Task 2 Statistical Learning 2020

Group 3

May 27th, 2020

Contents


```
# 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 images. Split the dataset into 500 train /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 or depending on the user requirements.

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

```
# 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$|PNG$")
```

```
# 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 an unique channel. We start with a batch size of 25.

```
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 a 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.

```
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`.

```
# 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.

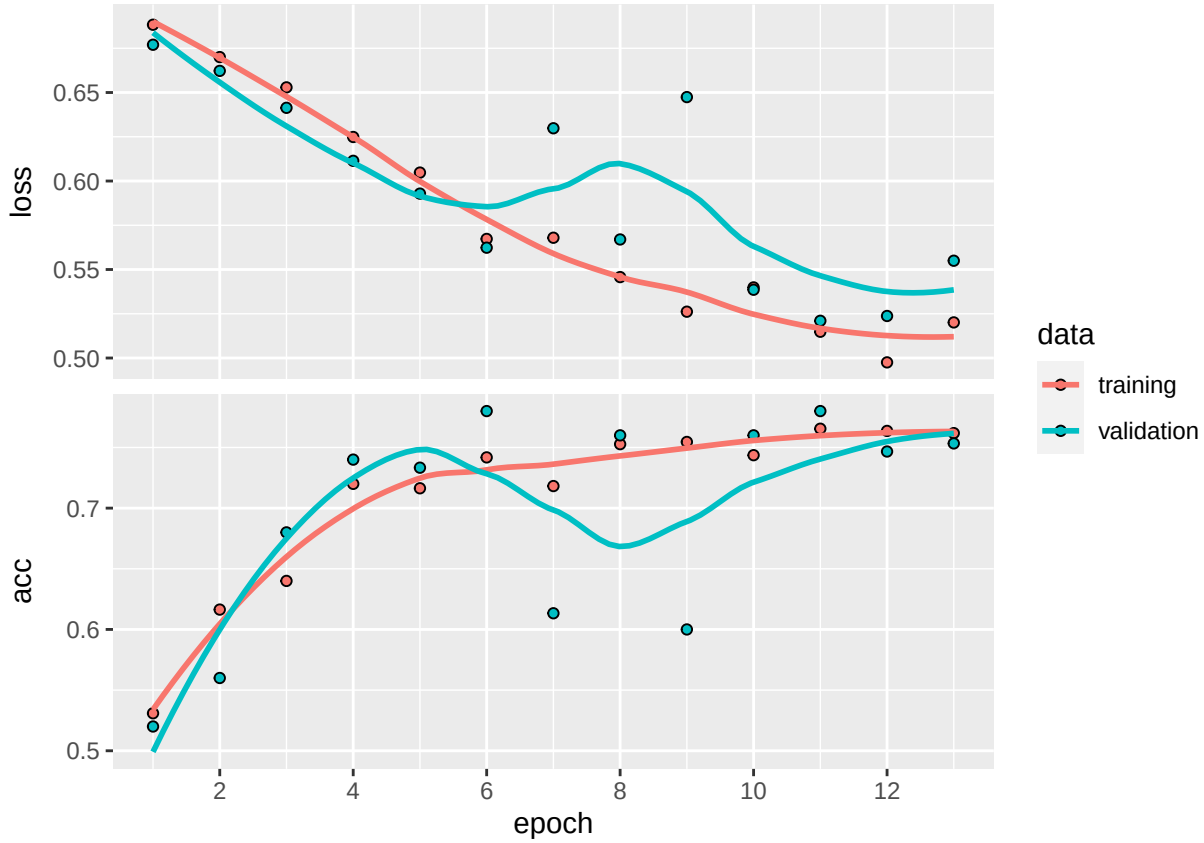

Figure 1: CNN accuracy on training and validation set

Table 1: Batch size tuning results - loss and accuracy

Batch size	Train. loss	Val. loss	Train. acc.	Val. acc
50	0.520	0.762	0.555	0.753
35	0.507	0.756	0.556	0.726
25	0.475	0.779	0.567	0.720

3 Question 4

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

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

```
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. We find a batch size of 50 gives the best loss and accuracy values.

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`.

```
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")
)
```

```
# 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.

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

stat_df <- as_tibble(cbind(predict[1:100,], test_generator$filenames, test_generator$classes)) %>%
  # assign prediction probability for filenames
  rename(
    predict_proba = V1,
    filename = V2,
    class = V3
  ) %>%
  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 0.72 on the test set. The confusion matrix is:

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 45 21
##          1  5 29
##
##                Accuracy : 0.74
##                  95% CI : (0.6427, 0.8226)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 8.337e-07
##
##                   Kappa : 0.48
##
##  Mcnemar's Test P-Value : 0.003264
##
```

```
##             Sensitivity : 0.9000
##             Specificity : 0.5800
##          Pos Pred Value : 0.6818
##          Neg Pred Value : 0.8529
##              Prevalence : 0.5000
##          Detection Rate : 0.4500
##    Detection Prevalence : 0.6600
##       Balanced Accuracy : 0.7400
##
##        '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`.

