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
```

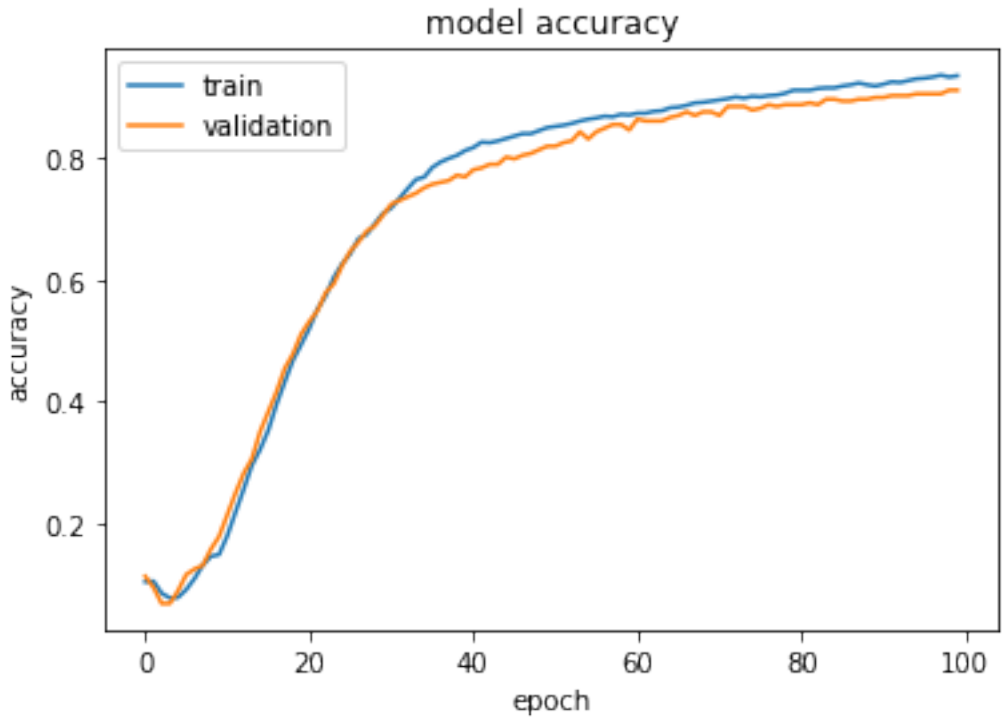
```
8/8 [==============================] - 0s 8ms/step - loss: 0.6024 - accuracy:
0.8311 - val_loss: 0.6487 - val_accuracy: 0.8012
Epoch 46/100
8/8 [==============================] - 0s 9ms/step - loss: 0.6008 - accuracy:
0.8358 - val_loss: 0.6374 - val_accuracy: 0.7982
Epoch 47/100
8/8 [==============================] - 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
```

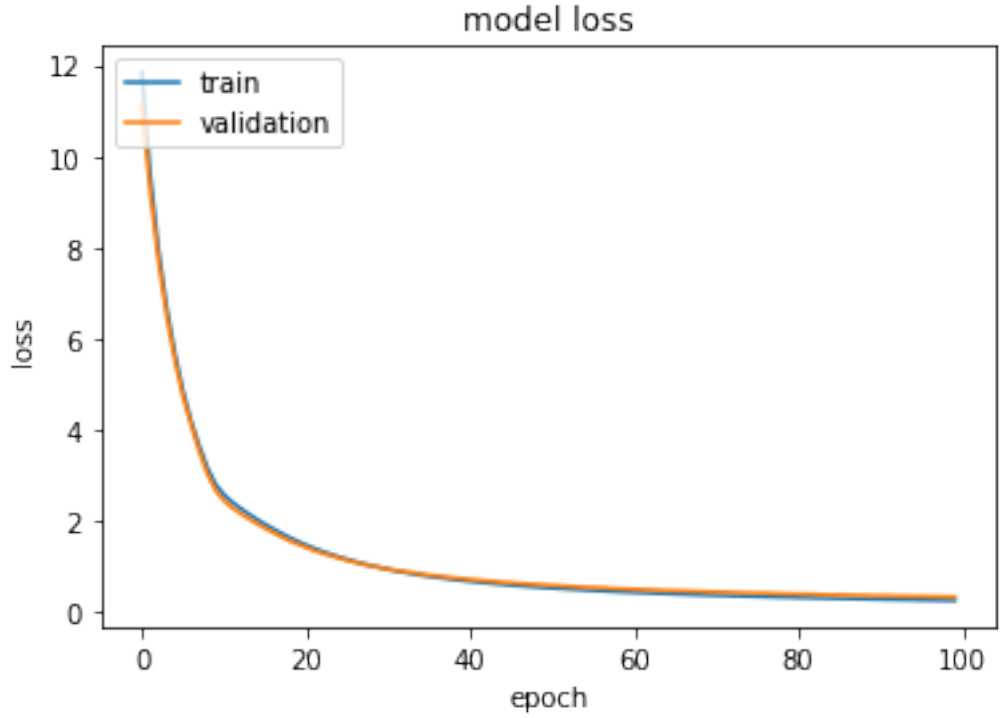
```
8/8 [==============================] - 0s 8ms/step - loss: 0.4104 - accuracy:
0.8781 - val_loss: 0.4931 - val_accuracy: 0.8635
Epoch 62/100
8/8 [==============================] - 0s 8ms/step - loss: 0.4394 - accuracy:
0.8723 - val_loss: 0.4892 - val_accuracy: 0.8605
Epoch 63/100
8/8 [==============================] - 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]]
```

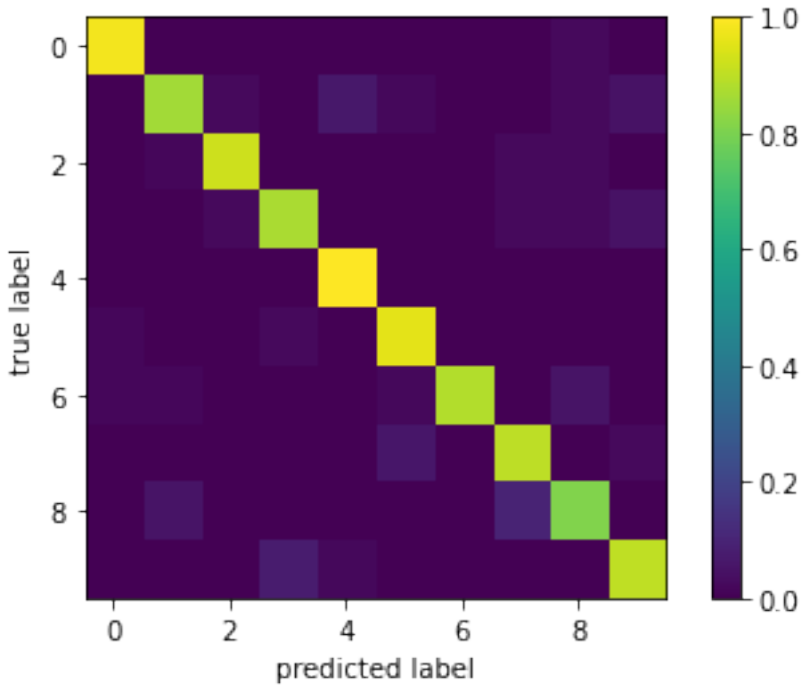
Normalize the confusion matrix to show graphically the prediction probability map.

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
```

AMIOT-TZDAKA_exercise_4_skin_cancer_assignment

December 18, 2020

1 Lab session 2 :

1.1 Exercise 4 : MBE skin cancer classification using a CNN

In this session you will develop a CNN to classify images of skin cancer into 7 categories.

The images come from the **HAM10000 ("Human Against Machine with 10000 training images")** dataset. It consists of 10015 dermatoscopic images which are released as a training set for academic machine learning purposes and are publicly available through the ISIC archive. This benchmark dataset can be used for machine learning and for comparisons with human experts.

It has 7 different classes of skin cancer which are listed below : **1. Melanocytic nevi 2. Melanoma 3. Benign keratosis-like lesions 4. Basal cell carcinoma 5. Actinic keratoses 6. Vascular lesions 7. Dermatofibroma**

We will follow these steps to classify moles into 7 classes.

Step 1: Importing Essential Libraries Step 2: Making Dictionary of images and labels Step 3: Reading and Processing Data Step 4: Data Cleaning Step 5: Exploratory data analysis (EDA) Step 6: Loading & Resizing of images Step 7: Train Test Split Step 8: Normalization Step 9: Label Encoding Step 10: Train validation split Step 11: Model Building (CNN) Step 12: Setting Optimizer & Annealing Step 13: Fitting the model Step 14: Model Evaluation (Testing and validation accuracy, confusion matrix, analysis of misclassified instances)

2 Step 1 : Auxiliary steps

importing essential python libraries

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os, sys, requests, tarfile, io, itertools
from PIL import Image
from time import time
np.random.seed(123)
from sklearn.preprocessing import label_binarize
from sklearn.metrics import confusion_matrix
```

```
import tensorflow as tf
print (tf.__version__)
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D
from keras import backend as K
from keras.layers.normalization import BatchNormalization
from keras.utils.np_utils import to_categorical # convert to one-hot-encoding
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
from sklearn.model_selection import train_test_split

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
!nvidia-smi
```

```
2.4.0
Found GPU at: /device:GPU:0
Fri Dec 18 16:37:25 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.45.01    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   50C    P0    29W /  70W |    227MiB / 15079MiB |      8%      Default |
|                               |                      |                 ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Your data will be stored in your google drive space. You need to mount your drive to access it from the notebook. You will be invited to confirm authorization in a separate window and copy&paste authorization code.

