nb_ch02_02a

June 14, 2025

1 MNIST digit classification with a fully connected network (fcNN)

Goal: In this notebook you will see how to use a fully connected networks (fcNN) in a classification task for images.

Usage: The idea of the notebook is that you try to understand the provided code by running it, checking the output and playing with it by slightly changing the code and rerunning it.

Dataset: You work with the MNIST dataset. We have 60'000 28x28 pixel greyscale images of digits and want to classify them into the right label (0-9).

Content: * load the MNIST data * transform the labels into the one hot encoding * visualize samples of the data * flatten the 2D images into a 1D vector * use keras to train a fcNN and look at the perfomance on new unseen test data * use different activation functions and more hidden layers

open in colab

Install correct TF version (colab only)

Imports In the next two cells, we load all the required libraries and functions.

[109]:
```
try: #If running in colab
    import google.colab
    IN_COLAB = True
    %tensorflow_version 2.x
except:
    IN_COLAB = False
```

[110]:
```
import tensorflow as tf
if (not tf.__version__.startswith('2')): #Checking if tf 2.0 is installed
    print('Please install tensorflow 2.0 to run this notebook')
print('Tensorflow version: ',tf.__version__, ' running in colab?: ', IN_COLAB)
```

```
Tensorflow version:  2.19.0  running in colab?:  False
```

[111]:
```
# load required libraries:
import numpy as np
import matplotlib.pyplot as plt
```

```
%matplotlib inline
plt.style.use('default')
from sklearn.metrics import confusion_matrix

import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Convolution2D, MaxPooling2D, Flatten␣
 ↪, Activation
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
```

Loading and preparing the MNIST data and transfering the labels into the one hot encoding Here we load the MNIST dataset form keras. The 8-bit greyscale images have values form 0 to 255, we divide all values with 255 so that the values are in a range between 0 and 1. In additon we transform the true labels, which are the numbers from 0 to 9 (the digit on the image) into the one hot encoding. We do this to make use of linear algebra in the calculation of the crossentropy loss.

The one hot encoding transforms the labels into a vector with the same length as we have labels (in our case 10). The resulting vector in the one hot encodig is zero everywhere except for the position of the true label, there it is 1. Let's look at some examples to make it more clear:

0 becomes [1,0,0,0,0,0,0,0,0,0]
1 becomes [0,1,0,0,0,0,0,0,0,0]
2 becomes [0,0,1,0,0,0,0,0,0,0]
…
9 becomes [0,0,0,0,0,0,0,0,0,1]

Listing 2.3 Loading the MNIST data

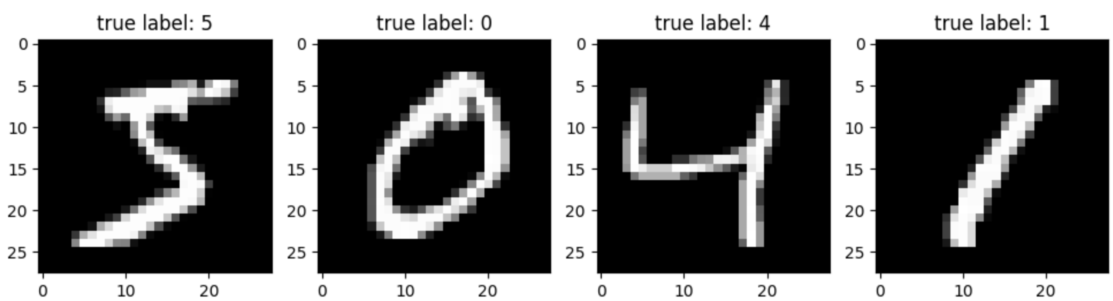
[112]:
```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# separate x_train in X_train and X_val, same for y_train
X_train=x_train[0:50000] / 255 #divide by 255 so that they are in range 0 to 1
Y_train=keras.utils.to_categorical(y_train[0:50000],10) # one-hot encoding

X_val=x_train[50000:60000] / 255
Y_val=keras.utils.to_categorical(y_train[50000:60000],10)

X_test=x_test / 255
Y_test=keras.utils.to_categorical(y_test,10)

del x_train, y_train, x_test, y_test

X_train=np.reshape(X_train, (X_train.shape[0],28,28,1))
X_val=np.reshape(X_val, (X_val.shape[0],28,28,1))
X_test=np.reshape(X_test, (X_test.shape[0],28,28,1))
```

```
print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_val.shape)
print(Y_test.shape)
```

```
(50000, 28, 28, 1)
(10000, 28, 28, 1)
(10000, 28, 28, 1)
(50000, 10)
(10000, 10)
(10000, 10)
```

Let's visualize the first 4 mnist images. It is very easy to recognise the true label of the digits.

