Task 2 Statistical Learning 2020

Group 3

May 30th, 2020

Contents

1 Question 1&2 **2**

2 Question 3 **3**

3 Question 4 **5**

4 Question 5 **6**

5 Question 6 & 7 **7**

6 Question 8 **10**

7 Question 9 **13**

8 Question 10 **17**

9 Question 11 **17**

10 Question 12 **19**

```r
# libraries
library(keras)
library(rstatix)
library(tidyverse)
library(caret)
library(ggplot2)
library(gridExtra)
library(tfruns)
library(kableExtra)
library(caret)
library(plyr)


# set seed for reproductibility
set.seed(42)
tensorflow::tf$random$set_seed(42)

#load saved data
load("data_q1to8.RData")
```

1 Question 1&2

Normalize the images. Split the dataset into 500 training /100 validation /100 test. Try to balance the two classes.

We create a function `random_files` to automatically split the data in the required folder structure and in training, validation and test sets.

We then execute our function and define the training, validation and test folders.

```r
# execute function
random_files('rxtorax/normal','rxtorax', 1, 250, 50, 50, pattern = "png$|PNG$")
random_files('rxtorax/effusion','rxtorax', 2, 250, 50, 50, pattern = "png$|PNG$")
```

```r
# define training, validation and test folders
train_dir<-"rxtorax/train"
validation_dir<-"rxtorax/validation"
test_dir<-"rxtorax/test"
```

We then use `image_data_generator` to normalize our data and `flow_images_from_directory` to define batches, resize our images and keep a unique channel. We start with a batch size of 25.

```r
b_size <- 25

train_datagen <- image_data_generator(rescale = 1/255)
train_generator <- flow_images_from_directory(
  train_dir,
  train_datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary"
)

validation_datagen <- image_data_generator(rescale = 1/255)
validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary"
)

test_datagen <- image_data_generator(rescale = 1/255)
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary",
  classes = c("effusion","normal"),
  shuffle = FALSE
)
# Now we have the images in the required format: 64x64 with a unique channel
```

2 Question 3

Implement a Convolutional Neural Network (CNN).

We then define a Convolutional Neural Network, with 2 convolution layers, each with its own pooling layer. Then we add a flatten layer and a dropout layer. At the bottom of the network we have two fully connected layers with 128 and 32 nodes respectively. Finally, the output layer has one unit and a sigmoid activation function. The total number of trainable parameters is 816 673.

```r
model <- keras_model_sequential() %>%
  # first convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = "relu",input_shape = c(64, 64, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # second convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.4) %>%
  # Outputs from dense layer are projected onto output layer
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d (Conv2D)                     (None, 62, 62, 32)              320
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 31, 31, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 29, 29, 32)              9248
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 14, 14, 32)              0
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 6272)                    0
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 6272)                    0
## ________________________________________________________________________________
## dense (Dense)                       (None, 128)                     802944
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 32)                      4128
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 1)                       33
## ================================================================================
## Total params: 816,673
## Trainable params: 816,673
## Non-trainable params: 0
## ________________________________________________________________________________
```

We compile and fit the model with a binary crossentropy loss function, compute the accuracy and use 13 epochs. We save our model in the attached file `cnn_model_batch25.h5`.

```r
# Compile the model
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)

# Fit the model
history <- model %>% fit_generator(
  train_generator,
  steps_per_epoch = 500/b_size,
  epochs = 13,
  validation_data = validation_generator,
  validation_steps = 100/b_size
)
model %>% save_model_hdf5("cnn_model_batch25.h5")
```

We obtain the following loss and accuracy values across epochs on the training and validation sets. We see the training and validation values stagnate and coincide after the 10th epoch. More than 13 epochs led to overfitting.

