ML-Tools-Assignment-2

Gyongyver Kamenar (2103380)

3/27/2022

```
library(tidyverse)
library(kableExtra)
library(keras)
library(ggplot2)
library(imager)
```

Problem 1

A)

What would be an appropriate metric to evaluate your models? Why?

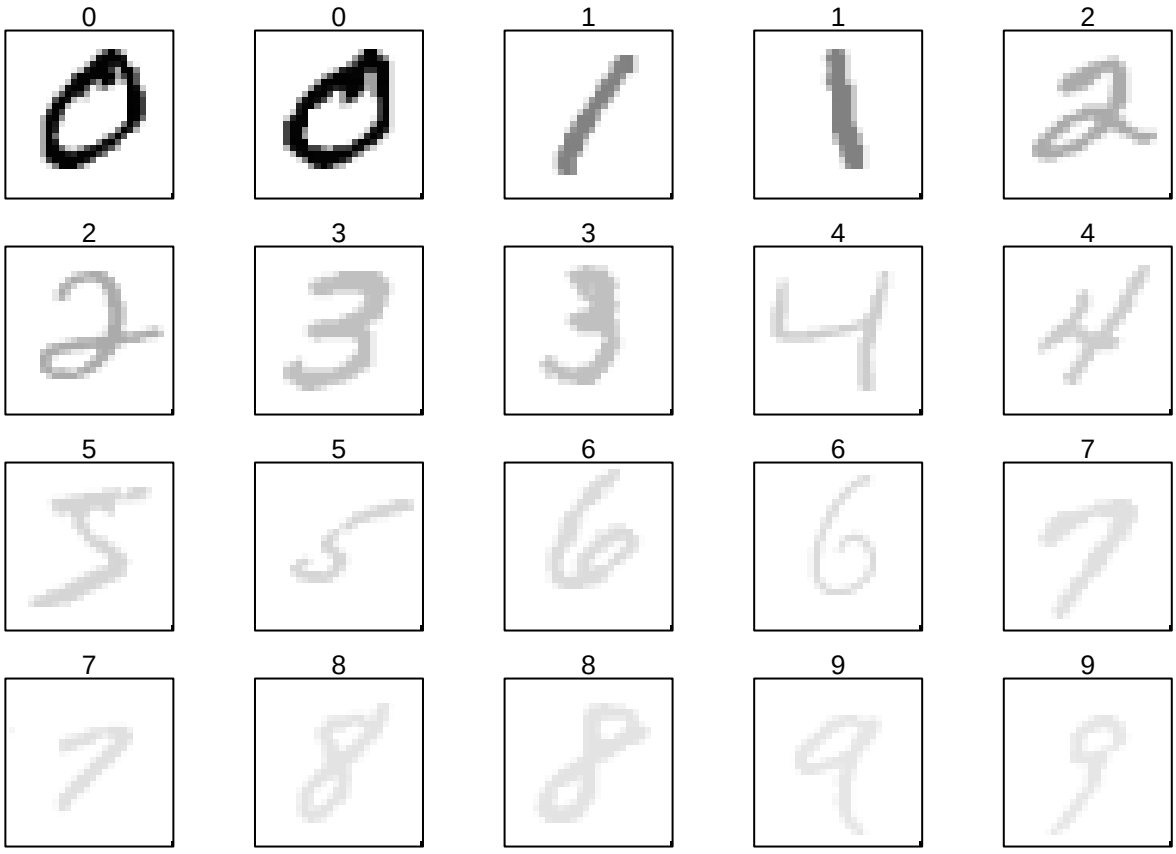
Accuracy would be an appropriate metric here, because it reports the correctly clasified images as the percentage of all and neglects the tradeoff between false classification. We can assume in this digit recognition project, that false classifications have equal loss. For example, we want to classify a digit 1 to label 1, but if it's not successful aka we did not classified it as 1, it does not matter which false label it has. I does not matter whether it's classified as 0,2,3,4 ... so on, anything other than 1. So the only thing matters to us it the rate of correctly classified cases, which is captured by accuracy.

B)

Get the data and show some example images from the data.

```
my_seed<-20200403
set.seed(my_seed)
# Get the data
data <-dataset_mnist()
ind <- sample(nrow(data$train$x),round(nrow(data$train$x)*0.8,0))

# train set
train_x<-as.matrix(as_tibble(data$train$x)[ind,])
train_y<-as.matrix(as_tibble(data$train$y)[ind,])

# validation set
valid_x<-as.matrix(as_tibble(data$train$x)[-ind,])
valid_y<-as.matrix(as_tibble(data$train$y)[-ind,])

# test set
test_x<-as.matrix(as_tibble(data$test$x))
test_y<-as.matrix(as_tibble(data$test$y))

# Merge features with labels for the plot
train <-cbind(as.tibble(data$train$x),y=data$train$y)

# Factorizing and scaling for plotting
```

```r
train <- train %>% mutate(
  y = as.factor(y),
  across(-y, ~./255)
)



# Plot the images
rotate <- function(x) t(apply(x, 2, rev))
par(mfrow=c(4, 5), pty='s', mai=c(0.1, 0, 0.15, 0.1))
for (lab in 0:9) {
  samp <- train %>%
    filter(y == lab)
  for (i in 1:2) {
    img <- matrix(as.numeric(samp[i, -1]), 28, 28, byrow = TRUE)
    image(t(rotate(img))[,28:1], axes = FALSE, col = grey(seq(1, 0, length = 256)),xlim = c(1,0))
    box(lty = 'solid')
    title(main = lab, font.main=22)
  }
}
```



