Task 2

Statistical Learning with Deep Artificial Neural Networks

Alexander J Ohrt

13. mai. 2022

Contents

The objective of this task is diagnosis of chest X-ray images. More specifically, we want to be able to separate between normal X-ray images and X-ray images with effusion.

We are given two data sets of images; 500 normal X-ray images and 500 X-ray images with effusion. The images are selected from the public NIH ChestXray14 data set. This data will be used to construct models for diagnosis of future (unseen) images.

1 Preprocessing

First we separate the data into two sets; a training set containing $\frac{2}{3}$ of the data and a test set containing the remaining $\frac{1}{3}$. In order to do this, we assume that both sets of images are saved in a directory called "images". This directory contains the 500 normal X-ray images in a sub directory called "normal", and the 500 X-ray images with effusion in a sub directory called "effusion". The images are given in png format.

We create three (or two, without the validation directory, depending on the parameter `DA.valid`) new sub directories within the "images" directory; one called "test", which contains the randomly sampled test data, one called "train", which contains the randomly sampled training data and one called "valid", which contains the randomly sampled validation data, which is sampled from the training data. Thus, this directory of images is constructed by splitting the training data into a "pure" training set and a validation data set, where the former takes $\frac{4}{5}$ of the values and the latter takes $\frac{1}{5}$ of the values (thus, in total data set terms, the former takes $\frac{8}{15}$ and the latter takes $\frac{2}{15}$ of the complete data set. The remaining $\frac{1}{3}$ is still left to the test set). The directory with validation data is only created if `params$DA.valid` is set to `FALSE`. The reason behind this parameter will be explained later. By the way, the naming of the parameter is shorthand for `Data Augmentation on Validation Data`. Perhaps this will make the point of the parameter clearer immediately. We assume that these directories have not been created on the computer earlier (this is very important in order for the data to be divided correctly between the directories!). We could implement some sort of deletion of old directories if they already exist, but I have chosen to skip this for now.

Thus, if the parameter `params$DA.valid` is set to `FALSE` we create the three different directories, which then will be used throughout the assignment. On the other hand, if the parameter is set to `TRUE`, we only create the train ($\frac{2}{3}$) and the test ($\frac{1}{3}$) directories of images. In this case, the validation set is created from the training set inside of R, while compiling the code. Thus, the splits of the data remains the same in both cases (train $\frac{8}{15}$, validation $\frac{2}{15}$ and test $\frac{1}{5}$), but there are some differences in the images. This will become clear once we start building the models.

```r
set.seed(params$seed) # Set seed to 1234, as chosen in params.
image.path <- "./images"
normal.path <- paste0(image.path, "/normal")
effusion.path <- paste0(image.path, "/effusion")

# New path for training and test images.
train.path <- paste0(image.path, "/train") # New path for training images.
test.path <- paste0(image.path, "/test") # New path for test images.

if(!params$DA.valid){
  valid.path <- paste0(image.path, "/valid") # New path for validation images.
}

# Sub directories of the above paths, for training and test images, split up into "normal" and "effusio
train.normal.path <- paste0(train.path, "/normal") # New path for normal train images .
train.effusion.path <- paste0(train.path, "/effusion") # New path for effusion train images.
test.normal.path <- paste0(test.path, "/normal") # New path for normal test images.
test.effusion.path <- paste0(test.path, "/effusion") # New path for effusion test images.

if(!params$DA.valid){
  valid.normal.path <- paste0(valid.path, "/normal") # New path for normal validation images.
  valid.effusion.path <- paste0(valid.path, "/effusion") # New path for effusion validation images.
}


# We collect the names of all the images.
normal.image.names <- list.files(normal.path)
effusion.image.names <- list.files(effusion.path)

# We create the two new directories. If they already exist, a warning is printed.
dir.create(train.path)
dir.create(test.path)

if(!params$DA.valid){
  dir.create(valid.path)
}

# We also create "effusion" and "normal" sub directories inside the new "train" and "test" directories.
# This is needed to get images for models later.
dir.create(train.normal.path)
dir.create(test.normal.path)
dir.create(train.effusion.path)
dir.create(test.effusion.path)

if(!params$DA.valid){
  dir.create(valid.normal.path)
  dir.create(valid.effusion.path)
}

# We fill the newly created sub folders with randomly sampled data.
# We know that the images are named (for example) "normal0.png" and "effusion0.png",
# with numbers ranging from 0 to 499. We confirm that this is the case below.
all(sort(parse_number(normal.image.names)) == 0:499)
```

```
#> [1] TRUE
```

```r
all(sort(parse_number(effusion.image.names)) == 0:499)
```

```
#> [1] TRUE
```

```r
# We sample from 0:499, in order to choose training numbers.
train.numbers <- sample(0:499, size = 2/3*500)
# Find testing numbers as well, simply using a set difference.
test.numbers <- setdiff(0:499, train.numbers)

if(!params$DA.valid){
  validation.numbers <- sample(train.numbers, size = 1/5*(2/3*500))
  train.numbers <- setdiff(train.numbers, validation.numbers)
  test.numbers <- setdiff(0:499, train.numbers)
  test.numbers <- setdiff(test.numbers, validation.numbers)
}

# Next we select the image names corresponding to the indices sampled above.
normal.image.names <- data.frame(normal.image.names)
effusion.image.names <- data.frame(effusion.image.names)
```

```r
# Select image names corresponding to training numbers sampled above.
train.normal <- normal.image.names %>% dplyr::filter(parse_number(normal.image.names) %in% train.number
train.normal <- train.normal$normal.image.names # Change data type of names to vector.
all(sort(parse_number(train.normal)) == sort(train.numbers)) # Check that we have selected the correct
```

```
#> [1] TRUE
```

```r
test.normal <- normal.image.names %>% dplyr::filter(parse_number(normal.image.names) %in% test.numbers)
test.normal <- test.normal$normal.image.names # Change data type of names to vector.
all(sort(parse_number(test.normal)) == sort(test.numbers)) # Check that we have selected the correct na
```

```
#> [1] TRUE
```

```r
if(!params$DA.valid){
  validation.normal <- normal.image.names %>% dplyr::filter(parse_number(normal.image.names) %in% valid
  validation.normal <- validation.normal$normal.image.names # Change data type of names to vector.
  print(all(sort(parse_number(validation.normal)) == sort(validation.numbers))) # Check that we have se
}

# We sample from 0:499, in order to choose training numbers. We do it again for the effusion images,
# in case the indices of the two types of images are not chosen randomly,
# and they present some sort of dependence (we do not know how the images are named by the source).
train.numbers <- sample(0:499, size = 2/3*500)
# Find testing numbers as well, simply using a set difference.
test.numbers <- setdiff(0:499, train.numbers)

if(!params$DA.valid){
  validation.numbers <- sample(train.numbers, size = 1/5*(2/3*500))
  train.numbers <- setdiff(train.numbers, validation.numbers)
  test.numbers <- setdiff(0:499, train.numbers)
  test.numbers <- setdiff(test.numbers, validation.numbers)
}

# Next we select the image names corresponding to the indices sampled above.
train.effusion <- effusion.image.names %>% dplyr::filter(parse_number(effusion.image.names) %in% train.
train.effusion <- train.effusion$effusion.image.names # Change data type of names to vector.
all(sort(parse_number(train.effusion)) == sort(train.numbers)) # Check that we have selected the correc
```

```
#> [1] TRUE
```

```r
test.effusion <- effusion.image.names %>% dplyr::filter(parse_number(effusion.image.names) %in% test.nu
test.effusion <- test.effusion$effusion.image.names # Change data type of names to vector.
all(sort(parse_number(test.effusion)) == sort(test.numbers)) # Check that we have selected the correct
```

```
#> [1] TRUE
```

```r
if(!params$DA.valid){
  validation.effusion <- effusion.image.names %>% dplyr::filter(parse_number(effusion.image.names) %in%
  validation.effusion <- validation.effusion$effusion.image.names # Change data type of names to vector
  print(all(sort(parse_number(validation.effusion)) == sort(validation.numbers))) # Check that we have
}

# Now that we have all the file names, we copy them into their respective directories.
for (i in 1:length(train.numbers)){
  file.copy(from = paste0(normal.path, "/", train.normal[i]), to = train.normal.path, overwrite = T)
  file.copy(from = paste0(effusion.path, "/", train.effusion[i]), to = train.effusion.path, overwrite =
}

for (i in 1:length(test.numbers)){
  file.copy(from = paste0(normal.path, "/", test.normal[i] ), to = test.normal.path, overwrite = T)
  file.copy(from = paste0(effusion.path, "/", test.effusion[i]), to = test.effusion.path, overwrite = T
}

if(!params$DA.valid){
  for (i in 1:length(validation.numbers)){
    file.copy(from = paste0(normal.path, "/", validation.normal[i]), to = valid.normal.path, overwrite =
    file.copy(from = paste0(effusion.path, "/", validation.effusion[i]), to = valid.effusion.path, over
  }
}

# Check that it worked!
train.normal.new <- list.files(train.normal.path)
length(train.normal.new)
```

```
#> [1] 333
```

```r
all(train.normal.new == train.normal)
```

```
#> [1] TRUE
```

```r
train.effusion.new <- list.files(train.effusion.path)
length(train.effusion.new)
```

```
#> [1] 333
```

```r
all(train.effusion.new == train.effusion)
```

```
#> [1] TRUE
```

```r
test.normal.new <- list.files(test.normal.path)
length(test.normal.new)
```

```
#> [1] 167
```

```r
all(test.normal.new == test.normal)
```

```
#> [1] TRUE
```

```r
test.effusion.new <- list.files(test.effusion.path)
length(test.effusion.new)
```

```
#> [1] 167
```

```r
all(test.effusion.new == test.effusion)
```

```
#> [1] TRUE
```

```r
if(!params$DA.valid){
  valid.normal.new <- list.files(valid.normal.path)
  print(length(valid.normal.new))
  print(all(valid.normal.new == validation.normal))
  valid.effusion.new <- list.files(valid.effusion.path)
  print(length(valid.effusion.new))
  print(all(valid.effusion.new == validation.effusion))
}
```

2 Image Data Generators

Simply loading all 1000 images into the memory is (in most cases) not feasible. We make image data generators in order to work with models. These are image processing helper tools from Keras, which are used to turn image files on disk into batches of pre-processed tensors. The function `image_data_generator` can be used to generate batches with real-time data augmentation. Thus, this function can be used to pre-process the images. We rescale the images in $[0, 1]$ using this function. Moreover, we apply some image augmentation, like shifting, zooming and change of brightness. This is done to avoid overfitting during model training, i.e. to improve the generalization abilities of the models. The function `flow_images_from_directory` takes, among other arguments, a directory and an image generator as input, and generates batches of images from the specified directory, following the pre-processing rules given in the generator. We also choose the target size of the images, i.e. the width and height, where the original images are 512×512. As mentioned earlier, if `params$DA.valid` is set to `TRUE`, we also choose a validation split of $\frac{1}{5}$, which means that one fifth of the training data will be used for model validation during fitting.

