AMIOT_TZDAKA_exercise_2_assignment

December 18, 2020

1 Lab session 1

This is the work of Eidan Tzdaka and Alexandre Amiot

1.1 Exercise 2 : Recognition of hand-written characters from the scikit-learn database using a perceptron

In this exercise you are to implement a simple hand-written digit classifier based on a perceptron (no convolutional networks).

This assignment contains examples of results you should reproduce with your code.

Import the **MNIST** dataset of the digits from scikit-learn

from sklearn import datasets

digits = datasets.load_digits()

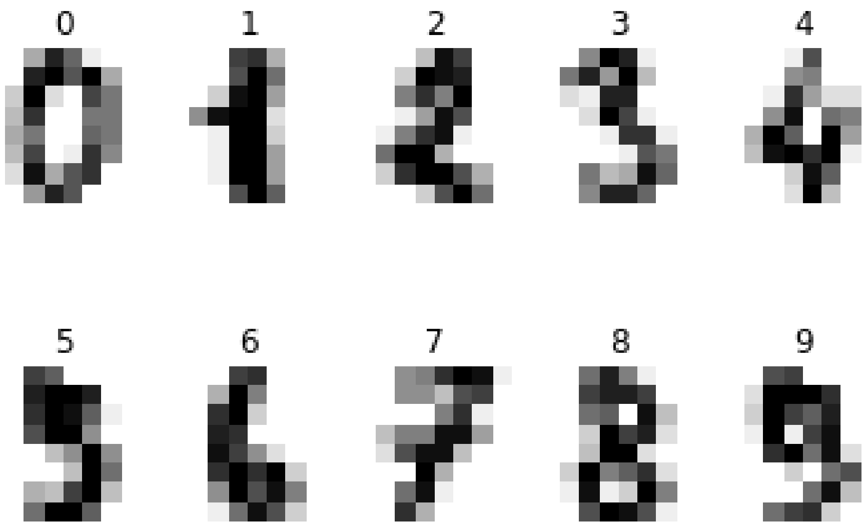
[1]:
```
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np

# Import the dataset of the digits from scikit-learn

from sklearn import datasets
digits = datasets.load_digits()
```

The data we are interested in are 8x8 images of digits. They are stored in the `images` attribute of the dataset.

Have a look at the data and the classes:

1. For every image, we know which digit it represents. It is the 'target' of the dataset.
2. Show the first 10 images and their target.

[2]:
```
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)
  plt.title('%i' % label)
```



To apply a perceptron classifier on this data, we will not use 8x8 images but the vectorized version stored in digits.data.

1. How many images do we have in the dataset?
2. How many classes do we have in the dataset?
3. What is the size of the vector representing the flattened version of the images?

1, The digit dataset is made of 1797 8x8 images

2, We have 10 classes in this dataset

3, The size of the vector representing the flattened version of the image is 10.

[3]:
```
print (digits.data.shape)
```

```
(1797, 64)
```

How many classes do we have?

[4]:
```
n_classes = len(np.unique(digits.target))
print ("We have a total of %d classes."%n_classes)
```

```
We have a total of 10 classes.
```

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()**.

[5]:
```
from tensorflow import keras
y = keras.utils.to_categorical(digits.target, n_classes)
print (y.shape)
```

```
(1797, 10)
```

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.

[6]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(digits.data, y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train)
```

1.3 create the sequential model

1. The model will the keras sequential model containing the one-layer perceptron. Instantiate the model using **model=Sequential()**, then add a **Dense()** layer from **keras.layers**. Adding layers can be done using the model's class function **model.add()**. Specify the input shape, and the number of classes to output. What kind of activation will you use?

We will use the softmax activation because the classification is for non binary data, we have more than 2 classes.

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.

[20]:
```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(n_classes, input_shape=digits.data[0].
 ↪shape,activation='softmax'))
model.compile(loss='categorical_crossentropy',␣
 ↪optimizer='adam',metrics=['accuracy'])
```

1.4 train the model

You need to specify the size of the batch and the number of epochs to train. You need to provide the training and the validation data.

Use the sequential model's class function **fit()** to train the model.

[21]:
```
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 [==============================] - 0s 24ms/step - loss: 11.7768 - accuracy:
0.0545 - val_loss: 10.9982 - val_accuracy: 0.0623
Epoch 2/100
8/8 [==============================] - 0s 8ms/step - loss: 9.9951 - accuracy:
0.0820 - val_loss: 8.9683 - val_accuracy: 0.0801
Epoch 3/100
8/8 [==============================] - 0s 8ms/step - loss: 7.8336 - accuracy:
0.1282 - val_loss: 7.4401 - val_accuracy: 0.1276
```