C)

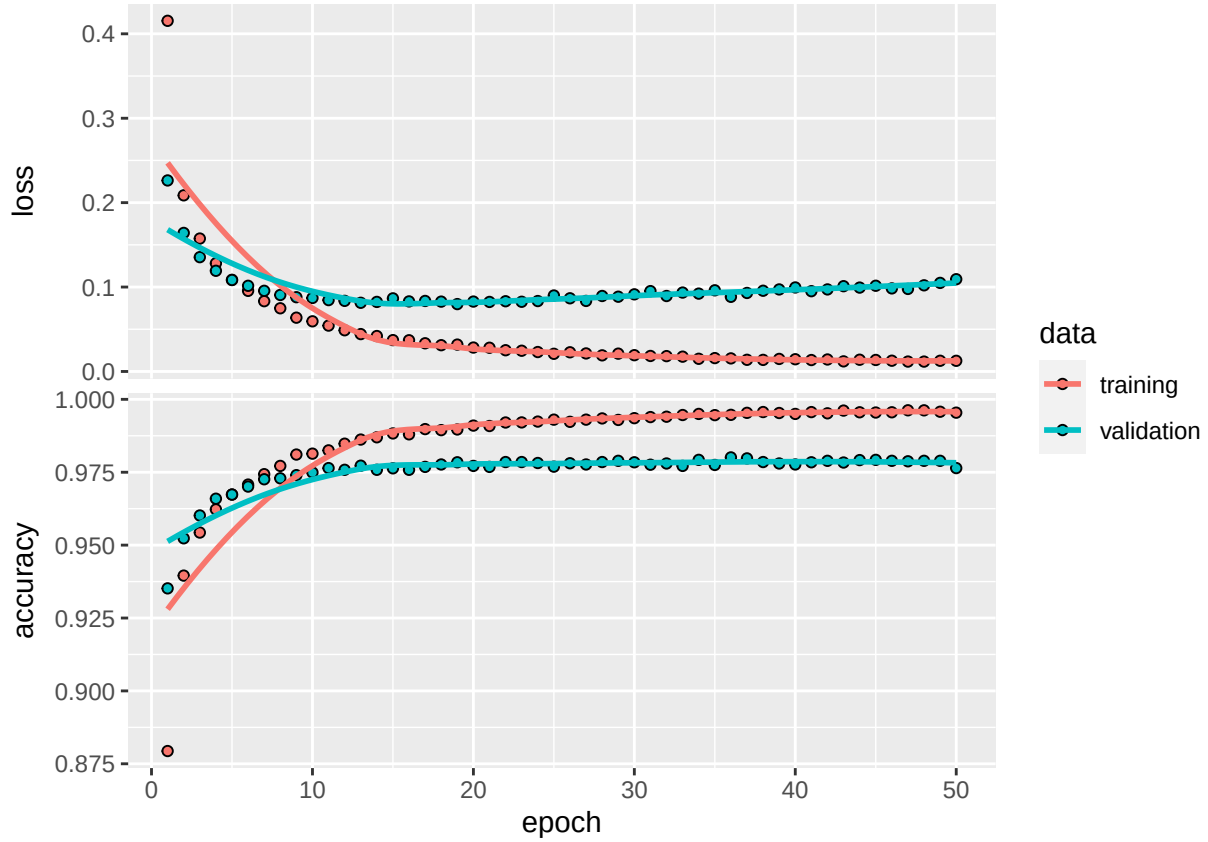
Train a simple fully connected network with a single hidden layer to predict the digits. Do not forget to normalize the data similarly to what we saw with FMNIST in class.

First, I normalized and converted the data into correct format. Then I build a model with 1 hidden layer using 128 neurons and the 'Relu' activation function. I used 0.2 dropout rate and output layer with 10 neurons using the softmax activation function. See the model summary below.

```
train_x<-array_reshape(train_x/255,c(nrow(train_x),784))
valid_x<-array_reshape(valid_x/255,c(nrow(valid_x),784))
test_x<-array_reshape(test_x/255,c(nrow(test_x),784))

train_y <- to_categorical(train_y, num_classes = 10)
valid_y <- to_categorical(valid_y, num_classes = 10)
test_y <- to_categorical(test_y, num_classes = 10)

simple_keras <- keras_model_sequential()
simple_keras |>
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(simple_keras)
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  dense_1 (Dense)                    (None, 128)                     100480
##
##  dropout (Dropout)                  (None, 128)                     0
##
##  dense (Dense)                      (None, 10)                      1290
##
## =========================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## _________________________________________________________________________
```

```
set.seed(my_seed)
compile(
    simple_keras,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits1_history <-fit(
    simple_keras, train_x, train_y,
    epochs = 50, batch_size = 100,
    validation_data = list(valid_x, valid_y)
)

# Plot results
plot(digits1_history)
```




```
digits1_history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.01262
##     accuracy: 0.9954
##     val_loss: 0.1093
## val_accuracy: 0.9764
```

```
res<-evaluate(simple_keras, valid_x, valid_y)
evaluate(simple_keras, valid_x, valid_y)
```

```
##      loss  accuracy
## 0.1092872 0.9764166
```

The accuracy on the test set is 97.64 %. This seems a good metric at first on a 10 class classification problem. If we suppose that there are the same number of observation from each categories, it would mean 10% accuracy with the simplest classification. Compared to that, the simple keras model performed really well.