[2]:
```
# -*- coding: utf-8 -*-
"""exercise_3_functions.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ou7s5cH5IcIYnpCQBsMBMuDxF3qRb4AY
"""

from matplotlib import pyplot as plt
import numpy as np
import itertools

# Function to plot confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')



#define a function to plot model's validation loss and validation accuracy
def plot_model_history(model_history):
    fig, axs = plt.subplots(1,2,figsize=(15,5))
```

```
    # summarize history for accuracy
    axs[0].plot(range(1,len(model_history.history['accuracy'])+1),model_history.
↪history['accuracy'])
    axs[0].plot(range(1,len(model_history.
↪history['val_accuracy'])+1),model_history.history['val_accuracy'])
    axs[0].set_title('Model Accuracy')
    axs[0].set_ylabel('Accuracy')
    axs[0].set_xlabel('Epoch')
    axs[0].set_xticks(np.arange(1,len(model_history.
↪history['accuracy'])+1),len(model_history.history['accuracy'])/10)
    axs[0].legend(['train', 'val'], loc='best')
    # summarize history for loss
    axs[1].plot(range(1,len(model_history.history['loss'])+1),model_history.
↪history['loss'])
    axs[1].plot(range(1,len(model_history.history['val_loss'])+1),model_history.
↪history['val_accuracy'])
    axs[1].set_title('Model Loss')
    axs[1].set_ylabel('Loss')
    axs[1].set_xlabel('Epoch')
    axs[1].set_xticks(np.arange(1,len(model_history.
↪history['loss'])+1),len(model_history.history['loss'])/10)
    axs[1].legend(['train', 'val'], loc='best')
    plt.show()
```

[3]:
```
if os.getcwd()=='/content':
    try:
        from google.colab import drive
        base_working_dir = '/content/drive/My Drive/Colab Notebooks/MBE DL␣
↪course'
        drive.mount('/content/drive')
    except:
        base_working_dir = os.getcwd()
        pass

sys.path.append(base_working_dir)

base_working_dir = os.path.join(base_working_dir, 'Skin_Cancer_class')

if not os.path.exists(base_working_dir):
    os.makedirs(base_working_dir)

print (base_working_dir)
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/MBE DL course/Skin_Cancer_class
```

3 Step 2 : Making Dictionary of labels

[4]:
```
# This dictionary is useful for displaying more human-friendly labels later on
lesion_type_dict = {
    'nv': 'Melanocytic nevi',
    'mel': 'Melanoma',
    'bkl': 'Benign keratosis-like lesions ',
    'bcc': 'Basal cell carcinoma',
    'akiec': 'Actinic keratoses',
    'vasc': 'Vascular lesions',
    'df': 'Dermatofibroma'}
```

4 Step 3 : Reading & Processing data

We made some new columns which is easily understood for later reference such as the path to the image_id, cell_type which contains the short name of lesion type.

We convert the categorical column cell_type_idx in which we have categorize the lesion type in to codes from 0 to 6

[5]:
```
try:
    skin_df = pd.read_csv(os.path.join(base_working_dir, 'HAM10000_metadata.
 ↪csv'))
except:
    # download images from a shared archive
    url = 'https://drive.google.com/uc?
 ↪authuser=0&id=1jdCVOmeJXI6bhWIfFVL7RqwyuOHiRjaE&export=download'
    r = requests.get(url, allow_redirects=True)
    open(os.path.join(base_working_dir, 'HAM10000_metadata.csv'), 'wb').write(r.
 ↪content)
    skin_df = pd.read_csv(os.path.join(base_working_dir, 'HAM10000_metadata.
 ↪csv'))

# Creating New Columns for better readability
skin_df['cell_type'] = skin_df['dx'].map(lesion_type_dict.get)
skin_df['cell_type_idx'] = pd.Categorical(skin_df['cell_type']).codes
```

Execute the cell below and look on the table. We have a table where each line references image and corresponding data (localization, lesion, consensus, sex, ...). Use this table to analyze distribution of various variables here.

[6]:
```
# Now lets see the sample of tile_df to look on newly made columns
skin_df.head()
```

[6]:

	lesion_id	image_id	…	cell_type	cell_type_idx
0	HAM_0000118	ISIC_0027419	…	Benign keratosis-like lesions	2
1	HAM_0000118	ISIC_0025030	…	Benign keratosis-like lesions	2
2	HAM_0002730	ISIC_0026769	…	Benign keratosis-like lesions	2

```
3  HAM_0002730  ISIC_0025661  …  Benign keratosis-like lesions              2
4  HAM_0001466  ISIC_0031633  …  Benign keratosis-like lesions              2

[5 rows x 9 columns]
```

5 Step 5 : Statistical Analysis of the Data

Look at different features of the **HAM10000_metadata.csv** dataset, their distributions and counts. Use the method **value_counts()** of the pandas datasheet **skin_df** to plot the following distributions: 1. the type of the cancer 2. the validation method used 3. the location of the cancer on the body 4. the age 5. the sex 6. the prevalence of each cancer type against the age. Use the **scatterplot** function from the **seaborn** package.

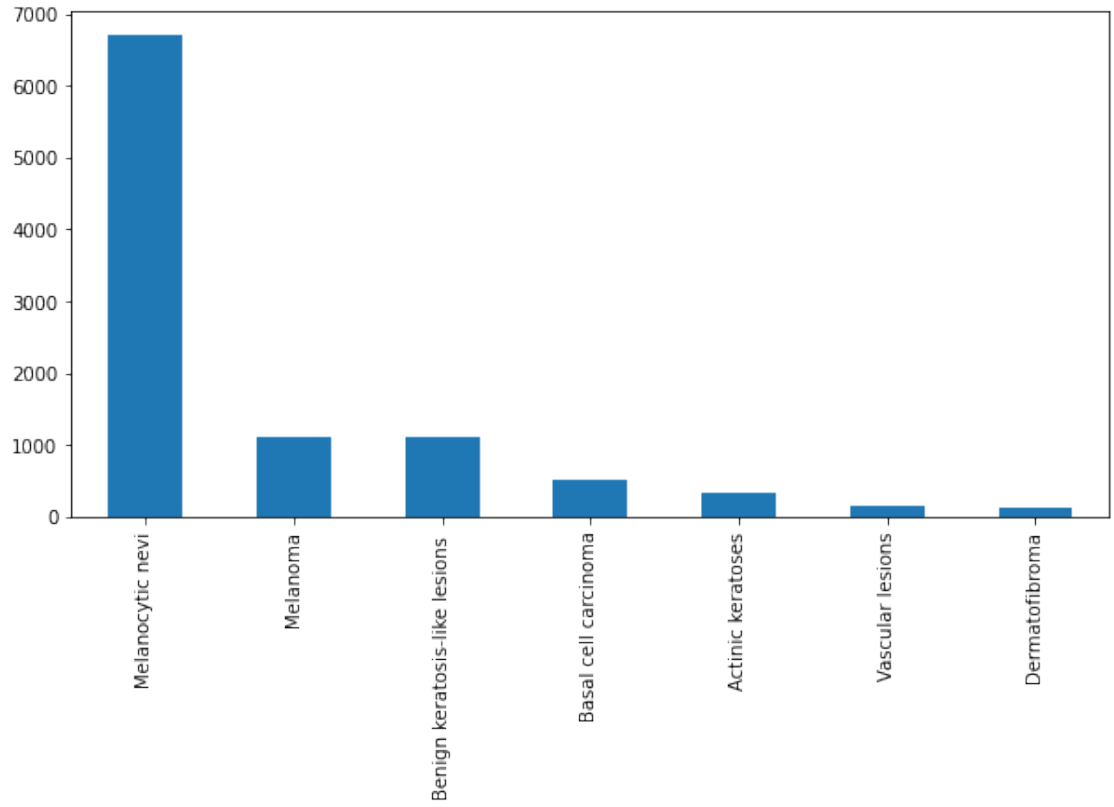
5.0.1 Q : What is the most frequently occuring cancer type in the database?

plot the histogram of the 'cell_type' occurences. Use the **value_counts()** method of the pandas table 'skin_df' as indicated below.

As we can see the most common cancer type in the database is the Melanocytic nevi.

