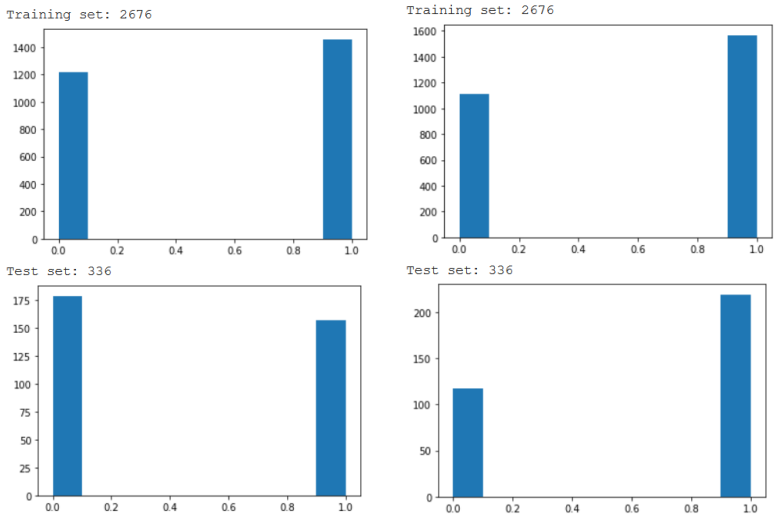
Contents

4 Task 2: From Scratch CNN

We developed different models of neural networks to work with the given dataset from scratch and we did different test to find the better hyperparameters to build the final model. This section is divided in three parts: in the first one we describe the main preprocessing applied to the data before the training of the models. Then we show how we built a model for classifying the dataset images between *mass* abnormality and *calfication* abnormality. In the last part, we describe the model built to classify between *benign* and *malignant* abnormality.

4.1 Data Preprocessing

Starting from the numpy arrays of the dataset images we were given, first of all we deleted all the samples associated with the *baseline patch* label. Since this labeled images were placed in the even positions of the dataset, we just selected all the odd-index samples al discard the others. This is done both for the training data and the test data.
Then we aggregated the labels according to the classification that we needed to do: in the *Task* 2.1 the classes *mass benign* and *mass malignant* were aggregated in a unique class *mass*, and so for the *calcification* classes. On the other hand, in the *Task* 2.2 we aggregated *mass benign* and *cacification benign* in the *benign* class, and the other labels in the *malignant* class.



(a) *Task* 2.1: 0 for *mass*, 1 for *calcification*

(b) *Task* 2.2: 0 for *benign*, 1 for *malignant*

Figure 1: Label distribution

For the first case, we can see from the figure that the labels are pretty much equally distributed. Instead, in the second case the dataset is a little more unbalanced towards the *benign* class.
Finally, we reshaped all the training images tensors in a single *numpy* array, that has been then normalized. The same has been done for the test images. With regards to the labels, they has been categorizated so that is possible to use the categorical loss function in the model.

4.2 Task 2.1: Mass-Calcification discriminator

We first tried to build a simple CNN, with some convolutional and max-pooling layers, and two final dense layers. We trained and tested the model several times, changing the hyperparameters to find a good model for out problem. The final architecture that has been used for the most relevant tests is shown in figure 2. It is made of five convolutional layers, with an increasing number of kernels. Each layer is followed by a max-pooling of $2x2$, except for the last one so that the input of the dense layer is not too reduced. The activation function used in the layers is *ReLu*.

After asserting the architecture, the model has been trained with different numbers of *hidden units* for the first fully connected layer, but also for different *batch sizes* and different *output functions*. In all the tests, the conclusion has been that the network was over-training, and this is the reason for which we have developed a second model, adding a data augmentation preprocessing.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 5, 5, 128)         73856
_________________________________________________________________
flatten (Flatten)            (None, 3200)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1638912
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1026
=================================================================
Total params: 1,778,786
Trainable params: 1,778,786
Non-trainable params: 0
```

Figure 2: First model of the CNN from scratch

In the figure 3, we present the results in term of accuracy and loss, but also the consufion matrix (fig. 4), related to a test of the model with 512 hidden units, a batch size of 50, and the *softmax* function for the output layer. The used optimizer is Adam, with the deafult learning rate of 0.001. As we said before, the gap between the training and validation graphs show that the model is over-training: even if the predictions on the test set are accurate, we cannot rely on this network as it is.

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

network.fit(train_images, train_labels, epochs=100, batch_size=50,
        validation_split=0.2, shuffle=True, callbacks=[callback])
```

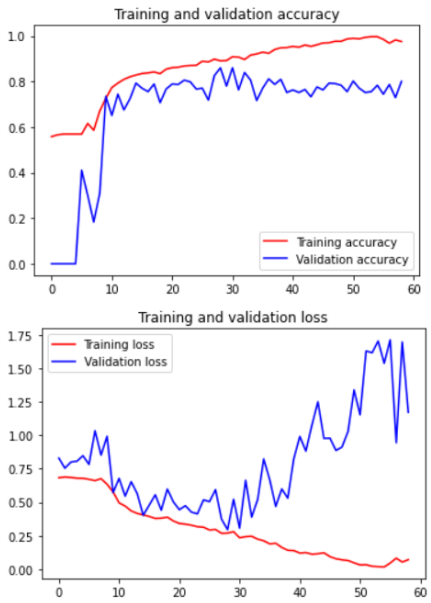


Figure 3: Accuracy and loss graphs for the first model training

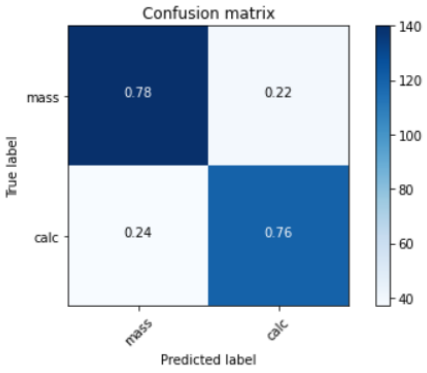


Figure 4: Normalized confusion matrix for the test set predictions

4.2.1 Data augmentation

To overcome the problem of over-fitting, we augmented our training using the *ImageDataGenerator* class of *keras*. The main variations we adopted were random rotation up to 40 degrees, random vertical and horizontal shifts with a range of 0.2, random zoom with a range of 0.2, random shear with a range of 0.2 and nearest fill model. To set these parameters we referred to the data in the papers we studied in the initial phase of the project. Before applying these variations to the train set, we splitted it to get the validation set separately.

```
train_set, val_set, train_classes, val_classes =
        train_test_split( train_images, train_labels, test_size=0.2)

train_datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=20,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

train_generator = train_datagen.flow(train_set,train_classes,batch_size=50)
```

We trained again our model with the augmented training set and with the validation set, and the results were more satisfying, having a mean accuracy around 80%. At this point, we wanted to have a more accurate analysis on some hyperparameters so we did several tests changing their values to get a model as good as possible.