COMP6685_Lab6b_Keras_CNN_CIFAR

March 2, 2025

1 Advanced Convolutional Neural Networks (CNNs) with CIFAR-10 dataset

In this tutorial we will learn how to use more complex CNNs, showing that the training of a **deeper** CNN can improve the performance of the model. We will also explore the concept of **data augmentation** to understand how to increase the variability of the training set by, for example, rotating the original images to generate new training stimuli.

This tutorail will use the CIFAR-10 training set.

CNN for CIFAR-10

To work with more complex CNNs, we will now use a more complex training dataset called **CIFAR-10**. https://www.cs.toronto.edu/~kriz/cifar.html . CIFAR-10 is a benchamark machine learning set of low-resolution, colour images. It includes 60000 32x32 colour (using 3 RGB colour channels) images in these 10 classes of objects: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. Each class has 6000. There are 50000 training images and 10000 test images. This dataset is enclosed in the default Anaconda KERAS package.

Initialisation of the program

The program starts with the importing of typical Keras and other Python service modules.

[ ]:
```
# importing of modules for CIFAR-10 CNN
from tensorflow.keras.datasets import cifar10
from tensorflow.keras import utils
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

# importing of service libraries
import numpy as np
import matplotlib.pyplot as plt

print('Libraries imported.')
```

The following constant and variable definitions are needed for the network and training parameters.

[ ]:
```
#training constants
BATCH_SIZE = 128
N_EPOCH = 20 # use 20 for best initial results
N_CLASSES = 10
VERBOSE = 1
VALIDATION_SPLIT = 0.2
OPTIM = RMSprop()

print('Main variables initialised.')
```

Constant definition for the training set images

[ ]:
```
# CIFAR_10 is a set of 60K images 32x32 pixels on 3 channels
IMG_CHANNELS = 3
IMG_ROWS = 32
IMG_COLS = 32

print('Image variables initialisation')
```

CIFAR-10 data loading and processing

Loading and preparation of the CIFAR-10 training set.

[ ]:
```
#load dataset
(input_X_train, output_y_train), (input_X_test, output_y_test) = cifar10.
 ↪load_data()
print('input_X_train shape:', input_X_train.shape)
print(input_X_train.shape[0], 'train samples')
print(input_X_test.shape[0], 'test samples')

# convert to categorical
output_Y_train = utils.to_categorical(output_y_train, N_CLASSES)
output_Y_test = utils.to_categorical(output_y_test, N_CLASSES)

# float and normalization
input_X_train = input_X_train.astype('float32')
input_X_test = input_X_test.astype('float32')
input_X_train /= 255
input_X_test /= 255
```

Visualisation of two sample CIFAR-10 images

Here we will visualise two sample images from the dataset.

[ ]:
```
# visualisation of the numerical vector and 2D colour plot of the sample CIFAR␣
 ↪immage 2
Selected_Image = 2
image = input_X_train[Selected_Image]
print ("Sample input image: " + str(image))
```

```
plt.imshow(image)
plt.show()

Selected_Image = 3
image = input_X_train[Selected_Image]
print ("Sample input image: " + str(image))
plt.imshow(image)
plt.show()
```

Simple CNN model definition

This code defines a simple CNN network. The model will learn 32 convolutional filters, each of a 3 x 3 size. The output dimension is the same one of the input shape, with a 32 x 32 filters (default stride of 1 is used). The activation function ReLU will be used. The network then has a max-pooling layer with pool size 2 x 2, and a dropout at 25%.

The next level of depth has a dense layer with 512 units and ReLU activation, followed by a dropout at 50%. Finally, a softmax layer is used with 10 units/classes as output, i.e. one for each of the 10 classes of objects encoded with one-hot coding.

[ ]:
```
# network definition

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(IMG_ROWS, IMG_COLS,␣
 ↪IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))

print('CNN network definition.')
```

Model compilation

This compiles the CNN model, and then shows its summary.

[ ]:
```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer=OPTIM,␣
 ↪metrics=['accuracy'])

model.summary()
```

Training of the CNN

This line of code trains the model, saving the results in the history variable.

```
[ ]: # training/fitting of the DNN model

history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,␣
 ↪epochs=N_EPOCH, validation_split=VALIDATION_SPLIT,  verbose=VERBOSE)
```

Saving of the model and of the trained weights

This saves the model definition and the weights, after training.

```
[ ]: #save model in json format into a file
model_json = model.to_json()
open('cifar10_architecture.json', 'w').write(model_json)

#save the trained weights
model.save_weights('cifar10.weights.h5', overwrite=True)

print('Files saved for model definition and for weights.')
```

Analysis of the results

This code generates the test scores, so we can visualise and inspect the model's peformance.