```
Epoch 4/100
8/8 [==============================] - 0s 7ms/step - loss: 6.8988 - accuracy:
0.1558 - val_loss: 6.3607 - val_accuracy: 0.1751
Epoch 5/100
8/8 [==============================] - 0s 6ms/step - loss: 5.8279 - accuracy:
0.1802 - val_loss: 5.5526 - val_accuracy: 0.1958
Epoch 6/100
8/8 [==============================] - 0s 7ms/step - loss: 5.1524 - accuracy:
0.2061 - val_loss: 4.9622 - val_accuracy: 0.1929
Epoch 7/100
8/8 [==============================] - 0s 7ms/step - loss: 4.6168 - accuracy:
0.2448 - val_loss: 4.4854 - val_accuracy: 0.2136
Epoch 8/100
8/8 [==============================] - 0s 7ms/step - loss: 4.1475 - accuracy:
0.2757 - val_loss: 4.0151 - val_accuracy: 0.2493
Epoch 9/100
8/8 [==============================] - 0s 7ms/step - loss: 3.7091 - accuracy:
0.3225 - val_loss: 3.5900 - val_accuracy: 0.2997
Epoch 10/100
8/8 [==============================] - 0s 7ms/step - loss: 3.2847 - accuracy:
0.3390 - val_loss: 3.1862 - val_accuracy: 0.3294
Epoch 11/100
8/8 [==============================] - 0s 6ms/step - loss: 2.8885 - accuracy:
0.3888 - val_loss: 2.8350 - val_accuracy: 0.3739
Epoch 12/100
8/8 [==============================] - 0s 7ms/step - loss: 2.6905 - accuracy:
0.3915 - val_loss: 2.5465 - val_accuracy: 0.3976
Epoch 13/100
8/8 [==============================] - 0s 8ms/step - loss: 2.4857 - accuracy:
0.4210 - val_loss: 2.3173 - val_accuracy: 0.4273
Epoch 14/100
8/8 [==============================] - 0s 7ms/step - loss: 2.1444 - accuracy:
0.4811 - val_loss: 2.1089 - val_accuracy: 0.4718
Epoch 15/100
8/8 [==============================] - 0s 8ms/step - loss: 1.9148 - accuracy:
0.5111 - val_loss: 1.9192 - val_accuracy: 0.4955
Epoch 16/100
8/8 [==============================] - 0s 7ms/step - loss: 1.7190 - accuracy:
0.5727 - val_loss: 1.7580 - val_accuracy: 0.5282
Epoch 17/100
8/8 [==============================] - 0s 7ms/step - loss: 1.6903 - accuracy:
0.5543 - val_loss: 1.6159 - val_accuracy: 0.5786
Epoch 18/100
8/8 [==============================] - 0s 6ms/step - loss: 1.5453 - accuracy:
0.6109 - val_loss: 1.5010 - val_accuracy: 0.6113
Epoch 19/100
8/8 [==============================] - 0s 7ms/step - loss: 1.4278 - accuracy:
0.6470 - val_loss: 1.3990 - val_accuracy: 0.6320
```

