Handwritten_digit_Tensor_Flow

January 9, 2020

1 Handwritten Digit Classifier Using Tensor Flow and Keras

Created by Giorgia Miniello: January 2020, Bari, Italy

1.1 Introduction

We perform the hadwritten digit recognition exercise we already did using a dataset from **MNIST**. The **MNIST** database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. This database is also widely used for Machine Learning (ML) training and testing and it is made of 28 x 28 pixel square images (784 pixels in total). For each dataset the training samples are 60,000 and the testing samples are 10,000. As you will see, this time we choose the high level API **Keras** as interface to **Tensor Flow** library to build a Neural Network. In this case the Neural Network (NN) is a simple Multi-Layer Perceptrons (MLP) which counts 784 input neurons. Two hidden layers are used with 512 neurons in hidden layer 1 and 256 neurons in hidden layer 2, followed by a fully connected layer of 10 neurons for taking the probabilities of all the class labels. We will see how making this model choice will increase the accuracy of this classification problem up to about 99%.

[1]:
```
#import libraries
# numpy is a library needed to perform vector/matrix operations, since to work␣
 ↪with images we will mconvert them in vectors or matrices
import numpy as np
# matplotlib is a 2D plotting library to visualize the behaviour of our network
import matplotlib.pyplot as plt
# Keras is an high-level API interfacing with Tensor Flow python library. We use␣
 ↪it to create NN models.
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Dropout
# using Keras to import datasets from MNIST
from keras.datasets import mnist
from keras.utils import np_utils
import pickle

# fix a random seed for reproducibility
np.random.seed(9)
```

```
Using TensorFlow backend.
```

1.2 Parameters in NNs

In a neural network, there are some parameters that could be tuned to obtain good results. Some experience is needed from users to pick up the "right" ones. The parameters that we set in this exercise are:

num_epoch:the number of iterations needed for the network to minimize the loss function.In this way the model learns the weights.

num_classes:the total number of class labels or classes involved in the classification problem. In this handwritten digit classification exercise the classes are 10 (the numbers from 0 to 9).

batch_size:the number of images given to the model at a particular instance.

train_size:the number of images used to train the model.

test_size:the number of images used to test the model.

v_length:the dimension of *flattened* input image size i.e. if input image size is 28x28 pixels, then v_length = 784.

[2]:
```
# user inputs
num_epoch = 30
num_classes = 10
batch_size = 128
train_size = 60000
test_size = 10000
v_length = 784
```

1.3 Loading and preparing the MNIST dataset

In order to load the MNIST dataset you just need to load the **mnist.load_data()** function in Keras. By default, it returns two tuples (train data and train label) hold in one tuple, and test data and test label in another tuple.

The first time this function is run, the MNIST dataset is automatically downloaded to a local folder ~/.keras/datasets/mnist.pkl.gz which is 14.6 MB in size.

After loading the dataset it must be pre-processed, i.e. it has to be modified in a way the model can manage and understand it. The main step of this dataset pre-processing are:

Reshaping:in Deep Learning, the raw pixel intensities of images are provided as inputs to the NNs. If you check the shape of original data and label, you see that each image has the dimension of 28x28 pixels. If we flatten it, we will get 28x28=784 pixel intensities. A NumPy's reshape function is used for the purpose.

Data type: this step comes after the reshaping. The pixel intensities are changed to float32 datatype so that a uniform representation can be obtained. Since the images are made by grayscale image pixels, the intensity of each pixel is an integer in the range [0-255]. It can be converted to floating point representations using .astype function provided by NumPy.

Normalize:Each floating point value must be normalized in the range (0-1) to improve computational efficiency and to follow the standards.

[3]:
```
#########if you have internet....

# split the mnist data into training and testing samples
(trainData, trainLabels), (testData, testLabels) = mnist.load_data()


#FOR THE TUTORIAL
# writing a file with pickle and dumping the (trainData,trainLabels)␣
↪(testData,testLabels) tuples
with open('mnist.pickle', 'wb') as f:
    pickle.dump([(trainData, trainLabels), (testData, testLabels)], f, pickle.
↪HIGHEST_PROTOCOL)


######### if you don't (starting from here for the tutorial)
# opening the file containing the (trainData,trainLabels) (testData,testLabels)␣
↪tuples and reading them with pickle
with open('mnist.pickle', 'rb') as f:
    (trainDatap, trainLabelsp), (testDatap, testLabelsp)=pickle.load(f)

#print(len(trainDatap))
#print(len(trainData))


#print(trainDatap[10])
#print(trainData[10])

print("TRAIN DATA SHAPE: {}".format(trainData.shape))
print("TEST DATA SHAPE: {}".format(testData.shape))
print("TRAIN DATA SAMPLES: {}".format(trainData.shape[0]))
print("TEST DATA SAMPLES: {}".format(testData.shape[0]))
```

```
TRAIN DATA SHAPE: (60000, 28, 28)
TEST DATA SHAPE: (10000, 28, 28)
TRAIN DATA SAMPLES: 60000
TEST DATA SAMPLES: 10000
```