[7]:
```
fig, ax1 = plt.subplots(1, 1, figsize= (10, 5))
skin_df['cell_type'].value_counts().plot(kind='bar', ax=ax1)
```

[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5723d16a58>
```

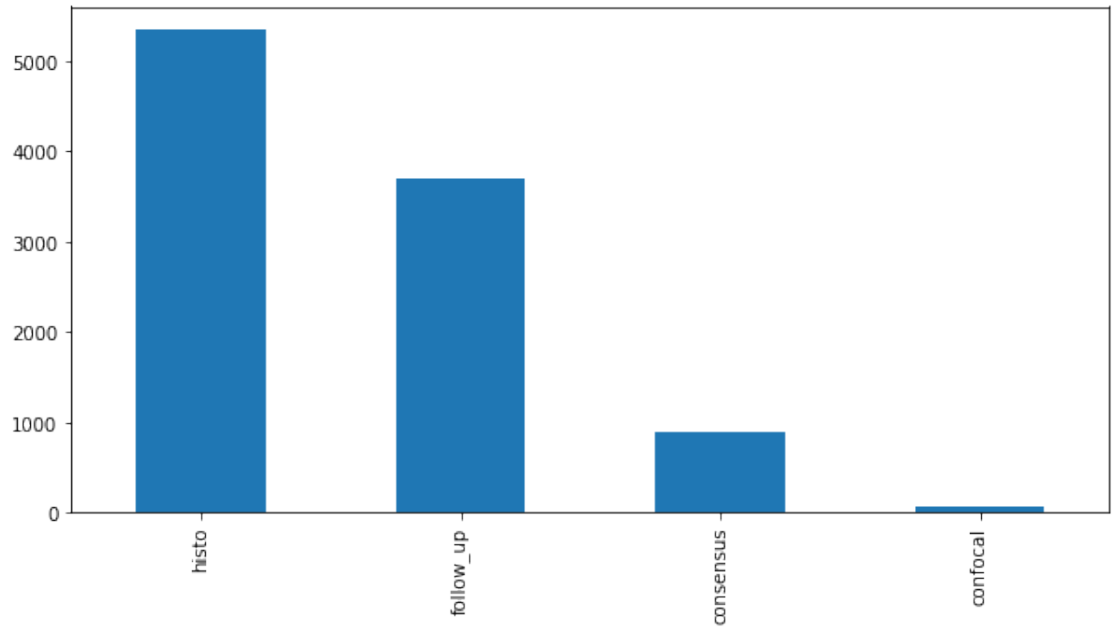


5.0.2 Plot the frequency of different validation methods.

The Technical Validation has been done by either of the following methods: **1. Histopathology(Histo): 2. Confocal:** In-vivo reflectance confocal microscope **3. Follow-up:** If nevi monitored by digital dermatoscopy did not show any changes during 3 follow-up visits or 1.5 years biologists accepted this as evidence of biologic benignity. Only nevi, but no other benign diagnoses were labeled with this type of ground-truth because dermatologists usually do not monitor dermatofibromas, seborrheic keratoses, or vascular lesions. **4. Consensus:** For typical benign cases without histopathology or followup biologists provide an expert consensus (labeled consensus only if both experts independently gave the same unequivocal diagnosis.

[8]:
```
fig, ax1 = plt.subplots(1, 1, figsize= (10, 5))
skin_df['dx_type'].value_counts().plot(kind='bar', ax=ax1)
```

[8]: <matplotlib.axes._subplots.AxesSubplot at 0x7f56bc15ee48>

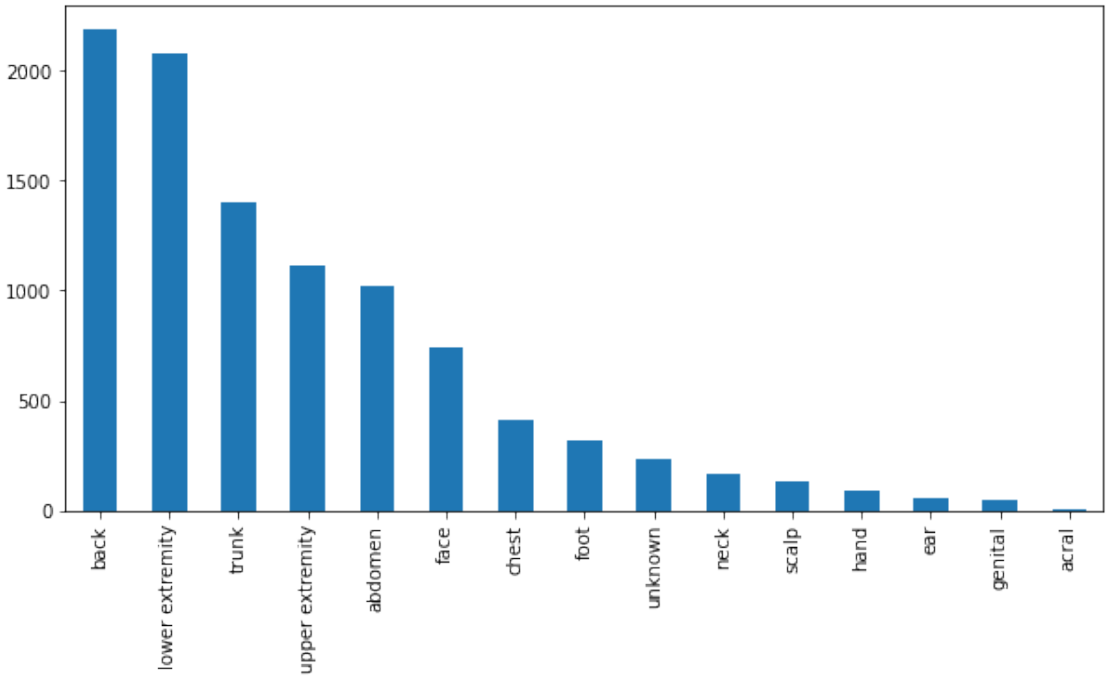


5.0.3 Q. What are the most heavily compromised regions of skin cancer?

The most heavily compromised region by the skin cancer is the back.

[9]:
```
fig, ax1 = plt.subplots(1, 1, figsize= (10, 5))
skin_df['localization'].value_counts().plot(kind='bar', ax=ax1)
```

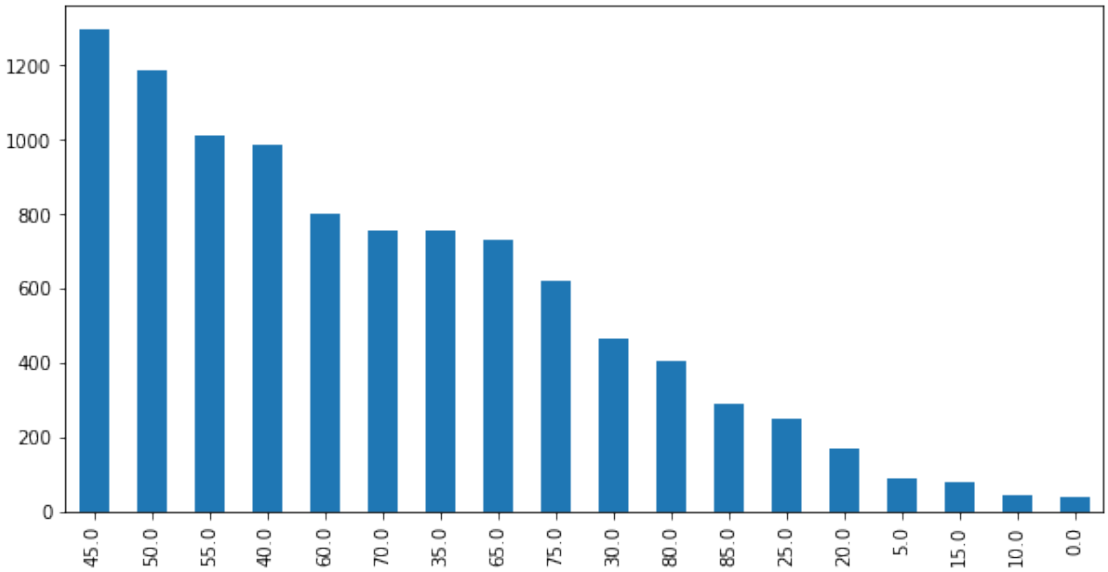
[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7f56bbc8dd68>



5.0.4 Q. Check the distribution of Age

[10]:
```
fig, ax1 = plt.subplots(1, 1, figsize= (10, 5))
skin_df['age'].value_counts().plot(kind='bar', ax=ax1)
```

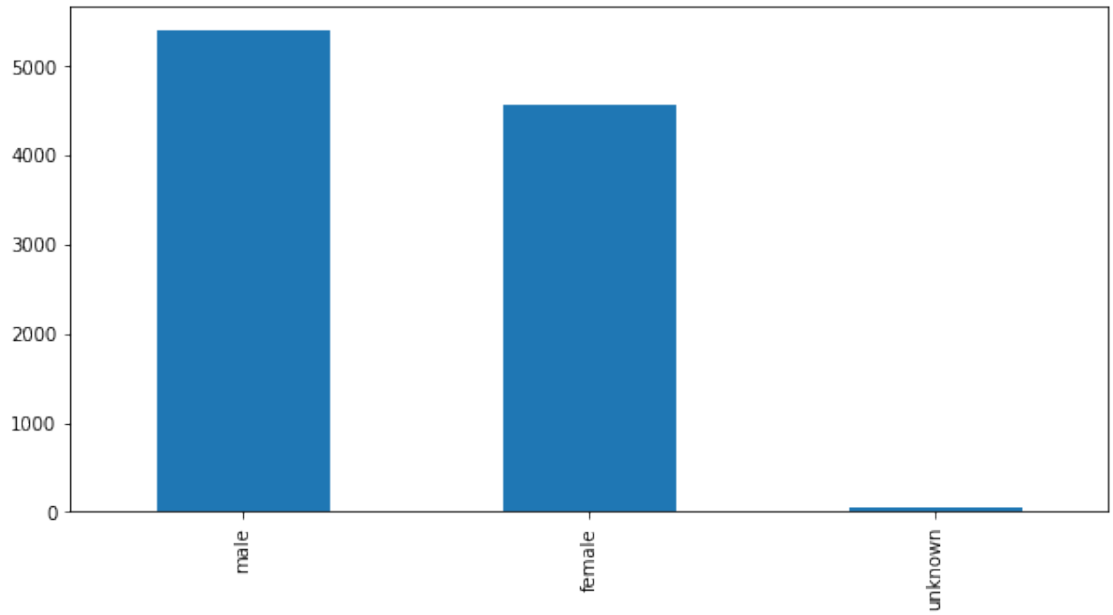
[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7f56bc16e2e8>



5.0.5 Q. Plot the distribution of males and females.

[11]:
```
fig, ax1 = plt.subplots(1, 1, figsize= (10, 5))
skin_df['sex'].value_counts().plot(kind='bar', ax=ax1)
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f56bbba0860>
```



5.0.6 Q. Visualize the age-wise distribution of skin cancer types.

What conclusions can be made about the prevalence of different cancer types regarding the age?

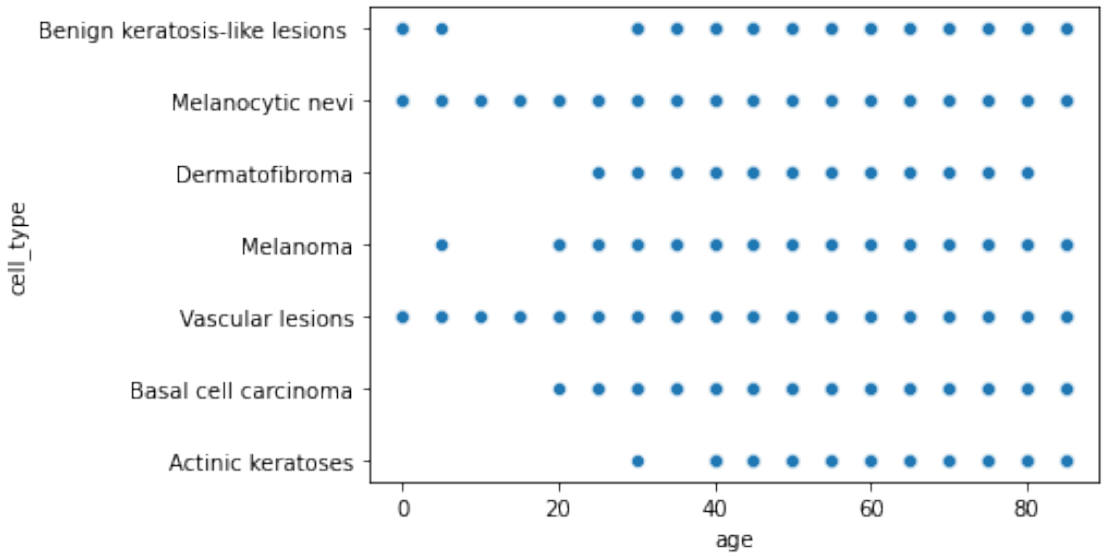
With have less cancer if the patient is less than 30 years old.

Use the seaborn method scatterplot.

[12]:
```
sns.scatterplot(skin_df['age'], skin_df['cell_type'] )
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWarning:
Pass the following variables as keyword args: x, y. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f56bbb02240>
```



6 Step 6: Loading the images

In the three following steps cancer images will be loaded from a shared archive to your personal Google Drive storage space. After this step a new file "HAM10000_images.tar" should be created in your Google Drive.

1) High-resolution images and the relief of the lesion are important featuers for dermatologists to examine visually skin moles.
2) We do not have access to the 3D relief using dermatoscopic images.
3) Processing big data is time-costly. In this lab session we will use on downsampled images.
4) Interested in trying later on mid-sized images? Use the file 'HAM10000_images_300x225.tar' (you will need to download it manually to the Skin_Cancer_class directory in your Gdrive first). Make the model deeper. Experiment with different-size models, regularization and batch normalization. Could you obtain better accuracy?

Yes, we can obtain better accuracy by making model keeper and with regularization, batch normalization... We can always improve our results.

[13]:
```
# select below to work with small images or large images