D)

Experiment with different network architectures and settings (number of hidden layers, number of nodes, type and extent of regularization, etc.). Train at least 5 models. Explain what you have tried, what worked and what did not. Make sure that you use enough epochs so that the validation error starts flattening out - provide a plot about the training history.

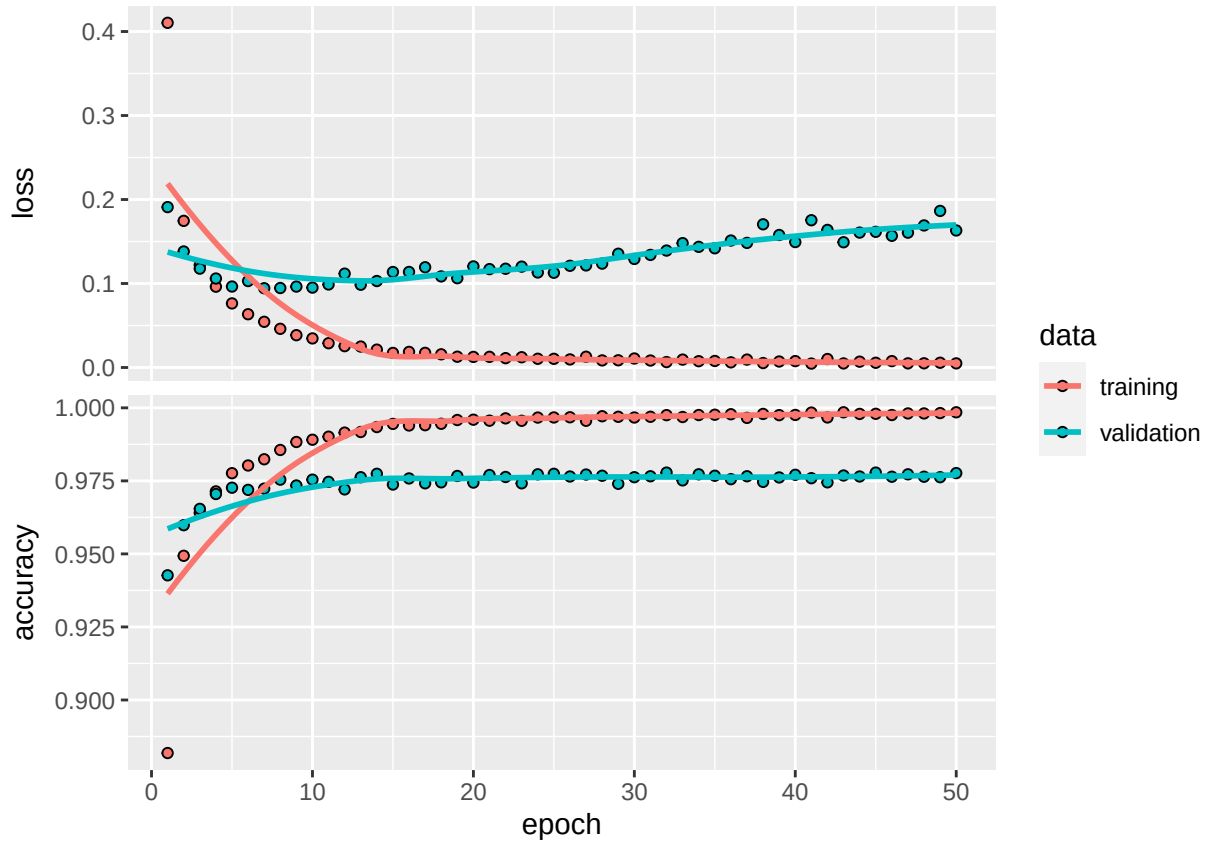
In my second model, I added one more hidden layer with 64 neurons and relu activation function. I expect that due to the plus layer, the model will perform better.

```
extended_keras1 <- keras_model_sequential()
extended_keras1 |>
```

```
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(extended_keras1)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_4 (Dense)                    (None, 128)                     100480
##
##  dense_3 (Dense)                    (None, 64)                      8256
##
##  dropout_1 (Dropout)                (None, 64)                      0
##
##  dense_2 (Dense)                    (None, 10)                      650
##
## ================================================================================
## Total params: 109,386
## Trainable params: 109,386
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
set.seed(my_seed)
compile(
    extended_keras1,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits2_history <-fit(
    extended_keras1, train_x, train_y,
    epochs = 50, batch_size = 100,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits2_history)
```




```
digits2_history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.004925
##     accuracy: 0.9985
##     val_loss: 0.1631
## val_accuracy: 0.9777
```

We can see, that a valuation accuracy increased from 0.9764 to 0.9777 in the second model. Besides, we can also see on the training plot, that the valuation accuracy reached this a bit earlier but is started from higher value, it is not so steep in the first few epochs as the simple model.

