size = (5,5),padding = 'Same', activation ='relu',
input_shape = (28,28,1)))
model.add(BatchNormalization())

model.add(Conv2D(filters = 64, kernel_size = (5,5),padding = 'Same', activation ='relu')
)
model.add(BatchNormalization())

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(BatchNormalization())

model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same', activation ='relu')
)
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 64, kernel_size = (3,3), padding = 'Same',  activation ='relu
'))
model.add(BatchNormalization())
```

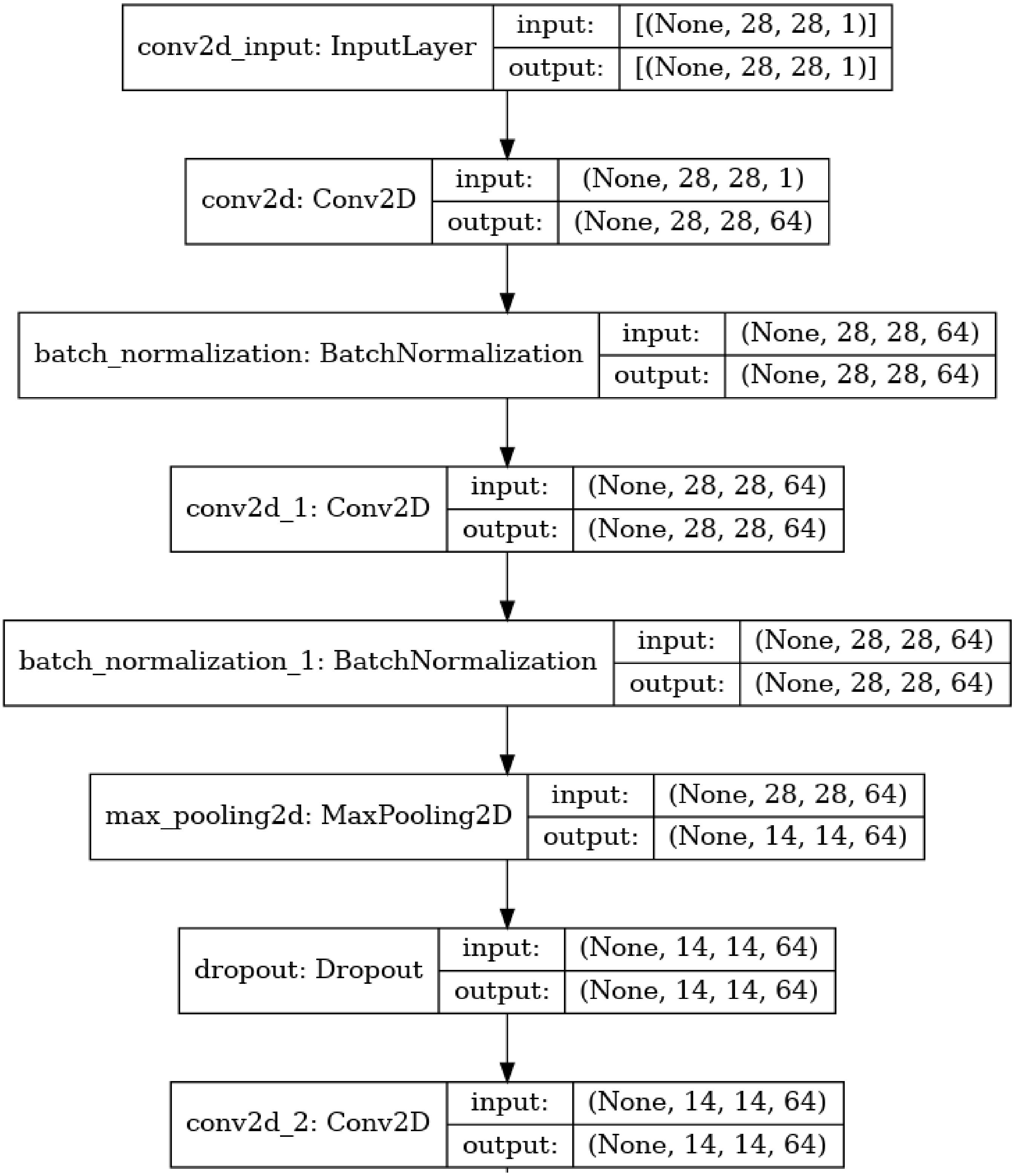
```
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation = "relu"))
model.add(BatchNormalization())
model.add(Dropout(0.25))