[113]:
```
# visualize the 4 first mnist images before shuffling the pixels
plt.figure(figsize=(12,12))
for i in range(0,4):
    plt.subplot(1,4,(i+1))
    plt.imshow((X_train[i,:,:,0]),cmap="gray")
    plt.title('true label: '+str(np.argmax(Y_train,axis=1)[i]))
    #plt.axis('off')
```



1.1 fcNN as classification model for MNIST data

Now we want to train a fcNN to classify the MNIST data. * we use a fcNN with 2 hidden layers and use the sigmoid activation function * train it on train data and check the performance on the test data

Flatten the the images into vectors Because we will use fcNN our input cannot be matrices or tensors. We need to flatten our inuput into a 1d vector. We do this in the next cell with reshap and look at the resulting shape of the flattened data.

[114]:
```
# prepare data for fcNN - we need a vector as input
```

```
X_train_flat = X_train.reshape([X_train.shape[0], 784])
X_val_flat = X_val.reshape([X_val.shape[0], 784])
X_test_flat = X_test.reshape([X_test.shape[0], 784])

# check the shape
print(X_train_flat.shape)
print(Y_train.shape)
print(X_val_flat.shape)
print(Y_val.shape)
```

```
(50000, 784)
(50000, 10)
(10000, 784)
(10000, 10)
```

1.1.1 Train the fcNN

Here we define the nework, we use two hidden layers with 100 and 50 nodes. In the output we predict the probability for the 10 digits with the softmax actication function, in the hidden layers we use the sigmoid activation function and our loss is the categroical crossentropy loss.

Listing 2.4 Definition of an fcNN for the MNIST data

[115]:
```
# define fcNN with 2 hidden layers
model = Sequential()

model.add(Dense(100, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dense(50))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))

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

```
c:\Users\nicol\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\layers\core\dense.py:93: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[116]:
```
# summarize model along with number of model weights
model.summary()
```

```
Model: "sequential_18"
```

Layer (type)	Output Shape	Param #
dense_49 (Dense)	(None, 100)	78,500
activation_46 (Activation)	(None, 100)	0
dense_50 (Dense)	(None, 100)	10,100
activation_47 (Activation)	(None, 100)	0
dense_51 (Dense)	(None, 50)	5,050
activation_48 (Activation)	(None, 50)	0
dense_52 (Dense)	(None, 10)	510
activation_49 (Activation)	(None, 10)	0

Total params: 94,160 (367.81 KB)

Trainable params: 94,160 (367.81 KB)

Non-trainable params: 0 (0.00 B)

[117]:
```
# train the model
history=model.fit(X_train_flat, Y_train,
                  batch_size=128,
                  epochs=10,
                  verbose=2,
                  validation_data=(X_val_flat, Y_val)
                 )
```

