TP_DeepLearning

May 23, 2020

Name: You ZUO

Date: 21/05/2020

1 Deep Learning Lab @ ENSIIE 2020

### Mohamed Ibn Khedher & Mounim A. El-Yacoubi

2 Introduction to Deep Learning

Classification of Handwritten Digits by a Convolutional Neural Network (CNN) This study is carried out MNIST, a dataset of handwritten numerals made of up of 60000 for training and 10000 for test. Each image has a size of 28x28 pixels, the gray level of each being between 0 and 255. The reference paper:

@inproceedingsLecun1998, Author = Y. Lecun and L. Bottou and Y. Bengio and P. Haffner, title = Gradient-based learning applied to document recognition, booktitle = Proceedings of the IEEE, year = 1998,

3 Demo:

This Lab contains 6 parts. The goal is to compelete the TO DO parts.

1. Data reading and splitting.
2. Data visualisation.
3. Define the model architecture
4. Model fiting
5. Model evaluation
6. New model architecture & evaluation

NB: In the Model evaluation part, Modify hyper parameters like **batch_size**, **epochs**, **validation_split**, etc., used so as to improve the results. Make an analysis and interpretation in light of the new results

3.1 Import the needed packages

[42]:
```
#### First, you should import libraires.
####

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
### for the color
import termcolor
todo=termcolor.colored('TO DO', color='red')
```

4 I) Data reading and splitting

This part consists of reading the MNIST dataset, split it into train and test sets and display the number of images per set.

[43]:
```
# I - Data reading & splitting

from keras.datasets import mnist

# 1) load data from MNIST Dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()


# Input image format
rows, cols, channels = 28,28,1

# 2) What does "x_train", "y_train", "x_test" and "y_test" present ?
# 3) Reshape "x_train" and "x_test" according to the input image format

x_train = x_train.reshape(60000,rows,cols,channels)
x_test = x_test.reshape(10000,rows,cols,channels)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

#4)Display the number of images in train and test sets

print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
y_train
```

```
60000 train samples
10000 test samples
```

```
[43]: array([5, 0, 4, …, 5, 6, 8], dtype=uint8)
```

From the part above, we have seperated our dataset into a training set with 60000 samples and a testing set with 10000 samples.

x_train represents the input of training dataset, which are the grey scales of 28*28 pixels of one handwriting; while the y_train are labels of each sample in training dataset, which refers to the real number it represents. x_test and y_test are the same except that they are for testing dataset.

5 II) Data visualisation

Which library is required to visualise images?
Complete the following commands to display the first 200 images from the training set.

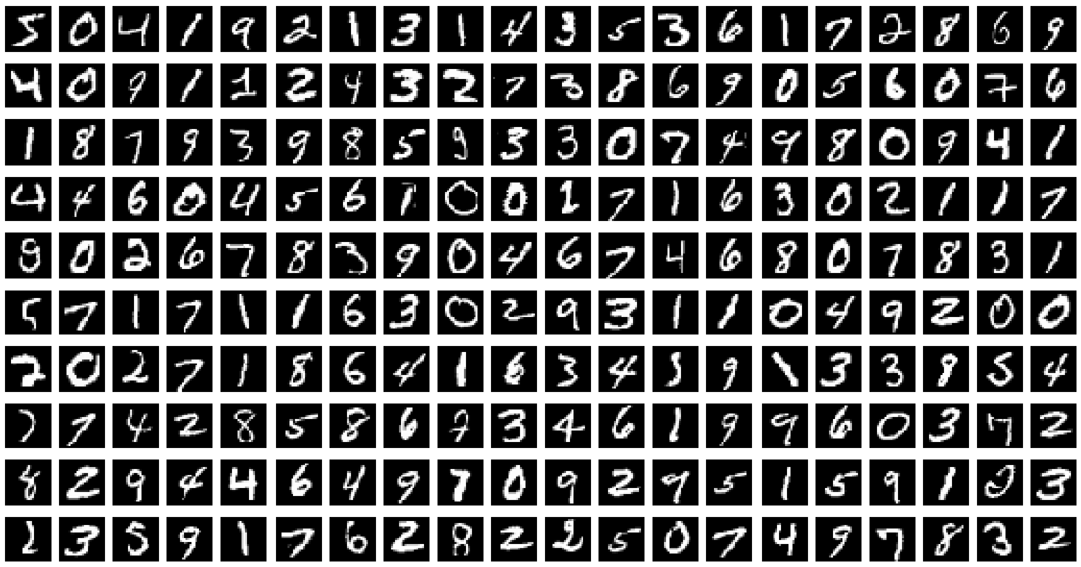
```
[44]: # II - Data visualisation

#1) Which library is required to  display images
import matplotlib.pyplot as plt

#2) Complete the script to display the first 200 images from the MNIST dataset

plt.figure(figsize=(7.195, 3.841), dpi=100)

for i in range(200):
    plt.subplot(10,20,i+1)
    plt.imshow(x_train[i].reshape([28,28]), cmap='gray')
    plt.axis('off')
```



