AMIOT_TZDAKA_exercise_3_assignment

December 18, 2020

1 Lab session 1

This is the work of Eidan TZDAKA and Alexandre Amiot

1.1 Exercise 3 : Recognition of hand-written characters from the scikit-learn database using a CNN

Start from the code you developed in exercise 2. Modify it to replace the perceptron by a two-stage CNN. The first stage will contain a unique filter layer and a maxpooling. The second stage will contain a flatten layer and one predicting dense layer.

[2]:
```
"""Import necessary libraries"""
import matplotlib.pyplot as plt
import numpy as np
```

[3]:
```
# Load the digits dataset from scikit-learn
# Import datasets and performance metrics
from sklearn import datasets
digits = datasets.load_digits()
```

The data we are interested in are 8x8 images of digits. A CNN can process images directly without having them to be vectorized. Work directly with the **digits.images** from sklearn.datasets .

[4]:
```
# The data we are interested in is made of 8x8 images of digits, let's
# have a look at the first 10 images, stored in the `images` attribute of the
# dataset.  If we were working from image files, we could load them using
# matplotlib.pyplot.imread.  Note that each image must have the same size. For␣
 ↪these
# images, we know which digit they represent: it is given in the 'target' of
# the dataset.
images_and_labels = list(zip(digits.images, digits.target))
for index, (image, label) in enumerate(images_and_labels[:10]):
    plt.subplot(2, 5, index + 1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Training: %i' % label)
```

What is the size of the images? How many images do we have?

We have 1797 images of size 8x8

[5]:
```
print (digits.images.shape)
print (digits.target[1:10])
```

```
(1797, 8, 8)
[1 2 3 4 5 6 7 8 9]
```

Again, convert the target into the matrix 'y' in one-hot format. This will allow the predictors output one at a time using the soft-max activation. What is the shape of the 'y' matrix. Use the keras function utils.to_categorical().

[6]:
```
from tensorflow import keras
n_classes = len(np.unique(digits.target))
y = keras.utils.to_categorical(digits.target, n_classes)
print (y.shape)
```

```
(1797, 10)
```

When working with images, keras CNN implicitly assume colour images.

The colours come in separate channels. These channels appear as the last dimension of the array.

Our hand-written characters are grey-tone images, that is one-channel images. We need to add this dimension to the array to make keras understand that we only have one channel.

Print the shape of your array X. You should have a 4-D array with the last dimension equal to 1.

[7]:
```
X = np.expand_dims(digits.images,axis=-1)
print (X.shape)
```

```
(1797, 8, 8, 1)
```

1.2 Prepare the train and test dataset.

1. split dataset into test and train dataset.
2. split further the train dataset into train and validation part Use the **keras.model_selection** function **train_test_split()** to separate a dataset. You can choose the percentage of the split, or leave the defaut value.

[8]:
```
digits.data.shape
```

[8]:
```
(1797, 64)
```

[9]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train)
```

[10]:
```
X_train.shape
```

[10]:
```
(1010, 8, 8, 1)
```

1.3 create a sequential model

1. The model will the keras sequential model containing a usual two-stage CNN. Instantiate the model using **model=Sequential()**. Add a first convolution layer **Conv2D()**. Specify the number of filters (start with 5), the filter size (say 3x3), the input shape, and the type of activation - choose 'relu'. Add a **MaxPooling2D()** layer to reduce the size. Add a layer to vectorize the data : **Flatten()**. Finally add a **Dense()** layer from **keras.layers**. Specify the number of classes to output. What kind of activation will you use?

We will use the relu activation function so that it doesnt activate all the neurons at the same time, they are not activated if the output of the linear transformation is less than 0 and thanks to that it is more computational efficient.

2. Compile the model. Specify the loss. Choose an optimizer and the evaluation metrics to use. Use the class function **compile()** of the sequential model you have created.

[21]:
```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
# create the sequential model
model = Sequential()
model.add(Conv2D(5,kernel_size=3, input_shape=(8,8,1),activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=None, padding="valid",␣
 ↪data_format=None))
model.add(Flatten())
model.add(Dense(n_classes, input_shape=(28,28,1),activation='softmax'))


