Practical 2: Implementing ConvNets in Keras
3rd IAPR Summer School on Document Analysis
August 2019

Task 1: Digit Recognition with ConvNets

In this task, you will carry out digit recognition (using MNIST database) using convolutional neural networks. The same problem was solved in the previous practical using vanilla ANNs. You may compare the performance of a traditional ANN with that of a ConvNet using the same experimental setup.

Convolutional Neural Networks

Convolutional Neural Networks (CNN) make the assumption that the input are images. The CNN architecture is defined by different layers typically including the Input Layer, Convolutional Layer, Activation Layer (ReLU), Pooling Layer, Fully-Connected Layer. The *input layer* normally has the shape 'height x width x depth'. In our example of digit recognition, the shape is *28x28x1* but in colored images the *depth* would be 3, one for each value in the RGB color model. The input is passed to a *convolutional layer* using multiple filters, it creates another matrix with different height, width and as deep as many filters defined. A filter is a small window that traverses the image to compute its features. Its output goes into and activation function which in our case is the ReLU function. A *pooling layer* is employed to reduce the (spatial) dimensionality. Then the *fully-connected layer* is a normal feed forward layer for classification. It is common to combine the *conv* layer with another *conv* to a *pooling* layer and repeat it *n* times before going to the *fully-connected* layer.

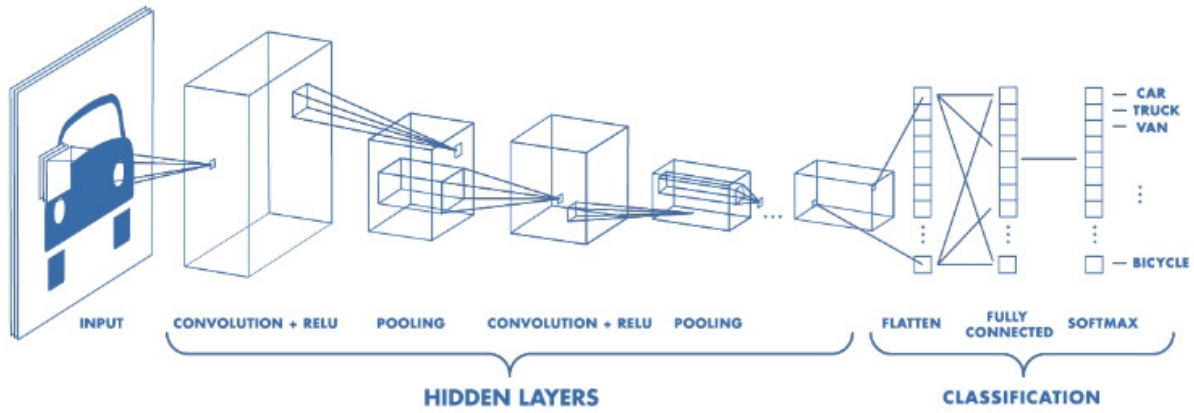

Figure 1 Typical architecture of a ConvNet

Libraries and modules

```
import numpy as np
from keras.datasets import mnist
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.utils import to_categorical
import matplotlib.pylab as plt

np.random.seed(123)  # for reproducibility
```

The keras Sequential model allows to add layers, each layer having its own architecture and purpose. The idea is to sequentially add different layers to our model until having a Neural Network architecture that can process and classify images of handwritten digits.

Data Preparation

Every Machine Learning model needs some data preparation. Typical data preparation steps include the following.

- Load dataset.
- Separate dataset into training and test datasets.

- Visualize data to get some intuition.
- Prepare input data to feed the input layer.
- Prepare data labels.

```
# input image dimensions
img_x, img_y = 28, 28
# load the MNIST data set, which already splits into train and test sets for us
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The load_data() function return two tuples, already splitting the data in training and test collections. The x_train.shape will be (60000, 28, 28), 60,000 examples of 28x28 pixels grayscale images of the 10 digits while y_train is a vector of 60,000 examples where each element is an integer from 0 to 9. Likewise, x_test and y_test contain 10,000 examples. You can visualize few of the digits (similar to Practical 1) to see how the images look like.

The next step is to reshape the input matrix to have the shape samples x height x width x channels, since these are grayscale images the channels will be 1, in other images the channels normally would be 3, one for each color in the RGB color model.