if True:
    # use small 150x100 images first
    url = 'https://drive.google.com/uc?
 ↪authuser=0&id=1--oGquD0y48lW-6WRz5ldM1rGqbJKOez&export=download'
    tgz_name = 'HAM10000_images_150x100.tar'
else:
```

```
    # high resolution is important for dermatologists to assess skin moles␣
↪visually
    # interested in training on larger images 300x250?
    # Download this file manually to your Google drive first
    # https://drive.google.com/uc?
↪authuser=0&id=1-4fKAGB_rpzp6eFFEzAYOVJgXtkDzLoW&export=download
    url = ''
    tgz_name = 'HAM10000_images_300x225.tar'
```

[14]:
```
try:
    t = tarfile.open(os.path.join(base_working_dir,tgz_name), 'r')
    print ('file %s found'%os.path.join(base_working_dir,tgz_name))
except:
    # download images from a shared archive
    r = requests.get(url, allow_redirects=True)
    open(os.path.join(base_working_dir,tgz_name), 'wb').write(r.content)
    print ('%s downloaded'%os.path.join(base_working_dir,tgz_name))
```

```
file /content/drive/My Drive/Colab Notebooks/MBE DL
course/Skin_Cancer_class/HAM10000_images_150x100.tar found
```

The images will be decompressed and loaded into the datasheet from the downloaded archive. Loading 10000 images takes several seconds.

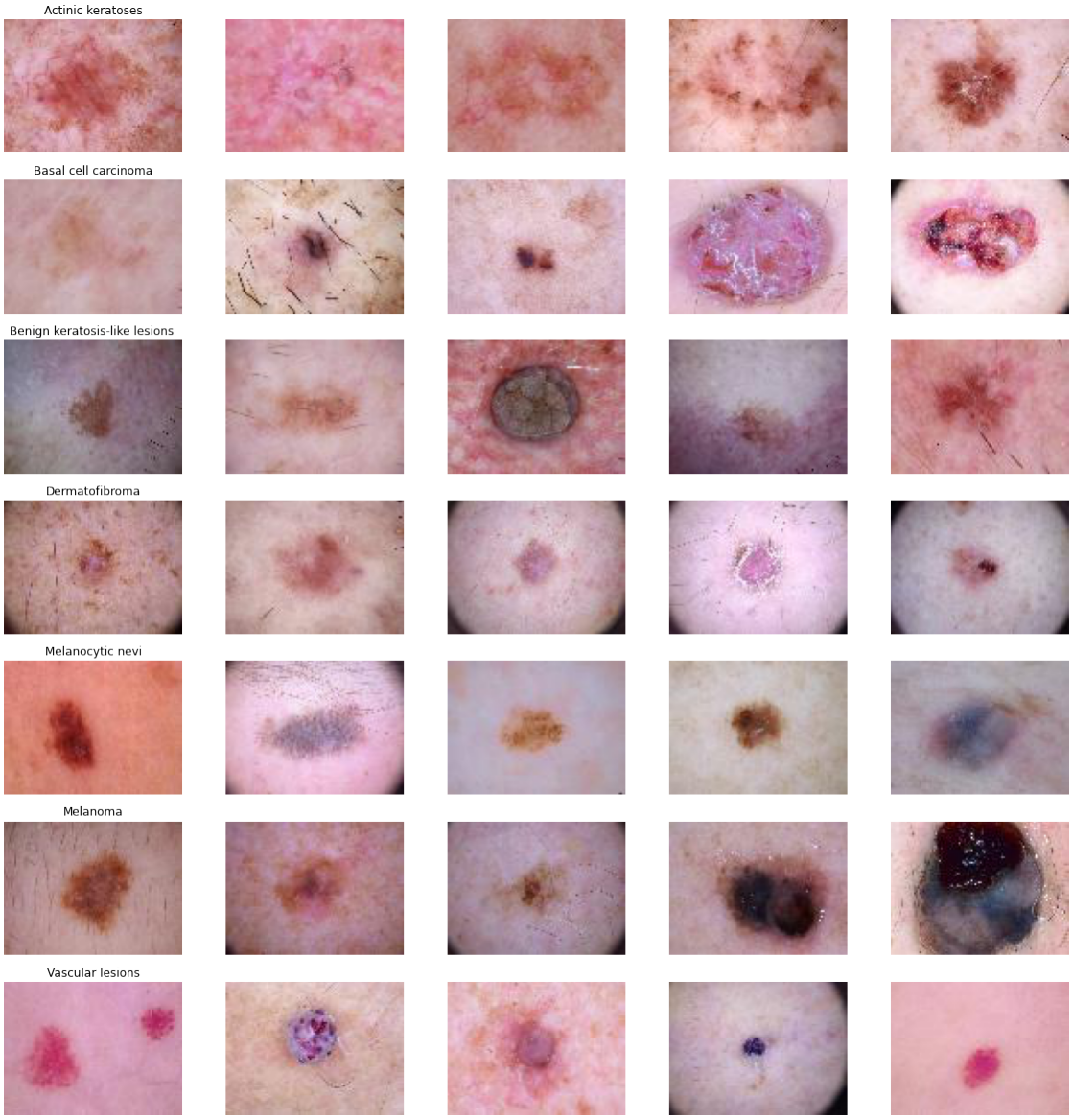
[15]:
```
# read each image and assign it as a cell content
t = tarfile.open(os.path.join(base_working_dir,tgz_name), 'r')