# compile the model. Use the categorical crossentropy, the adam optimizer and␣
 ↪accuracy
# as the evaluation metric
```

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

1.4 train the model

You can leave the same parameters as before.

[22]:
```
# train the model
history = model.fit(X_train, y_train, batch_size=128, epochs=100, verbose=1,␣
↪validation_data=(X_val,y_val))
```

```
Epoch 1/100
8/8 [==============================] - 1s 32ms/step - loss: 12.5485 - accuracy:
0.1020 - val_loss: 11.1362 - val_accuracy: 0.1128
Epoch 2/100
8/8 [==============================] - 0s 8ms/step - loss: 9.6634 - accuracy:
0.1146 - val_loss: 9.2436 - val_accuracy: 0.0950
Epoch 3/100
8/8 [==============================] - 0s 7ms/step - loss: 7.9350 - accuracy:
0.0926 - val_loss: 7.7247 - val_accuracy: 0.0682
Epoch 4/100
8/8 [==============================] - 0s 7ms/step - loss: 6.9333 - accuracy:
0.0806 - val_loss: 6.5110 - val_accuracy: 0.0682
Epoch 5/100
8/8 [==============================] - 0s 7ms/step - loss: 5.9480 - accuracy:
0.0721 - val_loss: 5.5323 - val_accuracy: 0.0890
Epoch 6/100
8/8 [==============================] - 0s 7ms/step - loss: 5.0212 - accuracy:
0.0818 - val_loss: 4.7244 - val_accuracy: 0.1157
Epoch 7/100
8/8 [==============================] - 0s 8ms/step - loss: 4.3627 - accuracy:
0.0980 - val_loss: 4.1001 - val_accuracy: 0.1246
Epoch 8/100
8/8 [==============================] - 0s 7ms/step - loss: 3.7323 - accuracy:
0.1362 - val_loss: 3.5145 - val_accuracy: 0.1306
Epoch 9/100
8/8 [==============================] - 0s 7ms/step - loss: 3.2331 - accuracy:
0.1387 - val_loss: 3.0135 - val_accuracy: 0.1573
Epoch 10/100
8/8 [==============================] - 0s 8ms/step - loss: 2.8066 - accuracy:
0.1575 - val_loss: 2.6591 - val_accuracy: 0.1780
Epoch 11/100
8/8 [==============================] - 0s 7ms/step - loss: 2.6232 - accuracy:
0.1754 - val_loss: 2.4391 - val_accuracy: 0.2136
Epoch 12/100
8/8 [==============================] - 0s 8ms/step - loss: 2.4012 - accuracy:
0.2176 - val_loss: 2.2837 - val_accuracy: 0.2493
Epoch 13/100
```