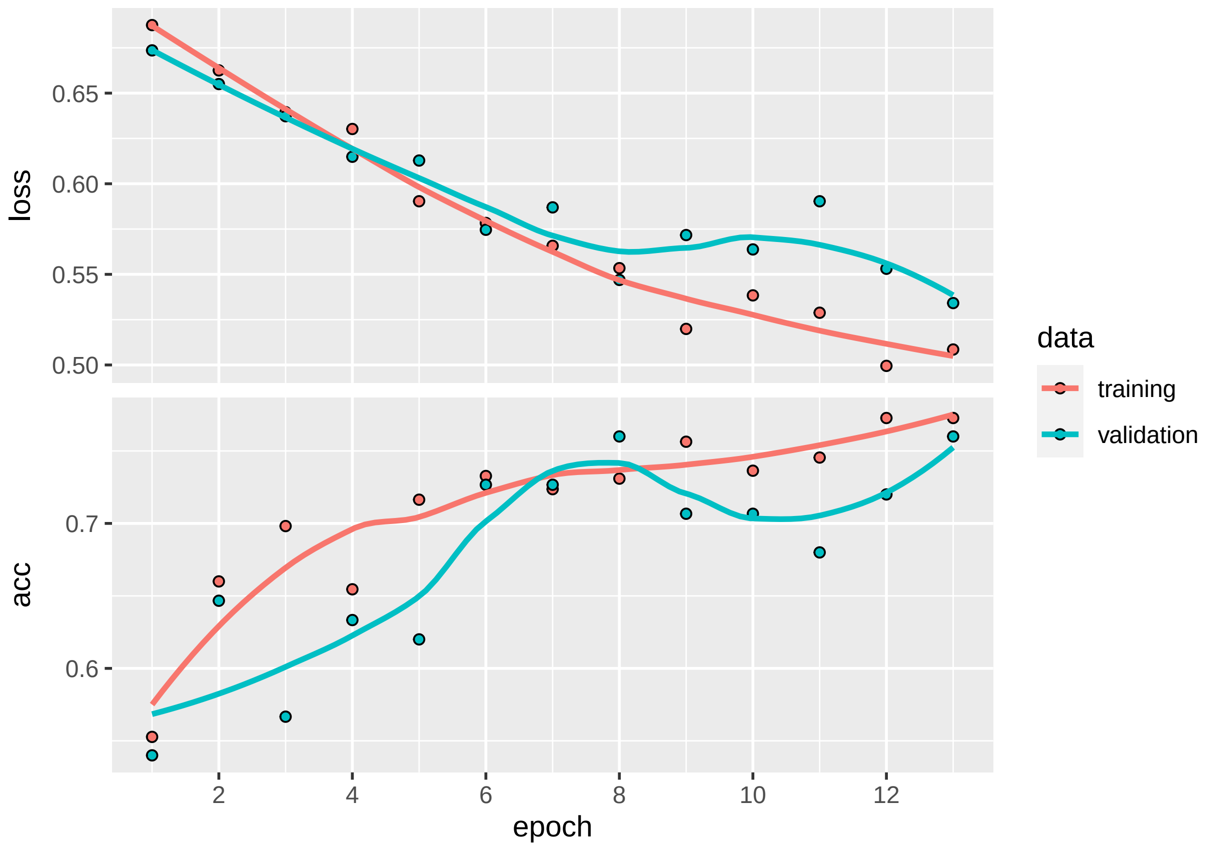

Figure 1: CNN accuracy on training and validation set

Table 1: Batch size tuning results - loss and accuracy

Batch size	Train. loss	Val. loss	Train. acc.	Val. acc
50	0.508	0.773	0.534	0.760
35	0.488	0.772	0.527	0.756
25	0.518	0.741	0.575	0.720

3 Question 4

Tune the hyperparameter batch_size checking the values 25, 35, 50.

We use `tfruns` to tune the batch size hyperparameter.

```r
b_s <- c(25,35,50)
for (b in b_s) {
  training_run("cnn_flags.r", flags = c(batch_size = b))
}
tuning_res <- tfruns::ls_runs()
write_csv(tuning_res,"tfruns_res_batchsize.csv")
```

In Table 1, we present the output of the tuning. The loss and accuracy values are very close to each other, the differences are probably not significant. We find a batch size of 50 gives the best accuracy value on the validation set so we set our batch size at 50, refit our model with that size and save our tuned model in the attached file `cnn_model_bestbatch.h5`.

```r
set.seed(43)
tensorflow::tf$random$set_seed(43)

b_size <- best_batch

# initialise the model
model_bestbatch <- keras_model_sequential() %>%
  # first convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = "relu",input_shape = c(64, 64, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # second convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.4) %>%
  # Outputs from dense layer are projected onto output layer
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

# Compile the model
model_bestbatch %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
```

```r
# Fit the model
history_bestbatch <- model_bestbatch %>% fit_generator(
  train_generator,
  steps_per_epoch = 500/b_size,
  epochs = 13,
  validation_data = validation_generator,
  validation_steps = 100/b_size
)
model_bestbatch %>% save_model_hdf5("cnn_model_bestbatch.h5")
```

4 Question 5

Assess the performance of the CNN predicting the categories of test images and obtain the confusion matrix.

We use `predict` to predict the class of our test images and gather the results in a dataframe.

```r
predict <- model_bestbatch %>% predict_generator(
  test_generator,
  steps = 100/b_size)

stat_df <- tibble(predict[1:100,], test_generator$filenames, test_generator$classes)
  # assign prediction probability for filenames
colnames(stat_df) <- c(
    "predict_proba",
    "filename",
    "class")
stat_df <- stat_df %>%
  mutate(predict_proba = as.double(predict_proba)) %>%
         mutate(predicted_class = ifelse(predict_proba > 0.5, 1, 0)) %>%
         mutate(predicted_class = as.integer(predicted_class)) %>%
  mutate(label_name = ifelse(predicted_class == 0, "effusion", "normal"))

test_accuracy <- mean(stat_df$class==stat_df$predicted_class)
```

We obtain an accuracy of 0.78 on the test set. The confusion matrix is:

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 44 16
##          1  6 34
##
##                Accuracy : 0.78
##                  95% CI : (0.6861, 0.8567)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 7.953e-09
##
##                   Kappa : 0.56
##
##  Mcnemar's Test P-Value : 0.05501
##
```

```
##             Sensitivity : 0.8800
##             Specificity : 0.6800
##          Pos Pred Value : 0.7333
##          Neg Pred Value : 0.8500
##              Prevalence : 0.5000
##          Detection Rate : 0.4400
##    Detection Prevalence : 0.6000
##       Balanced Accuracy : 0.7800
##
##        'Positive' Class : 0
##
```

5 Question 6 & 7

Re-fit the CNN including data augmentation. Was the use of augmentation an improvement? Compare these two CNN models.

We use `image_data_generator` to augment the training set of images and fit our model again with all the hyperparameters equal to our model without augmentation and the same seed. We save our augmented model in the attached file `cnn_model_bestbatch_augmented.h5`.