```
# reshape the data into a 4D tensor:
# (sample_number, x_img_size, y_img_size, num_channels)
# because the MNIST is greyscale, we only have a single channel
# RGB colour images would have 3
x_train = x_train.reshape(x_train.shape[0], img_x, img_y, 1)
x_test = x_test.reshape(x_test.shape[0], img_x, img_y, 1)

input_shape = (img_x, img_y, 1)
```

Next, normalize the input data to be in the range from 0 to 1, first by changing the type to float32 and dividing all elements by 255.

```
# convert the data to the right type
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

The labeled data is a 1-dimensional array with an integer value from 0 to 9 and needs to be converted to one-hot-encoding prior to model training and evaluation.

```
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Network Architecture

The keras Sequential model allows to generate a model for training by adding layers to it. The first layer is a Conv2D layer with a ReLU activation function, 32 filters and a 5x5 convolutional window. Then comes the Pooling layer, where we employ the MaxPooling2D keras layer with a 2x2 pool size. The next conv layer uses 64 filters each of size 5x5. We also use a Dropout layer for to prevent over fitting. After the convolutions, we aim to classify the data with fully connected layers, thus we use a Flatten layer to flat the data to a 1-dimensional vector to be used as input for two Dense (fully-connected) layers. We could use more or less Conv2D layers or change the hyperparameters in each layer, or the total nodes in one of the Dense layers, or use one Dense layer instead of two. Trying several architectures and measuring which generalize better is a common strategy to optimize the performance.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Compile and Train the Model

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])


history = model.fit(x_train, y_train,
          batch_size=128,
          epochs=10,
          verbose=1)
```

You can visualize the training accuracy and loss as a function of number of epochs.

```
plt.plot(history.history['acc'])
plt.plot(history.history['loss'])
plt.title('Model Accuracy and Loss')
plt.xlabel('Epoch')
plt.ylabel('Accuracy/Loss')
plt.legend(['Accuracy', 'Loss'], loc='upper left')

plt.show()
```

Evaluate the Model

The trained model can now be evaluated on the test data.

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

You may change the different hyper parameters and study the performance evolution.

Task 2: Recognizing CIFAR-10 Images with ConvNets and Saving Model

The CIFAR-10 dataset contains 60,000 color images of 32 x 32 pixels in 3 channels divided into 10 classes. Each class contains 6,000 images. The training set contains 50,000 images, while the test sets provides 10,000 images.

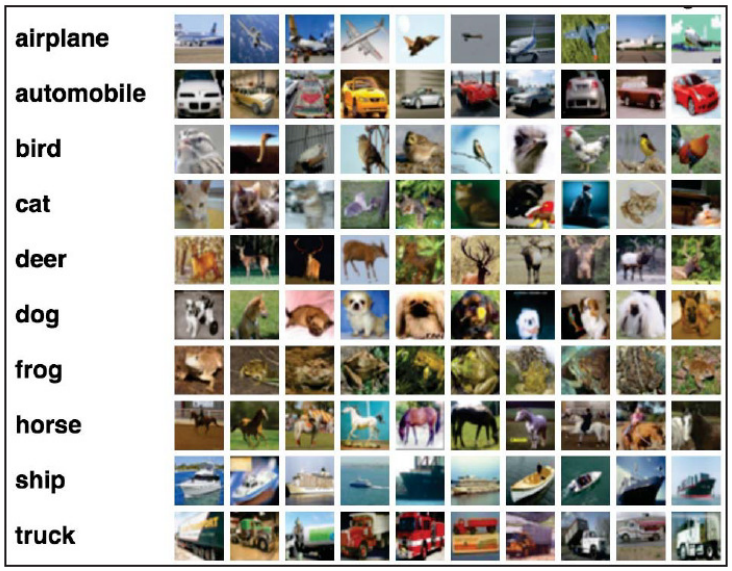

Figure 2 Sample images in the CIFAR-10 Database

As a first step, we import the useful modules and load (download) the dataset.

```
import numpy as np
from matplotlib import pyplot as plt
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dense, Flatten, Dropout,Activation
from keras.utils import to_categorical

seed=10
np.random.seed(seed)

(X_train,y_train),(X_test,y_test) = cifar10.load_data()
print(X_train.shape)

sample_image = X_train[1,:,:,:]
plt.imshow(sample_image), plt.axis('off')
plt.show()
```

The next step prepares the normalized data for model training and subsequent evaluation.

```
classes = 10
Y_train = to_categorical(y_train,classes)
Y_test = to_categorical(y_test,classes)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train = X_train/255
X_test = X_test/255
```

Create a ConvNet with convolutional, pooling and fully connected layers.

```
model = Sequential()
model.add(Conv2D(32,(3,3), padding='same',input_shape=(32,32,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
model.summary()
```