Notice that we have tried training the model both with and without data augmentation on the validation data, following the discussion (e.g.) in this thread. I wanted to test both variants, since it is not completely clear which is most coherent and I think both could make sense, depending on the philosophy one follows. It is not clear to me which is the best, even though the version with image augmentation in both data sets seems to be able to generalize better and give better results on the test set. Therefore, I have chosen to set the parameter `params$DA.valid` to `TRUE` during my discussion in this report, even though both options are aptly implemented and tested simply by changing the value of the parameter (and running the entire code, after deleting the old image directories created earlier, just to be safe). When wanting to add image augmentation to the validation data, we set the parameter `DA.valid` to `TRUE`, which then does not make the directory for validation data in the preprocessing above. The generators below are therefore defined in slightly different ways depending on the value of the parameter, where I have tried to explain the logic as clearly as possible in the code below.

```r
img_width <- img_height <- 128 # or 256, 128, 64 or 32.
# 128 seems to give alright performance, without taking too long to train.
target_size <- c(img_width, img_height)
batch_size <- 32 # Set the batch_size to the tuned hyperparameter following tfruns!
epochs <- 30
channels <- 1 # RGB = 3 channels
image_size <- c(target_size, channels)

# optional data augmentation.
train_data_gen <- image_data_generator(
  rescale = 1/255,
  validation_split = 1/5,
  # ,rotation_range = 40, # Not relevant for us.
  # Images will almost always be relatively straight.
  width_shift_range = 0.1, # Shift in x direction.
  height_shift_range = 0.1, # Shift in y direction.
  # shear_range = 0.2, # I do not want to use shearing either,
  # as it seems a bit irrelevant for our type of images.
  zoom_range = 0.2,
  # horizontal_flip = TRUE, # Deemed irrelevant.
  fill_mode = "constant", # Added constant fill-mode (cval = 0),
  # since the rest of the fill-modes seem to distort the images
  # in a very unnatural way. Therefore I think it is better to
  # simply set the points outside the boundaries of the input to 0.
  brightness_range = c(0.7, 1.3) # Play with the brightness.
)
```

```
#> Loaded Tensorflow version 2.7.1
```

```r
# We do not apply the data-augmentation (except from rescaling)
# to the testing data set, since it is important
# to validate on the true images we have been given.
test_data_gen <- image_data_generator(
  rescale = 1/255
)

# Then we load the data using the generators.


if(!params$DA.valid){
  # DO NOT USE data augmentation on the validation data.
  # training images.
  train.image_array_gen <- flow_images_from_directory(train.path,
                                                      train_data_gen,
                                                      class_mode = "binary",
                                                      seed = params$seed,
                                                      target_size = target_size,
                                                      batch_size = batch_size,
                                                      color_mode = "grayscale"
                                                      )

  # If we choose to NOT use data augmentation on the validation data.
  val.image_array_gen <- flow_images_from_directory(valid.path,
                                                     test_data_gen,
                                                     class_mode = "binary",
                                                     seed = params$seed,
                                                     target_size = target_size,
                                                     batch_size = batch_size,
                                                     color_mode = "grayscale"
                                                      )
} else {
  # USE data augmentation on the validation data.
  train.image_array_gen <- flow_images_from_directory(train.path,
                                                      train_data_gen,
                                                      class_mode = "binary",
                                                      seed = params$seed,
                                                      target_size = target_size,
                                                      batch_size = batch_size,
                                                      color_mode = "grayscale",
                                                      subset = "training"
                                                      )

  # If we choose to use data augmentation on the validation data,
  # then we use the same image data generator as for the training data.
  val.image_array_gen <- flow_images_from_directory(train.path,
                                                     train_data_gen,
                                                     class_mode = "binary",
                                                     seed = params$seed,
                                                     target_size = target_size,
                                                     batch_size = batch_size,
                                                     color_mode = "grayscale",
                                                     subset = "validation")

}

test.image_array_gen <- flow_images_from_directory(test.path,
                                                      test_data_gen,
                                                      class_mode = "binary",
                                                      seed = params$seed,
                                                      target_size = target_size,
                                                      batch_size = 1,
                                                      color_mode = "grayscale",
                                                      shuffle = F # Makes it easier to check with
                                                      # true class labels after predicting.
                                                      )

# number of training samples
(train_samples <- train.image_array_gen$n)
```

```
#> [1] 534
```

```
# number of validation samples
(valid_samples <- val.image_array_gen$n)
```

```
#> [1] 132
```

```
# number of testing samples
(test_samples <- test.image_array_gen$n)
```

```
#> [1] 334
```

```
# Class index decoding.
d1 <- cbind(c("effusion", "normal"),
            c(train.image_array_gen$class_indices$effusion,
              train.image_array_gen$class_indices$normal))
knitr::kable(d1, caption = "Encoding")
```

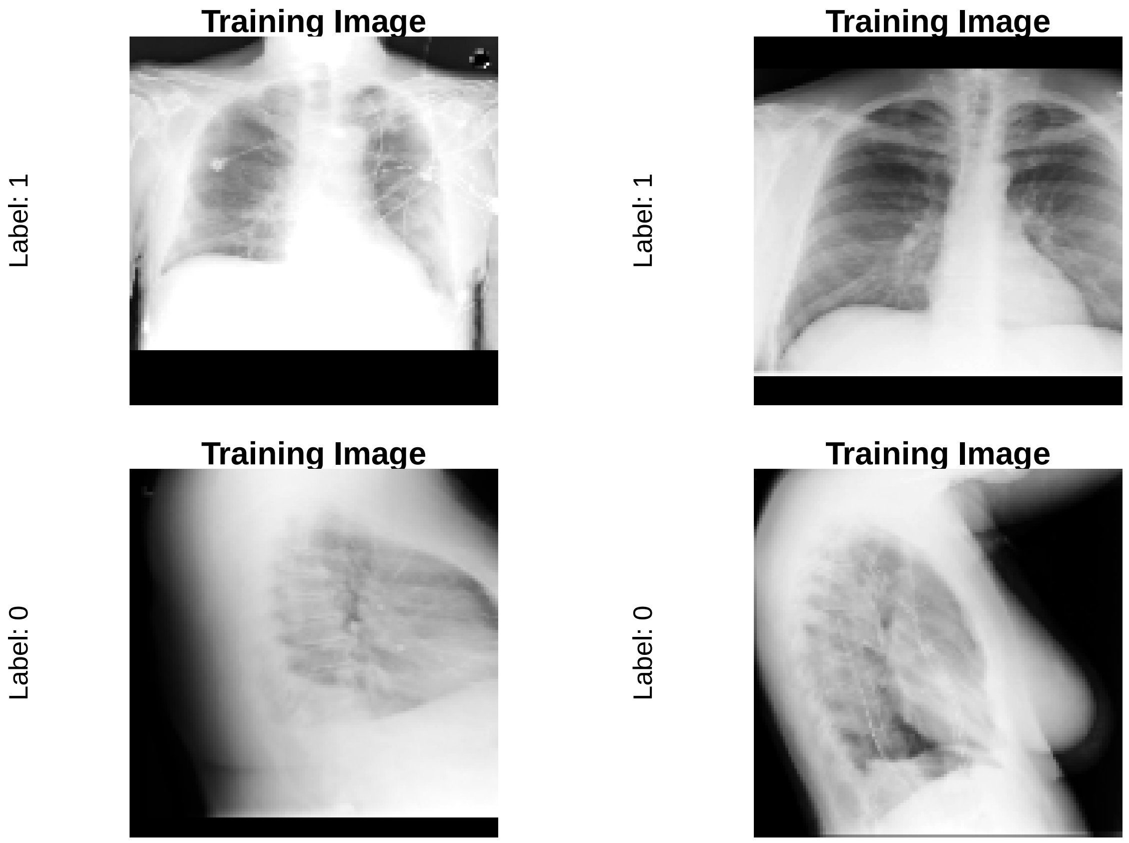
Table 1: Encoding

effusion	0
normal	1

The generators yield the batches indefinitely, which means that they act like infinite loops over the images in the specified directories. Notice that the normal class of images is coded as 1 and the effusion class of images is coded as 0. This choice is insignificant for us and is chosen by the image generators.

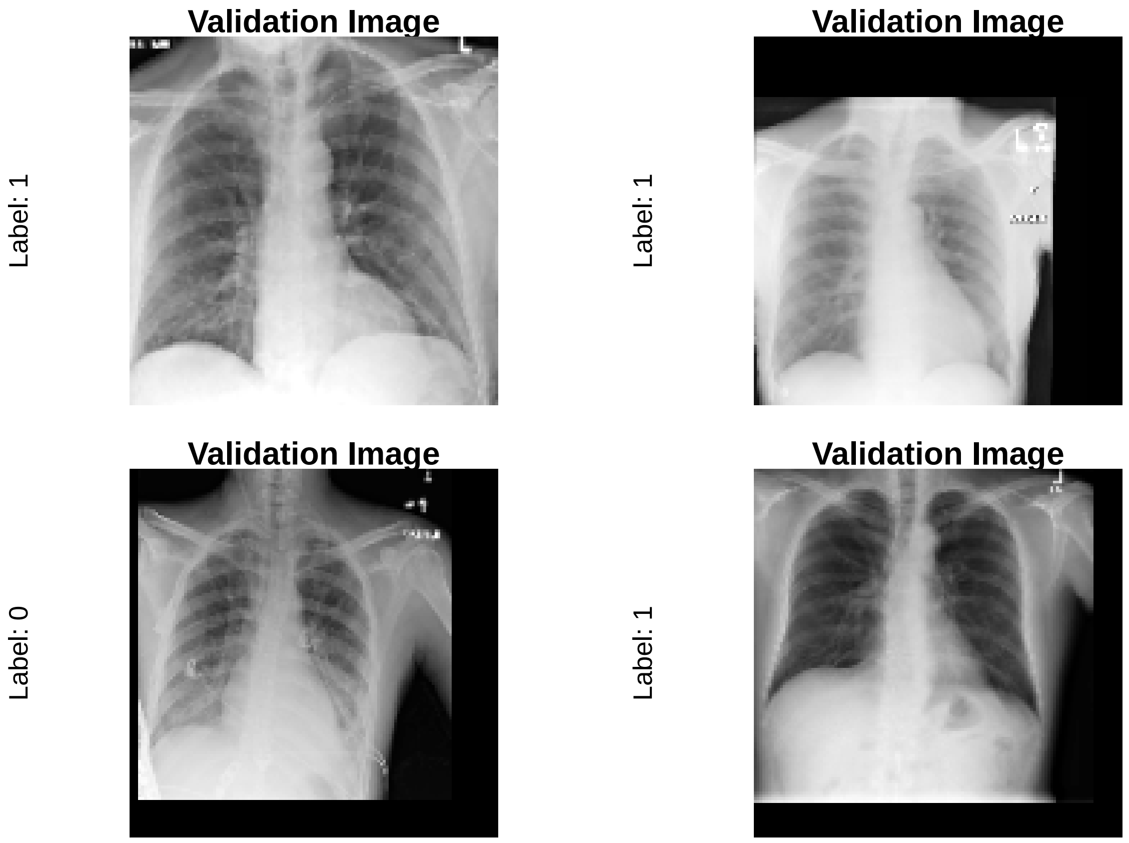
We have a short look at some of the images from the training data set, to get a feel for what the pictures may look like.

```
# Element generate.
batch <- generator_next(train.image_array_gen)
#str(batch)

op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in seq(5, 11, 2)) {
  #plot(as.raster(batch[[1]][i,,,]), main = paste0("Label: ", batch[[2]][i]))
  im <- matrix(batch[[1]][i,,,], nrow = img_height, ncol = img_height)
  rotate <- function(x) t(apply(x, 2, rev))
  image(1:img_height, 1:img_height, rotate(im), col=gray((0:255)/255), axes = F, asp = 1,
        ylab = paste0("Label: ", batch[[2]][i]), main = "Training Image")
}
```