```r
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE
)

set.seed(43)
tensorflow::tf$random$set_seed(43)

b_size <- best_batch

train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary"
)

validation_datagen <- image_data_generator(rescale = 1/255)
validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary"
)

test_datagen <- image_data_generator(rescale = 1/255)
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  color_mode = "grayscale",
  target_size = c(64, 64),
  batch_size = b_size,
  class_mode = "binary",
  classes = c("effusion","normal"),
  shuffle = FALSE
)

# initialise the model
model_bestbatch_aug <- keras_model_sequential() %>%
  # first convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = "relu",input_shape = c(64, 64, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # second convolutional hidden layer and max pooling
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.4) %>%
  # Outputs from dense layer are projected onto output layer
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")


# Compile the model
model_bestbatch_aug %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)

# Fit the model
history2 <- model_bestbatch_aug %>% fit_generator(
  train_generator,
  steps_per_epoch = 500/b_size,
  epochs = 13,
  validation_data = validation_generator,
  validation_steps = 100/b_size
)
model_bestbatch_aug %>% save_model_hdf5("cnn_model_bestbatch_augmented.h5")

# Prediction
predict_bestbatch_aug <- model_bestbatch_aug %>% predict_generator(
  test_generator,
  steps = 100/b_size)
```

```
stat_df_bestbatch_aug <- tibble(predict_bestbatch_aug, test_generator$filenames, test_generator$classes)
  # assign prediction probability for filenames
colnames(stat_df_bestbatch_aug) <- c(
    "predict_proba",
    "filename",
    "class")
stat_df_bestbatch_aug <- stat_df_bestbatch_aug %>%
  mutate(predict_proba = as.double(predict_proba)) %>%
           mutate(predicted_class = ifelse(predict_proba > 0.5, 1, 0)) %>%
           mutate(predicted_class = as.integer(predicted_class)) %>%
  mutate(label_name = ifelse(predicted_class == 0, "effusion", "normal"))

test_accuracy <- mean(stat_df_bestbatch_aug$class==stat_df_bestbatch_aug$predicted_class)
```

We compare the loss and accuracy on the training and validation sets before and after augmentation in Fig. 2. We see that the validation loss and accuracy metrics stagnate from epoch 10 in both cases eventhough the validation accuracy shows some variance after augmentation. The accuracy level reached before augmentation was higher than the level reached after augmentation: about 71% against 67%.

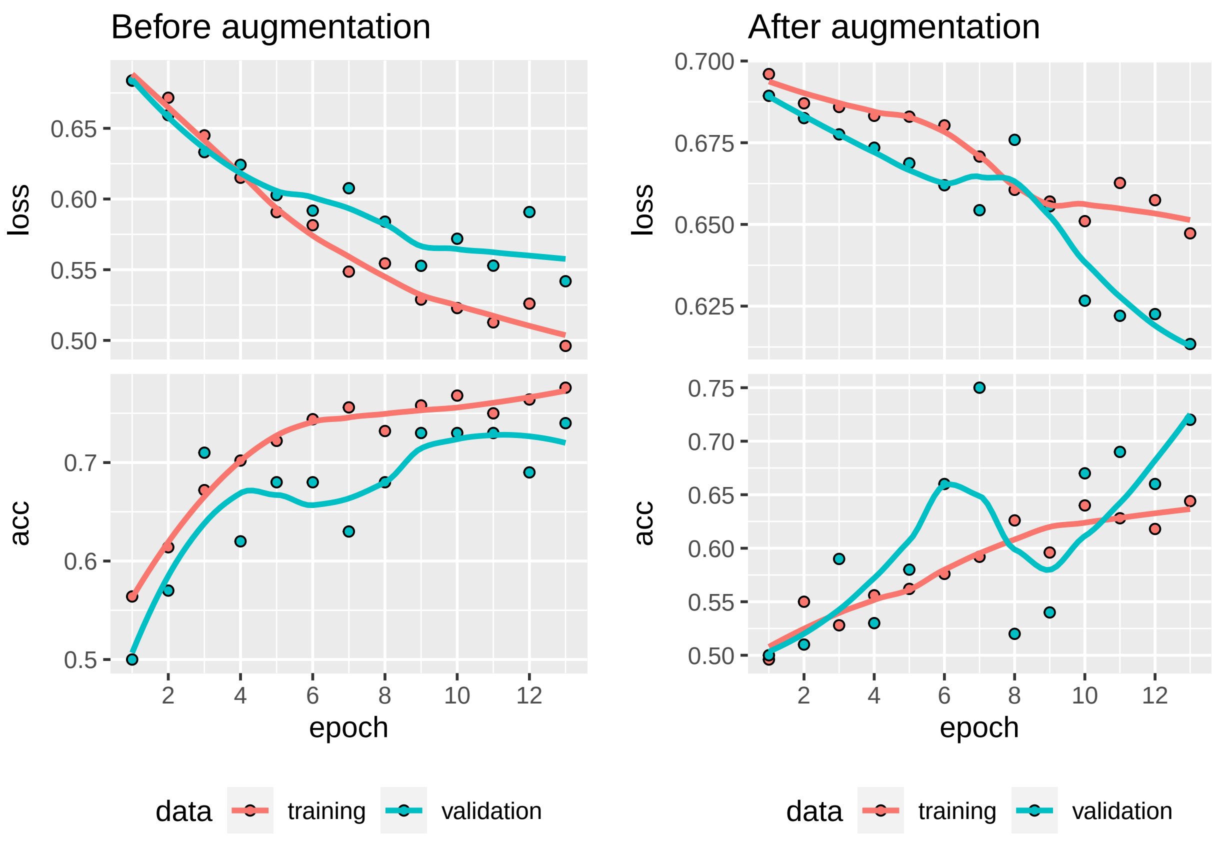

Figure 2: CNN loss and accuracy on training and validation set before and after augmentation

We also compare the two models on the test sample. For the CNN after augmentation we obtain an accuracy of 0.76 on the test set and the confusion matrix is:

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction  0  1
##          0 37 11
##          1 13 39
##
##                Accuracy : 0.76
##                  95% CI : (0.6643, 0.8398)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 9.05e-08
##
##                   Kappa : 0.52
##
##  Mcnemar's Test P-Value : 0.8383
##
##             Sensitivity : 0.7400
##             Specificity : 0.7800
##          Pos Pred Value : 0.7708
##          Neg Pred Value : 0.7500
##              Prevalence : 0.5000
##          Detection Rate : 0.3700
##    Detection Prevalence : 0.4800
##       Balanced Accuracy : 0.7600
##
##        'Positive' Class : 0
##
```