```
8/8 [==============================] - 0s 7ms/step - loss: 2.2838 - accuracy:
0.2557 - val_loss: 2.1600 - val_accuracy: 0.2819
Epoch 14/100
8/8 [==============================] - 0s 7ms/step - loss: 2.1601 - accuracy:
0.2912 - val_loss: 2.0447 - val_accuracy: 0.3056
Epoch 15/100
8/8 [==============================] - 0s 9ms/step - loss: 2.0254 - accuracy:
0.3198 - val_loss: 1.9372 - val_accuracy: 0.3501
Epoch 16/100
8/8 [==============================] - 0s 8ms/step - loss: 1.9540 - accuracy:
0.3499 - val_loss: 1.8348 - val_accuracy: 0.3828
Epoch 17/100
8/8 [==============================] - 0s 7ms/step - loss: 1.8465 - accuracy:
0.3882 - val_loss: 1.7376 - val_accuracy: 0.4154
Epoch 18/100
8/8 [==============================] - 0s 7ms/step - loss: 1.7501 - accuracy:
0.4100 - val_loss: 1.6458 - val_accuracy: 0.4540
Epoch 19/100
8/8 [==============================] - 0s 7ms/step - loss: 1.6432 - accuracy:
0.4583 - val_loss: 1.5613 - val_accuracy: 0.4777
Epoch 20/100
8/8 [==============================] - 0s 7ms/step - loss: 1.5445 - accuracy:
0.4896 - val_loss: 1.4845 - val_accuracy: 0.5104
Epoch 21/100
8/8 [==============================] - 0s 7ms/step - loss: 1.4772 - accuracy:
0.5220 - val_loss: 1.4137 - val_accuracy: 0.5312
Epoch 22/100
8/8 [==============================] - 0s 7ms/step - loss: 1.3910 - accuracy:
0.5440 - val_loss: 1.3440 - val_accuracy: 0.5490
Epoch 23/100
8/8 [==============================] - 0s 8ms/step - loss: 1.3382 - accuracy:
0.5648 - val_loss: 1.2845 - val_accuracy: 0.5786
Epoch 24/100
8/8 [==============================] - 0s 9ms/step - loss: 1.2901 - accuracy:
0.5945 - val_loss: 1.2300 - val_accuracy: 0.5935
Epoch 25/100
8/8 [==============================] - 0s 8ms/step - loss: 1.2116 - accuracy:
0.6219 - val_loss: 1.1782 - val_accuracy: 0.6231
Epoch 26/100
8/8 [==============================] - 0s 8ms/step - loss: 1.1813 - accuracy:
0.6234 - val_loss: 1.1279 - val_accuracy: 0.6469
Epoch 27/100
8/8 [==============================] - 0s 9ms/step - loss: 1.1512 - accuracy:
0.6615 - val_loss: 1.0853 - val_accuracy: 0.6617
Epoch 28/100
8/8 [==============================] - 0s 8ms/step - loss: 1.0435 - accuracy:
0.6775 - val_loss: 1.0413 - val_accuracy: 0.6795
Epoch 29/100
```