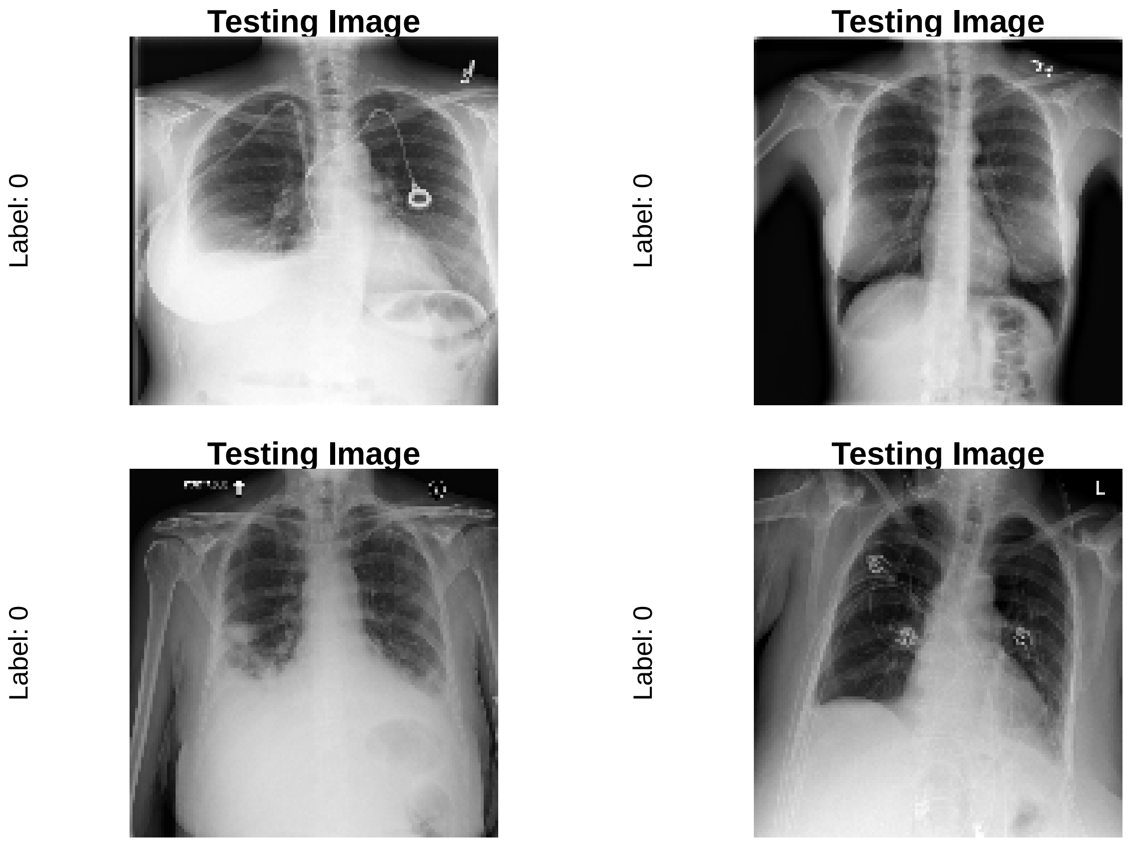
Some of the validation images are shown as well, which may be augmented, depending on the choice of the parameter `DA.valid`.

```
batch <- generator_next(val.image_array_gen)
#str(batch)

op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in seq(5, 11, 2)) {
  #plot(as.raster(batch[[1]][i,,,]), main = paste0("Label: ", batch[[2]][i]))
  im <- matrix(batch[[1]][i,,,], nrow = img_height, ncol = img_height)
  rotate <- function(x) t(apply(x, 2, rev))
  image(1:img_height, 1:img_height, rotate(im), col=gray((0:255)/255), axes = F, asp = 1,
        ylab = paste0("Label: ", batch[[2]][i]), main = "Validation Image")
}
```



Some of the testing images are shown as well, which have not been augmented.

```
op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in 1:4) {
  batch <- generator_next(test.image_array_gen)
  #plot(as.raster(batch[[1]][1,,,]))
  im <- matrix(batch[[1]][1,,,], nrow = img_height, ncol = img_height)
  rotate <- function(x) t(apply(x, 2, rev))
  image(1:img_height, 1:img_height, rotate(im), col=gray((0:255)/255), axes = F, asp = 1,
        ylab = paste0("Label: ", batch[[2]][1]), main = "Testing Image")
}
```



3 Convolutional Neural Network (CNN) Implementation

We have implemented a CNN with three convolutional layers and one fully connected dense hidden layer. This fully connected layer has the same amount of nodes as the image height / width. In between we have added some max pooling layers, some layer dropout and a kernel L2 regularizer in the fully connected dense hidden layer, in order to combat overfitting.

```r
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                padding="same",
                input_shape = image_size) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu",
                padding="same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = img_width, activation = "relu",
              kernel_regularizer = regularizer_l2(l = 0.01)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

```
#> Model: "sequential"
#> ________________________________________________________________________________
#>  Layer (type)                      Output Shape                   Param #
#> ================================================================================
#>  conv2d_2 (Conv2D)                 (None, 128, 128, 32)           320
#>
#>  max_pooling2d_2 (MaxPooling2D)    (None, 64, 64, 32)             0
#>
#>  dropout_1 (Dropout)               (None, 64, 64, 32)             0
#>
#>  conv2d_1 (Conv2D)                 (None, 64, 64, 64)             18496
#>
#>  max_pooling2d_1 (MaxPooling2D)    (None, 32, 32, 64)             0
#>
#>  conv2d (Conv2D)                   (None, 30, 30, 128)            73856
#>
#>  max_pooling2d (MaxPooling2D)      (None, 15, 15, 128)            0
#>
#>  flatten (Flatten)                 (None, 28800)                  0
#>
#>  dense_1 (Dense)                   (None, 128)                    3686528
#>
#>  dropout (Dropout)                 (None, 128)                    0
#>
#>  dense (Dense)                     (None, 1)                      129
#>
#> ================================================================================
#> Total params: 3,779,329
#> Trainable params: 3,779,329
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

4 CNN Training

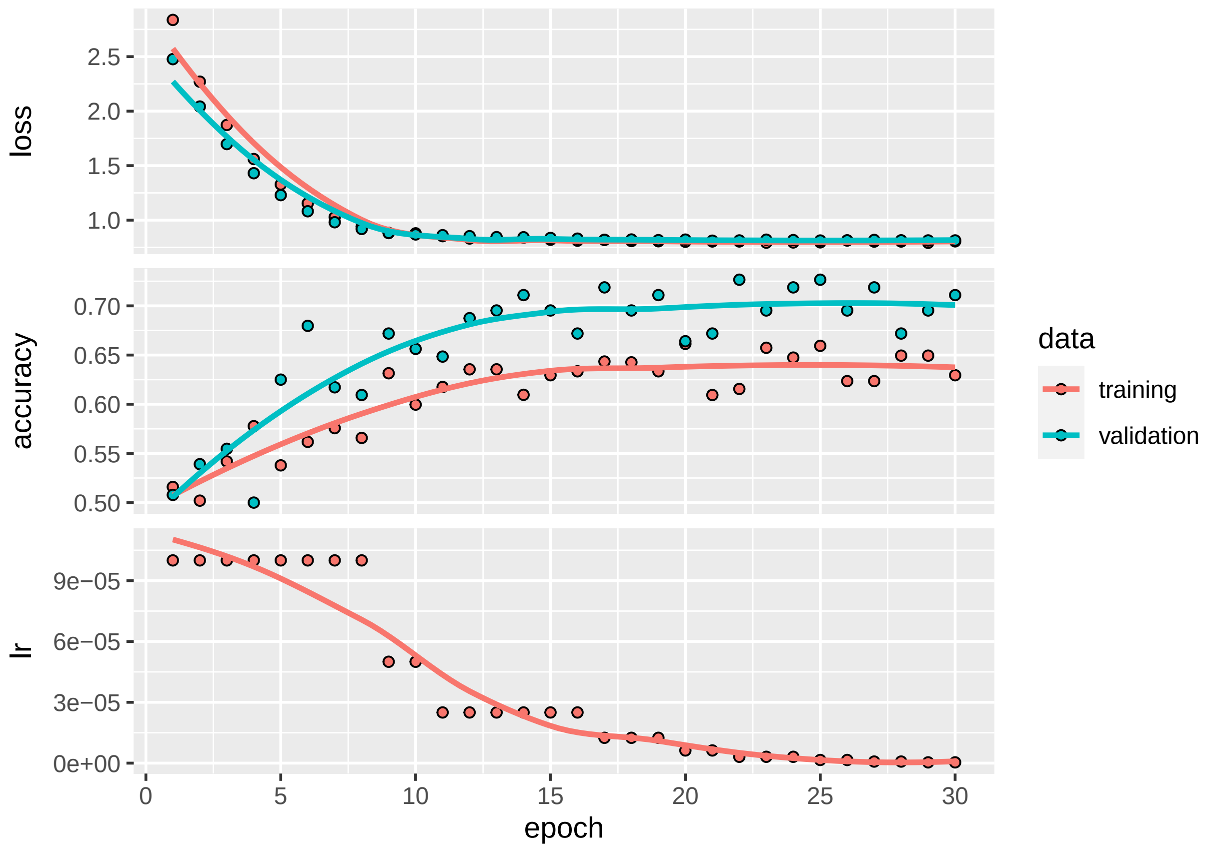
We define the CNN with choice of optimizer, loss function and metric, before we fit the model. We choose the `binary_crossentropy` as loss function, because we have defined the model with one node in the output layer, with sigmoid activation function. Moreover, we choose `rmsprop` as optimizer, because it seems like this gives the greatest performance when comparing to other optimizers, like `adam` and `adadelta`. Finally, we use `accuracy` as the metric, since we want the model to diagnose the images with the greatest accuracy possible.

After defining the model conveniently, we fit the model. Because the image generators loop endlessly over the images on disk, we need to specify the total number of steps, or batches, before declaring on epoch as finished. Since we are using a generator for both the training data and the validation data, we need to specify this value for both the training data and the validation data. This is done with the arguments `steps_per_epoch` and `validation_steps` below.