```
Epoch 20/100
8/8 [==============================] - 0s 7ms/step - loss: 1.3211 - accuracy:
0.6738 - val_loss: 1.3024 - val_accuracy: 0.6409
Epoch 21/100
8/8 [==============================] - 0s 7ms/step - loss: 1.2941 - accuracy:
0.6976 - val_loss: 1.2320 - val_accuracy: 0.6588
Epoch 22/100
8/8 [==============================] - 0s 7ms/step - loss: 1.1778 - accuracy:
0.7110 - val_loss: 1.1630 - val_accuracy: 0.6795
Epoch 23/100
8/8 [==============================] - 0s 7ms/step - loss: 1.1424 - accuracy:
0.7256 - val_loss: 1.0990 - val_accuracy: 0.6884
Epoch 24/100
8/8 [==============================] - 0s 7ms/step - loss: 1.1159 - accuracy:
0.7211 - val_loss: 1.0407 - val_accuracy: 0.7122
Epoch 25/100
8/8 [==============================] - 0s 7ms/step - loss: 1.0003 - accuracy:
0.7449 - val_loss: 0.9866 - val_accuracy: 0.7151
Epoch 26/100
8/8 [==============================] - 0s 7ms/step - loss: 0.9332 - accuracy:
0.7515 - val_loss: 0.9391 - val_accuracy: 0.7181
Epoch 27/100
8/8 [==============================] - 0s 7ms/step - loss: 0.9365 - accuracy:
0.7486 - val_loss: 0.8992 - val_accuracy: 0.7270
Epoch 28/100
8/8 [==============================] - 0s 6ms/step - loss: 0.9077 - accuracy:
0.7595 - val_loss: 0.8553 - val_accuracy: 0.7329
Epoch 29/100
8/8 [==============================] - 0s 7ms/step - loss: 0.9509 - accuracy:
0.7627 - val_loss: 0.8200 - val_accuracy: 0.7389
Epoch 30/100
8/8 [==============================] - 0s 6ms/step - loss: 0.8613 - accuracy:
0.7700 - val_loss: 0.7845 - val_accuracy: 0.7507
Epoch 31/100
8/8 [==============================] - 0s 6ms/step - loss: 0.7730 - accuracy:
0.7844 - val_loss: 0.7526 - val_accuracy: 0.7596
Epoch 32/100
8/8 [==============================] - 0s 7ms/step - loss: 0.7778 - accuracy:
0.8036 - val_loss: 0.7210 - val_accuracy: 0.7804
Epoch 33/100
8/8 [==============================] - 0s 6ms/step - loss: 0.7517 - accuracy:
0.7985 - val_loss: 0.6959 - val_accuracy: 0.7745
Epoch 34/100
8/8 [==============================] - 0s 6ms/step - loss: 0.7101 - accuracy:
0.8121 - val_loss: 0.6652 - val_accuracy: 0.7923
Epoch 35/100
8/8 [==============================] - 0s 6ms/step - loss: 0.6353 - accuracy:
0.8279 - val_loss: 0.6435 - val_accuracy: 0.7982
```