After loading the dataset it must be pre-processed, i.e. it has to be modified in a way the model can manage and understand it. The main step of this dataset pre-processing are:

Reshaping:in Deep Learning, the raw pixel intensities of images are provided as inputs to the NNs. If you check the shape of original data and label, you see that each image has the dimension of 28x28 pixels. If we flatten it, we will get 28x28=784 pixel intensities. A NumPy's reshape function is used for the purpose.

Data type: this step comes after the reshaping. The pixel intensities are changed to float32 datatype so that a uniform representation can be obtained. Since the images are made by grayscale image pixels, the intensity of each pixel is an integer in the range [0-255]. It can be converted to floating point representations using .astype function provided by NumPy.

Normalize:Each floating point value must be normalized in the range (0-1) to improve computational efficiency and to follow the standards.

[4]:
```
# reshape the dataset
trainData = trainData.reshape(train_size, v_length)
testData = testData.reshape(test_size, v_length)
trainData = trainData.astype("float32")
testData = testData.astype("float32")
trainData /= 255
testData /= 255

print("Printing some info:")
print("trainData shape: {}".format(trainData.shape))
print("testdata shape: {}".format(testData.shape))
print("Number of train samples: {}".format(trainData.shape[0]))
print("Number of test samples: {}".format(testData.shape[0]))
```

```
Printing some info:
trainData shape: (60000, 784)
testdata shape: (10000, 784)
Number of train samples: 60000
Number of test samples: 10000
```

1.4 One Hot Encoding

In digital circuits and ML, one-hot is a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0). Since this is a multi-label classification problem, we need to represent these 10 numeric digits into a binary form. This representation is an one-hot encoding.

It simply means that if we have a digit (e.g. 3), then we build table with a number of columns equal to the number classes (10 in this case as we have 10 digits) and we put zero in all the cells except 3 in which we put one. In Keras, we can simply use the **np_utils.to_categorical** function to transform numeric value to one-hot encoded representation. This function takes labels and number of class labels as input.

[5]:
```
# convert class vectors to binary class matrices
mTrainLabels = np_utils.to_categorical(trainLabels, num_classes)
mTestLabels = np_utils.to_categorical(testLabels, num_classes)
```

1.5 Creating the Model

A simple Multi-Layer Perceptron (MLP) is used as NN model with 784 input neurons.

Two hidden layers are used with 512 neurons in hidden layer 1 and 256 neurons in hidden layer 2, followed by a fully connected layer of 10 neurons for taking the probabilities of all the class labels.

ReLU is used as the activation function for hidden layers and **softmax** is used as the activation function for output layer. **Dropout** is a regularization function that, at each training iteration,

drops random neurons from the network with a probability p (typically from 25% to 50%, here 50%).

[7]:
```
# create the model
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation("relu"))
model.add(Dropout(0.5))
model.add(Dense(256))
model.add(Activation("relu"))
model.add(Dropout(0.2))
model.add(Dense(num_classes))
model.add(Activation("softmax"))

# summarize the model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               131328
_________________________________________________________________
activation_5 (Activation)    (None, 256)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                2570
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0
=================================================================
Total params: 535,818
Trainable params: 535,818
Non-trainable params: 0
_________________________________________________________________
```

1.6 Compiling the model

After creating the model, it must be compiled for optimization and learning. The **categorical_crossentropy** function is used as the loss function (as this is a multi-label classification problem), **adam** (gradient descent algorithm) is used as optimizer and accuracy as our performance metric.

[9]:
```
# compile the model
model.compile(loss="categorical_crossentropy", optimizer="adam",␣
↪metrics=["accuracy"])
```

1.7 Fit the Model

After compiling the model, it must be fit using **model.fit** function. This function requires some arguments created above. trainData and mTrainLabels go into 1st and 2nd position, followed by the validation_data. Then we have nb_epoch, batch_size and verbose. Verbose is just for debugging purposes. It can be observed during the fitting that the more you increase the number of the iterations (epochs) the greater is the accuracy.