```
8/8 [==============================] - 0s 7ms/step - loss: 1.0274 - accuracy:
0.6855 - val_loss: 1.0057 - val_accuracy: 0.6884
Epoch 30/100
8/8 [==============================] - 0s 8ms/step - loss: 1.0057 - accuracy:
0.7029 - val_loss: 0.9737 - val_accuracy: 0.7062
Epoch 31/100
8/8 [==============================] - 0s 7ms/step - loss: 0.9457 - accuracy:
0.7157 - val_loss: 0.9371 - val_accuracy: 0.7240
Epoch 32/100
8/8 [==============================] - 0s 8ms/step - loss: 0.8871 - accuracy:
0.7346 - val_loss: 0.9064 - val_accuracy: 0.7300
Epoch 33/100
8/8 [==============================] - 0s 8ms/step - loss: 0.8835 - accuracy:
0.7234 - val_loss: 0.8815 - val_accuracy: 0.7359
Epoch 34/100
8/8 [==============================] - 0s 7ms/step - loss: 0.8962 - accuracy:
0.7348 - val_loss: 0.8576 - val_accuracy: 0.7418
Epoch 35/100
8/8 [==============================] - 0s 7ms/step - loss: 0.8109 - accuracy:
0.7753 - val_loss: 0.8281 - val_accuracy: 0.7507
Epoch 36/100
8/8 [==============================] - 0s 7ms/step - loss: 0.7630 - accuracy:
0.7948 - val_loss: 0.8037 - val_accuracy: 0.7567
Epoch 37/100
8/8 [==============================] - 0s 23ms/step - loss: 0.7399 - accuracy:
0.7978 - val_loss: 0.7855 - val_accuracy: 0.7596
Epoch 38/100
8/8 [==============================] - 0s 8ms/step - loss: 0.7947 - accuracy:
0.7749 - val_loss: 0.7663 - val_accuracy: 0.7626
Epoch 39/100
8/8 [==============================] - 0s 7ms/step - loss: 0.7324 - accuracy:
0.8005 - val_loss: 0.7449 - val_accuracy: 0.7715
Epoch 40/100
8/8 [==============================] - 0s 7ms/step - loss: 0.7261 - accuracy:
0.7976 - val_loss: 0.7280 - val_accuracy: 0.7685
Epoch 41/100
8/8 [==============================] - 0s 7ms/step - loss: 0.6779 - accuracy:
0.8212 - val_loss: 0.7111 - val_accuracy: 0.7804
Epoch 42/100
8/8 [==============================] - 0s 8ms/step - loss: 0.6314 - accuracy:
0.8369 - val_loss: 0.6948 - val_accuracy: 0.7834
Epoch 43/100
8/8 [==============================] - 0s 7ms/step - loss: 0.6606 - accuracy:
0.8136 - val_loss: 0.6770 - val_accuracy: 0.7893
Epoch 44/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5990 - accuracy:
0.8376 - val_loss: 0.6640 - val_accuracy: 0.7893
Epoch 45/100
```

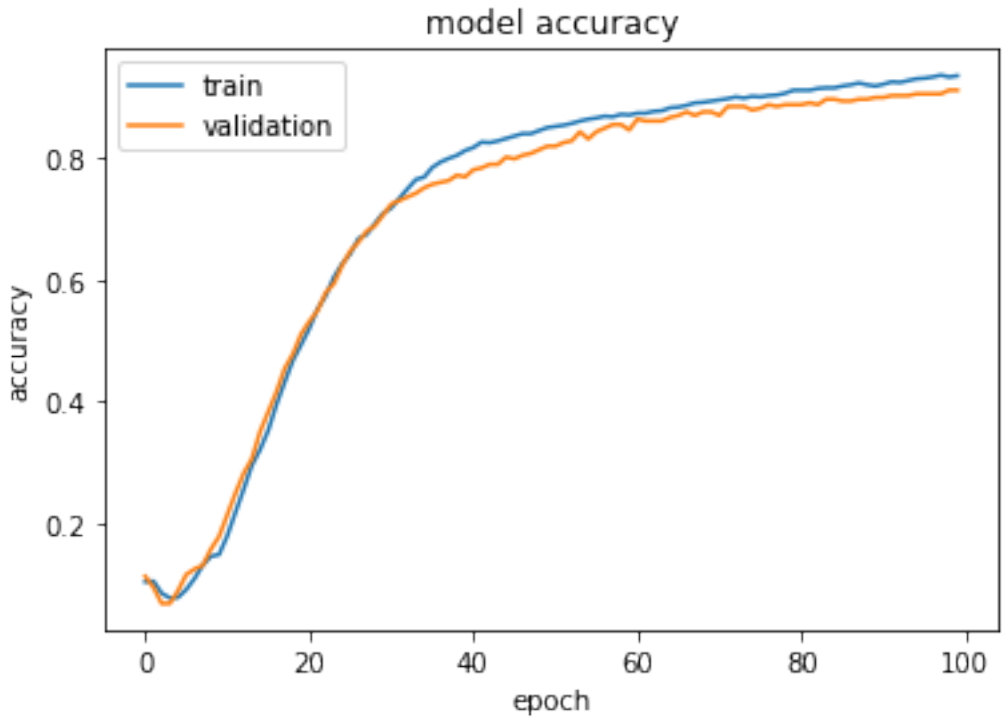
```
8/8 [==============================] - 0s 8ms/step - loss: 0.6024 - accuracy:
0.8311 - val_loss: 0.6487 - val_accuracy: 0.8012
Epoch 46/100
8/8 [==============================] - 0s 9ms/step - loss: 0.6008 - accuracy:
0.8358 - val_loss: 0.6374 - val_accuracy: 0.7982
Epoch 47/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5488 - accuracy:
0.8451 - val_loss: 0.6244 - val_accuracy: 0.8042
Epoch 48/100
8/8 [==============================] - 0s 8ms/step - loss: 0.5783 - accuracy:
0.8372 - val_loss: 0.6115 - val_accuracy: 0.8071
Epoch 49/100
8/8 [==============================] - 0s 8ms/step - loss: 0.5707 - accuracy:
0.8409 - val_loss: 0.6002 - val_accuracy: 0.8131
Epoch 50/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5528 - accuracy:
0.8336 - val_loss: 0.5906 - val_accuracy: 0.8190
Epoch 51/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5317 - accuracy:
0.8576 - val_loss: 0.5787 - val_accuracy: 0.8190
Epoch 52/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5344 - accuracy:
0.8462 - val_loss: 0.5694 - val_accuracy: 0.8249
Epoch 53/100
8/8 [==============================] - 0s 7ms/step - loss: 0.5124 - accuracy:
0.8601 - val_loss: 0.5597 - val_accuracy: 0.8279
Epoch 54/100
8/8 [==============================] - 0s 8ms/step - loss: 0.5087 - accuracy:
0.8550 - val_loss: 0.5484 - val_accuracy: 0.8427
Epoch 55/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4643 - accuracy:
0.8702 - val_loss: 0.5404 - val_accuracy: 0.8309
Epoch 56/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4668 - accuracy:
0.8642 - val_loss: 0.5327 - val_accuracy: 0.8427
Epoch 57/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4376 - accuracy:
0.8778 - val_loss: 0.5300 - val_accuracy: 0.8487
Epoch 58/100
8/8 [==============================] - 0s 8ms/step - loss: 0.5029 - accuracy:
0.8566 - val_loss: 0.5152 - val_accuracy: 0.8546
Epoch 59/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4841 - accuracy:
0.8577 - val_loss: 0.5072 - val_accuracy: 0.8546
Epoch 60/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4523 - accuracy:
0.8609 - val_loss: 0.5006 - val_accuracy: 0.8457
Epoch 61/100
```

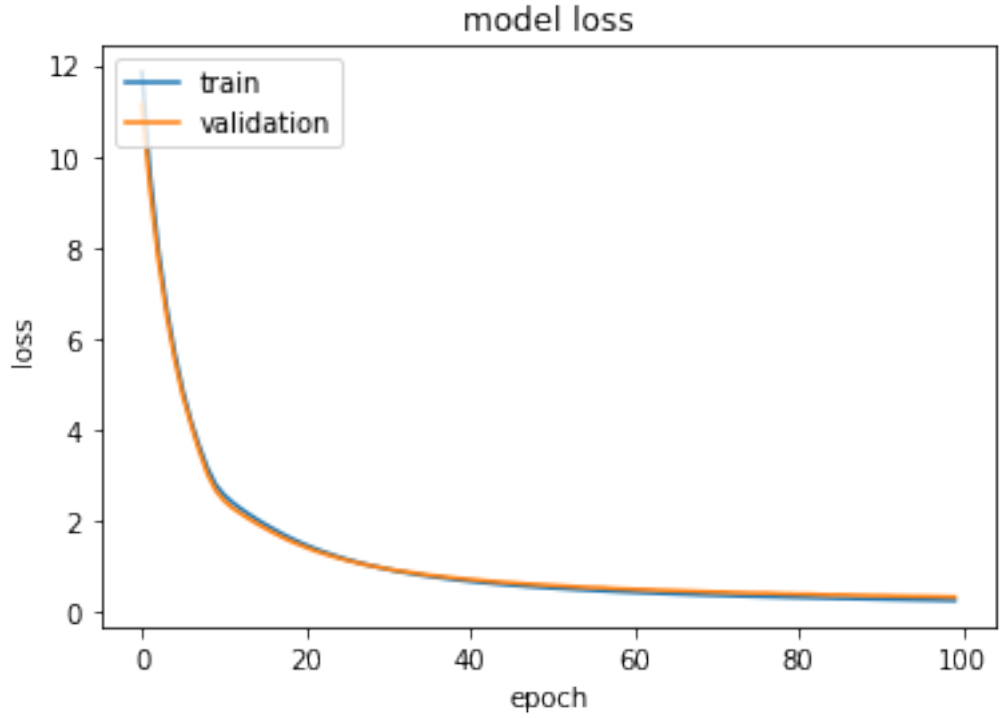
```
8/8 [==============================] - 0s 8ms/step - loss: 0.4104 - accuracy:
0.8781 - val_loss: 0.4931 - val_accuracy: 0.8635
Epoch 62/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4394 - accuracy:
0.8723 - val_loss: 0.4892 - val_accuracy: 0.8605
Epoch 63/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4200 - accuracy:
0.8711 - val_loss: 0.4824 - val_accuracy: 0.8605
Epoch 64/100
8/8 [==============================] - 0s 9ms/step - loss: 0.4069 - accuracy:
0.8833 - val_loss: 0.4736 - val_accuracy: 0.8605
Epoch 65/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4388 - accuracy:
0.8705 - val_loss: 0.4677 - val_accuracy: 0.8665
Epoch 66/100
8/8 [==============================] - 0s 9ms/step - loss: 0.4105 - accuracy:
0.8751 - val_loss: 0.4629 - val_accuracy: 0.8694
Epoch 67/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3992 - accuracy:
0.8802 - val_loss: 0.4569 - val_accuracy: 0.8754
Epoch 68/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3648 - accuracy:
0.8942 - val_loss: 0.4512 - val_accuracy: 0.8694
Epoch 69/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3681 - accuracy:
0.8942 - val_loss: 0.4443 - val_accuracy: 0.8754
Epoch 70/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3612 - accuracy:
0.8992 - val_loss: 0.4393 - val_accuracy: 0.8754
Epoch 71/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3677 - accuracy:
0.8867 - val_loss: 0.4340 - val_accuracy: 0.8694
Epoch 72/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3560 - accuracy:
0.8941 - val_loss: 0.4309 - val_accuracy: 0.8843
Epoch 73/100
8/8 [==============================] - 0s 10ms/step - loss: 0.3743 - accuracy:
0.8980 - val_loss: 0.4231 - val_accuracy: 0.8843
Epoch 74/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3510 - accuracy:
0.8976 - val_loss: 0.4176 - val_accuracy: 0.8843
Epoch 75/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3575 - accuracy:
0.8913 - val_loss: 0.4137 - val_accuracy: 0.8783
Epoch 76/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3561 - accuracy:
0.8888 - val_loss: 0.4088 - val_accuracy: 0.8813
Epoch 77/100
```