```
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"
)

# 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 <- as_tibble(cbind(predict_bestbatch_aug, test_generator$filenames, test_generato
  # assign prediction probability for filenames
  rename(
    predict_proba = V1,
    filename = V2,
    class = V3
  ) %>%
  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 accuracy stagnates from epoch 8 in both cases, but the level reached before augmentation was higher than the level reached after augmentation: about 71% against 67%.

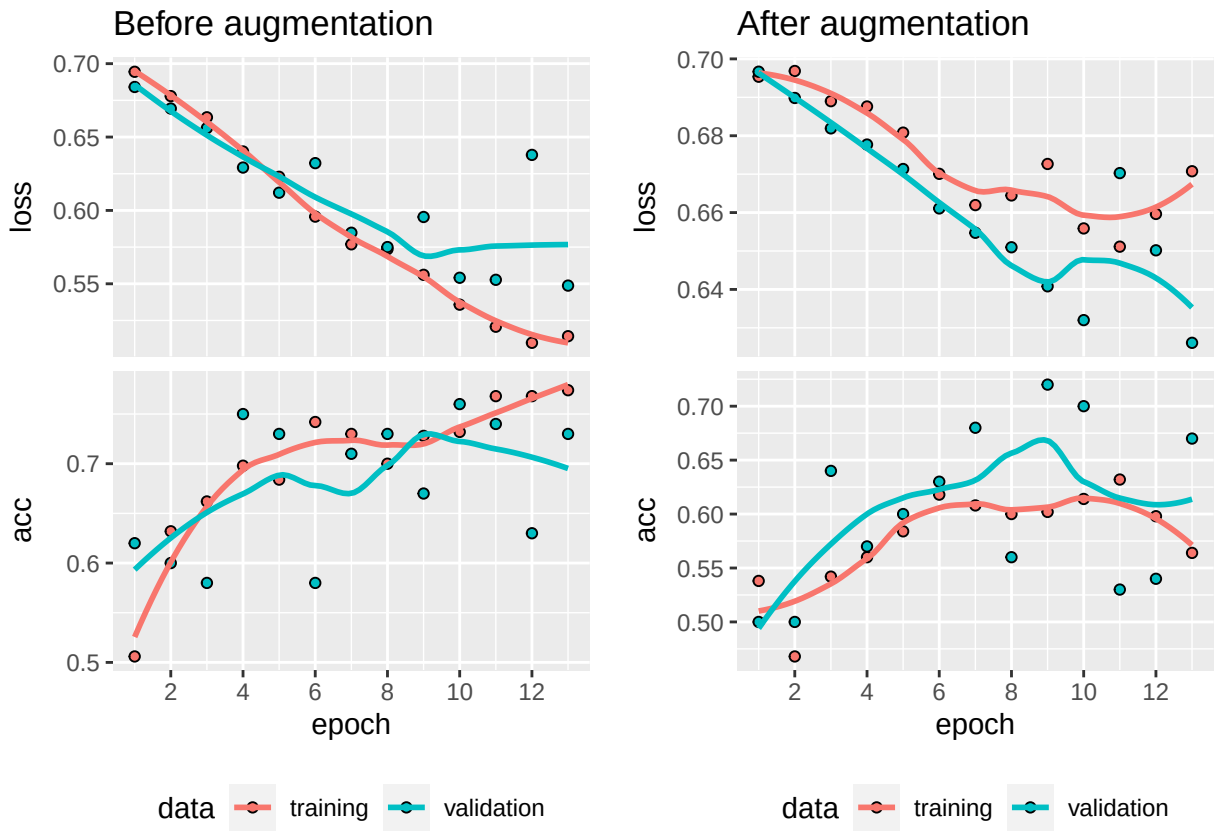

Figure 2: CNN loss and accuracy on training and validation set before and after augmentation

We also compare the two models on the test sample. For the CNN after augmentation we obtain an accuracy of 0.72 on the test set and the confusion matrix is:

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 30  8
##          1 20 42
##
##                Accuracy : 0.72
##                  95% CI : (0.6213, 0.8052)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 6.29e-06
##
##                   Kappa : 0.44
##
##  Mcnemar's Test P-Value : 0.03764
##
##             Sensitivity : 0.6000
##             Specificity : 0.8400
##          Pos Pred Value : 0.7895
##          Neg Pred Value : 0.6774
##              Prevalence : 0.5000
##          Detection Rate : 0.3000
```

```
##    Detection Prevalence : 0.3800
##       Balanced Accuracy : 0.7200
##
##        'Positive' Class : 0
##
```

The accuracy on the test set was slightly higher for the model before augmentation at 0.74. 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.

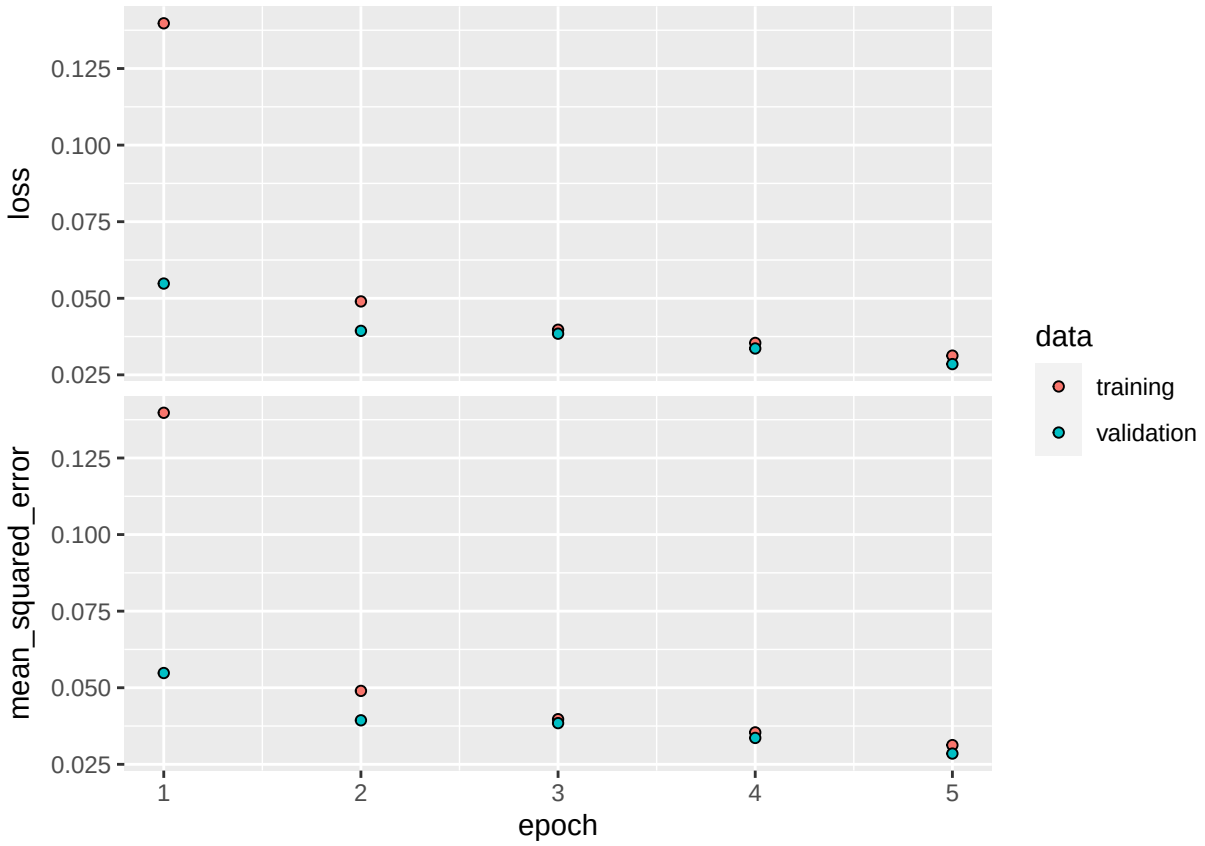

Figure 3: Autoencoder MSE on training and validation set

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")
```

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`.