Notice that, during fitting, we use a callback that reduces the learning rate if the validation accuracy stops improving for 2 epochs. This was added after fitting the model the first times, in order to try to increase the accuracy on both the training and the validation data. The history after training is plotted. We can see that the accuracy in both the training and the validation data stabilizes and "flattens out". This indicates that the model most likely won't be able to improve much more in this regard, with the given architecture, optimizer and hyperparameter choices.

```r
model %>% compile(
  loss = "binary_crossentropy",
  #optimizer = optimizer_adam(), # Adam is slow and gives worse performance it seems like.
  optimizer = optimizer_rmsprop(learning_rate = 1e-4), # decay does not seem to work very well.
  #optimizer = optimizer_adadelta() # Adadelta is slower and gives worse performance it seems like.
  metrics = c("accuracy")
)

# We save the model and its state (before fitting), so that
# it can be used when tuning the hyperparameter.
model %>% save_model_hdf5("convnet.h5")

cb <- callback_reduce_lr_on_plateau( # Reduce learning rate if val_loss plateaus.
    monitor="val_accuracy",
    factor = 0.5,
    patience = 2,
    min_lr = 10e-8,
    mode = "max")

history <- model %>% fit(
  train.image_array_gen,
  # epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = epochs,

  # validation data
  validation_data = val.image_array_gen,
  validation_steps = as.integer(valid_samples / batch_size),

  callbacks = cb
)
plot(history)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



5 Tuning of Hyperparameter

We tune the hyperparameter `batch_size` using the `tfruns` package. This is done exploring the grid 16, 32, 64 for the values of the hyperparameter.

This is done in separate files, which are delivered with this report and Rmd. For each of the values in the grid we want to explore, we load the saved model and fit it with the data generators defined using the respective batch size we want to test for. We need to redefine the data generators for each of the three models, because the batch sizes are set in the generators. Notice that this is only implemented for the variant of the solution where data augmentation is performed on both training data and validation data, i.e. when the parameter `params$DA.valid` is set to `TRUE`.

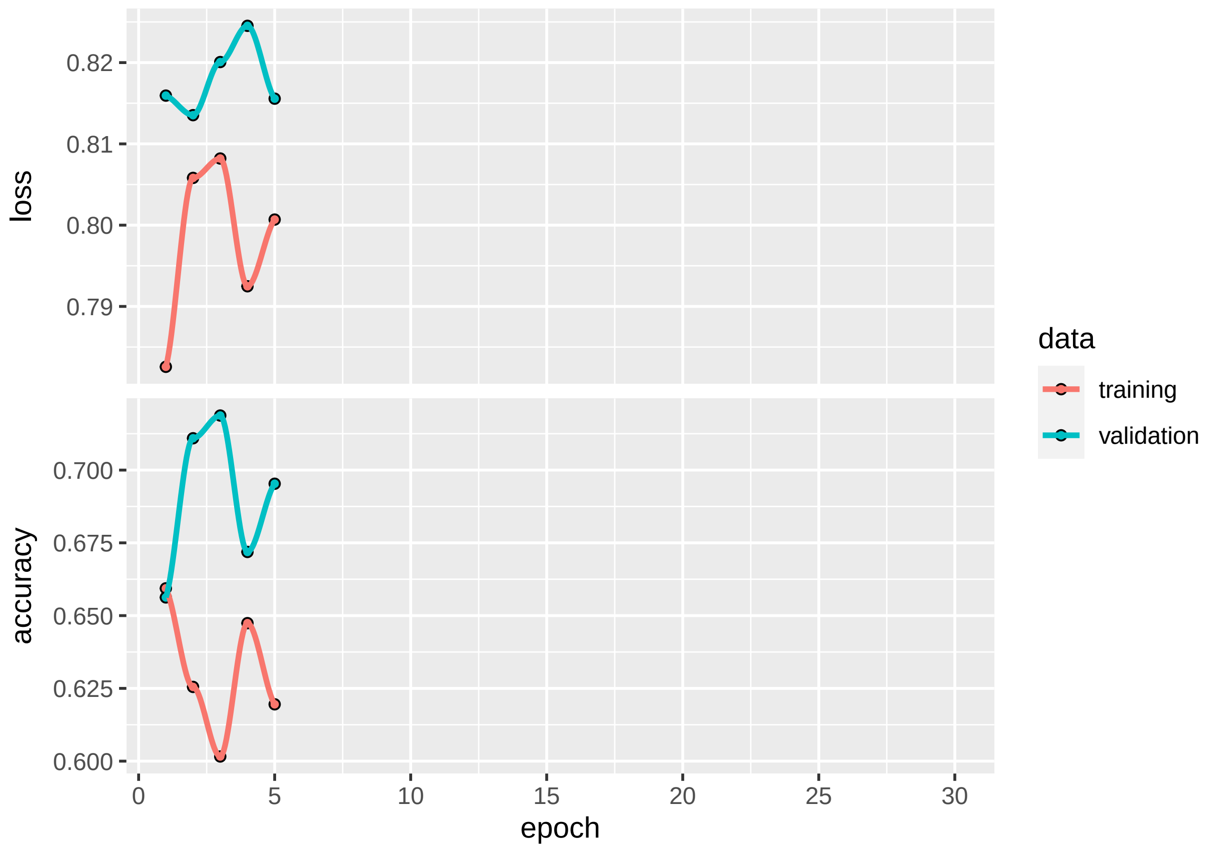
The results from running the file `task2_tfruns.R`, which calls on `task2_explore.R` for each different batch size on the grid, seem to show that a batch size of 32 yields the best results, according to the metric validation accuracy. In other words, the run where a batch size of 32 was used seems to yield the largest validation accuracy. Not only is the validation accuracy highest for this batch size, but the model yields slightly larger specificity and slightly larger positive predictive value. The AUC is also slightly larger. However, despite our conclusion of 32 as better, the runs are marginally different, as there is not much that separates the two models. The differences we see could very well be up to chance as well. Choosing either one of the hyperparameter values (either 32 or 64) would likely yield almost the same performance. The runs used during this deliberation are added to delivery, in case this is interesting information to the reader. Notice for example that if you regard the sensitivity and the negative predictive value as more important, which is a completely valid reflection to make, you might be inclined to choose the model run with a batch size of 64, but it is up to the each and every data scientist (and their teams) to decide.

6 Early Stopping

We implement early stopping in the model, using the `keras callbacks()` API. The training is interrupted when validation accuracy stops improving (increasing) for more than two epochs.

```r
callback.parameters <- callback_early_stopping(
  monitor = "val_accuracy",
  patience = 2, # This argument is used to interrupt training when
  # validation accuracy stops improving (increasing) for more than two epochs.
  verbose = 1,
  mode = "max",
  restore_best_weights = T
)

history <- model %>% fit(
  train.image_array_gen,
  # epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = epochs,

  # validation data
  validation_data = val.image_array_gen,
  validation_steps = as.integer(valid_samples / batch_size),

  # Add callbacks for early stopping.
  callbacks = callback.parameters
)
plot(history)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



We notice that it seems like a patience of 2 seems to be a bit strict for this model, since it stops training relatively early. If this was not a clear criteria in the assignment question, I would increase this patience a bit, in order to let the model train for longer.

7 Performance Assessment of CNN

We assess the performance of the CNN by predicting the categories of the test images. The confusion matrix is given.

First we evaluate the model to get an idea of how well the model can predict on the test images. Notice that we, again, have to specify the total number of steps before declaring the evaluation round finished, which is done using the `steps` argument in the `evaluate` function. For the test data generator we have defined the batch size as 1, such that we simply evaluate every test image once during evaluation and prediction.

```
model %>% evaluate(test.image_array_gen, steps = test_samples)
```

```
#>      loss  accuracy
#> 0.8219144 0.6616766
```

Next we make predictions on every test image.

```
#test.image_array_gen$reset() # Done in case such that nothing will go wrong.
y_pred <- model %>% predict(test.image_array_gen, steps = test_samples) %>%
    `>`(0.5) %>% k_cast("int32")
y_pred <- as.array(y_pred)
(mat.CNN <- confusionMatrix(as.factor(test.image_array_gen$classes), as.factor(y_pred)))
```

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

```
#>           Reference
#> Prediction   0   1
#>          0  75  92
#>          1  21 146
#>                                           
#>                Accuracy : 0.6617          
#>                  95% CI : (0.6082, 0.7123)
#>     No Information Rate : 0.7126          
#>     P-Value [Acc > NIR] : 0.9817          
#>                                           
#>                   Kappa : 0.3234          
#>                                           
#>  Mcnemar's Test P-Value : 4.547e-11       
#>                                           
#>             Sensitivity : 0.7812          
#>             Specificity : 0.6134          
#>          Pos Pred Value : 0.4491          
#>          Neg Pred Value : 0.8743          
#>              Prevalence : 0.2874          
#>          Detection Rate : 0.2246          
#>    Detection Prevalence : 0.5000          
#>       Balanced Accuracy : 0.6973          
#>                                           
#>        'Positive' Class : 0               
#>
```

The confusion matrix shows that the model performs better than random according to accuracy, where random prediction (or simply predicting one class always) would give 50% accuracy. From studying the confusion matrix a bit more, we can see that the model, for the most part, does a good job in predicting true normal images correctly. In fact, it predicts this case correctly in 0.8742515 percent of the cases, which is given by the negative predictive value in this case (because, by default, 0, or effusion, is regarded as a positive result from the test, since it is the first factor). On the other hand, we can see that the model struggles to predict images with true effusion correctly. The positive predictive value is only 0.4491018, which means that the model predicts effusion on a lot of the images that are normal. In this regard the model does a very bad job. The sensitivity and specificity are 0.78125 and 0.6134454 respectively. A larger sensitivity means that the model is able to detect a lot of the cases of true effusion, i.e. a large number of images with effusion are identified by the model. This means that it is likely that an image that has effusion is identified by the model. This is somewhat the case in this model. Similarly, a larger specificity means that the model is able to detect a lot of the cases of true normal images, i.e. a large number of images that are truly normal are identified as normal by the model. This is not really the case for this model. Thus, it is likely that a normal image is classified as positive, i.e. is classified as having effusion.

8 Convolutional Autoencoder (CAE) Implementation

We implement a CAE with 10 nodes in the z layer (the bottleneck). The rest of the architecture chosen is best seen in the code chunk below.