```
8/8 [==============================] - 0s 9ms/step - loss: 0.3207 - accuracy:
0.9057 - val_loss: 0.4103 - val_accuracy: 0.8872
Epoch 78/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3500 - accuracy:
0.8977 - val_loss: 0.4036 - val_accuracy: 0.8843
Epoch 79/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3314 - accuracy:
0.9012 - val_loss: 0.3956 - val_accuracy: 0.8872
Epoch 80/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3043 - accuracy:
0.9156 - val_loss: 0.3921 - val_accuracy: 0.8872
Epoch 81/100
8/8 [==============================] - 0s 8ms/step - loss: 0.2784 - accuracy:
0.9209 - val_loss: 0.3882 - val_accuracy: 0.8872
Epoch 82/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3264 - accuracy:
0.9032 - val_loss: 0.3872 - val_accuracy: 0.8902
Epoch 83/100
8/8 [==============================] - 0s 8ms/step - loss: 0.3294 - accuracy:
0.9024 - val_loss: 0.3819 - val_accuracy: 0.8872
Epoch 84/100
8/8 [==============================] - 0s 9ms/step - loss: 0.2755 - accuracy:
0.9227 - val_loss: 0.3753 - val_accuracy: 0.8961
Epoch 85/100
8/8 [==============================] - 0s 9ms/step - loss: 0.2734 - accuracy:
0.9278 - val_loss: 0.3706 - val_accuracy: 0.8961
Epoch 86/100
8/8 [==============================] - 0s 8ms/step - loss: 0.2742 - accuracy:
0.9297 - val_loss: 0.3725 - val_accuracy: 0.8932
Epoch 87/100
8/8 [==============================] - 0s 9ms/step - loss: 0.2733 - accuracy:
0.9265 - val_loss: 0.3680 - val_accuracy: 0.8932
Epoch 88/100
8/8 [==============================] - 0s 8ms/step - loss: 0.2849 - accuracy:
0.9229 - val_loss: 0.3624 - val_accuracy: 0.8961
Epoch 89/100
8/8 [==============================] - 0s 9ms/step - loss: 0.2922 - accuracy:
0.9199 - val_loss: 0.3593 - val_accuracy: 0.8961
Epoch 90/100
8/8 [==============================] - 0s 9ms/step - loss: 0.2459 - accuracy:
0.9312 - val_loss: 0.3530 - val_accuracy: 0.8991
Epoch 91/100
8/8 [==============================] - 0s 8ms/step - loss: 0.2895 - accuracy:
0.9137 - val_loss: 0.3539 - val_accuracy: 0.8991
Epoch 92/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2768 - accuracy:
0.9207 - val_loss: 0.3501 - val_accuracy: 0.9021
Epoch 93/100
```