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

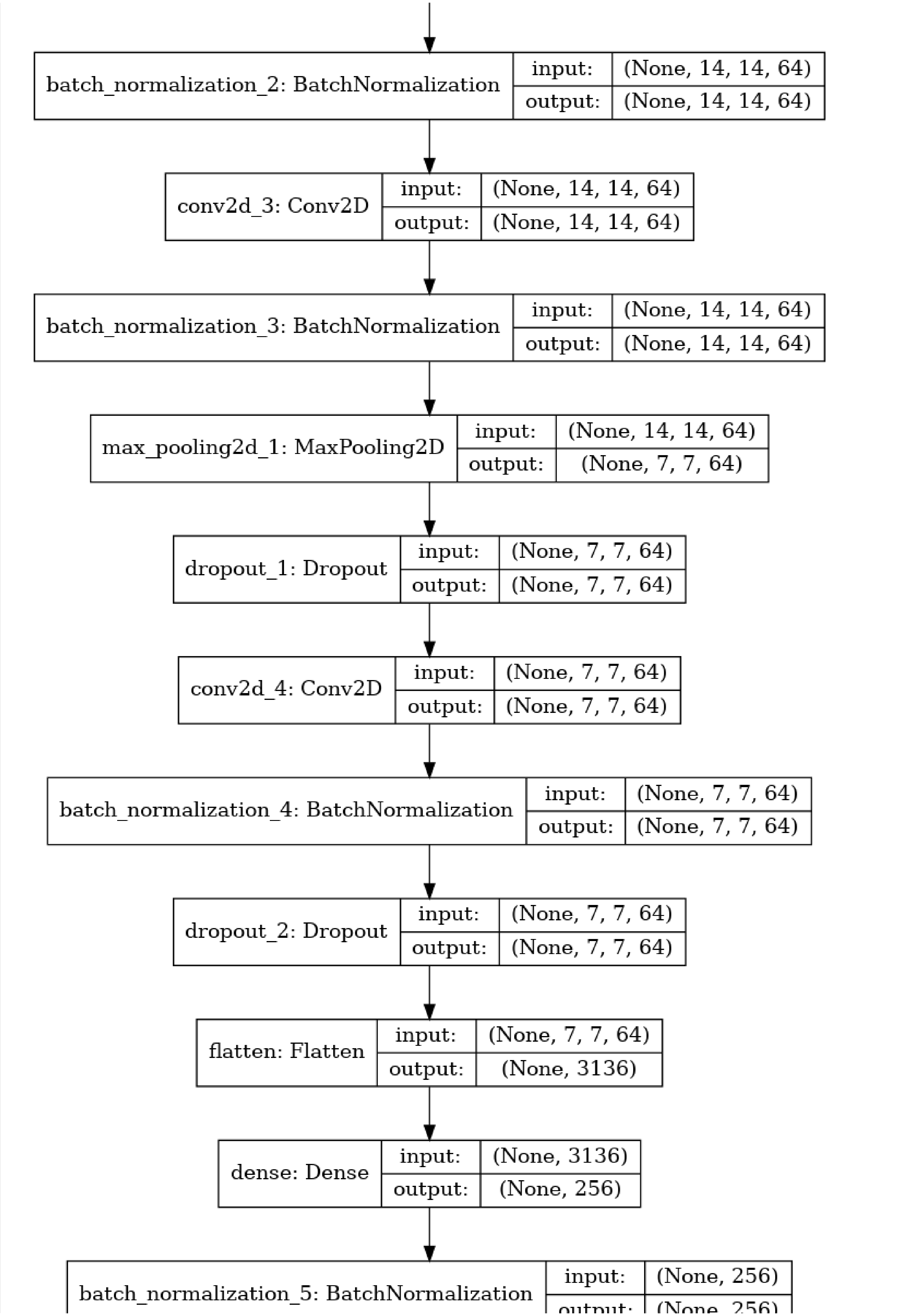
Plot CNN model

In [15]:

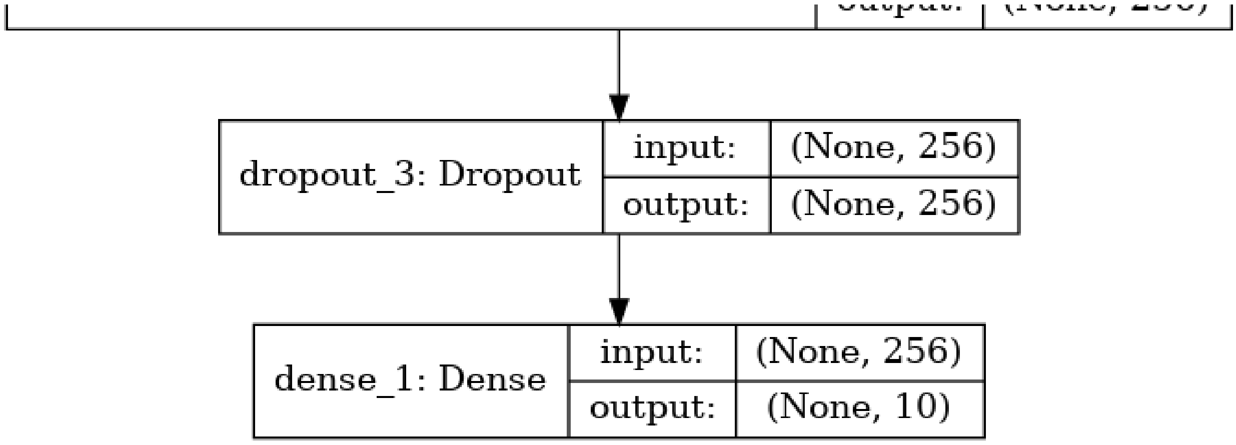
```
# print out model look
from keras.utils import plot_model
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
from IPython.display import Image
Image("model.png")
```

Out[15]:





batch_normalization_2: BatchNormalization
input:
(None, 14, 14, 64)
output:
(None, 14, 14, 64)
conv2d_3: Conv2D
input:
(None, 14, 14, 64)
output:
(None, 14, 14, 64)
batch_normalization_3: BatchNormalization
input:
(None, 14, 14, 64)
output:
(None, 14, 14, 64)
max_pooling2d_1: MaxPooling2D
input:
(None, 14, 14, 64)
output:
(None, 7, 7, 64)
dropout_1: Dropout
input:
(None, 7, 7, 64)
output:
(None, 7, 7, 64)
conv2d_4: Conv2D
input:
(None, 7, 7, 64)
output:
(None, 7, 7, 64)
batch_normalization_4: BatchNormalization
input:
(None, 7, 7, 64)
output:
(None, 7, 7, 64)
dropout_2: Dropout
input:
(None, 7, 7, 64)
output:
(None, 7, 7, 64)
flatten: Flatten
input:
(None, 7, 7, 64)
output:
(None, 3136)
dense: Dense
input:
(None, 3136)
output:
(None, 256)
batch_normalization_5: BatchNormalization
input:
(None, 256)



In [16]:

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

In [17]:

```
# Compile model
model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accur
acy"])
```

In [18]:

```
# Audjusting learning rate
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
```

In [19]:

```
#Adjusting epochs and batch_size
epochs = 50
batch_size = 128
```

Data augmentation

In [20]:

```
#Data Augmentation
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 he
ight)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images

#datagen.fit(X_train)
train_gen = datagen.flow(X_train,Y_train, batch_size=batch_size)
```

Model training

In [21]:

```
#Prediction model
```

```
history = model.fit(train_gen,
                              epochs = epochs,validation_data = (X_val,Y_val),
                              verbose = 2, steps_per_epoch=X_train.shape[0] // batch_siz
e
                              , callbacks=[learning_rate_reduction],
                             validation_steps = X_val.shape[0] // batch_size)
```