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	0.026	0.026	0.022	0.022
(4,4,32)	64	4	0.032	0.032	0.030	0.030

```
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, however a slightly larger ROC is achieved with the dimension (4,4,64). The sensitivity and specificity are comparable in the 3 cases.

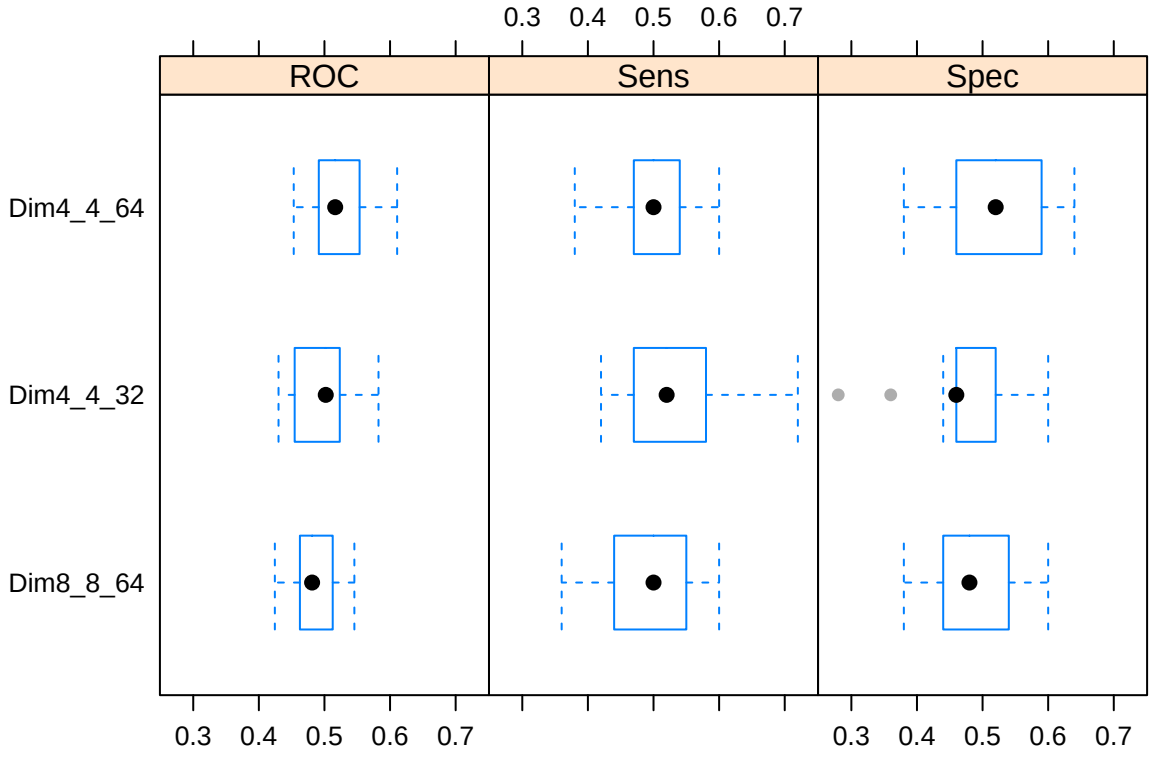

Figure 4: Adaboost model accuracy with different dimensions of compact layer Z

On the test set with dimension (4,4,32) we obtain the following confusion matrix and a ROC of 0.368.

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction effusion normal
##   effusion       21     32
##   normal         29     18
##
##                Accuracy : 0.39
##                  95% CI : (0.294, 0.4927)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.9895
##
##                   Kappa : -0.22
##
##  Mcnemar's Test P-Value : 0.7979
##
##             Sensitivity : 0.4200
##             Specificity : 0.3600
##          Pos Pred Value : 0.3962
##          Neg Pred Value : 0.3830
##              Prevalence : 0.5000
##          Detection Rate : 0.2100
##    Detection Prevalence : 0.5300
##       Balanced Accuracy : 0.3900
##
##        'Positive' Class : effusion
```

```
##
```

On the test set with dimension (4,4,64) we obtain the following confusion matrix and a ROC of 0.462.

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction effusion normal
##   effusion       26     26
##   normal         24     24
##
##                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 : 0.52
##             Specificity : 0.48
##          Pos Pred Value : 0.50
##          Neg Pred Value : 0.50
##              Prevalence : 0.50
##          Detection Rate : 0.26
##    Detection Prevalence : 0.52
##       Balanced Accuracy : 0.50
##
##        'Positive' Class : effusion
##
```

On the test set with dimension (8,8,64) we obtain the following confusion matrix and a ROC of 0.515.

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction effusion normal
##   effusion       27     26
##   normal         23     24
##
##                Accuracy : 0.51
##                  95% CI : (0.408, 0.6114)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.4602
##
##                   Kappa : 0.02
##
##  Mcnemar's Test P-Value : 0.7751
##
##             Sensitivity : 0.5400
##             Specificity : 0.4800
##          Pos Pred Value : 0.5094
##          Neg Pred Value : 0.5106
##              Prevalence : 0.5000
##          Detection Rate : 0.2700
```