The accuracy on the test set was slightly higher for the model before augmentation at 0.78. We note however that the two values are very close to each other. They each belong to the 95% confidence interval of the other. The Kappa measures are close to each other as well. Overall, the augmentation did not improve the model.

6 Question 8

Implement a convolutional autoencoder (CAE) network.

We use `image_data_generator`to normalize our data and `flow_images_from_directory` to split our images in batches of size 50 with mode `input`.

We define a convolutionnal autoencoder with 3 convolutionnal layers in the encoder and 4 convolutionnal layers in the decoder. The number of filters is a decresing function of the layer order in the encoder and we start from 64 filters. The pooling size is set at 2 in the first two layers and then 4 in the third one. The decoder structure is symetric to the encoder one, with an additional layer to obtain an image of the same dimension as the input. The total number of trainable parameters is 89,729.

```
#### Convolutional Encoder
filters_start <- 64
p_size <- 4

model_enc <- keras_model_sequential()
model_enc %>%
  layer_conv_2d(filters = filters_start, kernel_size = c(2,2), padding ="same",
                activation = "relu",input_shape = c(64, 64, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2,2),padding ="same") %>%

  layer_conv_2d(filters = filters_start, kernel_size = c(2,2), padding ="same",
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2),padding ="same") %>%

  layer_conv_2d(filters = filters_start/2, kernel_size = c(2,2), padding ="same",
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(p_size,p_size), padding ="same")
summary(model_enc)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 64, 64, 64)              320
## ________________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)      (None, 32, 32, 64)              0
## ________________________________________________________________________________
## conv2d_3 (Conv2D)                   (None, 32, 32, 64)              16448
## ________________________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)      (None, 16, 16, 64)              0
## ________________________________________________________________________________
## conv2d_4 (Conv2D)                   (None, 16, 16, 32)              8224
## ________________________________________________________________________________
## max_pooling2d_4 (MaxPooling2D)      (None, 4, 4, 32)                0
## ================================================================================
## Total params: 24,992
## Trainable params: 24,992
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
#### Convolutional Decoder

model_dec <- keras_model_sequential()
model_dec %>%
  layer_conv_2d(filters = filters_start/2, kernel_size = c(3,3),
                activation = "relu", padding = "same",
                input_shape = c(64/(2*2*p_size), 64/(2*2*p_size), filters_start/2))  %>%
  layer_upsampling_2d(size = c(2,2))  %>%

layer_conv_2d(filters = filters_start, kernel_size = c(3,3),
                activation = "relu", padding = "same")  %>%
  layer_upsampling_2d(size = c(2,2))  %>%

  layer_conv_2d(filters = filters_start, kernel_size = c(3,3),
                activation = "relu", padding = "same")  %>%
  layer_upsampling_2d(size = c(p_size,p_size))  %>%
  layer_conv_2d(filters = 1, kernel_size = c(1,1),
                activation = "relu")
summary(model_dec)
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_5 (Conv2D)                   (None, 4, 4, 32)                9248
## ________________________________________________________________________________
## up_sampling2d (UpSampling2D)        (None, 8, 8, 32)                0
## ________________________________________________________________________________
## conv2d_6 (Conv2D)                   (None, 8, 8, 64)                18496
## ________________________________________________________________________________
## up_sampling2d_1 (UpSampling2D)      (None, 16, 16, 64)              0
## ________________________________________________________________________________
## conv2d_7 (Conv2D)                   (None, 16, 16, 64)              36928
## ________________________________________________________________________________
## up_sampling2d_2 (UpSampling2D)      (None, 64, 64, 64)              0
## ________________________________________________________________________________
## conv2d_8 (Conv2D)                   (None, 64, 64, 1)               65
## ================================================================================
## Total params: 64,737
## Trainable params: 64,737
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
#### Autoencoder
model_auto<-keras_model_sequential()
model_auto %>%model_enc%>%model_dec
```

```
## Model
## Model: "sequential_3"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## sequential_1 (Sequential)           (None, 4, 4, 32)                24992
## ________________________________________________________________________________
## sequential_2 (Sequential)           (None, 64, 64, 1)               64737
## ================================================================================
## Total params: 89,729
## Trainable params: 89,729
## Non-trainable params: 0
## ________________________________________________________________________________
```