t1 = time()
skin_df['image'] = skin_df['image_id'].map(lambda x: np.asarray(Image.open(io.
↪BytesIO(t.extractfile(os.path.join('HAM10000_images',x+'.jpg')).read()))))
print ('%d images read in %d seconds.'%(len(skin_df['image']),time()-t1))
```

```
10015 images read in 6 seconds.
```

Now you have 10015 colour images of size 100x75 loaded into the computer memory.

Check a few sample images to see each cancer type

[16]:
```
n_samples = 5
fig, m_axs = plt.subplots(7, n_samples, figsize = (4*n_samples, 3*7))
for n_axs, (type_name, type_rows) in zip(m_axs,
                                         skin_df.sort_values(['cell_type']).
↪groupby('cell_type')):
    n_axs[0].set_title(type_name)
    for c_ax, (_, c_row) in zip(n_axs, type_rows.sample(n_samples,␣
↪random_state=1234).iterrows()):
        c_ax.imshow(c_row['image'])
        c_ax.axis('off')
```



7 Step 7 : Dataset preparation

Convert the target (the **cell_type_idx** column of the datasheet) into the one-hot encoding format. Recall, that one-hot encoding allows having several output neurons, with only one out of these firing one. Use the keras **to_categorical** function for the conversion.

Split the dataset into training and testing set of 80:20 ratio. The testing set will not be used during the training but rather in the end of this script to test the accuracy. The training set will be further split into training and validation. Use the scikit-learn **train_test_split** function.

[17]:
```
features = np.asarray(skin_df['image'])
target = to_categorical(skin_df['cell_type_idx'], num_classes = 7)


X_train, X_test, y_train, y_test = train_test_split(features, target,␣
↪test_size=0.20, random_state=42)
```

8 Step 8 : Normalization

If your data come from different acquisition devices (different dermatoscope devices in various labs) it is always a good idea to normalize your data. Normalize the x_train, x_test arrays by substracting the mean values and dividing by the standard deviation.

[18]:
```
X_train = np.asarray(X_train.tolist())
X_test = np.asarray(X_test.tolist())

X_train_mean = np.mean(X_train)
X_test_mean = np.mean(X_test)

X_train_std = np.std(X_train)
X_test_std = np.std(X_test)

X_train = (X_train - X_train_mean)/(X_train_std)
X_test = (X_test - X_test_mean)/(X_test_std)
```

9 Step 10 : Splitting training and validation

Split further the train set in two parts : 1. the majority (90%) is for training the model. 2. the rest, a small fraction (10%) for validation

[19]:
```
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
↪10, random_state=42)
```

10 Step 11: Model Building

11 CNN

Used the Keras **Sequential()** API to build your model. You can simply add one layer at a time, starting from the input, towards the output.

The first one is the convolutional **Conv2D()** layer. It is a set of learnable filters.

A second important type of layer in CNN is the pooling **MaxPool2D()** layer. This layer simply acts as a downsampling filter. It is used to reduce computational cost, and to reduce overfitting.

Combining convolutional and pooling layers, CNN are able to combine local features and learn more global features of the image.

Dropout() is a regularization method, where a proportion of nodes in the layer are randomly ignored (setting their wieghts to zero) for each training sample. This drops randomly a propotion of the network and forces the network to learn features in a distributed way. This technique improves generalization and reduces the overfitting.

For the activation, use the **relu** rectifier, replicating the max(0,x) function. It adds the non-linearity to the network.

The **Flatten()** layer is used to convert the final feature maps into a one single 1D vector allowing to use fully connected layers **Dense()** in the second CNN stage. You can use one or several dense layers. The last dense layer will have as many outputs as the categories in your target. In the last layerthe activation should be "softmax" to output the distribution of probability of each class.

When you use more that four or five layers in your model, a **BatchNormalization()** layer inserted every two or three layers will help to avoid the gradient vanishing problem and speed up the training.

[20]:
```
# Set the CNN model
# my CNN architechture is In -> [[Conv2D->relu]*2 -> MaxPool2D -> Dropout]*2 ->␣
 ↪Flatten -> Dense -> Dropout -> Out
num_classes = len(lesion_type_dict)
input_shape = X_train[0].shape

model = Sequential()
model.add(Conv2D(16, kernel_size=(3,3), activation='relu',␣
 ↪padding='Same',input_shape=input_shape))
model.add(Conv2D(16, kernel_size=(3,3), activation='relu', padding='Same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(24, kernel_size=(3,3), activation='relu', padding='Same'))
model.add(Conv2D(24, kernel_size=(3,3), activation='relu', padding='Same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

# model.add(Conv2D(32, kernel_size=(3,3), activation='relu', padding='Same'))
# model.add(Conv2D(32, kernel_size=(3,3), activation='relu', padding='Same'))
# model.add(BatchNormalization())
# model.add(MaxPool2D(pool_size=(2,2)))
# model.add(Dropout(0.25))

# model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='Same'))
# model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='Same'))
# model.add(BatchNormalization())
# model.add(MaxPool2D(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(num_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 75, 100, 16)       448
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 100, 16)       2320
_________________________________________________________________
batch_normalization (BatchNo (None, 75, 100, 16)       64
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 37, 50, 16)        0
_________________________________________________________________
dropout (Dropout)            (None, 37, 50, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 50, 24)        3480
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 37, 50, 24)        5208
_________________________________________________________________
batch_normalization_1 (Batch (None, 37, 50, 24)        96
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 18, 25, 24)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 18, 25, 24)        0
_________________________________________________________________
flatten (Flatten)            (None, 10800)             0
_________________________________________________________________
dense (Dense)                (None, 128)               1382528
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 903
=================================================================
Total params: 1,395,047
Trainable params: 1,394,967
Non-trainable params: 80
_________________________________________________________________
```

12 Step 12: Setting the Optimizer and Compilation of the model

Set up a score function, a loss function and an optimisation algorithm. The loss function is used to optimize the model using the gradient descent. Use the **categorical crossentropy**.

The metric is used to observe the performance of the model. This metric function is similar to the

loss function, except that the results from the metric evaluation are not used when training the model (only on the validation dataset to verify that the model does not overfit - in which case it decreases). Use the **accuracy**.

The most important function is the optimizer. This function will perform the gradient descent in the loss function to improve parameters (filters kernel values, weights and bias of neurons ...) and thus minimise the loss. Choose the **Adam** optimizer; it combines the advantages of two other stochastic gradient descent methods, that is the AdaGrad and RMSProp. Adam is a popular algorithm in the field of deep learning because it achieves good results fast and can be safely used with default parameter values.

[22]:
```
# Compile the model

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

13 Data Augmentation

In order to avoid overfitting problem, we need to expand artificially our HAM 10000 dataset to make the existing dataset even larger. The idea is to make the model robust to alterations in shape, size, position and rotation (naturally occuring with skin cancer moles).

Create a datagenerator object using the **ImageDataGenerator** function. Specify the rotation_range, zoom_range, width_shift_range, height_shift_range values, and set True the horizontal_flip and vertical_flip. Fit the datagenerator to the x_train data using the method **fit()**.

[23]:
```
# With data augmentation to prevent overfitting

datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the␣
↪dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=180,  # randomly rotate images in the range (degrees, 0␣
↪to 180)
        zoom_range = 0.2, # Randomly zoom image
        width_shift_range=0.3,  # randomly shift images horizontally (fraction␣
↪of total width)
        height_shift_range=0.3,  # randomly shift images vertically (fraction␣
↪of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=True)  # randomly flip images

datagen.fit(X_train)
```

14 Step 13: Fitting the model

Fit the model to x_train, y_train data. Observe the current accuracy during the training (it is the val_acc column).

A 50-epoch training lasts ~16 minutes. You can start preparing the report during the training. **The report is due in one week from today.**

You can also allow training just for 20 epochs. In 20 epochs (~7minutes) your model should almost converge.