In python, we use matplotlib.pyplot to draw all kinds of figures. From the above we have the pixel map for the first 200 images from the MNIST dataset.

6 III) Model architetcure

[45]:
```
#III - Model Architecture

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28,28,1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])


#1) In a seperate paper and for each model layer, specify the input/output␣
 ↪dimensions
```

```
#2) Verify your proposition using a predefined command

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_5 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               1179776
_________________________________________________________________
dropout_6 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

1. $(28, 28, 1) \rightarrow (26, 26, 32)$

   Our initial input dimension is 28*28*1, since we implemented the first convolutional layer with 32 filters of size 3*3 without padding, the pixels of cornners were "compressed" during the process of sliding the filter, so 28 became 28-(3-1). And each filter generated a new activation map, so from one channel and 32 filters we got 32 activation maps.

2. $(26, 26, 32) \rightarrow (24, 24, 64)$

   Same reason as above.

3. $(24, 24, 64) \rightarrow (12, 12, 64)$

   We used a 2*2 filter to do the maxing pooling with stride equals to 2, so we downsampled our data by 2. Moreover, the pooling process does not change the depth so 64 has not been modified.

4. $(12, 12, 64) \rightarrow (12, 12, 64)$

   The droplayer just dropped some samples of its input, so it did not change the dimension.

5. $(12, 12, 64) \rightarrow 9216$

   We wanted to do a full connection neuron network later so the implement of Flatten layer helped us to transform the three dimension data into a vector. $12 \times 12 \times 64 = 9216$

6. $9216 \rightarrow 128$

   $W_{9216 \times 128}$

7 IV) Fiting

[46]:
```
# IV - Fiting

# 1) convert labels to categorical type

num_classes=10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# 2) complete the following command to fit the Deep neural model.
# 3) select the hyperparameters values

model.fit(x_train, y_train,
          batch_size=256,
          epochs=3,
          verbose=1,
          validation_split=0.3)

# 4) What does each hyperparameter presents
```

```
Train on 42000 samples, validate on 18000 samples
Epoch 1/3
42000/42000 [==============================] - 93s 2ms/step - loss: 0.4280 -
acc: 0.8660 - val_loss: 0.1187 - val_acc: 0.9652
Epoch 2/3
42000/42000 [==============================] - 91s 2ms/step - loss: 0.1306 -
acc: 0.9605 - val_loss: 0.0901 - val_acc: 0.9714
Epoch 3/3
42000/42000 [==============================] - 79s 2ms/step - loss: 0.0941 -
acc: 0.9714 - val_loss: 0.0641 - val_acc: 0.9823
```

[46]:
```
<keras.callbacks.History at 0x631bb8190>
```

Here we have some hyperparameters which we could adjust:

1. batch_size: the number of training examples in one forward/backward pass. The higher the batch size, the more memory space we will need.
2. epochs: the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point.
3. validation_split: the rate of data we take from the training dataset to serve as a testing dataset.

Here I chose epochs = 3, so it went through three times the fitting process and finally picked up the one with the highest accuracy.

8 Model evaluation

[47]:
```
# V - Evaluate the model

# 1) complete the command to evaluate the model

score = model.evaluate(x_train, y_train, verbose=0)