```
Epoch 36/100
8/8 [==============================] - 0s 20ms/step - loss: 0.6136 - accuracy:
0.8319 - val_loss: 0.6169 - val_accuracy: 0.8042
Epoch 37/100
8/8 [==============================] - 0s 8ms/step - loss: 0.5601 - accuracy:
0.8538 - val_loss: 0.5927 - val_accuracy: 0.8101
Epoch 38/100
8/8 [==============================] - 0s 6ms/step - loss: 0.6159 - accuracy:
0.8398 - val_loss: 0.5768 - val_accuracy: 0.8071
Epoch 39/100
8/8 [==============================] - 0s 6ms/step - loss: 0.6109 - accuracy:
0.8314 - val_loss: 0.5538 - val_accuracy: 0.8160
Epoch 40/100
8/8 [==============================] - 0s 6ms/step - loss: 0.6802 - accuracy:
0.8279 - val_loss: 0.5359 - val_accuracy: 0.8249
Epoch 41/100
8/8 [==============================] - 0s 6ms/step - loss: 0.5304 - accuracy:
0.8615 - val_loss: 0.5203 - val_accuracy: 0.8279
Epoch 42/100
8/8 [==============================] - 0s 6ms/step - loss: 0.6016 - accuracy:
0.8402 - val_loss: 0.5018 - val_accuracy: 0.8279
Epoch 43/100
8/8 [==============================] - 0s 6ms/step - loss: 0.5242 - accuracy:
0.8675 - val_loss: 0.4871 - val_accuracy: 0.8338
Epoch 44/100
8/8 [==============================] - 0s 6ms/step - loss: 0.5046 - accuracy:
0.8642 - val_loss: 0.4675 - val_accuracy: 0.8457
Epoch 45/100
8/8 [==============================] - 0s 6ms/step - loss: 0.5613 - accuracy:
0.8477 - val_loss: 0.4550 - val_accuracy: 0.8516
Epoch 46/100
8/8 [==============================] - 0s 6ms/step - loss: 0.5152 - accuracy:
0.8611 - val_loss: 0.4442 - val_accuracy: 0.8576
Epoch 47/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4926 - accuracy:
0.8554 - val_loss: 0.4359 - val_accuracy: 0.8576
Epoch 48/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4498 - accuracy:
0.8712 - val_loss: 0.4240 - val_accuracy: 0.8635
Epoch 49/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4643 - accuracy:
0.8664 - val_loss: 0.4100 - val_accuracy: 0.8665
Epoch 50/100
8/8 [==============================] - 0s 6ms/step - loss: 0.4394 - accuracy:
0.8825 - val_loss: 0.3994 - val_accuracy: 0.8694
Epoch 51/100
8/8 [==============================] - 0s 6ms/step - loss: 0.4123 - accuracy:
0.8794 - val_loss: 0.3882 - val_accuracy: 0.8783
```

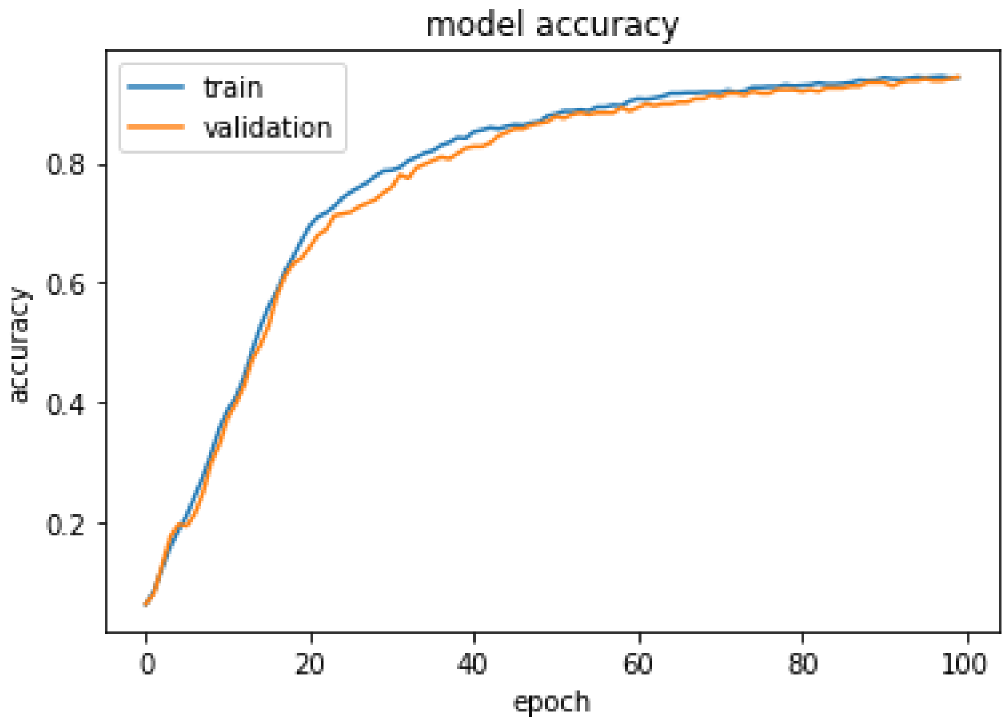
```
Epoch 52/100
8/8 [==============================] - 0s 7ms/step - loss: 0.4379 - accuracy:
0.8868 - val_loss: 0.3793 - val_accuracy: 0.8754
Epoch 53/100
8/8 [==============================] - 0s 7ms/step - loss: 0.3335 - accuracy:
0.9010 - val_loss: 0.3713 - val_accuracy: 0.8813
Epoch 54/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3500 - accuracy:
0.8942 - val_loss: 0.3620 - val_accuracy: 0.8843
Epoch 55/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3749 - accuracy:
0.8889 - val_loss: 0.3532 - val_accuracy: 0.8813
Epoch 56/100
8/8 [==============================] - 0s 9ms/step - loss: 0.3826 - accuracy:
0.8879 - val_loss: 0.3469 - val_accuracy: 0.8843
Epoch 57/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3828 - accuracy:
0.8958 - val_loss: 0.3431 - val_accuracy: 0.8843
Epoch 58/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3340 - accuracy:
0.8972 - val_loss: 0.3356 - val_accuracy: 0.8843
Epoch 59/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3232 - accuracy:
0.9072 - val_loss: 0.3255 - val_accuracy: 0.8932
Epoch 60/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3722 - accuracy:
0.9014 - val_loss: 0.3208 - val_accuracy: 0.8872
Epoch 61/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3161 - accuracy:
0.9069 - val_loss: 0.3162 - val_accuracy: 0.8932
Epoch 62/100
8/8 [==============================] - 0s 7ms/step - loss: 0.3574 - accuracy:
0.9055 - val_loss: 0.3110 - val_accuracy: 0.8991
Epoch 63/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3308 - accuracy:
0.9034 - val_loss: 0.3033 - val_accuracy: 0.8961
Epoch 64/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3563 - accuracy:
0.9107 - val_loss: 0.2979 - val_accuracy: 0.8991
Epoch 65/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3285 - accuracy:
0.9086 - val_loss: 0.2910 - val_accuracy: 0.8991
Epoch 66/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2980 - accuracy:
0.9204 - val_loss: 0.2894 - val_accuracy: 0.9021
Epoch 67/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3062 - accuracy:
0.9113 - val_loss: 0.2829 - val_accuracy: 0.9021
```

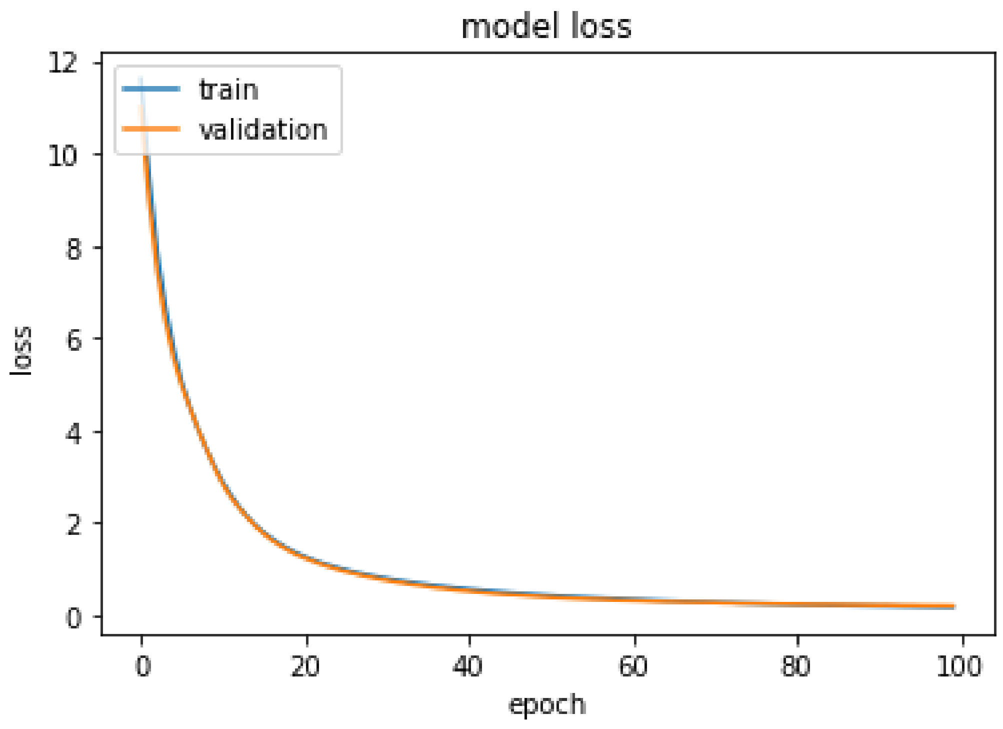
```
Epoch 68/100
8/8 [==============================] - 0s 6ms/step - loss: 0.3156 - accuracy:
0.9133 - val_loss: 0.2799 - val_accuracy: 0.9080
Epoch 69/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2919 - accuracy:
0.9133 - val_loss: 0.2768 - val_accuracy: 0.9080
Epoch 70/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2485 - accuracy:
0.9280 - val_loss: 0.2745 - val_accuracy: 0.9139
Epoch 71/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2707 - accuracy:
0.9094 - val_loss: 0.2697 - val_accuracy: 0.9110
Epoch 72/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2758 - accuracy:
0.9217 - val_loss: 0.2626 - val_accuracy: 0.9169
Epoch 73/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2947 - accuracy:
0.9137 - val_loss: 0.2612 - val_accuracy: 0.9169
Epoch 74/100
8/8 [==============================] - 0s 7ms/step - loss: 0.3109 - accuracy:
0.9062 - val_loss: 0.2544 - val_accuracy: 0.9139
Epoch 75/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2523 - accuracy:
0.9356 - val_loss: 0.2538 - val_accuracy: 0.9199
Epoch 76/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2401 - accuracy:
0.9327 - val_loss: 0.2528 - val_accuracy: 0.9169
Epoch 77/100
8/8 [==============================] - 0s 8ms/step - loss: 0.2648 - accuracy:
0.9170 - val_loss: 0.2483 - val_accuracy: 0.9169
Epoch 78/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2440 - accuracy:
0.9284 - val_loss: 0.2452 - val_accuracy: 0.9228
Epoch 79/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2145 - accuracy:
0.9380 - val_loss: 0.2436 - val_accuracy: 0.9228
Epoch 80/100
8/8 [==============================] - 0s 20ms/step - loss: 0.2267 - accuracy:
0.9349 - val_loss: 0.2395 - val_accuracy: 0.9228
Epoch 81/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2245 - accuracy:
0.9312 - val_loss: 0.2378 - val_accuracy: 0.9199
Epoch 82/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2391 - accuracy:
0.9248 - val_loss: 0.2345 - val_accuracy: 0.9228
Epoch 83/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2197 - accuracy:
0.9322 - val_loss: 0.2295 - val_accuracy: 0.9199
```