We use the mean squared error as loss function and 5 epochs.

```
# set seed for reproductibility
set.seed(42)
tensorflow::tf$random$set_seed(42)

model_auto %>% compile(
  loss = "mean_squared_error",
  #optimizer = optimizer_rmsprop(),
  optimizer = "adam",
  metrics = c("mean_squared_error")
)

# Fit the model
history_auto <- model_auto %>% fit_generator(
  train_generator,
  steps_per_epoch = 500/b_size,
  epochs = 5,
  validation_data = validation_generator,
  validation_steps = 100/b_size
)
model_auto %>% save_model_hdf5("auto_model.h5")
save.image("data_q1to8.RData")
```

We see that after 5 epochs, the training and validation metrics are similar and stagnate. More epochs would lead to overfitting.

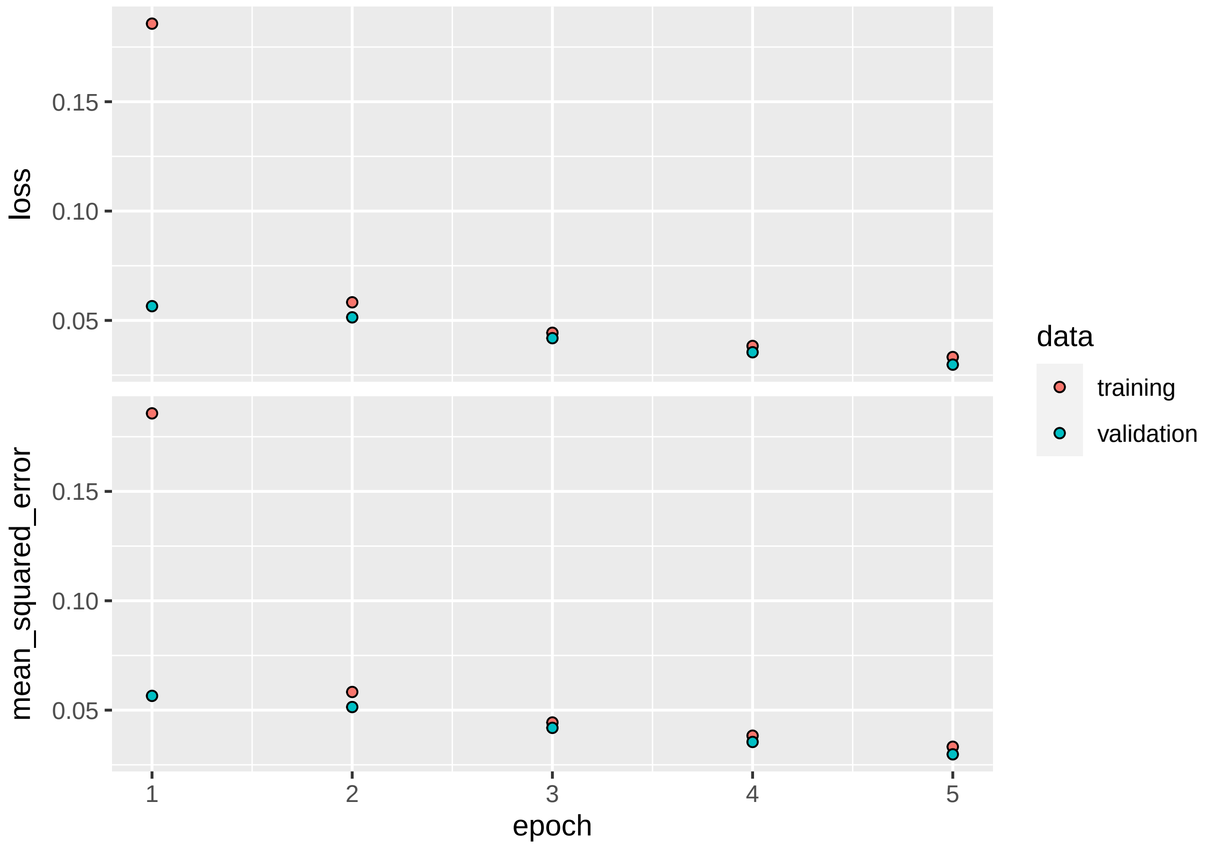

Figure 3: Autoencoder MSE on training and validation set

7 Question 9

Tune the more compact layer (z layer) with three configurations(width x heigth x filters) what you can free choose. To evaluate the z layer performance use this flattened layer as input in a random forest(or boosting) algorithm to classify the images.

We use `tfruns` to run our convolutionnal autoencoder with different configurations of the encoder to obtain three different dimensions of the compact layer Z: (4,4,32), (4,4,64) and (8,8,64).

```
training_run("cae_flags.R", flags = c(filters_nb_start = 64,
                                      pooling_size = 4))
training_run("cae_flags.R", flags = c(filters_nb_start = 128,
                                      pooling_size = 4))
training_run("cae_flags.R", flags = c(filters_nb_start = 128,
                                      pooling_size = 2))
tuning_res_dim <- tfruns::ls_runs() %>% filter(script == "cae_flags.R")
write_csv(tuning_res_dim,"tfruns_res_dim.csv")
```

Table 2: Z dimension tuning results - loss and MSE

Dimension Z	Filter param	Pooling size	Train. loss	Val. loss	Train. MSE.	Val. MSE
(8,8,64)	128	2	0.012	0.012	0.011	0.011
(4,4,64)	128	4	0.028	0.028	0.024	0.024
(4,4,32)	64	4	0.031	0.031	0.028	0.028

On the training and validation set, the compact layer Z of dimension (8,8,64) gives the lowest MSE as shown in Table 2.