# 2) complete the command to display model performance

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.04938362797953499
Test accuracy: 0.9853833333333334
```

NB: Modify hyper parameters like **batch_size**, **epochs**, **validation_split**, etc., used so as to improve the results. Make an analysis and interpretation in light of the new results

[66]:
```
# by modifying the batch_size:
batchSizes = [2^i for i in range(4,9)]
for batch in batchSizes:
    model.fit(x_train, y_train,
          batch_size=batch,
          epochs=1,
          verbose=0,
          validation_split=0.3)
    score = model.evaluate(x_train, y_train, verbose=0)
    print('batchsize = 2^%d: test loss is %f, test accuracy is␣
 ↪%f'%(batch,score[0],score[1]))
```

```
batchsize = 2^6: test loss is 0.046622, test accuracy is 0.988417
batchsize = 2^7: test loss is 0.040890, test accuracy is 0.988883
batchsize = 2^4: test loss is 0.041469, test accuracy is 0.988217
batchsize = 2^5: test loss is 0.035952, test accuracy is 0.989633
batchsize = 2^10: test loss is 0.036048, test accuracy is 0.989733
```

[68]:
```
# On fixing the batch_size = 2^10, we are going to modify the validation_split
import numpy as np
validation_splits = np.linspace(0.1,0.5,5)
for vs in validation_splits:
    model.fit(x_train, y_train,
          batch_size=2^10,
          epochs=1,
            verbose=0,
            validation_split=vs)
    score = model.evaluate(x_train, y_train, verbose=0)
    print('validation_split = %f: test loss is %f, test accuracy is␣
↪%f'%(vs,score[0],score[1]))
```

```
validation_split = 0.100000: test loss is 0.046145, test accuracy is 0.988350
validation_split = 0.200000: test loss is 0.036753, test accuracy is 0.989450
validation_split = 0.300000: test loss is 0.035987, test accuracy is 0.990150
validation_split = 0.400000: test loss is 0.033452, test accuracy is 0.991167
validation_split = 0.500000: test loss is 0.037538, test accuracy is 0.990150
```

From the results above, we can see that when batch_size = 2^{10} and validation_split = 0.4, we have the optimal accuracy 0.99. So we are going to keep it in the following process.

To analyze results, plot the confusion matrix using the following command

[71]:
```
from sklearn.metrics import confusion_matrix, classification_report
#### To analyze results, plot the confusion matrix using the following command
#Predict the test results
y_predict = model.predict_classes(x_test)
y_test_labels = y_test.argmax(1)
#confusion matrix and classification report
print("Confusion Matrix\n",confusion_matrix(y_test_labels,y_predict))
print("Classification Report\n",classification_report(y_test_labels,y_predict))
```

```
Confusion Matrix
 [[ 975    0    1    0    0    0    2    1    1    0]
 [   0 1131    0    3    0    0    0    1    0    0]
 [   3    3 1012    0    2    0    1    9    2    0]
 [   0    0    0 1000    0    4    0    5    1    0]
 [   1    0    1    0  973    0    3    0    2    2]
 [   2    2    0    5    1  878    2    1    0    1]
 [   6    3    1    1    2    4  939    0    2    0]
 [   1    6    5    2    0    0    0 1014    0    0]
 [   5    2    3    2    2    0    1    3  952    4]
 [   5    5    1    1    8    4    0    6    3  976]]
Classification Report
               precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       0.98      1.00      0.99      1135
           2       0.99      0.98      0.98      1032
           3       0.99      0.99      0.99      1010
           4       0.98      0.99      0.99       982
           5       0.99      0.98      0.99       892
           6       0.99      0.98      0.99       958
           7       0.97      0.99      0.98      1028
           8       0.99      0.98      0.98       974
           9       0.99      0.97      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.99      0.98      0.98     10000
weighted avg       0.99      0.98      0.98     10000
```

According to the comfusion matrix, we can see that we have actually an excellent result for our model. Almost all the digits have atteined a precision higher than 98%.

9 VI) New Model Architetcure & Evaluation

Now, It is time to create a new model and evaluate its performance on the MNIST dataset. We suggest to implement the following architecure.

9.1 Create model

Here we are going to build a very classical CNN but this time with the activation function changed to ELU, which is roughly a combination of ReLU and sigmoid. Convolutional layer with 30 feature maps of size 5×5. Pooling layer taking the max over 2*2 patches. Convolutional layer with 15 feature maps of size 3×3. Pooling layer taking the max over 2*2 patches. Dropout layer with a probability of 20Flatten layer. Fully connected layer with 128 neurons and rectifier activation. Fully connected layer with 50 neurons and rectifier activation. Output layer.

[73]:
```
from keras.optimizers import SGD
# VI - Create a new model and evaluate its performance
def New_model():
    # 1) create model
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3,␣
 ↪3),activation='elu',input_shape=(28,28,1)))
    model.add(Conv2D(32, (3, 3), activation='elu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, (3, 3), activation='elu'))
    model.add(Conv2D(64, (3, 3), activation='elu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(256, activation='elu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