[24]:
```
# Fit the model
from keras.callbacks import ModelCheckpoint, EarlyStopping

earlyStopping = EarlyStopping(monitor='val_accuracy', patience=20, verbose=0,␣
↪mode='max')
model_save = ModelCheckpoint('model_best.h5', save_best_only=True,␣
↪monitor='val_accuracy', mode='max')

epochs = 200
batch_size = 64
history = model.fit_generator(datagen.flow(X_train,y_train,␣
↪batch_size=batch_size),
                              epochs = epochs, validation_data = (X_val,y_val),
                              verbose = 1, steps_per_epoch=X_train.shape[0] //␣
↪batch_size,
                              callbacks = [earlyStopping, model_save])
```

```
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
```

```
Epoch 1/200
112/112 [==============================] - 21s 121ms/step - loss: 1.6297 -
accuracy: 0.5767 - val_loss: 1.8543 - val_accuracy: 0.4975
Epoch 2/200
112/112 [==============================] - 13s 117ms/step - loss: 1.0405 -
accuracy: 0.6530 - val_loss: 2.0287 - val_accuracy: 0.4875
Epoch 3/200
112/112 [==============================] - 13s 116ms/step - loss: 0.9966 -
accuracy: 0.6670 - val_loss: 1.7858 - val_accuracy: 0.5374
Epoch 4/200
112/112 [==============================] - 13s 117ms/step - loss: 0.9611 -
accuracy: 0.6753 - val_loss: 1.1005 - val_accuracy: 0.6384
Epoch 5/200
112/112 [==============================] - 13s 116ms/step - loss: 0.9442 -
accuracy: 0.6667 - val_loss: 0.9919 - val_accuracy: 0.6608
```

```
Epoch 6/200
112/112 [==============================] - 13s 119ms/step - loss: 0.9344 -
accuracy: 0.6713 - val_loss: 1.0317 - val_accuracy: 0.6608
Epoch 7/200
112/112 [==============================] - 13s 118ms/step - loss: 0.9216 -
accuracy: 0.6774 - val_loss: 0.9829 - val_accuracy: 0.6708
Epoch 8/200
112/112 [==============================] - 13s 118ms/step - loss: 0.9144 -
accuracy: 0.6682 - val_loss: 1.0192 - val_accuracy: 0.6596
Epoch 9/200
112/112 [==============================] - 13s 117ms/step - loss: 0.9224 -
accuracy: 0.6539 - val_loss: 0.9468 - val_accuracy: 0.6646
Epoch 10/200
112/112 [==============================] - 13s 117ms/step - loss: 0.8936 -
accuracy: 0.6826 - val_loss: 1.0096 - val_accuracy: 0.6708
Epoch 11/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8889 -
accuracy: 0.6772 - val_loss: 0.9250 - val_accuracy: 0.6895
Epoch 12/200
112/112 [==============================] - 13s 117ms/step - loss: 0.9136 -
accuracy: 0.6731 - val_loss: 0.9352 - val_accuracy: 0.6845
Epoch 13/200
112/112 [==============================] - 13s 117ms/step - loss: 0.8822 -
accuracy: 0.6830 - val_loss: 0.9364 - val_accuracy: 0.6683
Epoch 14/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8657 -
accuracy: 0.6754 - val_loss: 1.2831 - val_accuracy: 0.6796
Epoch 15/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8783 -
accuracy: 0.6778 - val_loss: 0.9750 - val_accuracy: 0.6895
Epoch 16/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8706 -
accuracy: 0.6750 - val_loss: 0.8751 - val_accuracy: 0.6820
Epoch 17/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8751 -
accuracy: 0.6803 - val_loss: 0.9091 - val_accuracy: 0.6696
Epoch 18/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8561 -
accuracy: 0.6850 - val_loss: 0.9068 - val_accuracy: 0.6746
Epoch 19/200
112/112 [==============================] - 13s 118ms/step - loss: 0.8659 -
accuracy: 0.6779 - val_loss: 0.8600 - val_accuracy: 0.7007
Epoch 20/200
112/112 [==============================] - 13s 119ms/step - loss: 0.8650 -
accuracy: 0.6777 - val_loss: 0.9198 - val_accuracy: 0.6833
Epoch 21/200
112/112 [==============================] - 13s 120ms/step - loss: 0.8441 -
accuracy: 0.6830 - val_loss: 0.8623 - val_accuracy: 0.6995
```

```
Epoch 22/200
112/112 [==============================] - 13s 120ms/step - loss: 0.8116 -
accuracy: 0.6977 - val_loss: 0.8645 - val_accuracy: 0.6845
Epoch 23/200
112/112 [==============================] - 13s 119ms/step - loss: 0.8332 -
accuracy: 0.6892 - val_loss: 0.8085 - val_accuracy: 0.7007
Epoch 24/200
112/112 [==============================] - 14s 121ms/step - loss: 0.8324 -
accuracy: 0.6875 - val_loss: 0.8277 - val_accuracy: 0.7057
Epoch 25/200
112/112 [==============================] - 13s 120ms/step - loss: 0.8413 -
accuracy: 0.6866 - val_loss: 0.8172 - val_accuracy: 0.6958
Epoch 26/200
112/112 [==============================] - 14s 121ms/step - loss: 0.8126 -
accuracy: 0.6985 - val_loss: 0.9325 - val_accuracy: 0.7095
Epoch 27/200
112/112 [==============================] - 14s 121ms/step - loss: 0.8258 -
accuracy: 0.6864 - val_loss: 0.8770 - val_accuracy: 0.6870
Epoch 28/200
112/112 [==============================] - 14s 121ms/step - loss: 0.8074 -
accuracy: 0.6964 - val_loss: 0.9154 - val_accuracy: 0.6933
Epoch 29/200
112/112 [==============================] - 14s 122ms/step - loss: 0.8089 -
accuracy: 0.7006 - val_loss: 0.8323 - val_accuracy: 0.7045
Epoch 30/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7918 -
accuracy: 0.7043 - val_loss: 0.9212 - val_accuracy: 0.6559
Epoch 31/200
112/112 [==============================] - 14s 122ms/step - loss: 0.8070 -
accuracy: 0.7003 - val_loss: 0.9737 - val_accuracy: 0.6933
Epoch 32/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7916 -
accuracy: 0.7035 - val_loss: 0.8159 - val_accuracy: 0.7007
Epoch 33/200
112/112 [==============================] - 14s 122ms/step - loss: 0.8041 -
accuracy: 0.6978 - val_loss: 0.8055 - val_accuracy: 0.7082
Epoch 34/200
112/112 [==============================] - 14s 123ms/step - loss: 0.8065 -
accuracy: 0.6956 - val_loss: 0.8275 - val_accuracy: 0.6808
Epoch 35/200
112/112 [==============================] - 14s 123ms/step - loss: 0.8038 -
accuracy: 0.6990 - val_loss: 0.8591 - val_accuracy: 0.7082
Epoch 36/200
112/112 [==============================] - 14s 122ms/step - loss: 0.7874 -
accuracy: 0.7053 - val_loss: 0.9509 - val_accuracy: 0.7020
Epoch 37/200
112/112 [==============================] - 14s 124ms/step - loss: 0.7952 -
accuracy: 0.6999 - val_loss: 0.8112 - val_accuracy: 0.6920
```

```
Epoch 38/200
112/112 [==============================] - 14s 126ms/step - loss: 0.7893 -
accuracy: 0.6993 - val_loss: 0.8374 - val_accuracy: 0.7007
Epoch 39/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7742 -
accuracy: 0.7082 - val_loss: 0.8798 - val_accuracy: 0.6995
Epoch 40/200
112/112 [==============================] - 14s 123ms/step - loss: 0.8057 -
accuracy: 0.6957 - val_loss: 0.8328 - val_accuracy: 0.7057
Epoch 41/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7797 -
accuracy: 0.6963 - val_loss: 0.7877 - val_accuracy: 0.7294
Epoch 42/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7843 -
accuracy: 0.7007 - val_loss: 0.8063 - val_accuracy: 0.7120
Epoch 43/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7918 -
accuracy: 0.7033 - val_loss: 0.7875 - val_accuracy: 0.7182
Epoch 44/200
112/112 [==============================] - 14s 124ms/step - loss: 0.7762 -
accuracy: 0.7025 - val_loss: 0.7748 - val_accuracy: 0.7269
Epoch 45/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7783 -
accuracy: 0.7022 - val_loss: 0.8688 - val_accuracy: 0.6870
Epoch 46/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7548 -
accuracy: 0.7117 - val_loss: 0.8201 - val_accuracy: 0.7244
Epoch 47/200
112/112 [==============================] - 14s 124ms/step - loss: 0.7742 -
accuracy: 0.7042 - val_loss: 0.7960 - val_accuracy: 0.7107
Epoch 48/200
112/112 [==============================] - 14s 123ms/step - loss: 0.7844 -
accuracy: 0.7040 - val_loss: 0.8059 - val_accuracy: 0.7257
Epoch 49/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7708 -
accuracy: 0.6935 - val_loss: 0.8749 - val_accuracy: 0.7157
Epoch 50/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7555 -
accuracy: 0.7204 - val_loss: 0.8051 - val_accuracy: 0.7232
Epoch 51/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7639 -
accuracy: 0.7087 - val_loss: 0.7930 - val_accuracy: 0.7057
Epoch 52/200
112/112 [==============================] - 14s 127ms/step - loss: 0.7789 -
accuracy: 0.7062 - val_loss: 0.8257 - val_accuracy: 0.7219
Epoch 53/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7670 -
accuracy: 0.7098 - val_loss: 0.7983 - val_accuracy: 0.7145
```

```
Epoch 54/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7476 -
accuracy: 0.7123 - val_loss: 0.7883 - val_accuracy: 0.7195
Epoch 55/200
112/112 [==============================] - 14s 126ms/step - loss: 0.7662 -
accuracy: 0.7010 - val_loss: 0.8717 - val_accuracy: 0.6833
Epoch 56/200
112/112 [==============================] - 14s 126ms/step - loss: 0.7576 -
accuracy: 0.7128 - val_loss: 0.7957 - val_accuracy: 0.7070
Epoch 57/200
112/112 [==============================] - 14s 124ms/step - loss: 0.7596 -
accuracy: 0.7059 - val_loss: 0.7837 - val_accuracy: 0.7257
Epoch 58/200
112/112 [==============================] - 14s 124ms/step - loss: 0.7569 -
accuracy: 0.7192 - val_loss: 0.8030 - val_accuracy: 0.7344
Epoch 59/200
112/112 [==============================] - 14s 122ms/step - loss: 0.7524 -
accuracy: 0.7222 - val_loss: 0.8114 - val_accuracy: 0.7207
Epoch 60/200
112/112 [==============================] - 14s 120ms/step - loss: 0.7720 -
accuracy: 0.7122 - val_loss: 0.7853 - val_accuracy: 0.7344
Epoch 61/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7576 -
accuracy: 0.7143 - val_loss: 0.7952 - val_accuracy: 0.7157
Epoch 62/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7643 -
accuracy: 0.7057 - val_loss: 0.8149 - val_accuracy: 0.7157
Epoch 63/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7370 -
accuracy: 0.7168 - val_loss: 0.7699 - val_accuracy: 0.7157
Epoch 64/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7594 -
accuracy: 0.7141 - val_loss: 0.7774 - val_accuracy: 0.7045
Epoch 65/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7616 -
accuracy: 0.7150 - val_loss: 0.7753 - val_accuracy: 0.7095
Epoch 66/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7275 -
accuracy: 0.7203 - val_loss: 0.7550 - val_accuracy: 0.7170
Epoch 67/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7258 -
accuracy: 0.7252 - val_loss: 0.7633 - val_accuracy: 0.7195
Epoch 68/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7334 -
accuracy: 0.7233 - val_loss: 0.7578 - val_accuracy: 0.7182
Epoch 69/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7402 -
accuracy: 0.7220 - val_loss: 0.7874 - val_accuracy: 0.7307
```

```
Epoch 70/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7118 -
accuracy: 0.7353 - val_loss: 0.8025 - val_accuracy: 0.7294
Epoch 71/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7388 -
accuracy: 0.7121 - val_loss: 0.8481 - val_accuracy: 0.6970
Epoch 72/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7326 -
accuracy: 0.7161 - val_loss: 0.7614 - val_accuracy: 0.7269
Epoch 73/200
112/112 [==============================] - 13s 119ms/step - loss: 0.7317 -
accuracy: 0.7278 - val_loss: 0.8006 - val_accuracy: 0.7195
Epoch 74/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7353 -
accuracy: 0.7186 - val_loss: 0.7589 - val_accuracy: 0.7157
Epoch 75/200
112/112 [==============================] - 13s 119ms/step - loss: 0.7422 -
accuracy: 0.7124 - val_loss: 0.7527 - val_accuracy: 0.7257
Epoch 76/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7646 -
accuracy: 0.7066 - val_loss: 0.7761 - val_accuracy: 0.7257
Epoch 77/200
112/112 [==============================] - 13s 116ms/step - loss: 0.7428 -
accuracy: 0.7190 - val_loss: 0.7758 - val_accuracy: 0.7232
Epoch 78/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7221 -
accuracy: 0.7197 - val_loss: 0.7820 - val_accuracy: 0.7357
Epoch 79/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7413 -
accuracy: 0.7183 - val_loss: 0.8280 - val_accuracy: 0.7357
Epoch 80/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7213 -
accuracy: 0.7275 - val_loss: 0.7779 - val_accuracy: 0.7307
Epoch 81/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7245 -
accuracy: 0.7181 - val_loss: 0.7440 - val_accuracy: 0.7319
Epoch 82/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7347 -
accuracy: 0.7227 - val_loss: 0.7523 - val_accuracy: 0.7195
Epoch 83/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7295 -
accuracy: 0.7202 - val_loss: 0.7790 - val_accuracy: 0.7244
Epoch 84/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7240 -
accuracy: 0.7192 - val_loss: 0.7616 - val_accuracy: 0.7195
Epoch 85/200
112/112 [==============================] - 13s 117ms/step - loss: 0.7303 -
accuracy: 0.7290 - val_loss: 0.7619 - val_accuracy: 0.7394
```

```
Epoch 86/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7114 -
accuracy: 0.7245 - val_loss: 0.7987 - val_accuracy: 0.7244
Epoch 87/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7186 -
accuracy: 0.7226 - val_loss: 0.7797 - val_accuracy: 0.7307
Epoch 88/200
112/112 [==============================] - 13s 119ms/step - loss: 0.7287 -
accuracy: 0.7154 - val_loss: 0.7612 - val_accuracy: 0.7120
Epoch 89/200
112/112 [==============================] - 13s 118ms/step - loss: 0.7134 -
accuracy: 0.7351 - val_loss: 0.7508 - val_accuracy: 0.7344
Epoch 90/200
112/112 [==============================] - 13s 119ms/step - loss: 0.7501 -
accuracy: 0.7063 - val_loss: 0.7504 - val_accuracy: 0.7170
Epoch 91/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7495 -
accuracy: 0.7162 - val_loss: 0.7452 - val_accuracy: 0.7269
Epoch 92/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7218 -
accuracy: 0.7257 - val_loss: 0.7495 - val_accuracy: 0.7232
Epoch 93/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7414 -
accuracy: 0.7146 - val_loss: 0.7364 - val_accuracy: 0.7219
Epoch 94/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7163 -
accuracy: 0.7276 - val_loss: 0.7588 - val_accuracy: 0.7382
Epoch 95/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7021 -
accuracy: 0.7331 - val_loss: 0.8055 - val_accuracy: 0.7120
Epoch 96/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7200 -
accuracy: 0.7318 - val_loss: 0.7916 - val_accuracy: 0.7195
Epoch 97/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7301 -
accuracy: 0.7196 - val_loss: 0.7673 - val_accuracy: 0.7170
Epoch 98/200
112/112 [==============================] - 14s 125ms/step - loss: 0.7304 -
accuracy: 0.7279 - val_loss: 0.7531 - val_accuracy: 0.7319
Epoch 99/200
112/112 [==============================] - 14s 122ms/step - loss: 0.7252 -
accuracy: 0.7270 - val_loss: 0.7405 - val_accuracy: 0.7269
Epoch 100/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7088 -
accuracy: 0.7243 - val_loss: 0.7609 - val_accuracy: 0.7107
Epoch 101/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7379 -
accuracy: 0.7261 - val_loss: 0.7798 - val_accuracy: 0.7282
```