```
8/8 [==============================] - 0s 7ms/step - loss: 0.2728 - accuracy:
0.9215 - val_loss: 0.3434 - val_accuracy: 0.9021
Epoch 94/100
8/8 [==============================] - 0s 10ms/step - loss: 0.2729 - accuracy:
0.9236 - val_loss: 0.3430 - val_accuracy: 0.9021
Epoch 95/100
8/8 [==============================] - 0s 22ms/step - loss: 0.2405 - accuracy:
0.9370 - val_loss: 0.3375 - val_accuracy: 0.9050
Epoch 96/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2624 - accuracy:
0.9346 - val_loss: 0.3384 - val_accuracy: 0.9050
Epoch 97/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2616 - accuracy:
0.9311 - val_loss: 0.3343 - val_accuracy: 0.9050
Epoch 98/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2577 - accuracy:
0.9372 - val_loss: 0.3335 - val_accuracy: 0.9050
Epoch 99/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2419 - accuracy:
0.9370 - val_loss: 0.3278 - val_accuracy: 0.9110
Epoch 100/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2465 - accuracy:
0.9324 - val_loss: 0.3251 - val_accuracy: 0.9110
```

[23]:
```python
import matplotlib.pyplot as plt
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], 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', 'validation'], loc='upper left')
plt.show()
```


model accuracy
train
validation
accuracy
0.2
0.4
0.6
0.8
0
20
40
60
80
100
epoch


model loss
train
validation
loss
0
2
4
6
8
10
12
0
20
40
60
80
100
epoch

1.5 Analyze the results.

1. Print a summary of the model - use the model's class function **summary()**. How many parameters does the model have? Is this model bigger or smaller than the previous one?
2. What was the accuracy you obtained at the end of the training? Do you use the training or validation accuracy to answer?
3. If you run the model several times, do you always obtain the same accuracy?

1, If we have only 5 filters, the model only have 510 parameters which is arround the same size as the previous one (610), this is a small model, if we increase the number of filters, like 64, we now have 6410 parameters, this model is bigger than the ones that we previously used. But in a general manner, it is still a small model.

2, At the end of the training we have an accuracy of 0.9324 on the training data and an accuracy of 0.9110 which represents the accuracy of the model. The results are not as good as the previous model. If can increase the accuracy a lot by adding filters, with 64 filters, we have a training accuracy of 1 and a validation accuracy of 0.9885.

3, No, every time we run it, the accuracies changes, for example, if we rerun it we can have an accuracy of 0.9145 instead of 0.9110.

1.6 Print the model summary.

[24]:
```
# print the model summary
print (model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 6, 6, 5)           50
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 5)           0
_________________________________________________________________
flatten_2 (Flatten)          (None, 45)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                460
=================================================================
Total params: 510
Trainable params: 510
Non-trainable params: 0
_________________________________________________________________
None
```

1.7 print the confusion matrix obtained on the test dataset.

Use the **confusion_matrix()** function provided in **sklearn.metrics**.

[25]:
```
Y_test_pred = model.predict(X_test)
from sklearn.metrics import confusion_matrix
```