Notice that the encoder consists of three convolutional layers, three max pooling layers and one dense fully connected layer. Additionally, notice that the decoder consists of three convolutional layers, one transpose convolutional layer, two upsampling layers and one dense fully connected layer. The decoder uses a transpose convolutional layer as the second hidden layer, in order to mirror the stride of 2 used in the last max pooling layer used in the encoder. This is perhaps not necessary, but it has nevertheless been done in this case.

```
# Taken inspiration from:
# https://blog.keras.io/building-autoencoders-in-keras.html
```

```r
# https://keras.io/examples/vision/autoencoder/

# Convolutional Encoder.
model_enc <- keras_model_sequential() %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
  activation = "relu", padding = "same",
  input_shape = image_size) %>%
  layer_max_pooling_2d(pool_size = c(2,2), padding = "same") %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
  activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(3,3), padding = "same") %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3),
  activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(4,4), strides = c(2,2),
                       padding = "valid") %>%
  layer_flatten() %>%
  layer_dense(10, activation = "relu")
summary(model_enc)
```

```
#> Model: "sequential_1"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  conv2d_5 (Conv2D)                  (None, 128, 128, 64)            640
#>
#>  max_pooling2d_5 (MaxPooling2D)     (None, 64, 64, 64)              0
#>
#>  conv2d_4 (Conv2D)                  (None, 64, 64, 32)              18464
#>
#>  max_pooling2d_4 (MaxPooling2D)     (None, 22, 22, 32)              0
#>
#>  conv2d_3 (Conv2D)                  (None, 22, 22, 16)              4624
#>
#>  max_pooling2d_3 (MaxPooling2D)     (None, 10, 10, 16)              0
#>
#>  flatten_1 (Flatten)                (None, 1600)                    0
#>
#>  dense_2 (Dense)                    (None, 10)                      16010
#>
#> ================================================================================
#> Total params: 39,738
#> Trainable params: 39,738
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

```r
# Convolutional Decoder.
model_dec <- keras_model_sequential() %>%
  layer_dense(units = 1600, activation = "relu", input_shape = c(10)) %>%
  layer_reshape(target_shape = c(10, 10, 16)) %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3),
      activation = "relu", padding = "same",
      input_shape = c(10, 10, 16)) %>%
  layer_conv_2d_transpose(filters = 16, kernel_size = c(4,4),
      activation = "relu", padding = "valid", strides = c(2,2)) %>%
  layer_conv_2d(filters = 32, activation = "relu",
      kernel_size = c(3,3), padding = "same") %>%
  layer_upsampling_2d(size = c(3,3)) %>%
  layer_conv_2d(filters = 1, kernel_size = c(3,3),
  activation = "relu", padding = "valid") %>%
  layer_upsampling_2d(size = c(2,2))

summary(model_dec)
```

```
#> Model: "sequential_2"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                     Param #
#> ================================================================================
#>  dense_3 (Dense)                    (None, 1600)                     17600
#>
#>  reshape (Reshape)                  (None, 10, 10, 16)               0
#>
#>  conv2d_8 (Conv2D)                  (None, 10, 10, 16)               2320
#>
#>  conv2d_transpose (Conv2DTranspose)  (None, 22, 22, 16)              4112
#>
#>  conv2d_7 (Conv2D)                  (None, 22, 22, 32)               4640
#>
#>  up_sampling2d_1 (UpSampling2D)     (None, 66, 66, 32)               0
#>
#>  conv2d_6 (Conv2D)                  (None, 64, 64, 1)                289
#>
#>  up_sampling2d (UpSampling2D)       (None, 128, 128, 1)              0
#>
#> ================================================================================
#> Total params: 28,961
#> Trainable params: 28,961
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

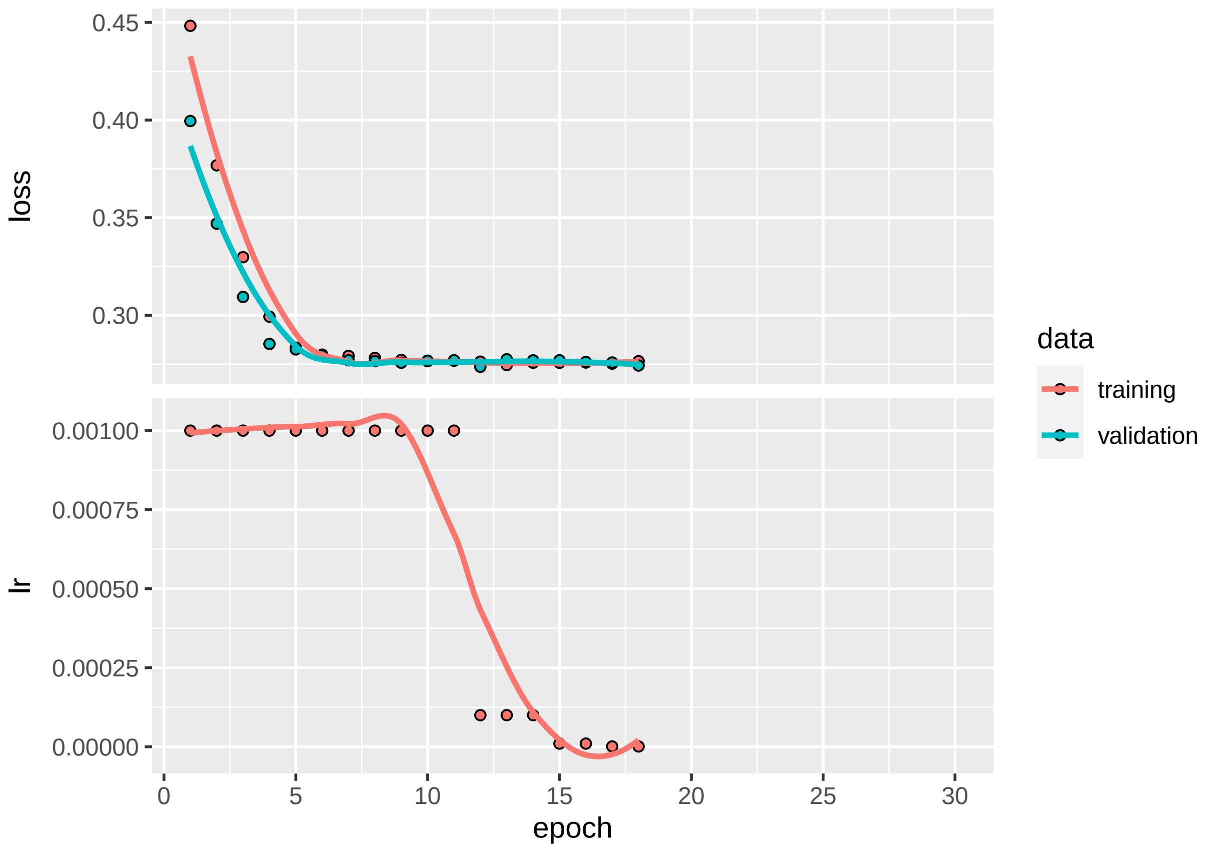
```
model.CAE <- keras_model_sequential()
# input dimension == output dimension

# Autoencoder.
model.CAE %>% model_enc %>% model_dec
```

```
#> Model
#> Model: "sequential_3"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                     Param #
#> ================================================================================
#>  sequential_1 (Sequential)          (None, 10)                       39738
#>
#>  sequential_2 (Sequential)          (None, 128, 128, 1)              28961
#>
#> ================================================================================
#> Total params: 68,699
#> Trainable params: 68,699
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

We compile and fit the model, adding callbacks for early stopping and reducing the learning rate on plateau. Notice that during the implementation we have tried fitting the model both with and without image augmentation (both on the training data and/or the validation data), and the results look better in the image augmentation case (`params$DA.valid` is `TRUE`). Because of this, in the end, we have simply chosen to use the same image generators as constructed earlier in the assignment.

```r
model.CAE %>% compile(
  loss = "mse",
  #optimizer = optimizer_rmsprop(),
  optimizer = optimizer_adam(learning_rate = 0.001, decay = 0.99)
  #,metrics = c("mean_squared_error")
)

callback.parameters.CAE <- list(
  callback_early_stopping(
  monitor = "val_loss",
    patience = 6, # This argument is used to interrupt training when
    # mean squared error stops improving (decreasing) for more than two epochs.
    verbose = 1,
    mode = "min",
    restore_best_weights = T),
  callback_reduce_lr_on_plateau( # Reduce learning rate if val_loss plateaus.
    monitor="val_loss",
    factor = 0.1,
    patience = 2,
    min_lr = 10e-8,
    mode = "min")
)

history.CAE <- model.CAE %>% fit(
  x = train.image_array_gen,
  y = train.image_array_gen,

  # epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = epochs,

  # validation data
  validation_data = val.image_array_gen,
  validation_steps = as.integer(valid_samples / batch_size),

  # Add callbacks for early stopping.
  callbacks = callback.parameters.CAE
)
plot(history.CAE)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



After training we do some predictions on the test data, which yields a first impression of the performance of the CAE; is it able to recreate the input images to some degree?

```r
model.CAE %>% evaluate(test.image_array_gen, steps = test_samples)
```

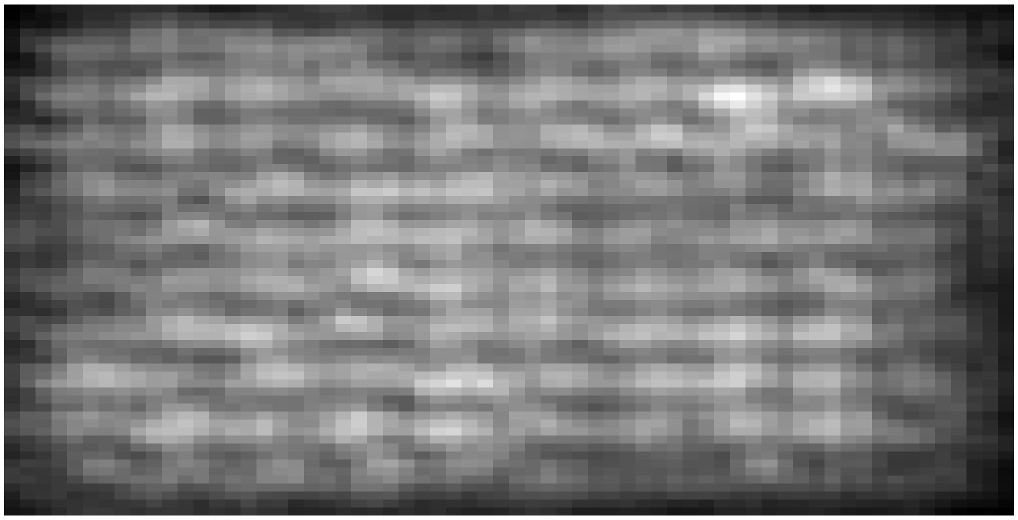
```
#>    loss
#> 0.26799
```

```r
index <- 1 # Only works with 1 index now, since batch size in testing is 1.

# Autoencoder
output_cae <- model.CAE %>% predict(test.image_array_gen, steps = test_samples)
#output_cae <- as.array(output_cae)
dim(output_cae)
```

```
#> [1] 334 128 128   1
```

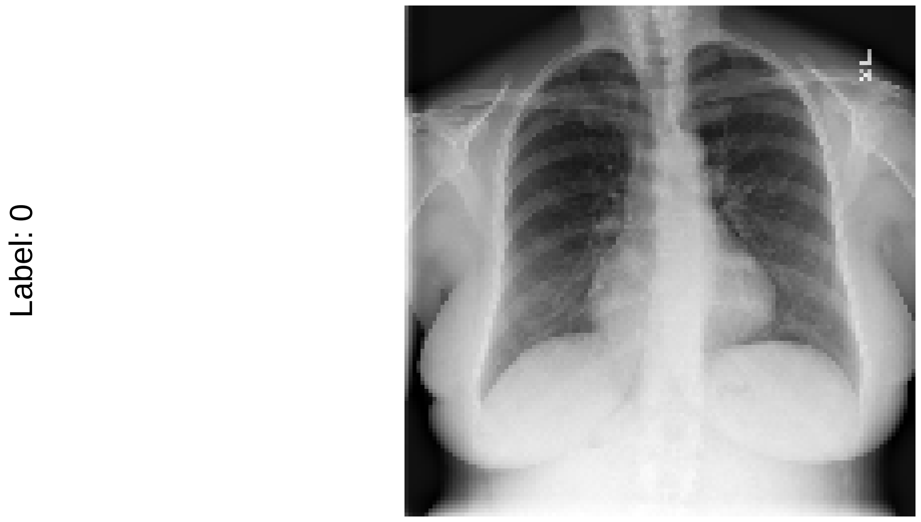
First of all we can see that the output images are of the correct dimension. We plot the first predicted image as an example of the output from the model.

```r
# Plot the first predicted image as an example.
im <- matrix(output_cae[index,,,], nrow = img_height, ncol = img_height)
image(1:img_height, 1:img_height, im, col=gray((0:255)/255),
      xlab = "", ylab = "", axes = F)
```



