CA2 Assignment Part II - Applying Convolutional Neural Network to MNIST Dataset ¶

Contents

Dataset and problem definition

The MNIST problem is the classic performance test of particular Convolutional Neural Networks (CNNs).The dataset contains 70,000 images of handwritten single digits from 0 to 9. All images are labelled. 60,000 of these images form the train set and 10,000 images form the test set. For this dataset, our task is a classification problem: we want to classify handwritten digits against their true labels.

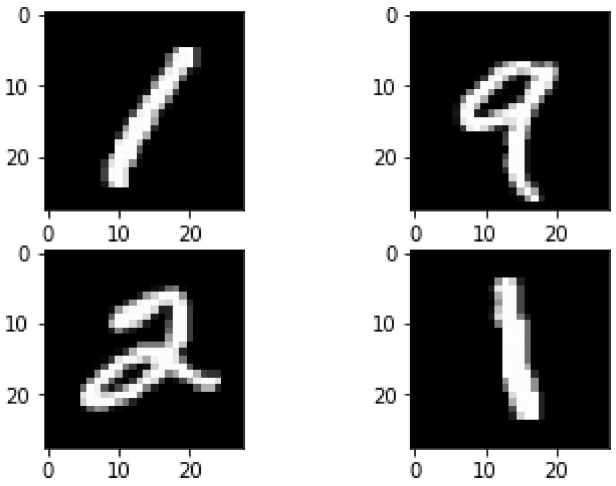
Our classification task is as follows:
a. Apply CNNs of different architectures to predict labels of different handwritten digits
c. [Personal goal] Try to achieve an excellent result (defined as less 1% error on Machine Learning Mastery)

A quick sense of the data and the labels can be obtained by loading some sample images below.

In [1]:

```
from keras.datasets import mnist
import matplotlib.pyplot as plt
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
for i in range (3,7):
    plt.subplot(218 + i)
    plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
plt.show()
print('label: %s' % (y_train[3:7]))
```

```
C:\Users\tanw\Anaconda2\envs\py35\lib\site-packages\h5py\__init__.py:36: F
utureWarning: Conversion of the second argument of issubdtype from `float`
to `np.floating` is deprecated. In future, it will be treated as `np.float
64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```




```
label: [1 9 2 1]
```

Data Preparation

This segment is reproduced from Practical 8, as I will be using Keras and there is not much variation in the method of preparation.

The first task in data preparation is to reshape the images so that it is suitable for input to a CNN. The Conv2D layer, which we will use quite a lot, expects pixel values in the following format: [pixels][width][height]. I will thus be converting all images into numpy arrays, and X_train will be a master array of all 60,000 training images.

For MNIST, the images are grayscale, hence the pixel channel is 1. This causes some issues when we use more advanced network architectures (as we will see later), since the later architectures are trained on ImageNet, which is a bank of millions of colour images.

In [2]:

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras.utils import plot_model
from keras import backend as K

K.set_image_dim_ordering('th')
seed = 1 #setting seed for reproducibility
numpy.random.seed(seed)

# reshape to format required by Keras -- [samples][pixels][width][height]
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')

# normalize inputs to a scale of 1 and one-hot encoding
X_train = X_train / 255
X_test = X_test / 255
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Considerations for Model Selection: Picking up from where we left off in Practical 8

I pick up from where we left off in Practical 8 -- the architecture named "A More Complex CNN Model" in Page 10 to 12. In reproducing this model, I will not run it (as it is already part of the Practical), but I want to examine its architecture

In [3]:

```
#Model 1 - "A More Complex CNN ModeL"
model1 = Sequential()
model1.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), activation='relu'))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Conv2D(15, (3, 3), activation='relu'))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Dropout(0.2))
model1.add(Flatten())
model1.add(Dense(128, activation='relu'))
model1.add(Dense(50, activation='relu'))
model1.add(Dense(num_classes, activation='softmax'))
# Compile model
model1.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
model1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 30, 24, 24)        780
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 12, 12)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 15, 10, 10)        4065
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 5, 5)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 5, 5)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 375)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               48128
_________________________________________________________________
dense_2 (Dense)              (None, 50)                6450
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
=================================================================
Total params: 59,933
Trainable params: 59,933
Non-trainable params: 0
_________________________________________________________________
```

While scouring the net for literature on the network selection segment, I realised that this looks like a modified version of Yann Lecun's 1988 LeNet-5.

Tracking the history of LeNet-5 up to recent models such as ResNet, and how they performed at competitions such as the ImageNet Large Scale Visual Recognition (ILSVRC) competition, I was able to know what were the available models out there, their relative performance, as well as pros and cons (e.g. accuracy, compute power required, etc.) I decided to investigate the following models:

1. **LeNet 5**: Since Practical 8's model was based off this, I want to see how the original architecture drawn up by Yann Lecun in 1998 holds up against all the other models.
2. **ZFNet**: I felt ZFNet was worth exploring as it won the ILSVRC 2013 with a fairly simple stacked layer architecture, modified from AlexNet. It achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate.
3. **VGG-16**: Finally, I decided to explore VGGNet, especially the VGG-16 variant. Although GoogLeNet won ILSVRC 2014, VGGNet came in as a close runner up. VGGNet is also featured extensively in Jeremy Howard's FastAI course which I have tried my hand at learning. Running various models (eg. up to VGG-19), I also surmised that the maximum depth which my compute power can take for a reasonable training time (about 45 minutes) is the VGG-16 variant.

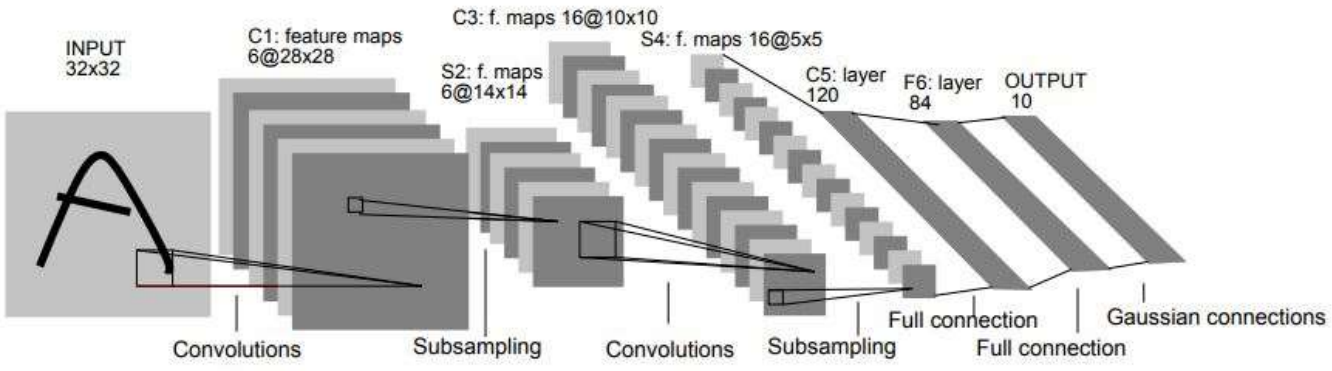
VGG-16 is thus the "frontier" model which is tha maximum my compute power can take, in reasonable boundaries of training time. I did not try other more "ground-breaking" models such as ResNet and GoogLeNet for the same reasons. All 3 architectures below are implemented ground-up by reading through the respective papers, checking for online references of people who have done it, and making the necessary adjustments (ZFNet and VGG were written for 3 channels and larger image input size, hence I had to scale it down).

Building a network from scratch: Modified LeNet-5

The LeNet-5 architecture has:

1. 2 sets of alternating convolutional and pooling layers. These are:
   a. **Convolutional layer with 6 filters**. b. **Pooling layer**. Different implementations online use different methods, but to keep things consistent with the other architecture, I use MaxPooling2D.
   c. **Convolutional layer with 16 filters**. b. **Pooling layer**. MaxPooling2D was used as per 1d.
2. A flattening layer in preparation for fully connected layers later
3. Two fully-connected layers. The size of the layers, at 120 and 84, are kept in accordance with that of the original paper.
4. Finally a softmax classifier with 10 classes.

The pictorial representation is as follows:



The only difference for the Practical 8 model differs is that it has one dropout later after the two sets of convolutional and pooling layer, as well as different sizes and feature maps for the convolutional and dense layers. The model was trained for 1 epoch. Validation on the test set yielded an error of 4.71%.

In [5]:

```
#Model 2 - Modified LeNet

model2 = Sequential()
#Conv Layer 1
model2.add(Conv2D(6, (5, 5), input_shape = (1, 28, 28), padding ='valid', activation =
'relu'))
#Pooling Layer
model2.add(MaxPooling2D(pool_size = 2, strides = 2))
#Conv Layer 2
model2.add(Conv2D(16, (5, 5), activation = 'relu'))
#Pooling Layer 2
model2.add(MaxPooling2D(pool_size = 2, strides = 2))
model2.add(Flatten())
#Fully connected layer 1
model2.add(Dense(units = 120, activation = 'relu'))
#Fully connected layer 2
model2.add(Dense(units = 84, activation = 'relu'))
#Output Layer
model2.add(Dense(units = 10, activation = 'softmax'))
model2.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accur
acy'])

model2.summary()

# Fit the model
model2.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=1, batch_size=200, verbose=True)
# Final evaluation of the model
scores = model2.evaluate(X_test, y_test, verbose=True)
print()
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 6, 24, 24)         156
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 6, 12, 12)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 16, 8, 8)          2416
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 16, 4, 4)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 120)               30840
_________________________________________________________________
dense_8 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_9 (Dense)              (None, 10)                850
=================================================================
Total params: 44,426
Trainable params: 44,426
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 58s - loss: 0.4201 - acc:
0.8785 - val_loss: 0.1495 - val_acc: 0.9529
 9984/10000 [============================>.] - ETA: 0s
CNN Error: 4.71%
```

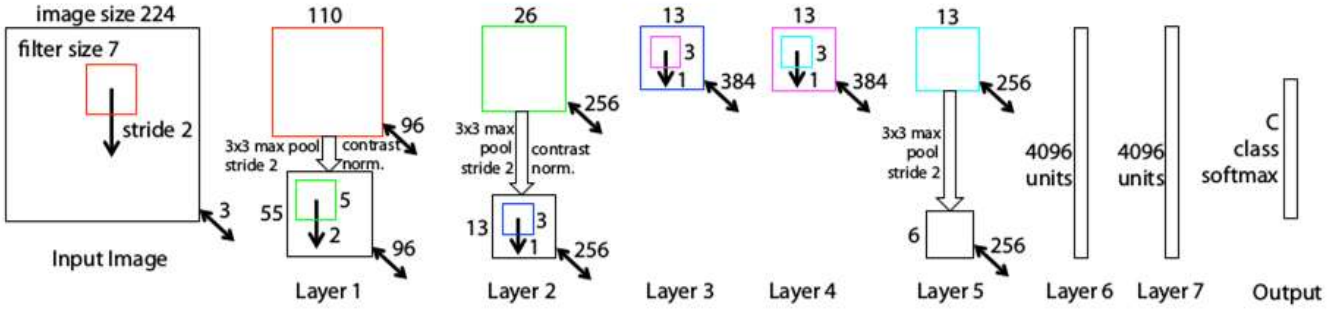
Building a network from scratch: Modified ZFNet

ZFNet was slightly trickier as it was meant for RGB images with 3 channels, and with an image size of 224 by 224. This meant that it had to be scaled down in order for it to fit with the MNIST dataset.

The architecture of ZFNet, along with details of how I modified it, are as follows:

1. A series of convolutional and pooling layers
   a. **Convolutional layer with 96 filters**. I changed the filter size significantly by taking a proportion of the original image dimension (96/224) and multiplying it to the MNIST image dimension (28)to get 12 feature maps. Did not apply the stride factor of 2 for the convolution -- I tried it, but ran into the issue of the out image dimension being too small for any further convolutions somewhere around the 7th layer. Hence kept the default stride at 1.
   b. **Pooling layer**. MaxPooling2D with stride of 2.
   c. **Convolutional layer with 256 filters**. I changed the filter size to 32 as per 1a by applying proportion.
   d. **Pooling layer**. MaxPooling2D, but I adjusted the stride again as my input image dimensions were a lot smaller. Used stride value of 1.
   e. **Convolutional layer with 384 filters**. I changed the filter size to 48 as per 1a by applying proportion.
   f. **Convolutional layer with 385 filters**. I changed the filter size to 48 as per 1a by applying proportion.
   g. **Convolutional layer with 256 filters**. I changed the filter size to 32 as per 1a by applying proportion.
2. A flattening layer in preparation for fully connected layers later
3. Two sets of alternating layers between fully connected and dropout.
   a. **Fully connected layer**. I changed the size of the layer to 1024, as 4096 was for a 224x224 image.
   b. **Dropout layer**. I changed the proportion of the dropout to 0.1 instead of 0.5 as per original architecture, as there isn't a lot of information in 28x28 pixels and my architecture cannot afford to lose too much information, or it will risk underfitting of the layer to 1024, as 4096 was for a 224x224 image.
   c. **Fully connected layer**. Same, as per 3a.
   d. **Dropout layer**. Same, as per 3b, but dropout value at 0.2 after some experimentation
   e. **Fully connected layer**. Scaled down the size, in preparation for the softmax prediction layer
4. Finally a softmax classifier with 10 classes.

The pictorial representation is as follows:



ZF Net Architecture

The model was trained for 1 epoch. Validation on the test set yielded an error of 4.03%.

In [6]:

```
#Model 3 - Modified ZFNet

model3 = Sequential()
#Conv Layer 1
model3.add(Conv2D(12,(7,7), input_shape=(1, 28, 28), padding='valid', activation='relu'
, kernel_initializer='uniform'))
#Pooling Layer
model3.add(MaxPooling2D(pool_size=2,strides=2))
#Conv Layer 2
model3.add(Conv2D(32,(5,5), padding='valid',activation='relu', kernel_initializer='unif
orm'))
#Pooling Layer
model3.add(MaxPooling2D(pool_size=1,strides=1))
model3.add(Conv2D(48,(3,3),padding='valid',activation='relu', kernel_initializer='unifo
rm'))
#Conv Layer 3
model3.add(Conv2D(48,(2,2),padding='valid',activation='relu', kernel_initializer='unifo
rm'))
#Conv Layer 4
model3.add(Conv2D(32,(2,2),padding='valid',activation='relu', kernel_initializer='unifo
rm'))
#Pooling Layer
model3.add(MaxPooling2D(pool_size=1,strides=1))
model3.add(Flatten())
#Fully Connected Layer 1
model3.add(Dense(units = 1024,activation='relu'))
model3.add(Dropout(0.1))
#Fully Connected Layer 2
model3.add(Dense(units = 1024,activation='relu'))
model3.add(Dropout(0.2))
#Fully Connected Layer 3
model3.add(Dense(units = 512,activation='relu'))
#Output Layer
model3.add(Dense(units = 10, activation='softmax'))
model3.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model3.summary()