[10]:
```
# fit the model
history = model.fit(trainData, mTrainLabels, validation_data=(testData,␣
↪mTestLabels), batch_size=batch_size, epochs=num_epoch, verbose=2)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from
tensorflow.python.ops.math_ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Use tf.cast instead.
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
 - 13s - loss: 0.3249 - acc: 0.9016 - val_loss: 0.1209 - val_acc: 0.9639
Epoch 2/30
 - 12s - loss: 0.1528 - acc: 0.9539 - val_loss: 0.0927 - val_acc: 0.9712
Epoch 3/30
 - 12s - loss: 0.1154 - acc: 0.9637 - val_loss: 0.0747 - val_acc: 0.9741
Epoch 4/30
 - 12s - loss: 0.0992 - acc: 0.9690 - val_loss: 0.0732 - val_acc: 0.9774
Epoch 5/30
 - 12s - loss: 0.0886 - acc: 0.9718 - val_loss: 0.0700 - val_acc: 0.9770
Epoch 6/30
 - 12s - loss: 0.0759 - acc: 0.9759 - val_loss: 0.0668 - val_acc: 0.9797
Epoch 7/30
 - 12s - loss: 0.0723 - acc: 0.9773 - val_loss: 0.0606 - val_acc: 0.9811
Epoch 8/30
 - 12s - loss: 0.0637 - acc: 0.9791 - val_loss: 0.0573 - val_acc: 0.9833
Epoch 9/30
 - 12s - loss: 0.0630 - acc: 0.9791 - val_loss: 0.0622 - val_acc: 0.9809
Epoch 10/30
 - 12s - loss: 0.0554 - acc: 0.9822 - val_loss: 0.0574 - val_acc: 0.9818
Epoch 11/30
 - 12s - loss: 0.0563 - acc: 0.9816 - val_loss: 0.0617 - val_acc: 0.9826
Epoch 12/30
 - 12s - loss: 0.0506 - acc: 0.9839 - val_loss: 0.0546 - val_acc: 0.9829
```

```
Epoch 13/30
 - 12s - loss: 0.0493 - acc: 0.9831 - val_loss: 0.0591 - val_acc: 0.9829
Epoch 14/30
 - 12s - loss: 0.0492 - acc: 0.9840 - val_loss: 0.0577 - val_acc: 0.9835
Epoch 15/30
 - 12s - loss: 0.0459 - acc: 0.9846 - val_loss: 0.0567 - val_acc: 0.9830
Epoch 16/30
 - 12s - loss: 0.0418 - acc: 0.9863 - val_loss: 0.0558 - val_acc: 0.9827
Epoch 17/30
 - 12s - loss: 0.0399 - acc: 0.9869 - val_loss: 0.0616 - val_acc: 0.9829
Epoch 18/30
 - 12s - loss: 0.0412 - acc: 0.9859 - val_loss: 0.0595 - val_acc: 0.9825
Epoch 19/30
 - 12s - loss: 0.0389 - acc: 0.9873 - val_loss: 0.0517 - val_acc: 0.9856
Epoch 20/30
 - 12s - loss: 0.0360 - acc: 0.9880 - val_loss: 0.0574 - val_acc: 0.9851
Epoch 21/30
 - 12s - loss: 0.0339 - acc: 0.9889 - val_loss: 0.0582 - val_acc: 0.9852
Epoch 22/30
 - 12s - loss: 0.0335 - acc: 0.9882 - val_loss: 0.0563 - val_acc: 0.9858
Epoch 23/30
 - 12s - loss: 0.0354 - acc: 0.9876 - val_loss: 0.0572 - val_acc: 0.9846
Epoch 24/30
 - 12s - loss: 0.0325 - acc: 0.9894 - val_loss: 0.0582 - val_acc: 0.9842
Epoch 25/30
 - 12s - loss: 0.0322 - acc: 0.9892 - val_loss: 0.0578 - val_acc: 0.9847
Epoch 26/30
 - 12s - loss: 0.0330 - acc: 0.9891 - val_loss: 0.0729 - val_acc: 0.9820
Epoch 27/30
 - 12s - loss: 0.0322 - acc: 0.9894 - val_loss: 0.0640 - val_acc: 0.9844
Epoch 28/30
 - 12s - loss: 0.0318 - acc: 0.9893 - val_loss: 0.0573 - val_acc: 0.9865
Epoch 29/30
 - 12s - loss: 0.0313 - acc: 0.9899 - val_loss: 0.0581 - val_acc: 0.9847
Epoch 30/30
 - 12s - loss: 0.0304 - acc: 0.9902 - val_loss: 0.0600 - val_acc: 0.9864
```