```
Epoch 84/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2426 - accuracy:
0.9286 - val_loss: 0.2288 - val_accuracy: 0.9258
Epoch 85/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2225 - accuracy:
0.9334 - val_loss: 0.2297 - val_accuracy: 0.9258
Epoch 86/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2211 - accuracy:
0.9310 - val_loss: 0.2265 - val_accuracy: 0.9258
Epoch 87/100
8/8 [==============================] - 0s 7ms/step - loss: 0.2206 - accuracy:
0.9353 - val_loss: 0.2229 - val_accuracy: 0.9288
Epoch 88/100
8/8 [==============================] - 0s 7ms/step - loss: 0.1903 - accuracy:
0.9452 - val_loss: 0.2220 - val_accuracy: 0.9288
Epoch 89/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2105 - accuracy:
0.9365 - val_loss: 0.2194 - val_accuracy: 0.9347
Epoch 90/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2174 - accuracy:
0.9330 - val_loss: 0.2165 - val_accuracy: 0.9347
Epoch 91/100
8/8 [==============================] - 0s 7ms/step - loss: 0.1761 - accuracy:
0.9432 - val_loss: 0.2142 - val_accuracy: 0.9347
Epoch 92/100
8/8 [==============================] - 0s 7ms/step - loss: 0.1916 - accuracy:
0.9444 - val_loss: 0.2165 - val_accuracy: 0.9288
Epoch 93/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2116 - accuracy:
0.9374 - val_loss: 0.2112 - val_accuracy: 0.9347
Epoch 94/100
8/8 [==============================] - 0s 6ms/step - loss: 0.1925 - accuracy:
0.9400 - val_loss: 0.2066 - val_accuracy: 0.9377
Epoch 95/100
8/8 [==============================] - 0s 7ms/step - loss: 0.1892 - accuracy:
0.9415 - val_loss: 0.2081 - val_accuracy: 0.9377
Epoch 96/100
8/8 [==============================] - 0s 7ms/step - loss: 0.1742 - accuracy:
0.9438 - val_loss: 0.2054 - val_accuracy: 0.9407
Epoch 97/100
8/8 [==============================] - 0s 6ms/step - loss: 0.1683 - accuracy:
0.9458 - val_loss: 0.2037 - val_accuracy: 0.9377
Epoch 98/100
8/8 [==============================] - 0s 6ms/step - loss: 0.2087 - accuracy:
0.9409 - val_loss: 0.2023 - val_accuracy: 0.9377
Epoch 99/100
8/8 [==============================] - 0s 6ms/step - loss: 0.1850 - accuracy:
0.9410 - val_loss: 0.2023 - val_accuracy: 0.9407
```