# Fit the model
model3.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=1, batch_size=200, verbose=True)
# Final evaluation of the model
scores = model3.evaluate(X_test, y_test, verbose=True)
print()
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 12, 22, 22)        600
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 12, 11, 11)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 7, 7)          9632
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 32, 7, 7)          0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 48, 5, 5)          13872
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 48, 4, 4)          9264
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 32, 3, 3)          6176
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 32, 3, 3)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_10 (Dense)             (None, 1024)              295936
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_11 (Dense)             (None, 1024)              1049600
_________________________________________________________________
dropout_3 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_12 (Dense)             (None, 512)               524800
_________________________________________________________________
dense_13 (Dense)             (None, 10)                5130
=================================================================
Total params: 1,915,010
Trainable params: 1,915,010
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 138s - loss: 0.4699 - acc:
0.8435 - val_loss: 0.1314 - val_acc: 0.9597
 9984/10000 [============================>.] - ETA: 0s
CNN Error: 4.03%
```

Modified VGG-16

VGG-16 was the hardest to build because given the depth of the layers, I had repeatedly run into the issue of the output dimension being too small to further run a convolution layer. VGGNet, like ZFNet, was also originally written for dimension 224x224x3 images. I applied the same adjustments as I did to ZFNet to scale down the architecture, by changing the number of filters, the filter sizes, as well as the strides and sizes of the pooling layers.

The architecture of VGG16, along with details of how I modified it, are as follows:

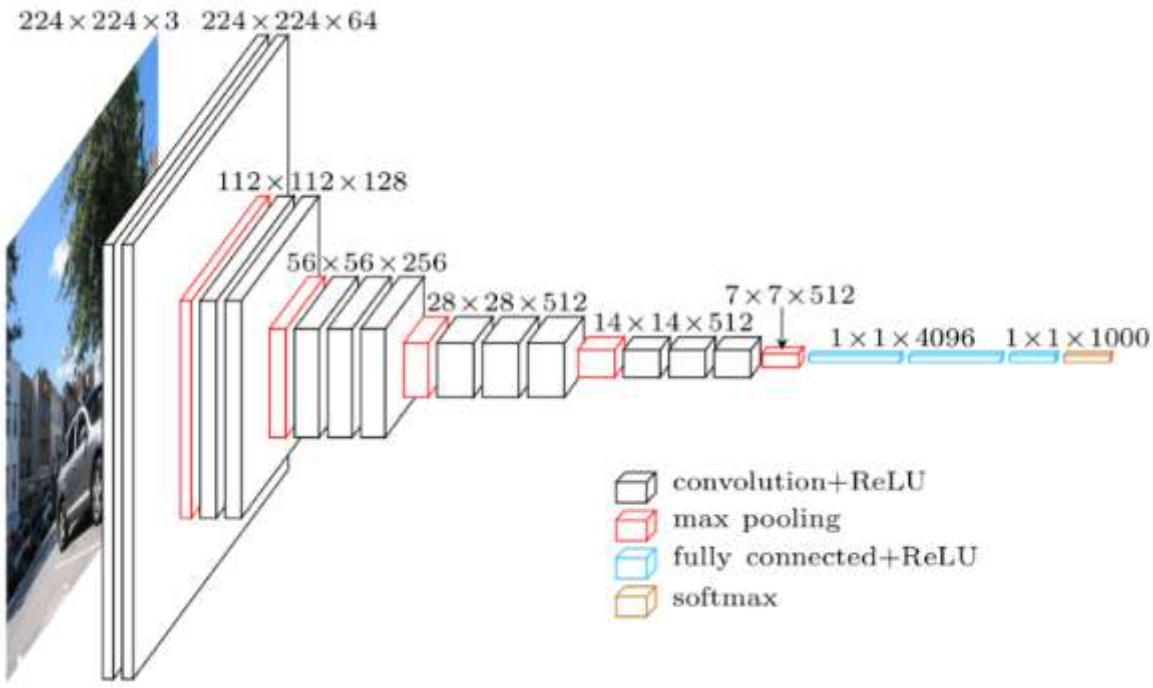
1. First set of convolutional and pooling layers
   a. **Convolutional layer with 64 filters**. As per ZFNet, I took a proportion of the original image dimension (64/224) and multiplied it to the MNIST image dimension to get 8 feature maps. Kept the dimensions of the filter size as they seem reasonable at 3x3, and kept the stride
   b. **Convolutional layer with 64 filters**. Modifications as per 1a.
   c. **Pooling layer**. MaxPooling2D with stride of 2.
2. Second set of convolutional and pooling layers
   a. **Convolutional layer with 128 filters**. Similar transformation by applying proportion to the number of filters, obtained 16 filters.
   b. **Convolutional layer with 128 filters**. Modifications as per 2a.
   c. **Pooling layer**. MaxPooling2D stride and size changed from 2 to 1, as the network is very deep, and if I do not change at thi point, somewhere in the 4th set of convolutional and pooling layers the dimensions will get too small.
3. Third set of convolutional and pooling layers
   a. **Convolutional layer with 256 filters**. Similar transformation by applying proportion to the number of filters, obtained 32 filters.
   b. **Convolutional layer with 256 filters**. Modifications as per 3a
   c. **Convolutional layer with 256 filters**. Modifications as per 3a
   b. **Pooling layer**. MaxPooling2D, but with stride and size value changed from 2 to 1
4. Third set of convolutional and pooling layers
   a. **Convolutional layer with 512 filters**. Similar transformation by applying proportion to the number of filters, obtained 64 filters.
   b. **Convolutional layer with 512 filters**. Modifications as per 4a
   c. **Convolutional layer with 512 filters**. Modifications as per 4a
   b. **Pooling layer**. MaxPooling2D, but with stride and size value changed from 2 to 1
5. Third set of convolutional and pooling layers
   a. **Convolutional layer with 512 filters**. Similar transformation by applying proportion to the number of filters, obtained 64 filters.
   b. **Convolutional layer with 512 filters**. Modifications as per 5a
   c. **Convolutional layer with 512 filters**. Modifications as per 5a
   b. **Pooling layer**. MaxPooling2D, but with stride and size value changed from 2 to 1
6. A flattening layer in preparation for fully connected layers later
7. A dropout layer to prevent overfitting. While many online implementations have set it at 0.5, I set it at 0.2 due to the smaller size of MNIST images.
8. A series of fully connected and dropout layers.
   a. **Fully connected layer of size 4096**. I changed the size of the layer to 512 by proportion, as 4096 was for a 224x224 image.
   b. **Dropout layer**. I changed the proportion of the dropout to 0.2 instead of 0.5 as per original architecture.
   c. **Fully connected layer of size 4096**. Same, as per 8a.
   d. **Dropout layer**. Same, as per 8b

e. **Fully connected layer of size 1000**. Scaled down the size to 125, in preparation for the softmax prediction layer

f. **Dropout layer**. Same, as per 8b.

9. Finally a softmax classifier with 10 classes.

The pictorial representation is as follows:



The model was trained for 1 epoch. Validation on the test set yielded an error of 1.83%. This was the best performance out of the 3 models investiated.

In [7]:

```
# Model 4 - Modified VGG 16

#Convolution set 1 - originally 64 filters
model4 = Sequential()
model4.add(Conv2D(8, (3,3), input_shape = (1, 28, 28), activation='relu', padding='sam
e'))
model4.add(Conv2D(8, (3,3), activation='relu', padding='same'))
model4.add(MaxPooling2D(pool_size = 2, strides = 2))

#Convolution set 2 - originally 128 filters
model4.add(Conv2D(16, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(16, (3,3), activation='relu', padding='same'))
model4.add(MaxPooling2D(pool_size = 1, strides = 1))

#Convolution set 3 - originally 256 filters
model4.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model4.add(MaxPooling2D(pool_size = 1, strides = 1))

#Convolution set 4 - originally 512 filters
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(MaxPooling2D(pool_size = 1, strides = 1))

#Convolution set 5 - originally 512 filters
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model4.add(MaxPooling2D(pool_size = 1, strides = 1))

# Fully connected and dropout layers
model4.add(Flatten())
model4.add(Dropout(0.2))
model4.add(Dense(units = 512, activation='relu'))
model4.add(Dropout(0.2))
model4.add(Dense(units = 512, activation='relu'))
model4.add(Dropout(0.2))
model4.add(Dense(units = 125, activation='relu'))
model4.add(Dense(units = 10, activation='softmax'))

model4.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics=['accuracy'
])
model4.summary()

# Fit the model
model4.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=1, batch_size=200, verbose=True)
# Final evaluation of the model
scores = model4.evaluate(X_test, y_test, verbose=True)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 8, 28, 28)	80
conv2d_13 (Conv2D)	(None, 8, 28, 28)	584
max_pooling2d_10 (MaxPooling	(None, 8, 14, 14)	0
conv2d_14 (Conv2D)	(None, 16, 14, 14)	1168
conv2d_15 (Conv2D)	(None, 16, 14, 14)	2320
max_pooling2d_11 (MaxPooling	(None, 16, 14, 14)	0
conv2d_16 (Conv2D)	(None, 32, 14, 14)	4640
conv2d_17 (Conv2D)	(None, 32, 14, 14)	9248
conv2d_18 (Conv2D)	(None, 32, 14, 14)	9248
max_pooling2d_12 (MaxPooling	(None, 32, 14, 14)	0
conv2d_19 (Conv2D)	(None, 64, 14, 14)	18496
conv2d_20 (Conv2D)	(None, 64, 14, 14)	36928
conv2d_21 (Conv2D)	(None, 64, 14, 14)	36928
max_pooling2d_13 (MaxPooling	(None, 64, 14, 14)	0
conv2d_22 (Conv2D)	(None, 64, 14, 14)	36928
conv2d_23 (Conv2D)	(None, 64, 14, 14)	36928
conv2d_24 (Conv2D)	(None, 64, 14, 14)	36928
max_pooling2d_14 (MaxPooling	(None, 64, 14, 14)	0
flatten_5 (Flatten)	(None, 12544)	0
dropout_4 (Dropout)	(None, 12544)	0
dense_14 (Dense)	(None, 512)	6423040
dropout_5 (Dropout)	(None, 512)	0
dense_15 (Dense)	(None, 512)	262656
dropout_6 (Dropout)	(None, 512)	0
dense_16 (Dense)	(None, 125)	64125
dense_17 (Dense)	(None, 10)	1260

Total params: 6,981,505
Trainable params: 6,981,505
Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples
Epoch 1/1

```
60000/60000 [==============================] - 1507s - loss: 0.3996 - acc:
0.8663 - val_loss: 0.0558 - val_acc: 0.9817
10000/10000 [==============================] - 122s
CNN Error: 1.83%
```

Performance boosting: Data Augmentation

Data Augmentation

I wanted to see if it was still possible to improve on the best performing model (i.e. VGG-16). Hence I decided to apply some tweaks. One of it is data augmentation, which I learnt about through this site: https://machinelearningmastery.com/image-augmentation-deep-learning-keras/ (https://machinelearningmastery.com/image-augmentation-deep-learning-keras/). Essentially this involved creating synthetic training samples by making slight transformations to the data without changing its integrity. This could include whitening, rotation, flipping or other geometric changes, which would lead to very new numpy arrays.