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

variables	statistic	p	p.adj
X2337	3.11	0.002	0.989
X1825	2.97	0.003	0.989
X3233	2.66	0.008	0.989
X2500	-2.42	0.016	0.989
X1953	2.32	0.021	0.989
X353	2.29	0.022	0.989
X2362	2.25	0.025	0.989
X2562	-2.24	0.026	0.989
X1781	-2.22	0.027	0.989
X2345	2.18	0.030	0.989

```
##    Detection Prevalence : 0.5300
##       Balanced Accuracy : 0.5100
##
##        'Positive' Class : effusion
##
```

On the test set, models with dimensions (4,4,64) and (8,8,64) have very similar accuracy, however the ROC is larger for (8,8,64). We choose (8,8,64) as our best dimension.

8 Question 10

Once it was selected the best performing z layer configuration, to detect in which variables (nodes) there are significant differences between the two classes of images, a statistical test will be performed to determine those that are significant.

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.

In Table 3, we list the top 10 variables with the smallest p-values. In total we have 31 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 Volcano plot.

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 one variable has 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 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)
```

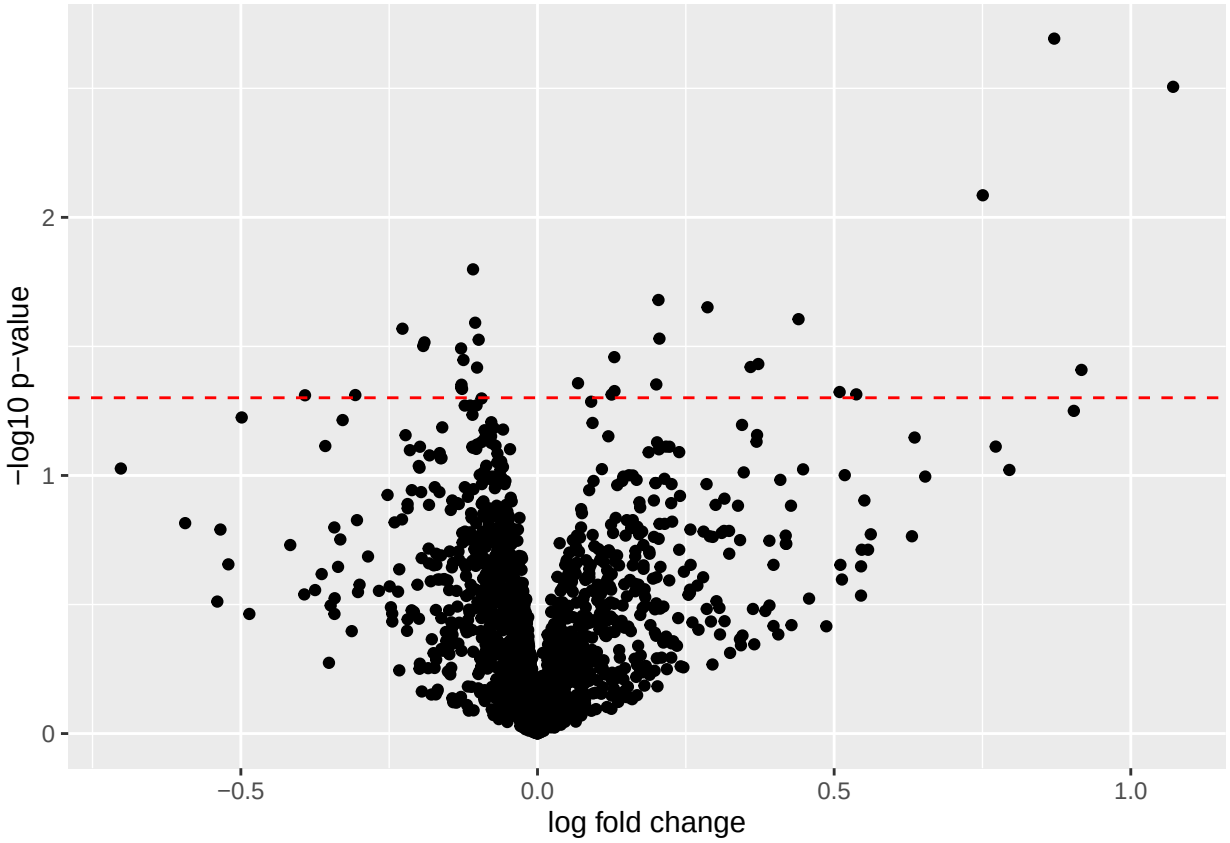

Figure 5: Volcano plot of differences in means of Z varibales