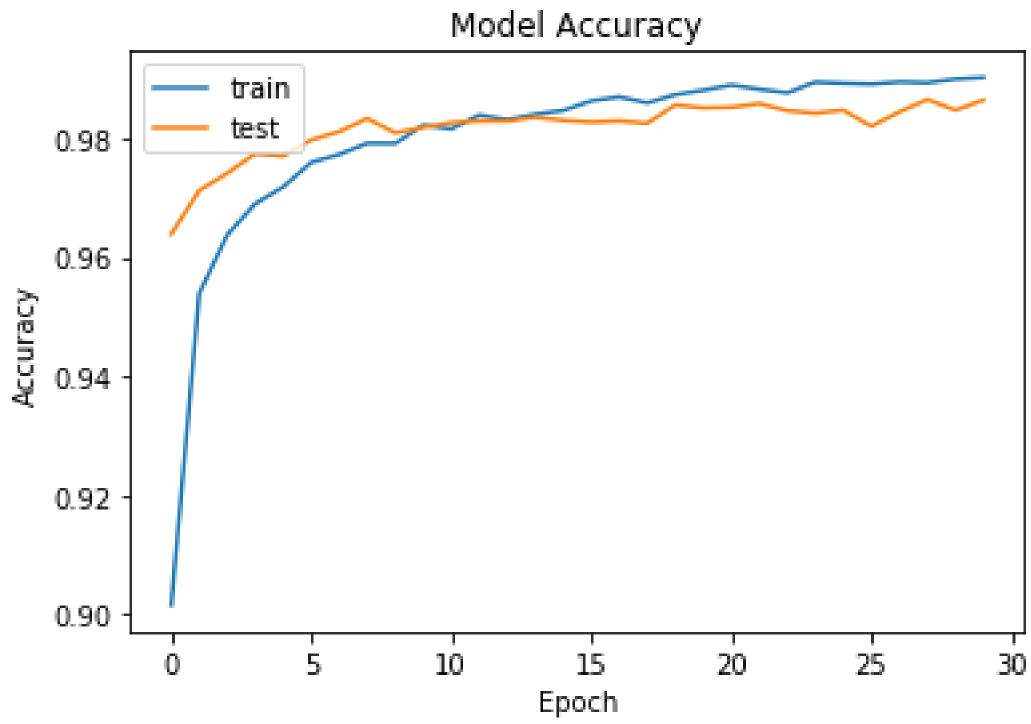
1.8 Evaluation of the Model

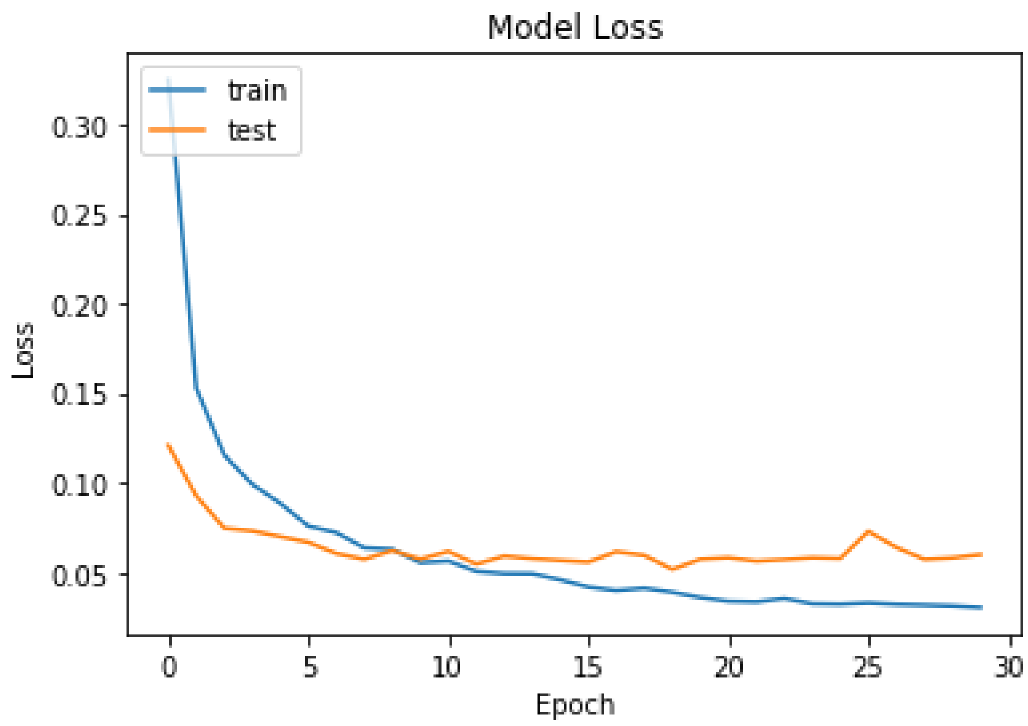
After fitting the model, it can be evaluated test its perfomance on the unseen test data. Using **model.evaluate** Kera function in Keras. As usal, using test data as input for the model just trained predictions can be made and comparing the predictions with testing data labels the accuracy can be computed.we can give test data and test labels to the model and make predictions. Furthermore, matplotlib can be used to visualize how our model reacts at different epochs on both training and testing data. In the two plots the Model Accuracy and the Model Loss vs the epochs are presented. It's worth to remind that in a ML model the **loss function** measures how far the prediction is away from the true values represented by the labels.