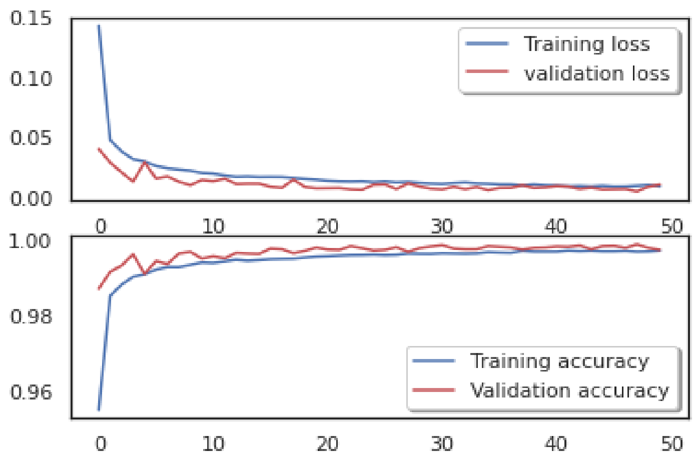
```
Epoch 1/50
787/787 - 36s - loss: 0.1432 - accuracy: 0.9552 - val_loss: 0.0407 - val_accuracy: 0.9871
Epoch 2/50
787/787 - 28s - loss: 0.0482 - accuracy: 0.9852 - val_loss: 0.0293 - val_accuracy: 0.9916
Epoch 3/50
787/787 - 28s - loss: 0.0385 - accuracy: 0.9883 - val_loss: 0.0211 - val_accuracy: 0.9933
Epoch 4/50
787/787 - 28s - loss: 0.0321 - accuracy: 0.9903 - val_loss: 0.0135 - val_accuracy: 0.9962
Epoch 5/50
787/787 - 28s - loss: 0.0301 - accuracy: 0.9910 - val_loss: 0.0296 - val_accuracy: 0.9911
Epoch 6/50
787/787 - 28s - loss: 0.0265 - accuracy: 0.9922 - val_loss: 0.0161 - val_accuracy: 0.9945
Epoch 7/50
787/787 - 28s - loss: 0.0246 - accuracy: 0.9929 - val_loss: 0.0179 - val_accuracy: 0.9937
Epoch 8/50
787/787 - 28s - loss: 0.0235 - accuracy: 0.9929 - val_loss: 0.0134 - val_accuracy: 0.9964
Epoch 9/50
787/787 - 29s - loss: 0.0225 - accuracy: 0.9935 - val_loss: 0.0106 - val_accuracy: 0.9969
Epoch 10/50
787/787 - 28s - loss: 0.0207 - accuracy: 0.9941 - val_loss: 0.0148 - val_accuracy: 0.9951
Epoch 11/50
787/787 - 28s - loss: 0.0202 - accuracy: 0.9940 - val_loss: 0.0138 - val_accuracy: 0.9957
Epoch 12/50
787/787 - 28s - loss: 0.0186 - accuracy: 0.9944 - val_loss: 0.0159 - val_accuracy: 0.9952
Epoch 13/50
787/787 - 28s - loss: 0.0175 - accuracy: 0.9948 - val_loss: 0.0115 - val_accuracy: 0.9966
Epoch 14/50
787/787 - 29s - loss: 0.0178 - accuracy: 0.9945 - val_loss: 0.0118 - val_accuracy: 0.9964
Epoch 15/50
787/787 - 28s - loss: 0.0173 - accuracy: 0.9947 - val_loss: 0.0118 - val_accuracy: 0.9963
Epoch 16/50
787/787 - 27s - loss: 0.0174 - accuracy: 0.9950 - val_loss: 0.0092 - val_accuracy: 0.9978
Epoch 17/50
787/787 - 28s - loss: 0.0174 - accuracy: 0.9950 - val_loss: 0.0084 - val_accuracy: 0.9976
Epoch 18/50
787/787 - 28s - loss: 0.0166 - accuracy: 0.9951 - val_loss: 0.0151 - val_accuracy: 0.9965
Epoch 19/50
787/787 - 28s - loss: 0.0158 - accuracy: 0.9954 - val_loss: 0.0091 - val_accuracy: 0.9971
Epoch 20/50
787/787 - 28s - loss: 0.0151 - accuracy: 0.9956 - val_loss: 0.0079 - val_accuracy: 0.9980
Epoch 21/50
787/787 - 28s - loss: 0.0142 - accuracy: 0.9957 - val_loss: 0.0081 - val_accuracy: 0.9975
Epoch 22/50
787/787 - 28s - loss: 0.0137 - accuracy: 0.9959 - val_loss: 0.0082 - val_accuracy: 0.9974
Epoch 23/50
787/787 - 28s - loss: 0.0136 - accuracy: 0.9960 - val_loss: 0.0071 - val_accuracy: 0.9984
Epoch 24/50
787/787 - 28s - loss: 0.0138 - accuracy: 0.9960 - val_loss: 0.0067 - val_accuracy: 0.9979
Epoch 25/50
787/787 - 28s - loss: 0.0131 - accuracy: 0.9961 - val_loss: 0.0109 - val_accuracy: 0.9972