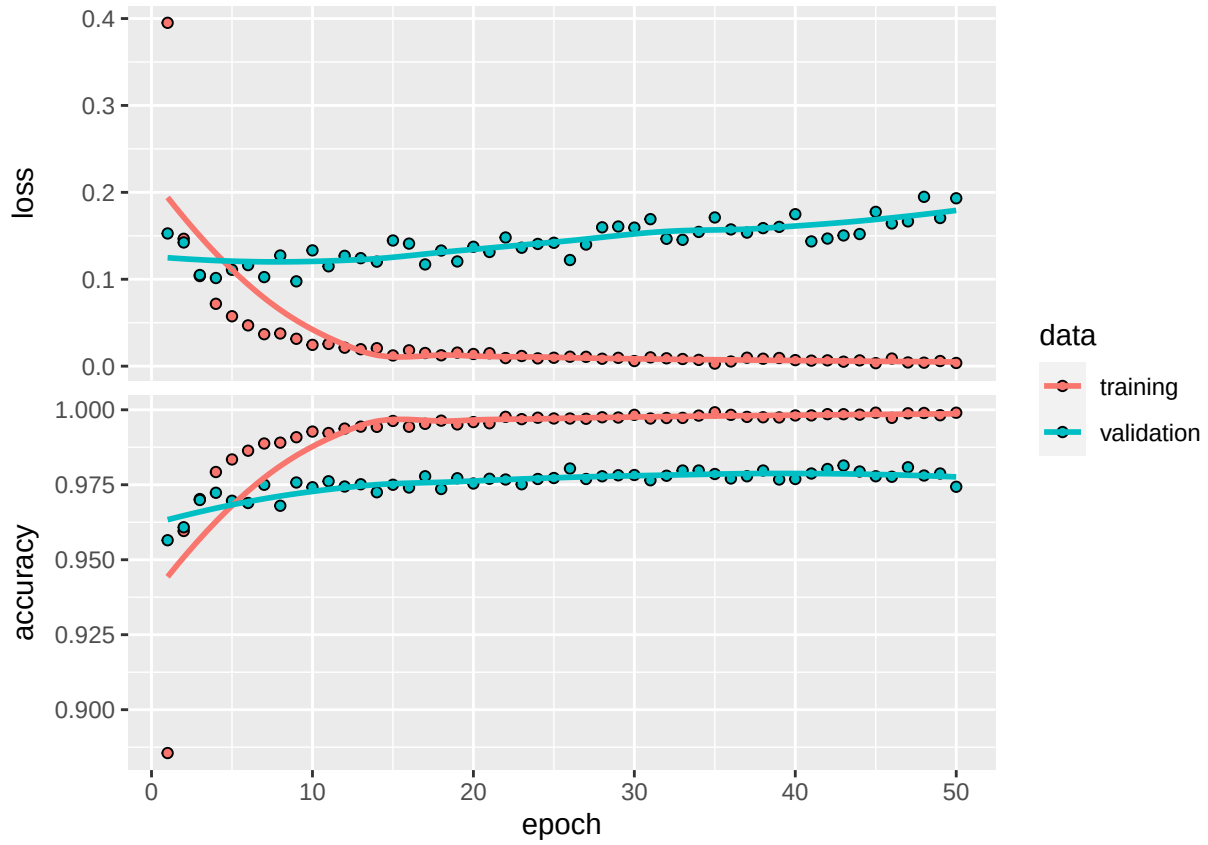
In my 3rd model, I add two more layers and more neurons, to test, whether it will similarly increase accuracy.

```
extended_keras2 <- keras_model_sequential()
extended_keras2 |>
    layer_dense(units = 256, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'relu', input_shape = c(784)) |>
  layer_dense(units = 32, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(extended_keras2)
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_9 (Dense)                    (None, 256)                     200960
```

```
##
##  dense_8 (Dense)                  (None, 128)                   32896
##
##  dense_7 (Dense)                  (None, 64)                    8256
##
##  dense_6 (Dense)                  (None, 32)                    2080
##
##  dropout_2 (Dropout)              (None, 32)                    0
##
##  dense_5 (Dense)                  (None, 10)                    330
##
## ================================================================================
## Total params: 244,522
## Trainable params: 244,522
## Non-trainable params: 0
## --------------------------------------------------------------------------------
```

```
set.seed(my_seed)
compile(
    extended_keras2,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits3_history <-fit(
    extended_keras2, train_x, train_y,
    epochs = 50, batch_size = 100,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits3_history)
```




```
digits3_history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.00367
##     accuracy: 0.999
##     val_loss: 0.1932
## val_accuracy: 0.9743
```