[11]:

```
# print the history keys
print(history.history.keys())

# evaluate the model
scores = model.evaluate(testData, mTestLabels, verbose=0)

# history plot for accuracy
plt.plot(history.history["acc"])
plt.plot(history.history["val_acc"])
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(["train", "test"], loc="upper left")
plt.show()

# history plot for accuracy
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title("Model Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(["train", "test"], loc="upper left")
plt.show()

# print the results
print("Test score : {}".format(scores[0]))
print("Test accuracy: {}".format(scores[1]))
```

```
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```


Model Accuracy
train
test
Accuracy
0.98
0.96
0.94
0.92
0.90
0
5
10
15
20
25
30
Epoch


Model Loss
train
test
Loss
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20
25
30
Epoch

```
Test score : 0.05998402663118104
Test accuracy: 0.9864
```

1.9 Testing the Model

In order to visualize our model performance, 25 random images were chosen from the testing dataset. The predictions made using the **model.predict_classes** function are printed in red on the top of each of them.

[12]:
```
import matplotlib.pyplot as plt

# picking some test images from the test data
test_images = testData[0:25]
#print(len(test_images))

# reshape the test images to standard 28x28 MNIST format
test_images = test_images.reshape(test_images.shape[0], 28, 28)
print("Test images shape: {}".format(test_images.shape))

pred_list=[]
plt.figure(figsize=(10,10))
# loop over each of the test images
for i, test_image in enumerate(test_images):
    # grab a copy of test image for viewing
    org_image = test_image

    # reshape the test image to [1x784] format so that our model can understand␣
↪it
    test_image = test_image.reshape(1,784)

    # make prediction on test image using our trained model
    prediction = model.predict_classes(test_image, verbose=0)
    pred_list.append(prediction)

#print (pred_list)

    # display the prediction and image
    print("The prediction for the digit ", i+1 ," is {}".format(prediction[0]))

    plt.subplot(5,5,i+1)
    plt.axis("OFF")
    plt.title("Prediction: %i" % prediction[0], fontsize=9, fontweight='bold',␣
↪color='r')
    plt.imshow(255-org_image, cmap=plt.get_cmap('gray'))

plt.show()
```

```
Test images shape: (25, 28, 28)
The prediction for the digit  1  is 7
The prediction for the digit  2  is 2
The prediction for the digit  3  is 1
The prediction for the digit  4  is 0
The prediction for the digit  5  is 4
The prediction for the digit  6  is 1
The prediction for the digit  7  is 4
The prediction for the digit  8  is 9
The prediction for the digit  9  is 5
The prediction for the digit  10  is 9
The prediction for the digit  11  is 0
The prediction for the digit  12  is 6
The prediction for the digit  13  is 9
The prediction for the digit  14  is 0
The prediction for the digit  15  is 1
The prediction for the digit  16  is 5
The prediction for the digit  17  is 9
The prediction for the digit  18  is 7
The prediction for the digit  19  is 3
The prediction for the digit  20  is 4
The prediction for the digit  21  is 9
The prediction for the digit  22  is 6
The prediction for the digit  23  is 6
The prediction for the digit  24  is 5
The prediction for the digit  25  is 4
```

1.10 Conclusions

A simple MLP was built as to solve a handwritten digit recognition exercise using MNIST dataset. Note that we haven't used Convolutional Neural Networks (CNN) yet. The results were provided in terms of accuracy, which is about 99% with a Dropout of 50%, and model loss, which is about 6%, as a function of epochs. As a further improvement Convolutional Neural Networks (CNN) can be chosen keeping the others parameters fixed to evaluate any possible increase in accuracy.