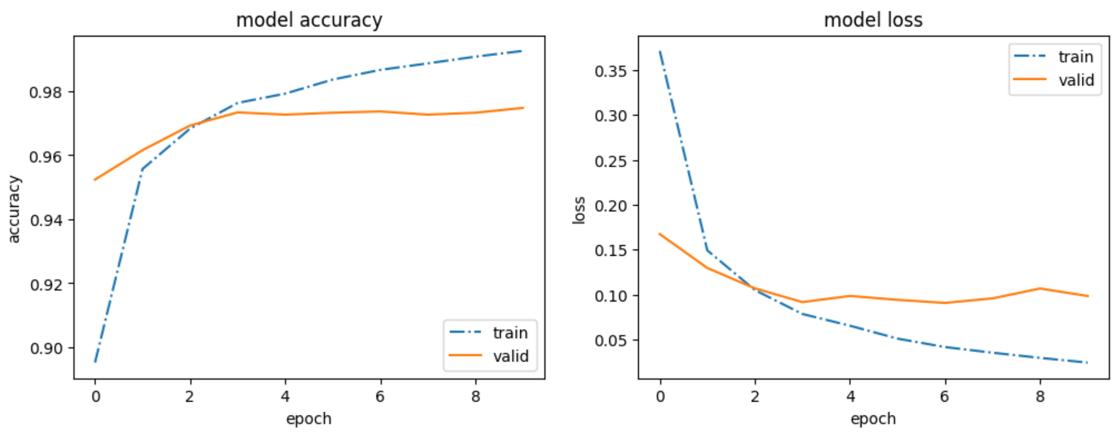
```
Epoch 1/10
391/391 - 2s - 5ms/step - accuracy: 0.8952 - loss: 0.3712 - val_accuracy: 0.9524
- val_loss: 0.1674
Epoch 2/10
391/391 - 1s - 2ms/step - accuracy: 0.9557 - loss: 0.1492 - val_accuracy: 0.9616
- val_loss: 0.1296
Epoch 3/10
391/391 - 1s - 2ms/step - accuracy: 0.9683 - loss: 0.1050 - val_accuracy: 0.9693
- val_loss: 0.1070
```

```
Epoch 4/10
391/391 - 1s - 2ms/step - accuracy: 0.9764 - loss: 0.0784 - val_accuracy: 0.9734
- val_loss: 0.0916
Epoch 5/10
391/391 - 1s - 2ms/step - accuracy: 0.9793 - loss: 0.0654 - val_accuracy: 0.9727
- val_loss: 0.0985
Epoch 6/10
391/391 - 1s - 2ms/step - accuracy: 0.9837 - loss: 0.0510 - val_accuracy: 0.9733
- val_loss: 0.0942
Epoch 7/10
391/391 - 1s - 2ms/step - accuracy: 0.9867 - loss: 0.0416 - val_accuracy: 0.9737
- val_loss: 0.0907
Epoch 8/10
391/391 - 1s - 2ms/step - accuracy: 0.9887 - loss: 0.0353 - val_accuracy: 0.9727
- val_loss: 0.0958
Epoch 9/10
391/391 - 1s - 2ms/step - accuracy: 0.9908 - loss: 0.0295 - val_accuracy: 0.9733
- val_loss: 0.1068
Epoch 10/10
391/391 - 1s - 2ms/step - accuracy: 0.9926 - loss: 0.0243 - val_accuracy: 0.9748
- val_loss: 0.0984
```

In the next cell we plot the accuray and loss of the train and validation vs the number of train eprochs to see how the development

[118]:
```
# plot the development of the accuracy and loss during training
plt.figure(figsize=(12,4))
plt.subplot(1,2,(1))
plt.plot(history.history['accuracy'],linestyle='-.')
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='lower right')
plt.subplot(1,2,(2))
plt.plot(history.history['loss'],linestyle='-.')
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='upper right')
```

[118]:
```
<matplotlib.legend.Legend at 0x1808d6d49b0>
```



Prediction on the original test set after training on original data Now, let's use the fcNN that was trained on the flattened MNIST data to predict new unseen data (our testdata). We determine the confusion matrix and the accuracy on the testdata to evaluate the classification performance.

[119]:
```
pred=model.predict(X_test_flat)
print(confusion_matrix(np.argmax(Y_test,axis=1),np.argmax(pred,axis=1)))
acc_fc_orig = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("Acc_fc_orig_flat = " , acc_fc_orig)
```

```
313/313                  0s 747us/step
[[ 965    1    1    2    1    0    5    1    3    1]
 [   0 1126    1    1    0    0    1    2    4    0]
 [   5    8  983   10    3    0    2   13    8    0]
 [   0    1    2  994    2    2    0    5    3    1]
 [   1    2    3    0  952    0    3    6    1   14]
 [   3    0    0   19    1  845    5    4   10    5]
 [   4    3    1    2    4    4  936    0    4    0]
 [   0    3    3    5    0    0    0 1009    1    7]
 [   2    2    0   20    6    2    0    2  937    3]
 [   3    4    0    8    6    1    1    9    0  977]]
Acc_fc_orig_flat =  0.9724
```

We get an accuray of around 97% on the test data!

Play the deep learning game and stack more layers and change the activation function from sigmoid to relu

Exercise: Try to improve the fcNN by adding more hidden layers and/or changing the activation function from "sigmoid" to "relu". What do you observe? can you improve the performace on the testset?

1.1.2 Lets train a Convolutional Neural Network (CNN)!

- Use the previous code to complete this part of the task
- Change our fcNN to a CNN, it should look something like this:

- We don't need to flatten our input this time!
- Train the CNN and use the same parameters as before (10 epochs, batchsize 128, etc.)
- Visualize the loss/accuracy like before, but include the loss of our fcNN so we can see a difference
- Use the testset to create a confusion matrix and compare it with our fcNN

1.1.3 Optional

`Run your network on the Edge TPU by following the instructions in this`

Notebook.