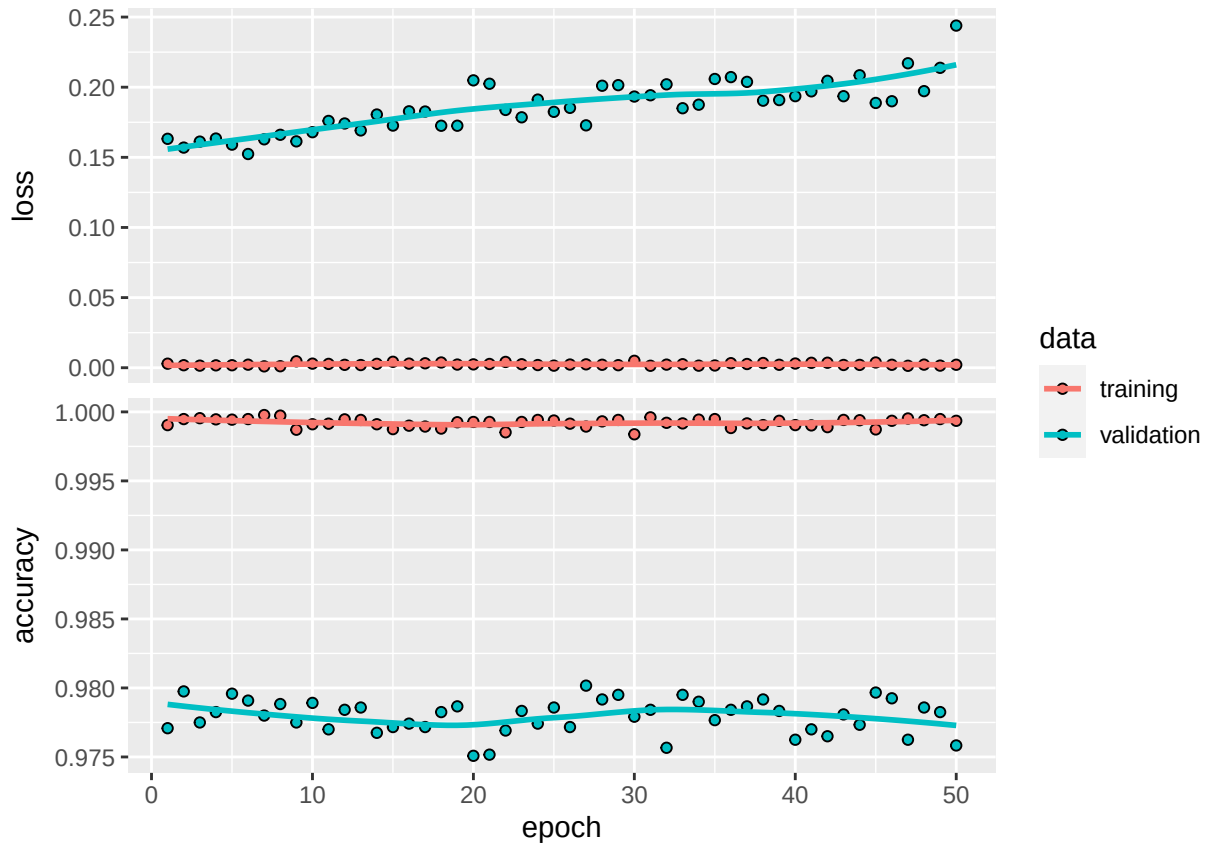
In the next model, I increase batch size from 100 to 200, and I use 2 hidden layers just like in the second model.

```
extended_keras3 <- keras_model_sequential()
extended_keras3 |>
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(extended_keras3)
```

```
## Model: "sequential_3"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_12 (Dense)                   (None, 128)                     100480
##
##  dense_11 (Dense)                   (None, 64)                      8256
##
##  dropout_3 (Dropout)                (None, 64)                      0
##
```

```
##  dense_10 (Dense)                    (None, 10)                        650
##
## ================================================================================
## Total params: 109,386
## Trainable params: 109,386
## Non-trainable params: 0
## _________________________________________________________________________________
```

```
set.seed(my_seed)
compile(
    extended_keras3,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits4_history <-fit(
    extended_keras1, train_x, train_y,
    epochs = 50, batch_size = 200,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits4_history)
```




```
digits4_history
```

```
##
## Final epoch (plot to see history):
```