In the runs, we extracted and flattened the compact layer Z for each dimension for the training and the test sets and saved it in Rdata files. We fit an adaboost model with stumps on the training Z for each combination. For computation time and power, we ran the code in a Google Colab virtual machine with the attached notebook `Adaboost on encoder output.ipynb`.

```r
combina <-c("filter128_pool2","filter128_pool4","filter64_pool4")
for (com in combina){
  load(paste0("Conv_Encod_Flat_",com,".RData"))
  training <- data.frame(y=as.factor(y_radio_train),predict_enc_train)
  nzv <- nearZeroVar(training)
  training <- training[, -nzv]
  #test <- data.frame(y=as.factor(y_radio_test),predict_enc_test)
  training$y <- revalue(training$y, c("0"="effusion", "1"="normal"))
  #test$y <- revalue(test$y, c("0"="effusion", "1"="normal"))

  control <- trainControl(
    method = "repeatedcv",
    number = 5,
    repeats = 3,
    classProbs = TRUE
    allowParallel = TRUE,
    summaryFunction = twoClassSummary
  )

  grid <- expand.grid(
    .interaction.depth = 1,
    .n.trees = c(500, 1500, 3000),
    .shrinkage = .01,
    .n.minobsinnode = 10
  )

  metric <- "Accuracy"

  stump_adaboost <- train(y ~ .,
    data = training,
    method = "gbm",
    bag.fraction = 0.5,
    distribution = "adaboost",
    trControl = control,
    tuneGrid = grid,
    verbose = FALSE,
    metric = metric
  )

  save(stump_adaboost,file=paste0("stump_adaboost",com,".Rdata"))
}
```

We compare the accuracy obtained with the different dimensions in the training set in Fig.4 where we plot the distributions of the ROC, Specificity and Sensitivity accross resamples. We see that in all the cases the ROC is poor, on average the dimensions (4,4,64) abd (8,8,64) have larger ROC tha (4,4,32) but the boxplots between the three are not well separated. The sensitivity and specificity are comparable in the 3 cases.

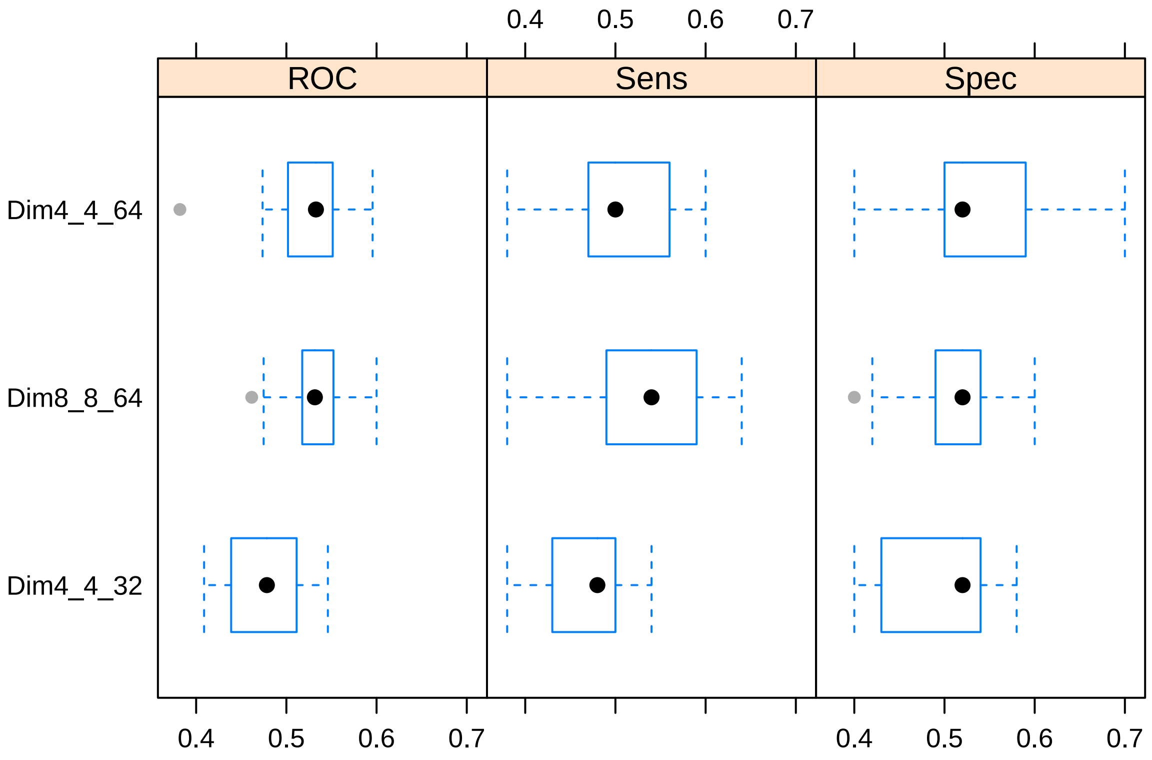

Figure 4: Adaboost model accuracy with different dimensions of compact layer Z

On the test set with dimension (4,4,32) we obtain the following confusion matrix and a ROC of 0.511.