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


# 2) Model evaluation
# build the model
model = New_model()
# Fit the model
model.fit(x_train, y_train, validation_split=0.3, epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(x_test, y_test, verbose=0)
print("New Model Error: %.2f%%" % (100-scores[1]*100))
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
Train on 42000 samples, validate on 18000 samples
Epoch 1/10
42000/42000 [==============================] - 93s 2ms/step - loss: 0.5410 -
acc: 0.8270 - val_loss: 0.1352 - val_acc: 0.9619
Epoch 2/10
42000/42000 [==============================] - 76s 2ms/step - loss: 0.1624 -
acc: 0.9502 - val_loss: 0.0915 - val_acc: 0.9732
Epoch 3/10
42000/42000 [==============================] - 76s 2ms/step - loss: 0.1202 -
acc: 0.9620 - val_loss: 0.0694 - val_acc: 0.9789
Epoch 4/10
42000/42000 [==============================] - 76s 2ms/step - loss: 0.0986 -
acc: 0.9690 - val_loss: 0.0629 - val_acc: 0.9805
Epoch 5/10
42000/42000 [==============================] - 77s 2ms/step - loss: 0.0876 -
acc: 0.9726 - val_loss: 0.0552 - val_acc: 0.9839
Epoch 6/10
42000/42000 [==============================] - 93s 2ms/step - loss: 0.0804 -
acc: 0.9742 - val_loss: 0.0513 - val_acc: 0.9842
Epoch 7/10
42000/42000 [==============================] - 88s 2ms/step - loss: 0.0700 -
acc: 0.9777 - val_loss: 0.0500 - val_acc: 0.9848
Epoch 8/10
42000/42000 [==============================] - 78s 2ms/step - loss: 0.0645 -
acc: 0.9798 - val_loss: 0.0480 - val_acc: 0.9851
Epoch 9/10
42000/42000 [==============================] - 75s 2ms/step - loss: 0.0613 -
acc: 0.9811 - val_loss: 0.0451 - val_acc: 0.9862
Epoch 10/10
42000/42000 [==============================] - 76s 2ms/step - loss: 0.0572 -
acc: 0.9815 - val_loss: 0.0426 - val_acc: 0.9867
```

```
New Model Error: 0.94%
Test loss: 0.027130232962354783
Test accuracy: 0.9906
```

We find that the score of the new model is also pretty high, the best val_acc is 0.9867 and the evaluation score is better than our first model.

[40]:
```
#### To analyze results, plot the confusion matrix using the following command
```

[74]:
```
from sklearn.metrics import confusion_matrix
#### To analyze results, plot the confusion matrix using the following command
#Predict the test results
y_predict = model.predict_classes(x_test)
y_test_labels = y_test.argmax(1)
#confusion matrix and classification report
print("Confusion Matrix\n",confusion_matrix(y_test_labels,y_predict))
```

```
Confusion Matrix
 [[ 976    0    1    0    0    0    1    1    1    0]
 [   0 1131    2    0    0    0    1    1    0    0]
 [   2    1 1026    0    1    0    0    2    0    0]
 [   1    0    2 1000    0    3    0    2    2    0]
 [   0    0    0    0  978    0    0    0    1    3]
 [   2    0    0    7    0  879    1    1    0    2]
 [   5    1    1    0    1    2  946    0    2    0]
 [   1    2    5    2    0    0    0 1015    1    2]
 [   3    0    2    0    1    1    0    1  965    1]
 [   3    2    1    1    4    2    0    4    2  990]]
```