Epoch 26/50
787/787 - 28s - loss: 0.0137 - accuracy: 0.9960 - val_loss: 0.0112 - val_accuracy: 0.9974
Epoch 27/50
787/787 - 28s - loss: 0.0128 - accuracy: 0.9961 - val_loss: 0.0073 - val_accuracy: 0.9981
Epoch 28/50
787/787 - 28s - loss: 0.0133 - accuracy: 0.9964 - val_loss: 0.0120 - val_accuracy: 0.9969
Epoch 29/50
787/787 - 27s - loss: 0.0124 - accuracy: 0.9964 - val_loss: 0.0095 - val_accuracy: 0.9979
Epoch 30/50
787/787 - 28s - loss: 0.0119 - accuracy: 0.9963 - val_loss: 0.0077 - val_accuracy: 0.9983
Epoch 31/50
787/787 - 28s - loss: 0.0116 - accuracy: 0.9965 - val_loss: 0.0070 - val_accuracy: 0.9987
Epoch 32/50
787/787 - 28s - loss: 0.0123 - accuracy: 0.9964 - val_loss: 0.0092 - val_accuracy: 0.9978
Epoch 33/50
```

```
787/787 - 28s - loss: 0.0129 - accuracy: 0.9964 - val_loss: 0.0072 - val_accuracy: 0.9976
Epoch 34/50
787/787 - 29s - loss: 0.0120 - accuracy: 0.9965 - val_loss: 0.0090 - val_accuracy: 0.9976
Epoch 35/50
787/787 - 28s - loss: 0.0117 - accuracy: 0.9968 - val_loss: 0.0064 - val_accuracy: 0.9984
Epoch 36/50
787/787 - 28s - loss: 0.0112 - accuracy: 0.9967 - val_loss: 0.0083 - val_accuracy: 0.9982
Epoch 37/50
787/787 - 27s - loss: 0.0111 - accuracy: 0.9966 - val_loss: 0.0084 - val_accuracy: 0.9980
Epoch 38/50
787/787 - 29s - loss: 0.0104 - accuracy: 0.9971 - val_loss: 0.0105 - val_accuracy: 0.9975
Epoch 39/50
787/787 - 28s - loss: 0.0111 - accuracy: 0.9969 - val_loss: 0.0082 - val_accuracy: 0.9979
Epoch 40/50
787/787 - 28s - loss: 0.0104 - accuracy: 0.9969 - val_loss: 0.0087 - val_accuracy: 0.9980
Epoch 41/50
787/787 - 28s - loss: 0.0101 - accuracy: 0.9969 - val_loss: 0.0098 - val_accuracy: 0.9983
Epoch 42/50
787/787 - 28s - loss: 0.0095 - accuracy: 0.9972 - val_loss: 0.0094 - val_accuracy: 0.9982
Epoch 43/50
787/787 - 28s - loss: 0.0098 - accuracy: 0.9971 - val_loss: 0.0073 - val_accuracy: 0.9986
Epoch 44/50
787/787 - 27s - loss: 0.0094 - accuracy: 0.9972 - val_loss: 0.0085 - val_accuracy: 0.9976
Epoch 45/50
787/787 - 28s - loss: 0.0100 - accuracy: 0.9970 - val_loss: 0.0070 - val_accuracy: 0.9984
Epoch 46/50
787/787 - 28s - loss: 0.0093 - accuracy: 0.9970 - val_loss: 0.0071 - val_accuracy: 0.9985
Epoch 47/50
787/787 - 29s - loss: 0.0094 - accuracy: 0.9971 - val_loss: 0.0073 - val_accuracy: 0.9979
Epoch 48/50
787/787 - 28s - loss: 0.0100 - accuracy: 0.9969 - val_loss: 0.0053 - val_accuracy: 0.9988
Epoch 49/50
787/787 - 28s - loss: 0.0106 - accuracy: 0.9970 - val_loss: 0.0092 - val_accuracy: 0.9979
Epoch 50/50
787/787 - 28s - loss: 0.0096 - accuracy: 0.9972 - val_loss: 0.0111 - val_accuracy: 0.9974
```