The image does not make much sense; it does not resemble our data. Let us plot the true image that the model was given as input when producing the output shown.

```r
# Plot the real image.
batch <- generator_next(test.image_array_gen)
#plot(as.raster(batch[[1]][index,,,]))
im <- matrix(batch[[1]][index,,,],
      nrow = img_height, ncol = img_height)
rotate <- function(x) t(apply(x, 2, rev))
image(1:img_height, 1:img_height, rotate(im), col=gray((0:255)/255),
      axes = F, asp = 1, ylab = paste0("Label: ", batch[[2]][1]), xlab = "")
```



They do not look similar, which suggests that the model is not able to reproduce the images. Is it useless? Before we give up on this model, we use the CAE as a pre-training model for prediction of emission or not, to see what the performance is like in this regard. The code is given in the code chunk below, where the model is defined, compiled and fitted. The history after the fitting is plotted.

```r
# First we build the encoder again, using the encoder we fit earlier.
model.input <- layer_input(shape = c(img_height, img_width, channels, channels))
model.output <- model.input %>%
  model_enc
model.CAE.pre <- keras_model(model.input, model.output)
```

```r
# Then we add some dense layers in the end, with a sigmoid output (one node).
model.output2 <- model.output %>%
  layer_flatten() %>%
  layer_dense(100,activation = "relu") %>% # Couple with fully connected layers (DNN).
  layer_dropout(0.2) %>%
  layer_dense(1,activation = "sigmoid")
model.CAE.pred <- keras_model(model.input, model.output2)

# Then we freeze the weights of the CAE, such that we only train the weights of the dense
# fully connected part now.
summary(model.CAE.pred)
```

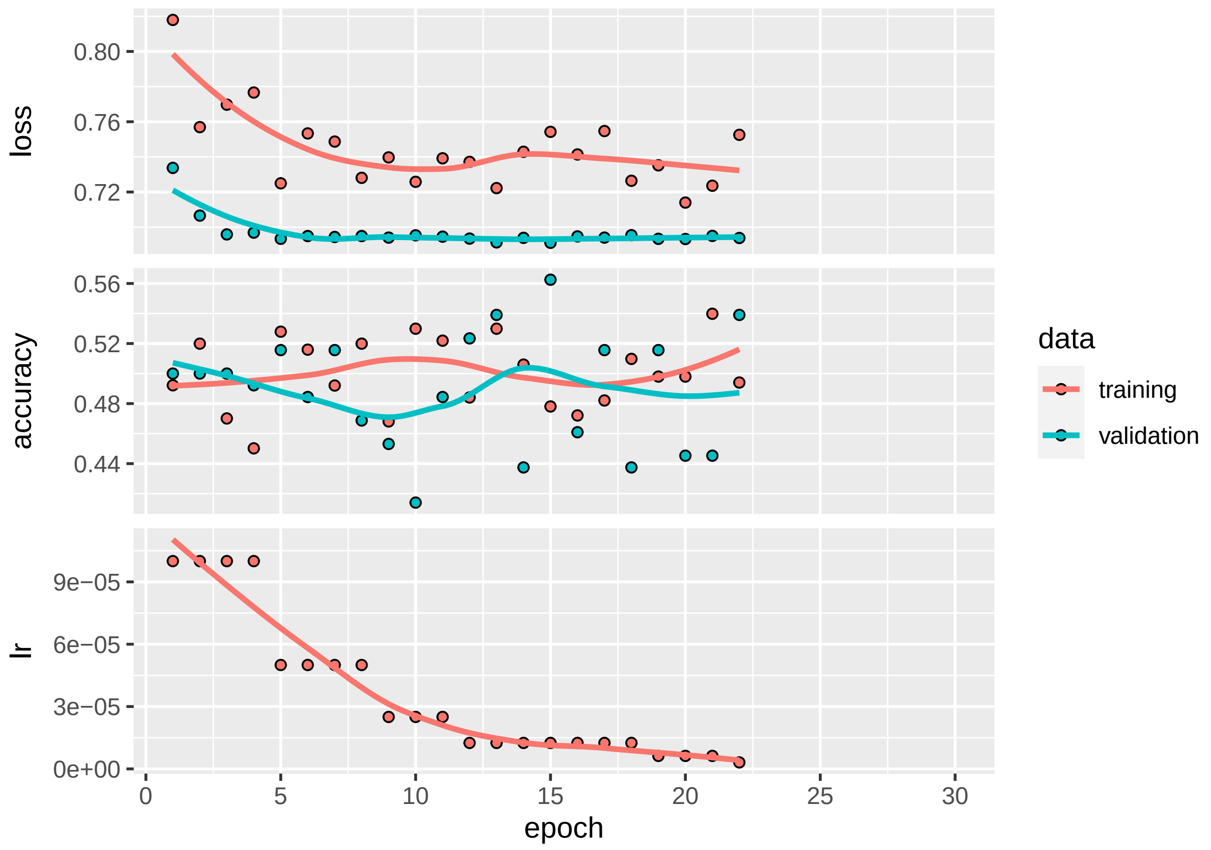
```
#> Model: "model_1"
#> _________________________________________________________________________________
#>  Layer (type)                       Output Shape                     Param #
#> =================================================================================
#>  input_1 (InputLayer)               [(None, 128, 128, 1, 1)]         0
#>
#>  sequential_1 (Sequential)          (None, 10)                       39738
#>
#>  flatten_2 (Flatten)                (None, 10)                       0
#>
#>  dense_5 (Dense)                    (None, 100)                      1100
#>
#>  dropout_2 (Dropout)                (None, 100)                      0
#>
#>  dense_4 (Dense)                    (None, 1)                        101
#>
#> =================================================================================
#> Total params: 40,939
#> Trainable params: 40,939
#> Non-trainable params: 0
#> _________________________________________________________________________________
```

```r
freeze_weights(model.CAE.pred, from = 1, to = 2)
summary(model.CAE.pred)
```

```
#> Model: "model_1"
#> _________________________________________________________________________________
#>  Layer (type)                       Output Shape                     Param #
#> =================================================================================
#>  input_1 (InputLayer)               [(None, 128, 128, 1, 1)]         0
#>
#>  sequential_1 (Sequential)          (None, 10)                       39738
#>
#>  flatten_2 (Flatten)                (None, 10)                       0
#>
#>  dense_5 (Dense)                    (None, 100)                      1100
#>
#>  dropout_2 (Dropout)                (None, 100)                      0
#>
#>  dense_4 (Dense)                    (None, 1)                        101
#>
#> =================================================================================
#> Total params: 40,939
#> Trainable params: 1,201
#> Non-trainable params: 39,738
#> ________________________________________________________________________________
```

```
model.CAE.pred %>% compile(
  loss = "binary_crossentropy",
  #optimizer = optimizer_adam(), # Adam is slow and gives worse performance it seems like.
  optimizer = optimizer_rmsprop(learning_rate = 1e-4), # decay does not seem to work very well.
  #optimizer = optimizer_adadelta() # Adadelta is slower and gives worse performance it seems like.
  metrics = c("accuracy")
)

cb <- list(callback_early_stopping(
  monitor = "val_accuracy",
    patience = 7, # This argument is used to interrupt training when
    # mean squared error stops improving (decreasing) for more than two epochs.
    verbose = 1,
    mode = "max",
    restore_best_weights = T),
  callback_reduce_lr_on_plateau( # Reduce learning rate if val_loss plateaus.
    monitor="val_accuracy",
    factor = 0.5,
    patience = 3,
    min_lr = 10e-8,
    mode = "max")
)

history.CAE.pred <- model.CAE.pred %>% fit(
  train.image_array_gen,
  # epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = epochs,

  # validation data
  validation_data = val.image_array_gen,
  validation_steps = as.integer(valid_samples / batch_size),

  callbacks = cb
)
plot(history.CAE.pred)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



After the training we do some predictions and compare the performance of this model to that of the CNN built earlier in the assignment.

```
model.CAE.pred %>% evaluate(test.image_array_gen, steps = test_samples)
```

```
#>      loss  accuracy
#> 0.6964842 0.3952096
```

```
#test.image_array_gen$reset() # Done in case such that nothing will go wrong.
y_pred.CAE <- model.CAE.pred %>% predict(test.image_array_gen, steps = test_samples) %>%
    `>`(0.5) %>% k_cast("int32")
y_pred.CAE <- as.array(y_pred.CAE)
confusionMatrix(as.factor(test.image_array_gen$classes), as.factor(y_pred.CAE))
```

```
#> Confusion Matrix and Statistics
#> 
#>           Reference
#> Prediction   0   1
#>          0  76  91
#>          1 111  56
#> 
#>                Accuracy : 0.3952
#>                  95% CI : (0.3424, 0.4499)
#>     No Information Rate : 0.5599
#>     P-Value [Acc > NIR] : 1.0000
#> 
#>                   Kappa : -0.2096
#> 
```