The machine essentially reads this as new training points, and we have now increased our training set.The data augmenting transformation I applied here is to rotate each image by 20 degrees. This leads to 120,000 training samples.

Learning Rate

I also learnt about another performance boosting measure, i.e. how the learning rate parameter could affect model performance, here: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 (https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10). Using a smaller learning rate could lead to a better result as it ensures that the algorithm will not overshoot the minima when trying to reduce the loss function. This was especially important, as I learnt after several iterations of the code below: my model was hovering around a loss function of 0.08, and at times the loss function would increase with time, then go back down again.

I dropped the learning rate to 1e-4.

In [11]:

```
#Fitting the CNN model with performance boosting measures


#Data augmentation
X_train1 = X_train.reshape(-1,1,28,28)
X_test1 = X_test.reshape(-1,1,28,28)

from keras.preprocessing.image import ImageDataGenerator
X_train2 = numpy.array(X_train, copy=True)
y_train2 = numpy.array(y_train, copy=True)
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=Tru
e,rotation_range=20)
datagen.fit(X_train1)

# Concatenating the old data with the augmented data
augmented_x  = numpy.concatenate((X_train1, X_train2), axis=0)
augmented_y  = numpy.concatenate((y_train, y_train2), axis=0)

#Setting the learning rate to a slower one
from keras import optimizers
adam = optimizers.Adam(lr=1e-4)
model4.compile(loss='categorical_crossentropy', optimizer = adam, metrics=['accuracy'])

#Fitting the VGG-16 model to the augmented data, at a slower learning rate
model4.fit(augmented_x,augmented_y, validation_data=(X_test1, y_test),
          epochs=1, batch_size=200, verbose=True)

# Final evaluation of the model
scores = model4.evaluate(X_test1, y_test, verbose=True)
print()
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

```
Train on 120000 samples, validate on 10000 samples
Epoch 1/1
120000/120000 [==============================] - 2960s - loss: 0.0449 - ac
c: 0.9867 - val_loss: 0.0317 - val_acc: 0.9903
10000/10000 [==============================] - 116s

CNN Error: 0.97%
```

Applying the performance boosters above led to a 0.97% error rate -- this is around state-of-the-art performance (as highlighted in Practical 8).

Evaluation - Confusion Matrix

We know the accuracies of the different models, but for a more granular measure of what was classified correctly and what was not, we can plot a confusion matrix of the predicted labels against the true labels. SK-learn already has a template for plotting confusion matrices for plotting multi-class predictions, so I will reuse the code (provided here: https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html (https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html)):

In [13]:

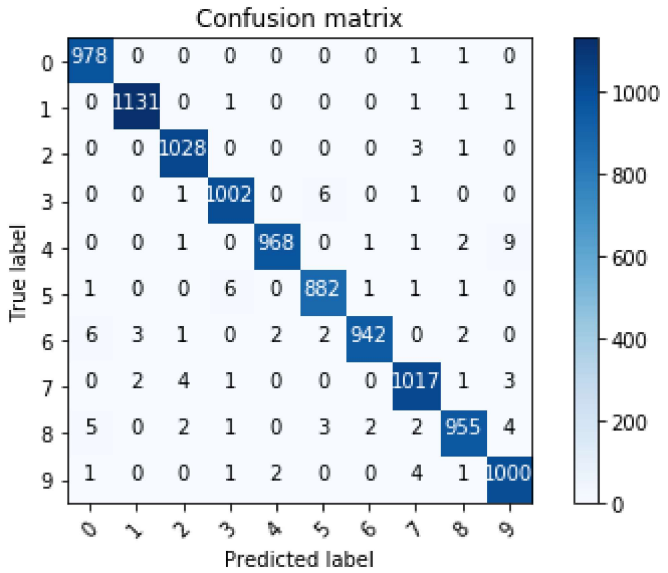
```
#Confusion Matrix Template from SK-Learn Documentation

import itertools
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

"""
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
"""
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    plt.rcParams["figure.figsize"] = [35,35]
    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')
```

Plotting the confusion matrix shows us where the main errors take place. It seems that:
a) Some 6s were predicted as 0s
b) Some 4s were predicted as 9s
c) Many 3s were predicted as 5s, and vice versa

This could be due to ambiguities in the handwriting (e.g. the loop in 6 closing near to the top than in the middle, and the slope in 4 being more rounded hence giving the model the impression that it was a 9). Nevertheless, there was no single mis-classification that had more than 10 instances -- a definitely satisfying performance, for a training of only 1-2 epochs!

In [14]:

```
# Predict the values from the validation dataset
Y_pred = model4.predict(X_test1)
# 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_test,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))
plt.show()
```



Conclusion

In this notebook and exercise, I have investigated and built using the Keras API three different CNN models. The VGG-16 architecture was able to achieve near state-of-the-art results after training on augmented data, and with a slower learning rate. If this were to be continued, and with more compute power, I would be interested to try building GoogLeNet or Resnet-50 as these are models that have outperformed even VGG in different runs of ILSVRC.

Resources

1. Adit Deshpande, The 9 Deep Learning Papers You Need to Know About. https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html (https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html)
2. Prakash Jay, Image Classification Architectures Review. https://medium.com/@14prakash/image-classification-architectures-review-d8b95075998f (https://medium.com/@14prakash/image-classification-architectures-review-d8b95075998f)
3. Yann Lecun, Leon Bottou, Yoshua Bengio, Patrick Haffner. Gradient Based Learning Applied to Document Recognition. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf((http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf()**Original LeNet-5 Paper**)
4. Muhammad Rizwan, LeNet-5: A Classic CNN Architecture. https://engmrk.com/lenet-5-a-classic-cnn-architecture/ (https://engmrk.com/lenet-5-a-classic-cnn-architecture/)
5. Matthew D Zeiler, Rob Fergus, Visualising and Understanding Convolutional Networks. https://arxiv.org/abs/1311.2901 (https://arxiv.org/abs/1311.2901) (**Original ZFNet Paper**)
6. Pantelis Kaplanoglou, ZFNet Architecture. https://www.researchgate.net/figure/ZFNet-architecture-The-apparent-differences-with-AlexNet-are-located-in-the-first-two_fig13_318277197 (https://www.researchgate.net/figure/ZFNet-architecture-The-apparent-differences-with-AlexNet-are-located-in-the-first-two_fig13_318277197)
7. Karen Simonyan & Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/pdf/1409.1556.pdf (https://arxiv.org/pdf/1409.1556.pdf) (**Original VGGNet Paper**)
8. Fabio Perez, Reading the VGG Network Paper and Implementing It From Scratch with Keras: https://hackernoon.com/learning-keras-by-implementing-vgg16-from-scratch-d036733f2d5 (https://hackernoon.com/learning-keras-by-implementing-vgg16-from-scratch-d036733f2d5)
9. Jonathan Hui, Keras Tutorial. https://jhui.github.io/2018/02/11/Keras-tutorial/ (https://jhui.github.io/2018/02/11/Keras-tutorial/)
10. Moghazy, Kaggle: A Guide to CNNs with Data Augmentation in Keras. https://www.kaggle.com/moghazy/guide-to-cnns-with-data-augmentation-keras (https://www.kaggle.com/moghazy/guide-to-cnns-with-data-augmentation-keras)
11. Jason Brownie: Image Augmentation for Deep Learning with Keras. https://machinelearningmastery.com/image-augmentation-deep-learning-keras/ (https://machinelearningmastery.com/image-augmentation-deep-learning-keras/)
12. Jonas Krause, Image Classification using Convolutional Neural Networks (CNNs). https://lipyeow.github.io/ics491f17/morea/deepnn/ImageClassification-CNN.pdf (https://lipyeow.github.io/ics491f17/morea/deepnn/ImageClassification-CNN.pdf)
13. Siddarth Das, CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5 (https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5)
14. Stack Exchange, Q126238: What are the advantages of ReLU over sigmoid function in deep neural networks?https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks (https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks)
15. Fabio Perez, Reading the VGG Network Paper and Implementing It From Scratch with Keras. https://hackernoon.com/learning-keras-by-implementing-vgg16-from-scratch-d036733f2d5 (https://hackernoon.com/learning-keras-by-implementing-vgg16-from-scratch-d036733f2d5)
16. Jason Brownlee, Image Augmentation for Deep Learning With Keras. https://machinelearningmastery.com/image-augmentation-deep-learning-keras/ (https://machinelearningmastery.com/image-augmentation-deep-learning-keras/)