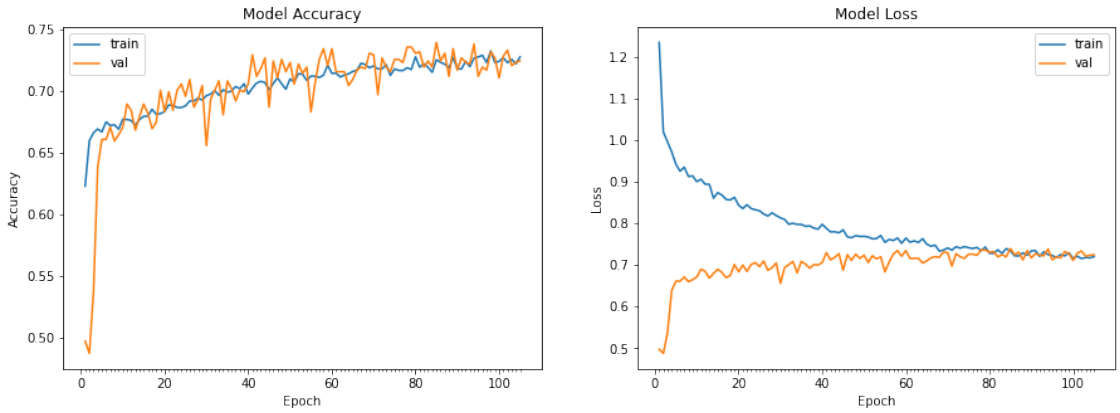
```
Epoch 102/200
112/112 [==============================] - 13s 120ms/step - loss: 0.7146 -
accuracy: 0.7207 - val_loss: 0.7621 - val_accuracy: 0.7332
Epoch 103/200
112/112 [==============================] - 14s 122ms/step - loss: 0.7010 -
accuracy: 0.7283 - val_loss: 0.7822 - val_accuracy: 0.7207
Epoch 104/200
112/112 [==============================] - 14s 122ms/step - loss: 0.7150 -
accuracy: 0.7200 - val_loss: 0.8356 - val_accuracy: 0.7232
Epoch 105/200
112/112 [==============================] - 14s 121ms/step - loss: 0.7152 -
accuracy: 0.7325 - val_loss: 0.7582 - val_accuracy: 0.7244
```

15 Step 14: Model Evaluation

Use the helper function **plot_model_history** from the **exercise_functions** to plot the model learning history.