```
#>  Mcnemar's Test P-Value : 0.1813
#>
#>             Sensitivity : 0.4064
#>             Specificity : 0.3810
#>          Pos Pred Value : 0.4551
#>          Neg Pred Value : 0.3353
#>              Prevalence : 0.5599
#>          Detection Rate : 0.2275
#>    Detection Prevalence : 0.5000
#>       Balanced Accuracy : 0.3937
#>
#>        'Positive' Class : 0
#>
```

We can see that the performance of this model is worse than the CNN, when it comes to accuracy. Similar comments about the other metrics can be done here as well, but will be skipped (the reader is referred to the discussion after prediction with the CNN). In general, this is not very good performance from a classifier, which might be because of a small amount of data and/or badly tuned hyperparameters in the models. The architectures might be lacking as well. For this pre-trained CAE, with 10 nodes in the bottleneck layer, I imagine that there are not enough nodes for this philosophy to be effective. I tried increasing the number of nodes in the trainable fully connected layer further - this seemed to deteriorate the performance. I also tried using a pre-trained CNN without the final dense layer in the encoder, i.e. with the final `max_pooling2d` layer of dimension `(None, 10, 10, 16)`, as final output layer, which gave a more similar accuracy to the CNN. Thus, it seems to me like we give up too much information in the images when reducing their dimension simply to 10 nodes, if prediction is the ultimate goal. I imagine that the goal here is more about demonstrating what happens in the bottleneck layer, which of course is easier when the dimension is lower. Some representations of this will be shown next.

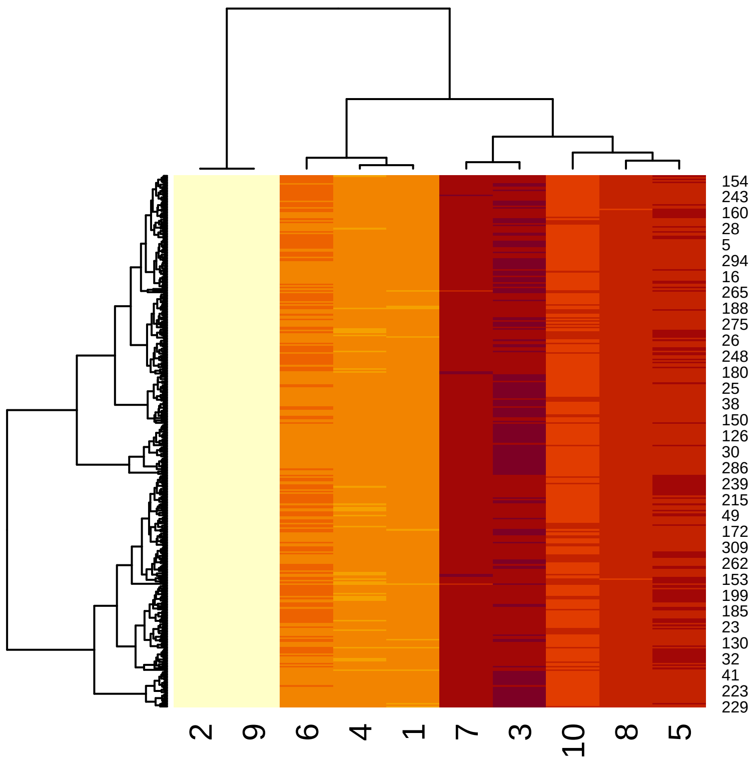
9 Graphical Representation of Results from CAE

In order to get more insight about the results from the CAE, we add some more graphical representations of what happens in the z-layer, i.e. the bottleneck. We predict on the test set using the encoder, such that the output is a dataset with 10 numerical values, representing the outputs from the 10 nodes, for each image in the test set.

```
# Prediction on test images using the encoder.
enc_output <- model_enc %>% predict(test.image_array_gen, steps = test_samples)
dim(enc_output)
```

```
#> [1] 334  10
```

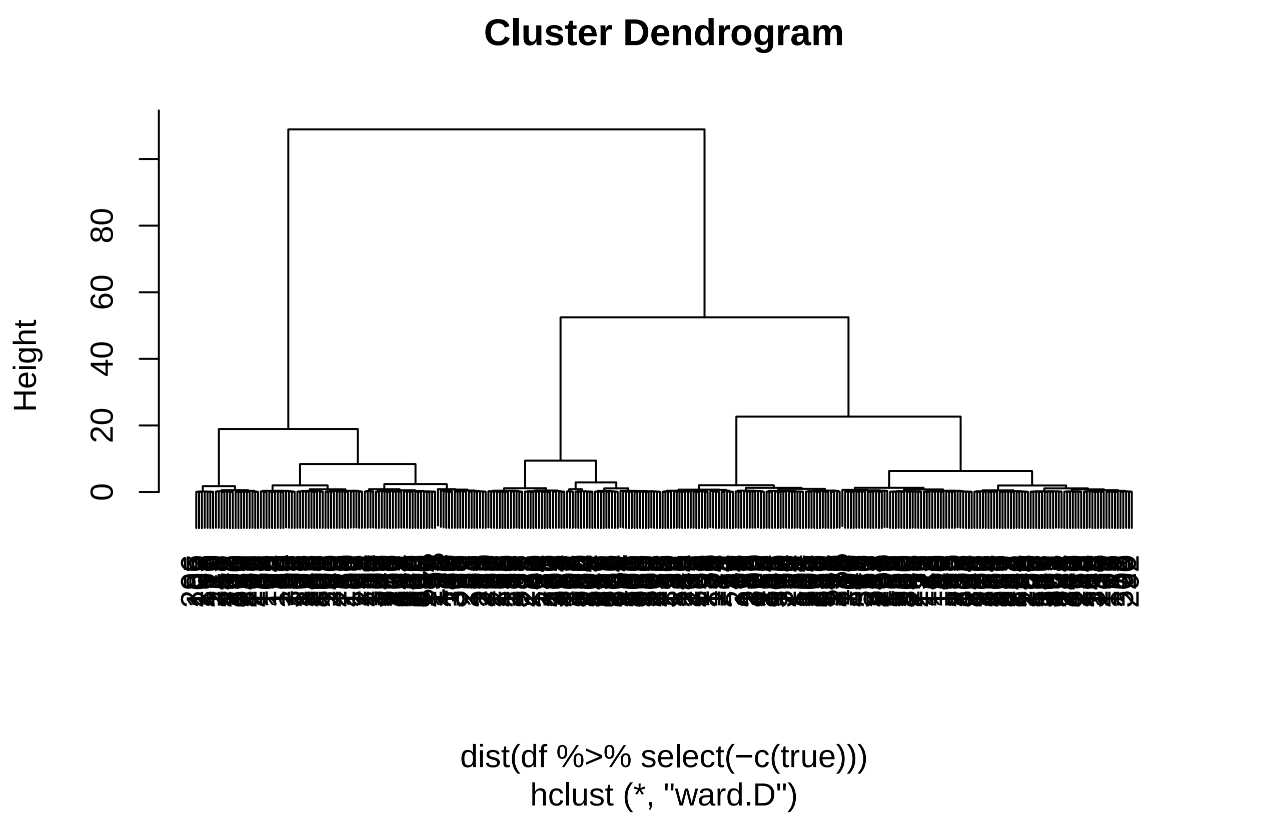
```
heatmap(enc_output)
```



The heatmap signals that the 10 nodes in the bottleneck have not been able to separate between the images, displayed row-wise on the right side of the heatmap. There is no clear difference in color in any of the columns, comparing different image numbers, meaning that it does not look like the nodes are able to represent any patterns, i.e. any split between the two types of images. We investigate this further.

The first method we use with the intention of understanding what is going on in the bottleneck, is Hierarchical Clustering. We cluster the predictions below.

```r
df.enc <- as.data.frame(enc_output)
df <- cbind(df.enc, "true" = test.image_array_gen$classes)
hcl1 <- hclust(dist(df %>% select(-c(true))) , method="ward.D") # Euclidean distance, ward.D method.
plot(hcl1)
```



We cut the dendrogram at two clusters, since we know that the data is split up into two clusters.

```
clusts <- cutree(hcl1, 2)
table(clusts)
```

```
#> clusts
#>   1   2
#> 230 104
```

```
# Add the cluster label to each of the data points.
df$cluster <- clusts
```

If the model has been able to generalize the data, there should be about half and half of each of the clusters, meaning 167 points in each class. This is not entirely true, as seen above, but it is not terrible at first glance. Since we know the true labels, it is interesting to see what percentage of cluster indices are different from the true labels.

```
count(df %>% filter(true != (cluster-1)))
```

```
#>     n
#> 1 183
```

We can see that a large amount of the data set - over half! This might be because of a reversed order of naming the clusters, but the number is still large. Let us investigate how many of each of the true labels are clustered in each of the two clusters.

```
table(df %>% select(c(true, cluster)))
```

```
#>     cluster
#> true   1   2
#>    0 107  60
#>    1 123  44
```

This does not look promising; the clusters do not clearly differentiate between normal X-rays (labeled 1) and X-rays with effusion (labeled 0).

We do the same investigation with K-means clustering, with $K = 2$.

```r
df2.enc <- as.data.frame(enc_output)
df2 <- cbind(df2.enc, "true" = test.image_array_gen$classes)
km <- kmeans(df2 %>% select(-c(true)), 2)
df$kmcluster <- km$cluster

# Count the number of wrongly predicted clusters.
count(df %>% filter(true != (kmcluster-1)))
```

```
#>     n
#> 1 183
```

The amount of wrongly clustered images is slightly lower compared with hierarchical clustering (given that the naming convention is the same in the two methods). We check how many clusters are unequal between the two methods.

```r
count(df %>% filter(kmcluster == cluster))
```

```
#>     n
#> 1 282
```

We can see that most of the assigned clusters are different, meaning that the two methods do not agree (given that they have labeled the clusters in the similar "direction". We will see in the plots later that the two clustering methods have probably just flipped the order in which the clusters are given their names, since they look very similar there). The fact that these auxiliary methods do not always agree is something to keep in mind, because they influence the conclusions we make about what the bottleneck in the CAE has done, which is our main objective here. How does the clustering look compared to the true labels for K-means clustering?

```r
table(df %>% select(c(true, kmcluster)))
```

```
#>     kmcluster
#> true  1  2
#>    0 81 86
#>    1 97 70
```

Still not promising.

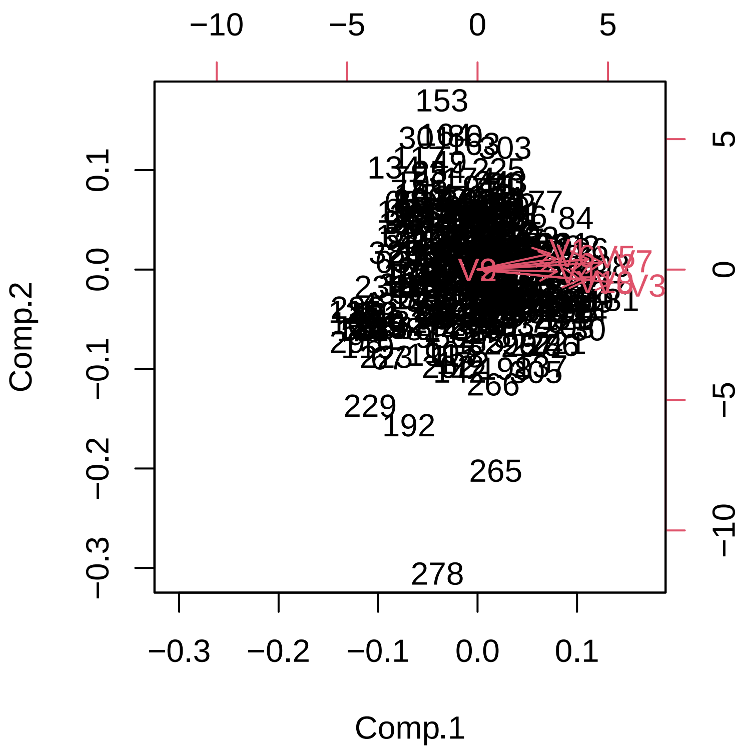
In order to even better illustrate what the bottleneck does with the images, i.e. how it represents the images, we reduce the dimensions of the predictions using Principal Component Analysis (PCA).