Training and validation curves

In [22]:

```python
# Draw the loss and accuracy curves of the training set and the validation set.
# Can judge whether it is under-fitting or over-fitting
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['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```



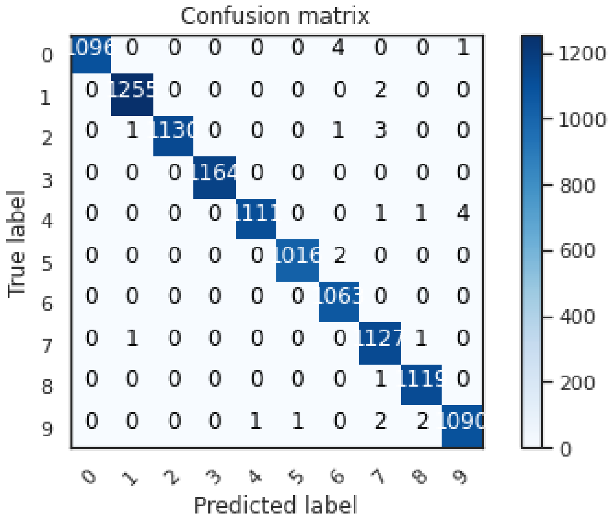
In [23]:

```python
# Draw a confusion matrix that can be used to observe high false positives
```

```python
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');
```

In [24]:

```python
# 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))
```



In [25]:

```python
# Show some wrong results, and the difference between the predicted label and the real la
be
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]
```

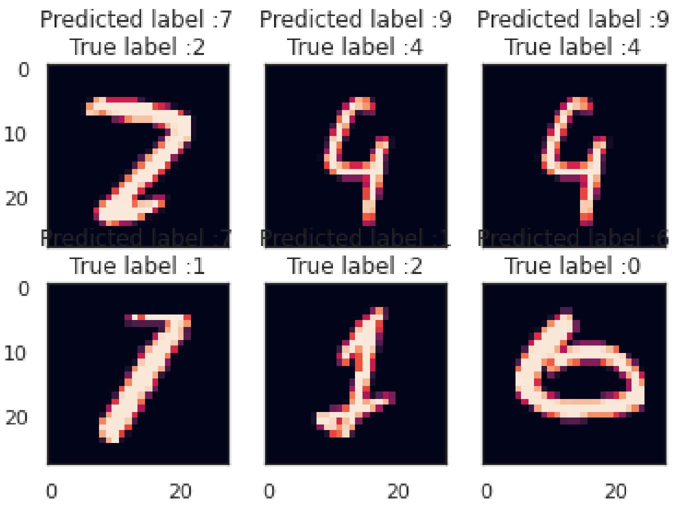
Displaying The Errors And Showing The Top 6 Errors And It's True Label

In [26]:

```python
def display_errors(errors_index,img_errors,pred_errors, obs_errors):
    """ This function shows 6 images with their predicted 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 :{}\nTrue label :{}".format(pred_erro
rs[error],obs_errors[error]))
            n += 1
```

In [27]:

```python
# 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)
```



Prediction and submition

In [28]:

```python
# Make predictions about test sets
results = model.predict(test)

# Convert one-hot vector to number
results = np.argmax(results,axis = 1)

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

In [29]:

```python
# Save the final result in submission.csv
```

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

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