Subsequently, the model can be trained and evaluated.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])


history=model.fit(X_train, Y_train,
          batch_size=128,
          epochs=10,
          verbose=1)

score = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

In some cases, it is desirable to save the model architecture and weights so that a fresh training is not required every time classification is to be carried out. The model is typically saved as 'json' file while the weights are saved as ".h5" file.

```
model_json = model.to_json()
open('cifar10Net.json','w').write(model_json)
model.save_weights('cifar10weights.h5',overwrite=True)
```

Task 3: Recognizing an Object using a Trained Model

It is common to load a saved model (and weights) and perform predictions.

```
from keras.models import model_from_json
import cv2
import matplotlib.pyplot as plt


#Import model
model_arch = 'cifar10Net.json'
model_weights='cifar10weights.h5'

model =model_from_json(open(model_arch).read())
model.load_weights(model_weights)
```

By default, the images loading the cv2 library are in BGR format. You need to convert them to RGB format before feeding to the model.

```
#Image to recognize
image_path="G:/Academics/Bahria/Teaching/Spring 2019/Deep Learning/Implementation/Book/dog2.jpg"
#Load Image
img = cv2.imread(image_path,cv2.IMREAD_COLOR)
img_rgb = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
print(img.shape)
```

The image also needs to be resized to 32x32 to match the input layer of the previously trained network (in Task 2).

```
#Resize image to fit the model input
img_res = cv2.resize(img,(32,32))
#Display image
plt.imshow(img_res)
plt.axis("off")
plt.show()
```

The pixel values can be normalized and the image can be converted to 4-D tensor.

```
#Normalize pixel values and reshape
img_res= img_res/255
print(img_res.shape)
#Conver to 4D tensor (1,32,32,3)
img_res=img_res.reshape((1,)+img_res.shape)
print(img_res.shape)
```

The model predicts the class label of the input image as follows.

```
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
predictions = model.predict_classes(img_res)
print(predictions)
```

Task 4: Data Augmentation

When working with image classification, the input which a user will give can vary in many aspects like angles, zoom etc. Hence, we should train our model to accept and make sense of almost all types of inputs. This can be done by training the model for all possibilities. But we can't go around clicking the same training picture in every possible angles especially once the number of classes is large. This can be easily be solved by a technique called Image Data Augmentation, which takes an image, converts it and save it in all the possible forms we specify. We will employ the ImageDataGenerator class in Keras for this purpose. The ImageDataGenerator takes the following argument:

- **rotation_range**: amount of rotation
- **width_shift_range , height_shift_range** : amount of shift in width, height
- **shear_range** : shear angle in counter-clockwise direction as radians

- **zoom_range** : range for random zoom
- **horizontal_flip** : Boolean (True or False). Randomly flip inputs horizontally
- **fill_mode** : One of {"constant", "nearest", "reflect" or "wrap"}. Points outside the boundaries of the input are filled according to the given mode

Setting the **rotation_range** argument to 90 for instance, randomly applies rotation on the image up to 90 degrees.

The following program applies data augmentation to an image saves all images in a directory.

```
import cv2
import matplotlib.pyplot as plt
from keras.preprocessing.image import ImageDataGenerator

image_path="dog.jpg"
img = cv2.imread(image_path,cv2.IMREAD_COLOR)
img_rgb = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
print(img.shape)

plt.imshow(img_rgb)
plt.axis("off")
plt.show()

#Convert 3D Data to 4D: Add one dimensions
img_rgb = img_rgb.reshape((1,)+img_rgb.shape)
print(img_rgb.shape)

datagen = ImageDataGenerator(rotation_range=40,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip=True,
                             fill_mode='nearest')
i = 0
for batch in datagen.flow(img_rgb,save_to_dir='preview', save_prefix='dog', save_format='jpeg'):
    i += 1
    if i > 9:
        break
```

We run the loop 10 times and for each iteration we use **`datatgen.flow()`** function. Ten images of the dog with changes that were specified in **`datagen`** will be produced and will be stored in the folder called '*preview*'.