```r
pca <- princomp(df %>% select(-c(true, cluster, kmcluster)))
summary(pca)
```

```
#> Importance of components:
#>                           Comp.1     Comp.2     Comp.3      Comp.4      Comp.5
#> Standard deviation     0.7711388 0.07860712 0.06714155 0.061628978 0.042855399
#> Proportion of Variance 0.9683616 0.01006228 0.00734100 0.006185039 0.002990774
#> Cumulative Proportion  0.9683616 0.97842392 0.98576492 0.991949959 0.994940733
#>                             Comp.6      Comp.7     Comp.8 Comp.9 Comp.10
#> Standard deviation     0.035412771 0.031593561 0.02923346      0       0
#> Proportion of Variance 0.002042172 0.001625435 0.00139166      0       0
#> Cumulative Proportion  0.996982905 0.998608340 1.00000000      1       1
```

We can see that the first principal component seems to capture almost all the variance in the predictions.

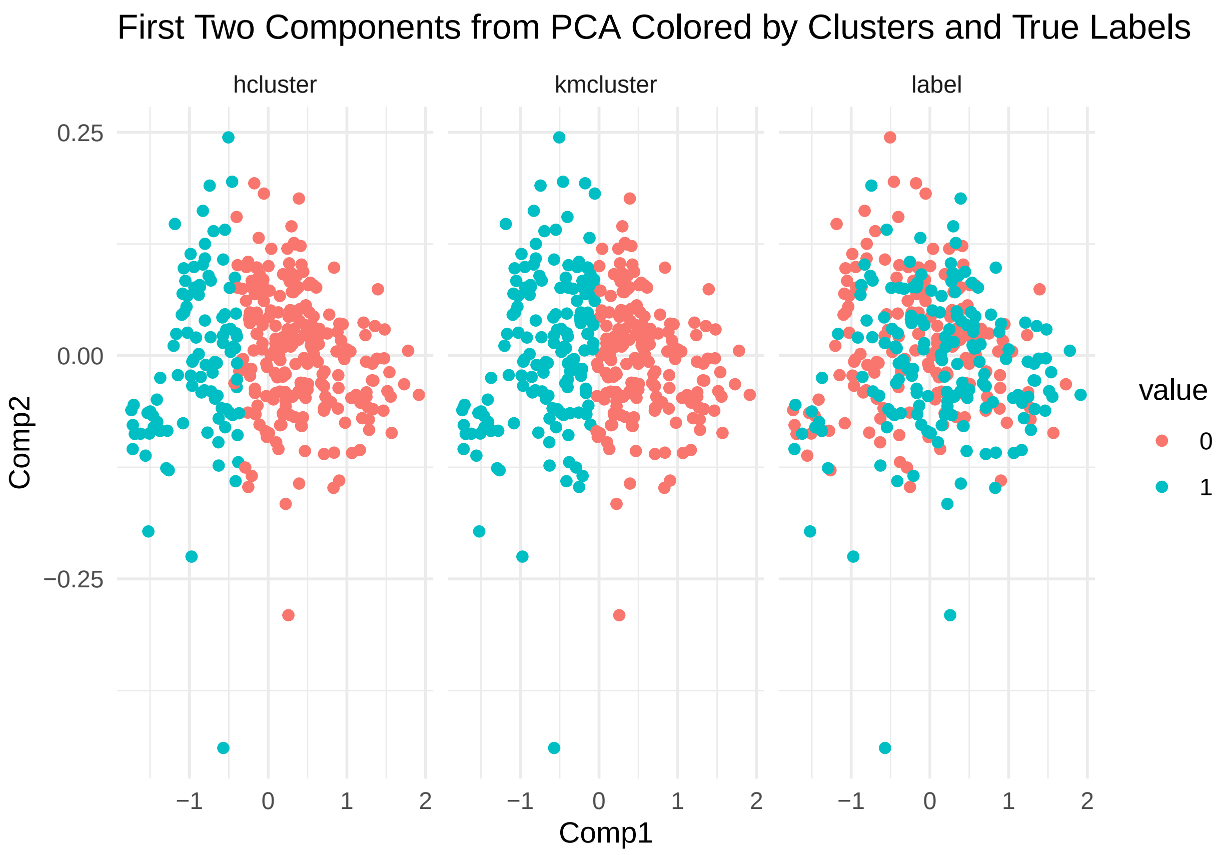
```
biplot(pca)
```



From the biplot we can see that most nodes have large loadings in the first component, while the second component almost does not differentiate between them. No clear clustering tendencies in the scores.

We plot the first component scores, coloring in the labels of the type of image (normal or effusion) and the clusters of each score.

```
df.plotting <- data.frame(Comp1 = pca$scores[,1], Comp2 = pca$scores[,2], label = factor(df$true),
                          hcluster = factor(df$cluster-1), kmcluster = factor(df$kmcluster-1))

df.final.plot <- df.plotting %>% gather(key = type, value = value, -c(Comp1, Comp2))

#df2 <- data.frame(Comp1 = pca$scores[,1], Comp2 = pca$scores[,2], label = factor(df$true))
df.final.plot %>% ggplot(aes(x = Comp1, y = Comp2)) +
   geom_point(aes(color = value)) +
   theme_minimal() +
   ggtitle("First Two Components from PCA Colored by Clusters and True Labels") +
   facet_wrap(~type)
```

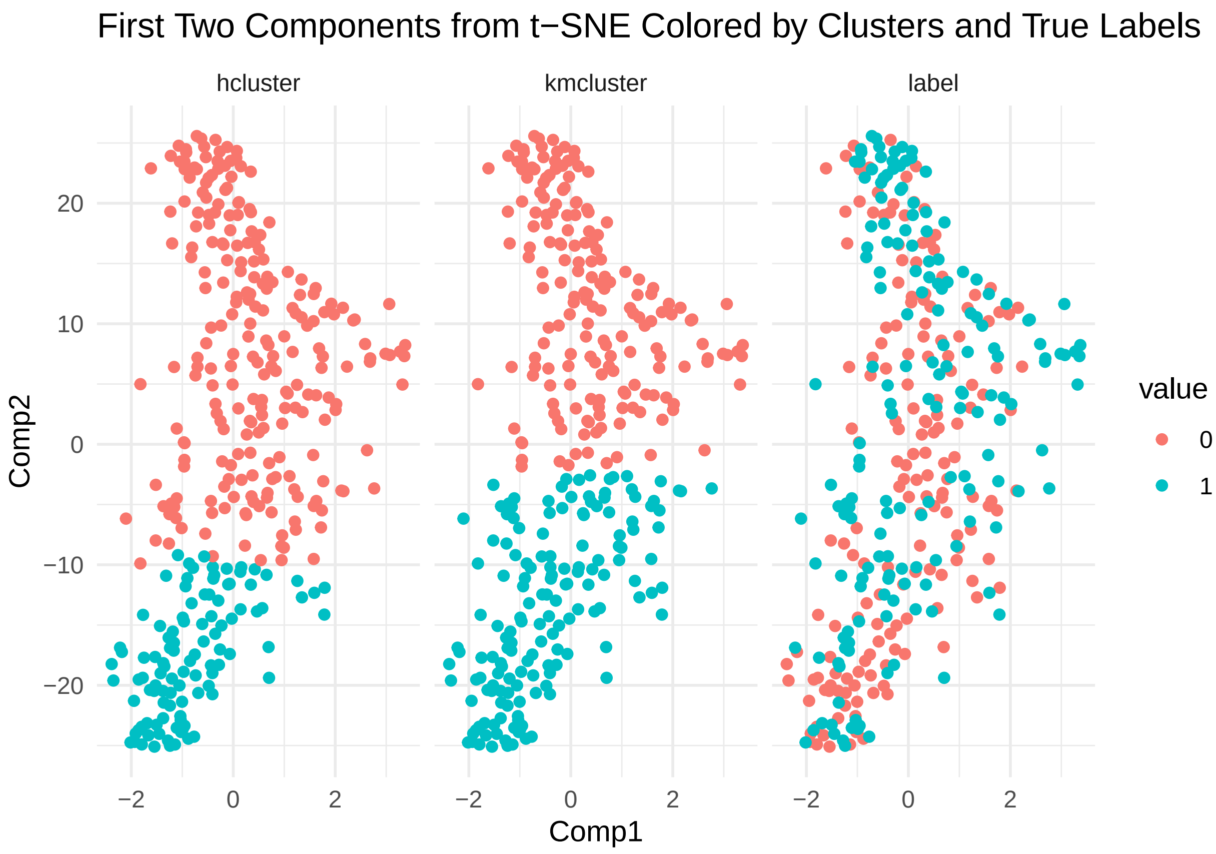


```r
# df3 <- data.frame(Comp1 = pca$scores[,1], Comp2 = pca$scores[,2], cluster = factor(df$cluster))
# df3 %>% ggplot(aes(x = Comp1, y = Comp2)) +
#    geom_point(aes(color = cluster)) +
#    theme_minimal() +
#    ggtitle("First Two Components from PCA Colored by Cluster")
```

This plot is very illustrative. We can see that the clusters are clearly separable in the two first principal components, while the true labels are not at all. Thus, it seems like the method of PCA, which is a linear dimensionality reduction method, cannot capture non-linear relationships in the bottleneck. It is unsure if this observed non-separability of the true images following the predictions using the encoder is a result of the encoder itself, i.e. the bottleneck not being able to differentiate between them, or if it is a drawback of PCA. Therefore, we try a non-linear dimensionality reduction technique as well. There exists many different such methods, for example ISOMAP, Kernel PCA, Local MDS and t-SNE.

Our non-linear dimensionality reduction method of choice is t-SNE. We use it to reduce to two dimensions, which might be very strict. The results from the algorithm are plotted below. Notice that the hyperparameter `perplexity` needs to be set for this algorithm, which is set somewhat arbitrarily or heuristically in this case.

```r
tsne <- Rtsne(df %>% select(-c(true, cluster, kmcluster)), dims = 2, perplexity = 40, theta = 0)
tsne.points <- tsne$Y

df.plotting2 <- data.frame(Comp1 = tsne.points[,1], Comp2 = tsne.points[,2], label = factor(df$true),
                           hcluster = factor(df$cluster-1), kmcluster = factor(df$kmcluster-1))

df.final.plot2 <- df.plotting2 %>% gather(key = type, value = value, -c(Comp1, Comp2))

#df2 <- data.frame(Comp1 = pca$scores[,1], Comp2 = pca$scores[,2], label = factor(df$true))
df.final.plot2 %>% ggplot(aes(x = Comp1, y = Comp2)) +
   geom_point(aes(color = value)) +
   theme_minimal() +
   ggtitle("First Two Components from t-SNE Colored by Clusters and True Labels") +
   facet_grid(~type)
```



Also for this method, we can see that we cannot separate between the normal and the effusion X-ray images in two dimensions. I am inclined to conclude that the CAE is not able to represent the two different types of images to a sufficient accuracy with only 10 nodes in the bottleneck, as briefly discussed earlier when trying to use the CAE as a pre-training model. Thus, it seems to me like the model does a poor job at differentiating between normal images and X-ray images with effusion in the bottleneck.

10 Final Remarks and Further Work

As we have seen, neither of the models employed do an impressingly good job in separating between normal X-ray images and images with effusion. Because of shortage of time close to the end of the spring semester, it was not possible to keep improving these models. I have already tried to combat overfitting problems (especially in the CNN) by using data augmentation in the image generators, adding dropout and L2 kernel regularization. This is something I would have liked to do to a greater extent before delivering this report, but perhaps it can be done after. Notice also that we do not have a lot of data, which for sure is a reason to why the models do not yield greater performance. Lastly, I think the CAE suffers from the restriction of having 10 nodes in the bottleneck, as I found better performance when having convolutional layers in the bottleneck, but I understand that the purpose here is to analyze a bit more what happens in the bottleneck, which is easier with fewer nodes.