```
CM = confusion_matrix (np.argmax(y_test,axis=1), np.argmax(Y_test_pred,axis=1))

print (CM)
```

```
[[54  0  0  0  0  0  0  0  1  0]
 [ 0 50  1  0  3  1  0  0  1  2]
 [ 0  1 37  0  0  0  0  1  1  0]
 [ 0  0  1 34  0  0  0  1  1  2]
 [ 0  0  0  0 46  0  0  0  0  0]
 [ 1  0  0  1  0 47  0  0  0  0]
 [ 1  1  0  0  0  1 39  0  2  0]
 [ 0  0  0  0  0  3  0 36  0  1]
 [ 0  3  0  0  0  0  0  4 30  0]
 [ 0  0  0  3  1  0  0  0  0 38]]
```

Normalize the confusion matrix to show graphically the prediction probability map.

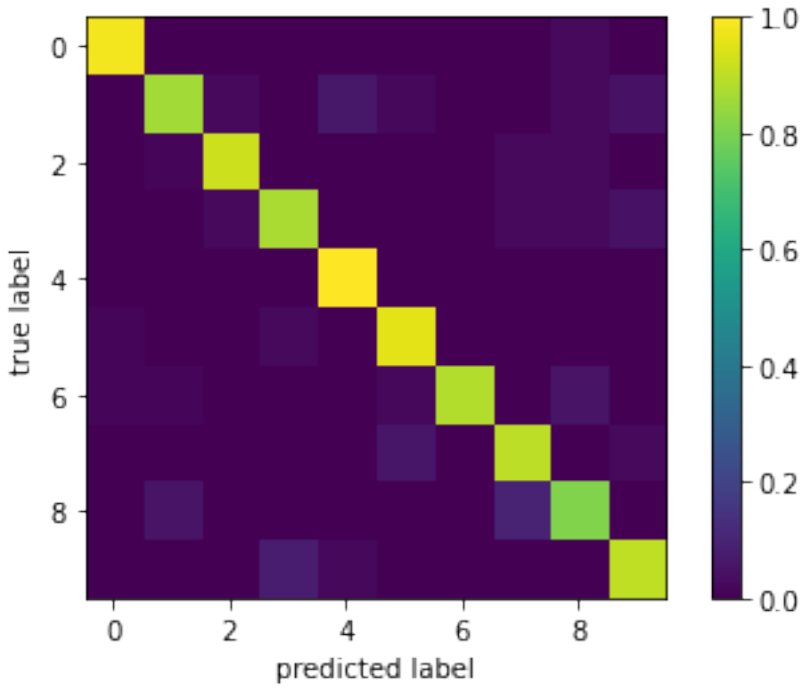
Hint: The confusion matrix needs to be normalized by the counts of each class. Print the counts of elements in each class in the test dataset. Use the numpy function **unique()** to count the elements in y_test.

[26]:
```
_,count = np.unique(np.argmax(y_test,axis=1),return_counts=True)
print (count)
CM = CM/count
CM = np.round(1000*CM)/1000

plt.figure('confusion matrix')
ax = plt.imshow(CM); plt.colorbar()
plt.ylabel('true label')
plt.xlabel('predicted label')
```

```
[55 58 40 39 46 49 44 40 37 42]
```

[26]:
```
Text(0.5, 0, 'predicted label')
```



Evaluate the model on the test dataset. What is the accuracy on the test dataset? Use the model's class function **evaluate()**.

For 5 filters: The loss and accuracy on the test dataset : 0.290320, 0.913333

For 64 filters: The loss and accuracy on the test dataset : 0.049495, 0.984444

[27]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test)


print ("The loss and accuracy on the test dataset : %f, %f"%(test_loss,␣
↪test_accuracy))
```

```
15/15 [==============================] - 0s 2ms/step - loss: 0.2903 - accuracy:
0.9133
The loss and accuracy on the test dataset : 0.290320, 0.913333
```

1.8 Conclusions:

Conclude your report by answering the following questions:

1. Did your training converge?
2. Was the training successful?
3. Try to improve the model. Try adding the number of filters in the filter layer. Does the accuracy improve?
4. Could you obtain better results than with the perceptron?

1, Yes the training converged toward a good accuracy, by plotting the model loss, we can also say that we have a very small overfitting but we discard it because it doesn't impact the model (too small).

2, We can say that the training was successful, the test score are good, we have an accuracy of 0.9133 and a model loss of 0.2903.

3, A good way to improve our model is by adding filters, here, we tried with 64 filters, which greatly improved our results. Now we have a 0.984444 accuracy for the test set with a model loss of only 0.049495.

4, Yes, the results can be way better than with the perceptron, but only if we set a good number of filters.

[ ]:
```
from google.colab import drive
base_working_dir = '/content/drive/My Drive/Colab Notebooks/MBE DL course'
drive.mount('/content/drive')

!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-generic-recommended

!jupyter nbconvert --to pdf  "drive/My Drive/Colab Notebooks/MBE DL course/
↪exercise_3_assignment.ipynb" #>/dev/null 2>/dev/null
```