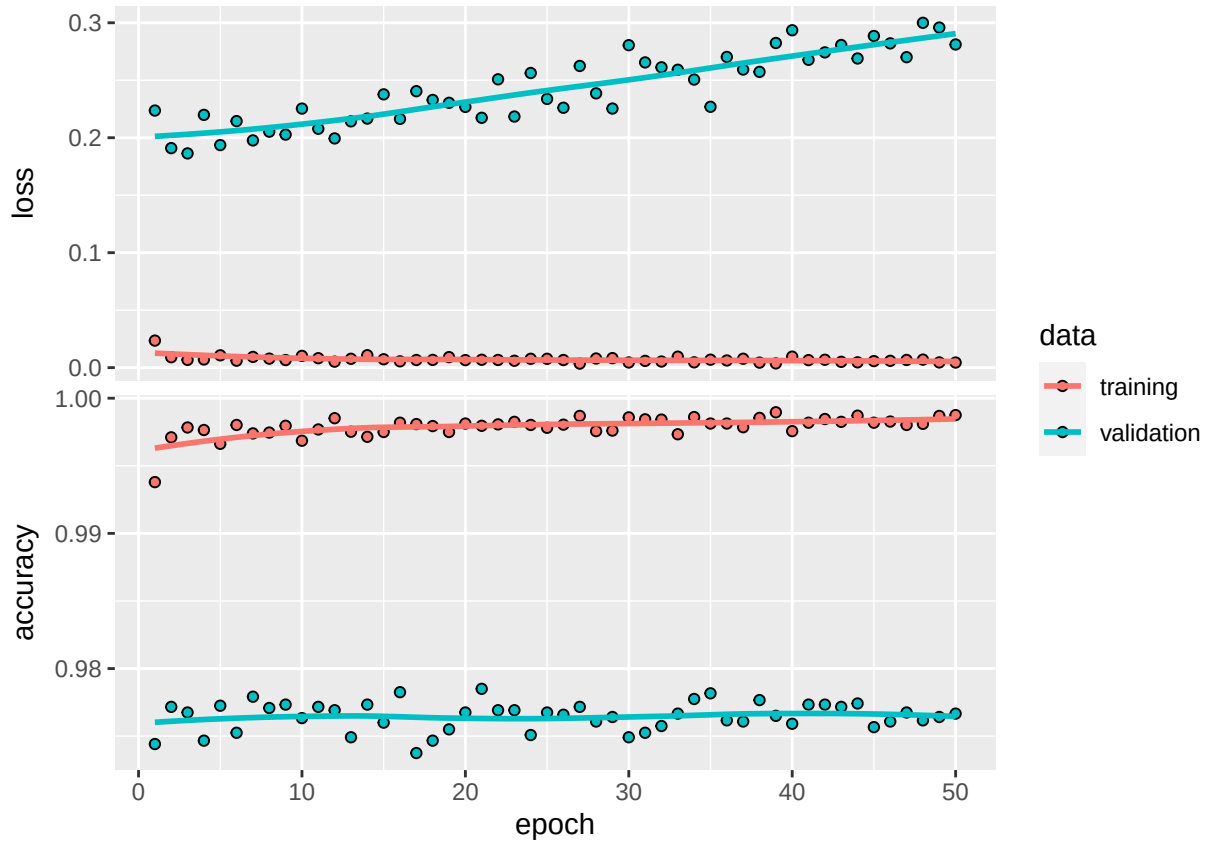
```
##         loss: 0.002116
##     accuracy: 0.9994
##     val_loss: 0.2439
## val_accuracy: 0.9758
```

The bigger batch size clearly not helped the model. The valuation accuracy is continously decreasing. Therefore, I try the same with smaller batch size = 50.

```
extended_keras4 <- keras_model_sequential()
extended_keras4 |>
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(extended_keras4)
```

```
## Model: "sequential_4"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  dense_15 (Dense)                   (None, 128)                     100480
##
##  dense_14 (Dense)                   (None, 64)                      8256
##
##  dropout_4 (Dropout)                (None, 64)                      0
##
##  dense_13 (Dense)                   (None, 10)                      650
##
## =========================================================================
## Total params: 109,386
## Trainable params: 109,386
## Non-trainable params: 0
## _________________________________________________________________________
```

```
set.seed(my_seed)
compile(
    extended_keras4,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits5_history <-fit(
    extended_keras1, train_x, train_y,
    epochs = 50, batch_size = 50,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits5_history)
```




```
digits5_history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.004452
##     accuracy: 0.9987
##     val_loss: 0.2811
## val_accuracy: 0.9767
```

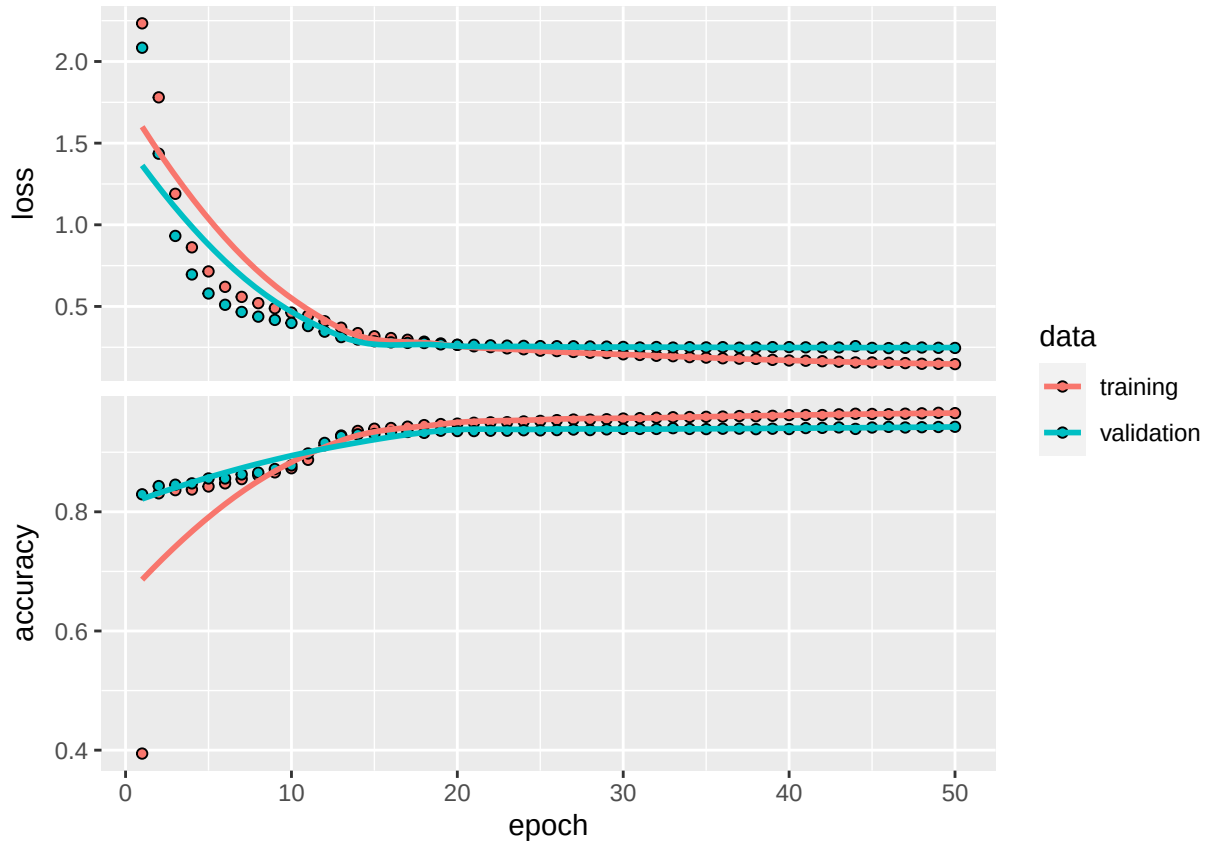
It seems like the smaller batch size also performed worse than model 2 and model 1 with batch size=100, so it seems like 100 is about the appropriate value in this setting. In the 5th extended model, I try using other activation function, softmax instead of the previously used relu in the hidden layers.

```
extended_keras5 <- keras_model_sequential()
extended_keras5 |>
    layer_dense(units = 128, activation = 'softmax', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'softmax', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'softmax')

summary(extended_keras5)
```

```
## Model: "sequential_5"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_18 (Dense)                   (None, 128)                     100480
##
##  dense_17 (Dense)                   (None, 64)                      8256
##
```

```
##  dropout_5 (Dropout)                (None, 64)                      0
##
##  dense_16 (Dense)                   (None, 10)                      650
##
## ================================================================================
## Total params: 109,386
## Trainable params: 109,386
## Non-trainable params: 0
## --------------------------------------------------------------------------------
```

```
set.seed(my_seed)
compile(
    extended_keras5,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

digits6_history <-fit(
    extended_keras5, train_x, train_y,
    epochs = 50, batch_size = 100,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits6_history)
```



```
digits6_history
```