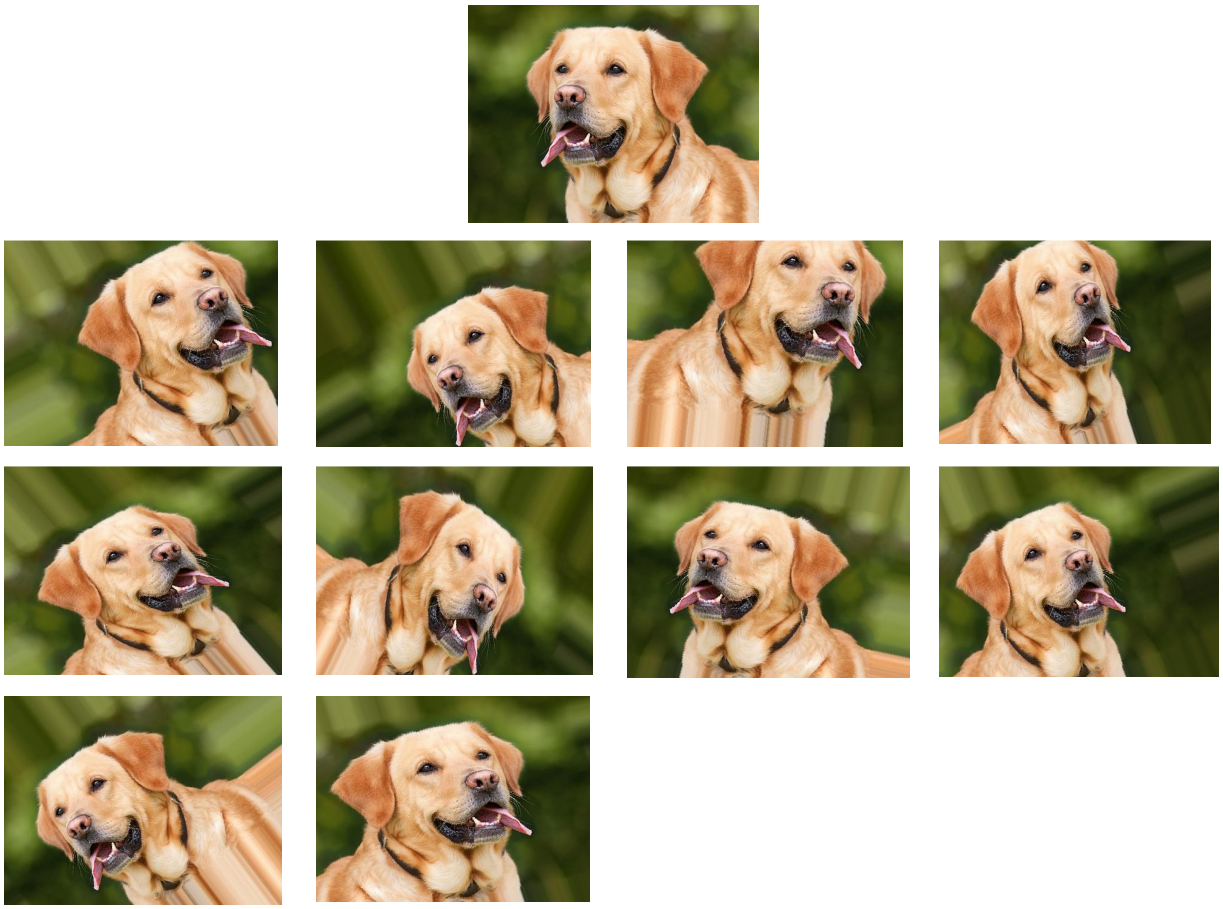

Figure 3 An Original Image and its augmented versions

Rather than first generating the augmented images and saving them to disk, the image data generator API is designed to be iterated by the deep learning model fitting process, creating augmented image data for you just-in-time. This reduces the memory overhead, but adds some additional time cost during model training.

After you have created and configured your ImageDataGenerator, you must fit it on your data. This will calculate any statistics required to actually perform the transforms to your image data. You can do this by calling the fit() function on the data generator and pass it your training dataset.

```
datagen = ImageDataGenerator(rotation_range=40,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip=True,
                             fill_mode='nearest')

datagen.fit(X_train)
```

The data generator itself is in fact an iterator, returning batches of image samples when requested. We can configure the batch size and prepare the data generator and get batches of images by calling the flow() function.

```
datagen.flow(datagen.flow(X_train,Y_train,batch_size=128))
```

Finally we can make use of the data generator. Instead of calling the fit() function on our model, we must call the fit_generator() function and pass in the data generator and the desired length of an epoch as well as the total number of epochs on which to train.

```
history = model.fit_generator(datagen.flow(X_train,Y_train,batch_size=128),
                              steps_per_epoch=X_train.shape[0]/128,
                              epochs = 2,
                              verbose=1)
```

Exercise:

Compare the classification performance on CIFAR-10 dataset by using deeper ConvNets and Data Augmentation.

++++++++++++