```
Epoch 100/100
8/8 [==============================] - 0s 6ms/step - loss: 0.1681 - accuracy:
0.9383 - val_loss: 0.1983 - val_accuracy: 0.9436
```

[22]:
```
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()
```





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 a big or a small model?
2. What was the accuracy we 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, The model have 650 parameters but we only have one layer made of 10 neurons, so it is a small model.

2, At the end of the training we have an accuracy of 0.9383 on the training data and an accuracy of 0.9436 which represents the accuracy of the model.

3, No, the results will always be different, for example, when I run it again I have an accuracy of 0.9258 instead of 0.9436.

[23]:
```
print (model.summary())
```

```
Model: "sequential_1"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 650
```

```
Trainable params: 650
Non-trainable params: 0
_________________________________________________________________
None
```

1.6 print the confusion matrix obtained on the test dataset.

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

[24]:
```
 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)
```

```
[[36  0  1  0  0  0  0  0  0  0]
 [ 0 47  3  0  0  0  0  0  1  0]
 [ 0  0 43  2  0  0  0  0  1  0]
 [ 0  1  0 39  0  0  0  2  1  0]
 [ 0  1  0  0 45  0  0  0  0  2]
 [ 0  2  0  2  1 40  0  0  0  1]
 [ 0  0  0  0  0  0 38  0  1  0]
 [ 0  4  0  0  3  0  0 45  0  0]
 [ 0  2  1  0  0  0  0  0 37  1]
 [ 0  1  0  1  0  3  0  0  4 38]]
```

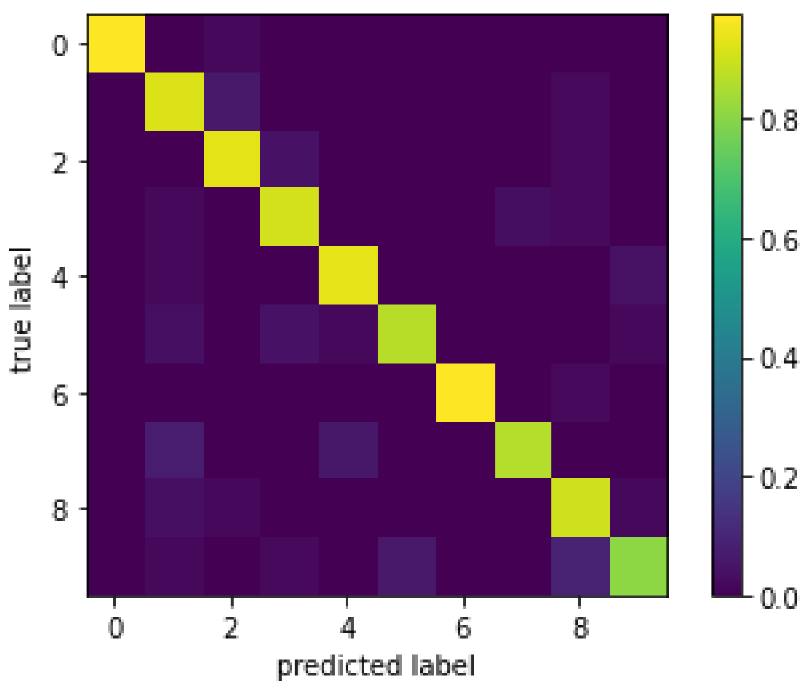
Normalize the confusion matrix to show graphically the prediction probability map.

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.

[25]:
```
_,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')
```

```
[37 51 46 43 48 46 39 52 41 47]
```

[25]:
```
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()**.

We have an accuracy of 0.908 on the dataset and a loss of 0.313.

[26]:
```
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: 0.3250 - accuracy:
0.9067
The loss and accuracy on the test dataset : 0.324952, 0.906667
```

1.7 Conclusions:

Conclude your report by answering the following questions:

1. Did your training converge?
2. Was the training successful?
3. Do you always obtain the same accuracy when you run the code several times?
4. Do all classes always predict with the same accuracy?
5. Are there classes where the prediction happens to fail?

1, Yes the training converged toward a good accuracy, by plotting the model loss, we can also say that we have no overfitting

2, We can say that the training was successful, the test score are pretty good, we have an accuracy of 0.906667 and a model loss of 0.3250.

3, No we don't get the same accuracy when we run the code again, this is most likely due to the fact that our algorithm use randomness during learning, ensuring a different model is trained each run. We could resolve this issue by setting a random_state.

4, No, some classes have a almost perfect score like for the number 0, but others have a lot have misclassification like the numbers 5,7,9. We can see this with the conufsion matrix.

5, The classification never entirely fails, but some works less than others, the worst class here is the 9th, with 9 missclassification.

```
[ ]:
```

```
[ ]:
```