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction effusion normal
##   effusion       24     24
##   normal         26     26
##
##                Accuracy : 0.5
##                  95% CI : (0.3983, 0.6017)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.5398
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : 0.8875
##
```

```
##              Sensitivity : 0.48
##              Specificity : 0.52
##           Pos Pred Value : 0.50
##           Neg Pred Value : 0.50
##               Prevalence : 0.50
##           Detection Rate : 0.24
##     Detection Prevalence : 0.48
##        Balanced Accuracy : 0.50
##
##         'Positive' Class : effusion
##
```

On the test set with dimension (4,4,64) we obtain the following confusion matrix and a ROC of 0.408.

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction effusion normal
##   effusion       21     24
##   normal         29     26
##
##                Accuracy : 0.47
##                  95% CI : (0.3694, 0.5724)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.7579
##
##                   Kappa : -0.06
##
##  Mcnemar's Test P-Value : 0.5827
##
##             Sensitivity : 0.4200
##             Specificity : 0.5200
##          Pos Pred Value : 0.4667
##          Neg Pred Value : 0.4727
##              Prevalence : 0.5000
##          Detection Rate : 0.2100
##    Detection Prevalence : 0.4500
##       Balanced Accuracy : 0.4700
##
##        'Positive' Class : effusion
##
```

On the test set with dimension (8,8,64) we obtain the following confusion matrix and a ROC of 0.545.

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction effusion normal
##   effusion       28     24
##   normal         22     26
##
##                Accuracy : 0.54
##                  95% CI : (0.4374, 0.6402)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.2421
##
```

```
##                   Kappa : 0.08
##
##  Mcnemar's Test P-Value : 0.8828
##
##             Sensitivity : 0.5600
##             Specificity : 0.5200
##          Pos Pred Value : 0.5385
##          Neg Pred Value : 0.5417
##              Prevalence : 0.5000
##          Detection Rate : 0.2800
##    Detection Prevalence : 0.5200
##       Balanced Accuracy : 0.5400
##
##        'Positive' Class : effusion
##
```

On the test set, all performance metrics are similar and accuracies 95% confidence intervals overlap. However, (8,8,64) has the largest accuracy and also the largest ROC. We choose (8,8,64) as our best dimension.

8 Question 10

Once the best performing z layer configuration has bee selected, perform a statistcal test to detect in which variables (nodes) there are significant differences between the two classes of images.

We extract again the compact layer for the training set with dimension (8,8,64). For each variable, we run a t-test comparing the average value between the samples from the two classes and adjust for false discovery ratio with the Benjamini-Hochberg method.

```r
load("Conv_Encod_Flat_filter128_pool2.RData")
training <- data.frame(y=as.factor(y_radio_train),predict_enc_train)
nzv <- nearZeroVar(training)
training <- training[, -nzv]
training$y <- revalue(training$y, c("0"="effusion", "1"="normal"))


training.long <- training %>%
  pivot_longer(-y, names_to = "variables", values_to = "value")

stat.test <- training.long %>%
  group_by(variables) %>%
  t_test(value ~ y, comparisons = list(c("effusion","normal"))) %>%
  adjust_pvalue(method = "BH") %>%
  add_significance()
write_csv(stat.test,"t_test_results.csv")
```

In Table 3, we list the top 10 variables with the smallest p-values. In total we have 79 variables with significant difference in mean with a p-value cut off of 5% before adjustment. However, none of the variable has a significant difference in mean after adjustment for multi-testing.

9 Question 11

Visualize the results of the previous item using a Volcano plot.

Table 3: T-test on compact layer variables means between classes

variables	statistic	p	p.adj
X3798	-2.92	0.004	0.999
X2350	-2.91	0.004	0.999
X2355	-2.82	0.005	0.999
X2726	-2.77	0.006	0.999
X3412	-2.76	0.006	0.999
X2582	-2.73	0.007	0.999
X2070	-2.66	0.008	0.999
X4052	-2.64	0.009	0.999
X1029	-2.57	0.010	0.999
X1428	-2.56	0.011	0.999

We compute the log fold change ($\log_2$) between the two classes and plot the variabes p-values in a log10 scale against the log fold change in Fig.5. The volcano plot shows a group of variables has significant p-values but among those very little have a large fold change: only three variables have a log fold change larger than 1 in absolute value. A lot of our significant unajusted p-values might actually highlight very small changes in means. This volcano plot tells us that overall our variables have little difference betweenn the two classes, which is in line with the fact thay we find they have little classification power.