[25]:
```
plot_model_history(history)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:54:
MatplotlibDeprecationWarning: Passing the minor parameter of set_xticks()
positionally is deprecated since Matplotlib 3.2; the parameter will become
keyword-only two minor releases later.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:62:
MatplotlibDeprecationWarning: Passing the minor parameter of set_xticks()
positionally is deprecated since Matplotlib 3.2; the parameter will become
keyword-only two minor releases later.
```



15.0.1 Evaluate the model on the validation and test data.

Use the method **evaluate** of your keras Sequential model.

[26]:
```
model.evaluate(X_test,y_test)
```

```
63/63 [==============================] - 0s 6ms/step - loss: 0.6985 - accuracy:
0.7304
```

[26]:
```
[0.6985369920730591, 0.7304043769836426]
```

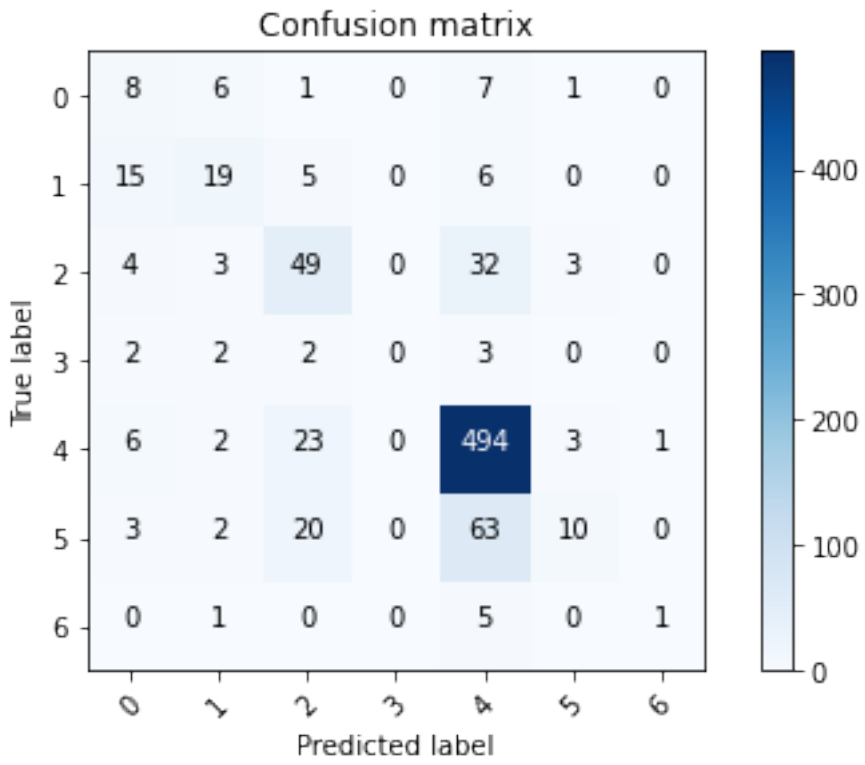
Make the prediction on the validation dataset. Plot the confusion matrix and check the missclassified count of each type.

[27]:
```
# Predict the values from the validation dataset
Y_pred_classes1 = model.predict(X_val)
# Convert predictions classes to one hot vectors
Y_pred_classes = to_categorical(Y_pred_classes1, num_classes = 7,dtype='int')
# Convert validation observations to one hot vectors
Y_true = to_categorical(y_val, num_classes = 7,dtype='int')
# compute the confusion matrix
# CM = confusion_matrix (np.argmax(y_test,axis=1), np.
 ↪argmax(Y_test_pred,axis=1))

confusion_mtx = confusion_matrix(np.argmax(y_val,axis=1), np.
 ↪argmax(Y_pred_classes1,axis=1))
```

15.0.2 Plot the confusion matrix on the test dataset.

Use the **plot__confusion__function** from the **exercise__functions** module.

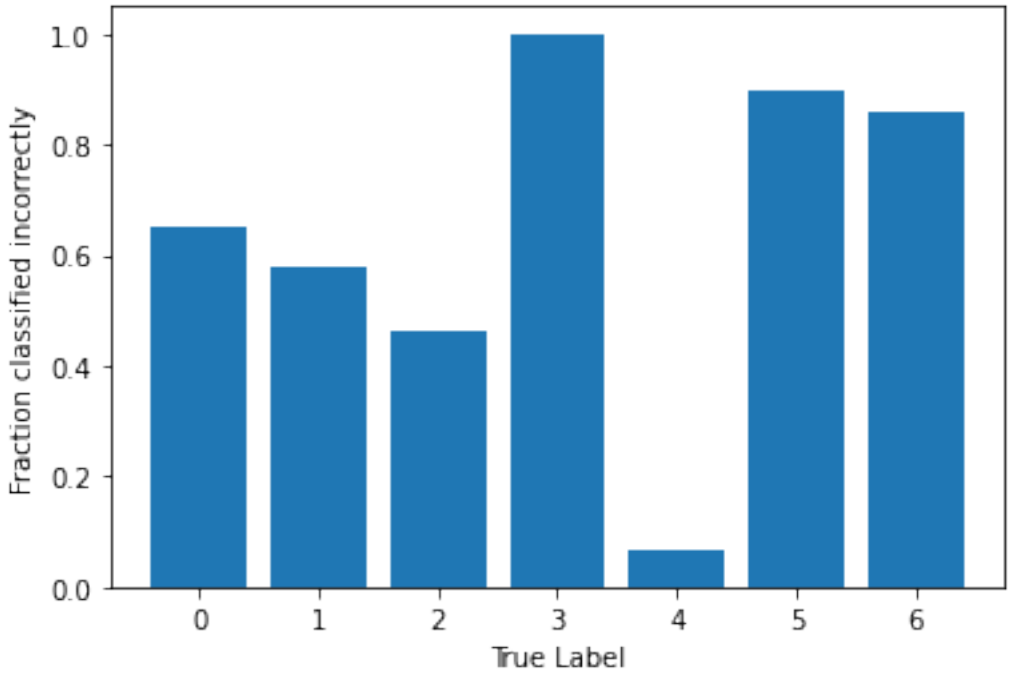
[28]:
```
plot_confusion_matrix(confusion_mtx, classes = range(7))
```



Now, lets see which category has most incorrect predictions

[29]:
```
label_frac_error = 1 - np.diag(confusion_mtx) / np.sum(confusion_mtx, axis=1)
plt.bar(np.arange(7),label_frac_error)
plt.xlabel('True Label')
plt.ylabel('Fraction classified incorrectly')
```

[29]:
```
Text(0, 0.5, 'Fraction classified incorrectly')
```



###When you are happy with your model, let it train longer, say for 100 to 150 epochs, or until it starts to overfit. Then proceed to conclusions here below.

16 Conclusion

Provide in your report analysis of the data. Give various distributions you could plot: the location of the mole, the patient's age and sex, the cancer type prevalence, … Comment these distributions. What cancer is the most frequent one. Is the age distribution the same for all? …

The Melanocytic nevi and vascular lesions has the same frequency at any age, and the nevi is the most frequent one.

16.0.1 Conclude your report by answering to the following questions?

1. **Did your model converge?**

Yes, it converged, we can see that thanks to the plot of the loss and accuracy functions

2. **Did you not overfit?**

We can see no overfitting, we have no gap enlargment in the model loss and the validation is not decreasing.

3. **What was the accuracy you could reach?**

From the test set, we got the accuracy of 0.7304 and a model loss of 0.6985

4. **What were the most misclassified classes? What was the moest easily classified class?**

The most misclassified class is the number 3 (Dermatofibroma) with almost every one of them being misclassified and the 4 (melanocytic nevi) one is the best with almost no misclassifications. Which is makes sense, because the most frequent is the nevi and the least frequent one is the dermatofibroma.

5. **Is your model competitive to human expert having ~77% accuracy? Any suggestion to improve the accuracy?**

The model can be considered competitive to humans because its accuracy is quite close to 77% (73%), and we can still improve the accuracy with bigger models, but due to computational time being larger, we didn't. We noticed that the data is quite unbalanced, so the most frequent cancer type will have a bigger impact onto the loss function. So we can improve the results by balancing the data or by using a weighted loss function.