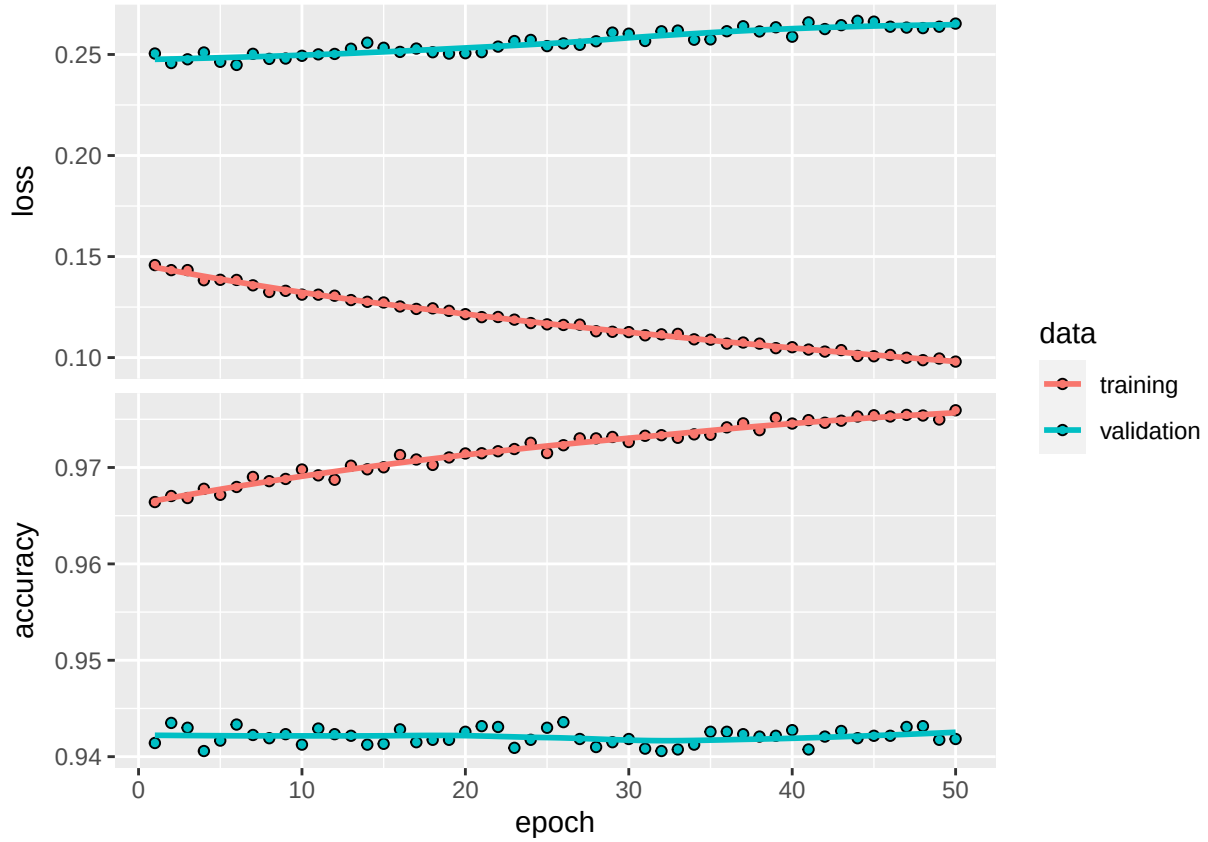
```
##
## Final epoch (plot to see history):
##         loss: 0.1468
##     accuracy: 0.9656
##     val_loss: 0.2459
## val_accuracy: 0.9427
```