```
stat.test <- stat.test %>% ungroup() %>% arrange(variables)

lfc<- training.long %>%
  dplyr::group_by(variables,y) %>%
  dplyr::summarise(log_mean = log2(mean(value)))

lfc <- pivot_wider(lfc, id_cols=variables, names_from = y, values_from = log_mean) %>%
  mutate(lfc=effusion-normal) %>%
  arrange(variables)

stat.test$lfc <- lfc$lfc
stat.test$log10pval <- -log10(stat.test$p)
```

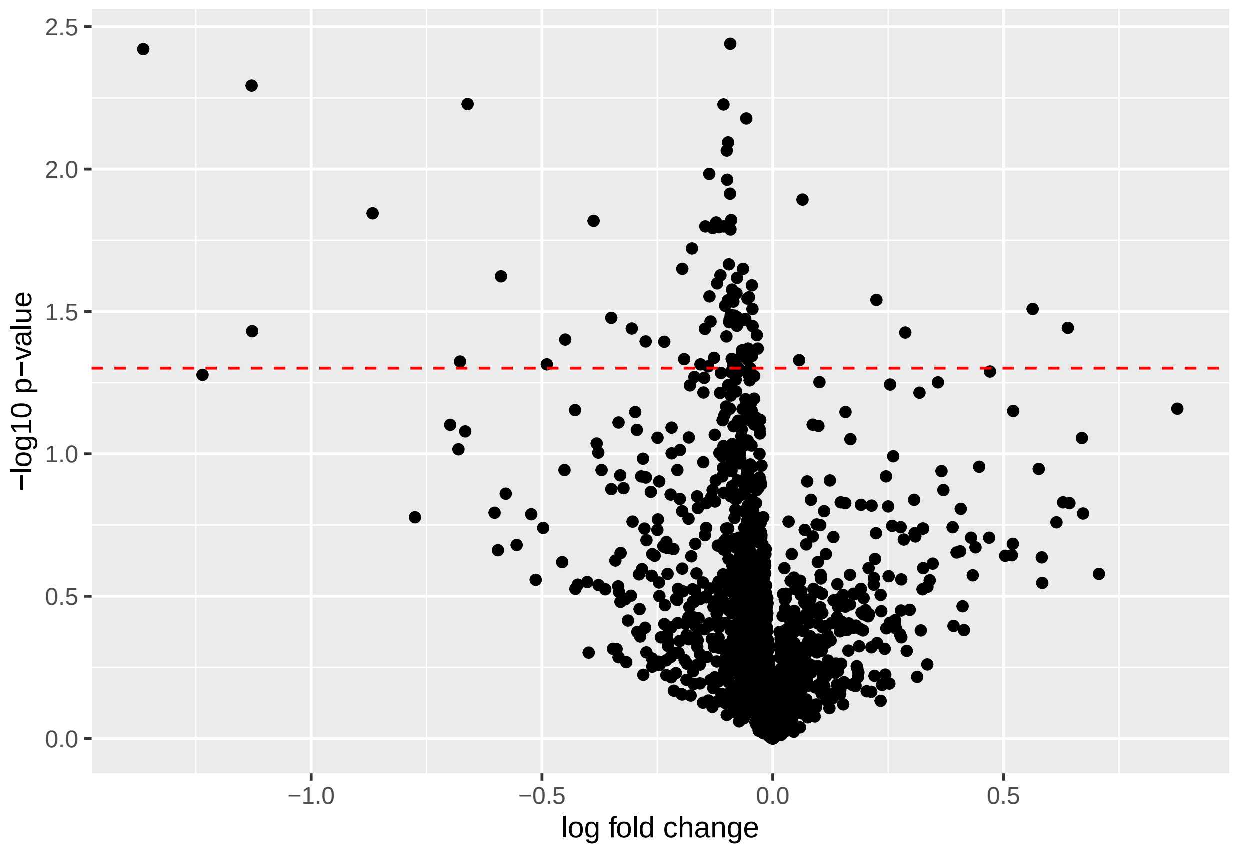

Figure 5: Volcano plot of differences in means of Z varibales

10 Question 12

CNNs are mostly used for processing images. The input can generally be 3 channels (for color images) or 1 channel (for grayscale images). The 1 or 3 channels form the volume of the input image (using 2D matrixes for each channel) and is reflected in the layers of the network. The network uses filters in its layers to find patterns and one can use different lens' sizes (3x3, 5x5, etc, generally odd numbers). Another important element of this type networks if the pooling layer, which is a sort of non-linear down-sampling layer. The network allows for network reconfiguration in each training epoch in order to improve accuracy and contribute to gradient calculation speed. On the output side of the CNN, after all convolutions are concluded, output is generated via a fully connected dense layer.

CAEs consist of two CNNs connected back to back, where one is referred to as the encoder (input network) and the other decoder (output network). The task of the encoder is to produce a data representation of the input data in a lower dimensional space (occupying less bytes) using a non-linear transformation. The task of the decoder is to re-interpret the encoded information and produce a result that attempts to resemble the original / input information, with as less possible loss of data as possible. Generally this is used also for images. The benefit of having two networks work separately is that, eventually, when all the information has been encoded, the two networks can be disconnected and use a different approach in place of the decoder.

In this work, the CNN classification has proved to be much more accurate than the classification based on the compact layer output by the encoder. Such results must be considered carefully given the limitations of our models and tuning process. Our CNN has much more parameters than our CAE which was limited by design. We also experienced a lot of variance in performance metrics on validation and test sets which complicated the tuning of hyperparameters. With more samples, more time in our hands and more computing power to perform a extensive grid search with good estimates of performance metrics, but also more experience, the results would have been certainly different. A lesson learned here is that given the difficulties in training a good netwrok, relying on pre-trained neural networks as base structure is a very reasonable option when training ressources (time, computing power, samples) are scarce.