It also plots the accuracy and loss values along the training timescale.

```
[ ]: #Testing
score = model.evaluate(input_X_test, output_Y_test, batch_size=BATCH_SIZE,␣
 ↪verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])

# list all data in history
print(history.history.keys())

# summarize history for accuracy
#plt.plot(mo)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```

```
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

1.1 A deeper CNN

To improve the performance of the network on the CIFAR-10 dataset, it is possible to use a deeper CNN, with a chain of multiple convolution and pooling layers. The following network will be used:

conv+conv+maxpool+dropout+conv+conv+maxpool

The final classification layers will use the standard:

dense+dropout+dense

All the layers will use the reLu function, except the final one with the Softmax function necessary for the categorical classification

[ ]:
```
# REUSE THE SAME INITIALISATION CODE AND THE TRAINING AND TEST DATA SET LOADING␣
↪OPERATION

N_EPOCH = 40 # bigger network will benefit from extra training epochs

# Complex DNN model definition
model = Sequential()

model.add(Conv2D(32, kernel_size=3, padding='same', input_shape=(IMG_ROWS,␣
↪IMG_COLS, IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(Conv2D(32, kernel_size=3, padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=3, padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))
```

```
# OPTIM = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
OPTIM = RMSprop()
model.compile(loss='categorical_crossentropy', optimizer=OPTIM,␣
↪metrics=['accuracy'], run_eagerly=True)

model.summary()
```

Training of the deeper CNN

Let's train (fit) this new model.

[ ]:
```
# training/fitting of the complex DNN model
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,␣
↪epochs=N_EPOCH, validation_split=VALIDATION_SPLIT,  verbose=VERBOSE)
```

Analysis of the Deeper CNN results

This generates the test scores and plots for the new, deeper DNN.

[ ]:
```
#Testing
score = model.evaluate(input_X_test, output_Y_test, batch_size=BATCH_SIZE,␣
↪verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])

# list all data in history
print(history.history.keys())

# summarize history for accuracy
#plt.plot(mo)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

1.2 Data Augmentation

To further improve the performance of the model, it is advisable to use a larger training set, to expose the network to more variations of the images. One way to achieve this, without having to collect new images from the real world, is to **augment** the existing images with multiple types of transformations of the dataset stimuli. This can include rotation of the image, rescaling, horizontal/vertical flip, zooming, channel shift, etc.

Below is an example of the code that augments the current datase.

[ ]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
#from keras.datasets import cifar10

#load dataset
#(input_X_train, output_y_train), (input_X_test, output_y_test) = cifar10.
↪load_data()

# augumenting
print("Augmenting training set images...")

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')

# rotation_range is a value in degrees (0 - 180) for randomly rotating pictures
# width_shift and height_shift are ranges for randomly translating pictures␣
↪vertically or horizontally
# zoom_range is for randomly zooming pictures
# horizontal_flip is for randomly flipping the images horizontally
# fill_mode fills in new pixels that can appear after a rotation or a shift
```

Training with augmented data

The function below used the dynamic generation of the augmented data during the training (just in time).

[ ]:
```
#fit the dataset
datagen.fit(input_X_train)



# train by fitting the model on batches with real-time data augmentation
history = model.fit_generator(datagen.flow(input_X_train, output_Y_train,␣
↪batch_size=BATCH_SIZE), steps_per_epoch=input_X_train.shape[0],␣
↪epochs=N_EPOCH, verbose=VERBOSE)
```

Analysis of the Data Augmented, Deeper CNN results

This generates the test scores and plots for the deeper DNN trained on the augmented data.

[ ]:
```
#Testing
score = model.evaluate(input_X_test, output_Y_test, batch_size=BATCH_SIZE,␣
 ↪verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])

# list all data in history
print(history.history.keys())

# summarize history for accuracy
#plt.plot(mo)
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Below is a commented different example of a data augmentation approach.

But we have carried out plenty of slow, long simulations for this class, and we can stop here.

[ ]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

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=0,  # randomly rotate images in the range (degrees, 0 to␣
 ↪180)
        width_shift_range=0.1,  # randomly shift images horizontally (fraction␣
 ↪of total width)
```

```
        height_shift_range=0.1,  # randomly shift images vertically (fraction␣
↪of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

datagen.fit(input_X_train)
```

1.3 Conclusions

Today we learned to train more complex DNNs, and to use data augmentation to further improve the network training and performance.

Copyright (c) 2022 Angelo Cangelosi, MIT License. Code and examples adapted from Gulli & Pal (2017) Deep Learning with Kera. Punkt Publishing. With support from Wenjie Huang.