We can see from the accuracy measures, that the softmax activation function performes worse than the relu. So in my last model, I set all of them to relu and also change the optimizer from adam to rmsprop.

```
extended_keras6 <- keras_model_sequential()
extended_keras6 |>
    layer_dense(units = 128, activation = 'relu', input_shape = c(784)) |>
    layer_dense(units = 64, activation = 'relu', input_shape = c(784)) |>
    layer_dropout(rate = 0.2) |>
    layer_dense(units = 10, activation = 'relu')

summary(extended_keras6)
```

```
## Model: "sequential_6"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  dense_21 (Dense)                   (None, 128)                     100480
##
##  dense_20 (Dense)                   (None, 64)                      8256
##
##  dropout_6 (Dropout)                (None, 64)                      0
##
##  dense_19 (Dense)                   (None, 10)                      650
##
## ================================================================================
## Total params: 109,386
## Trainable params: 109,386
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
set.seed(my_seed)
compile(
    extended_keras6,
    loss = 'categorical_crossentropy',
    optimizer = optimizer_rmsprop(),
    metrics = c('accuracy')
)

digits7_history <-fit(
    extended_keras5, train_x, train_y,
    epochs = 50, batch_size = 100,
    validation_data = list(valid_x, valid_y)
)

# Plot results

plot(digits7_history)
```




```
digits7_history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.09802
##     accuracy: 0.9759
##     val_loss: 0.2653
## val_accuracy: 0.9418
```

This model laso performed poorly compared to the first two, so I definiately wanto to use a model with adam optimizer.

E)

Choose a final model and evaluate it on the test set. How does test error compare to validation error?

My chose model is the first one, the simple model with 1 hidden layer. It's simple, and it has the highest validation accuracy. See the summary below.

```
summary(simple_keras)
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  dense_1 (Dense)                    (None, 128)                     100480
##
##  dropout (Dropout)                  (None, 128)                     0
##
##  dense (Dense)                      (None, 10)                      1290
##
```

```
## ================================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## --------------------------------------------------------------------------------
```

Evaluation on the test set:

```
evaluate(simple_keras,test_x,test_y)
```

```
##      loss  accuracy
## 0.1103476 0.9792000
```

The test accuracy is around the same/ a bit higher than the validation accuracy on the model. It's still a bit lower than the training accuracy, because that is around 100%. It's a good sign that the test is higher than the validation accuracy, it can be partly due lack, partly due to the huge smaller sample size of the test. This model is certainly not overfitted on the validation set, so I would be confident using it.

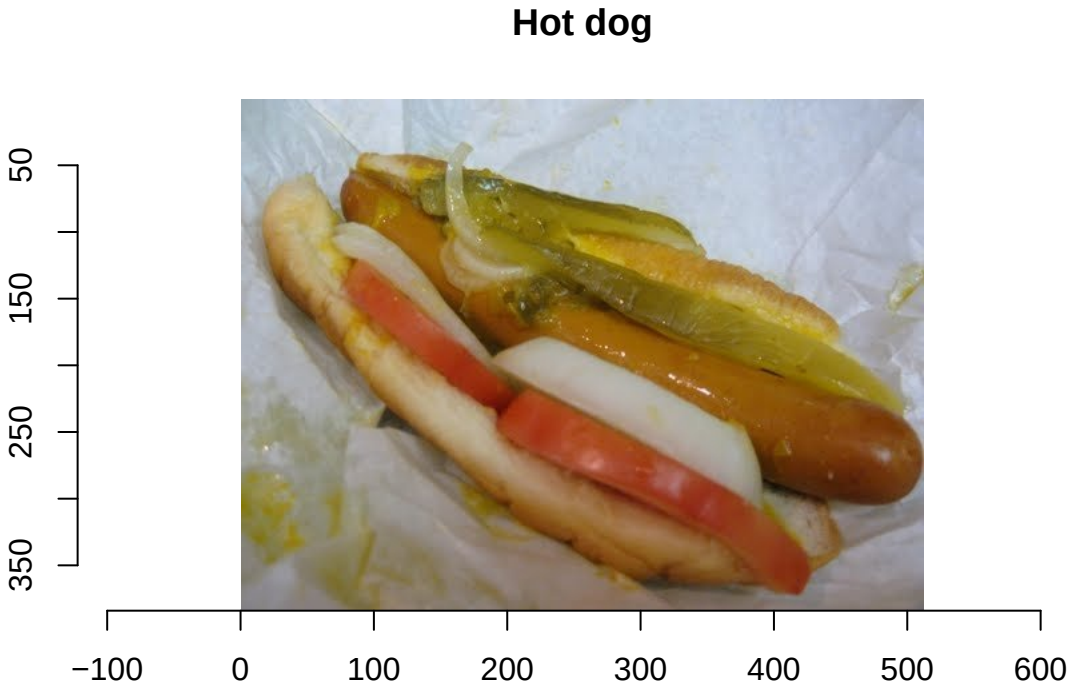
Problem 2

A)

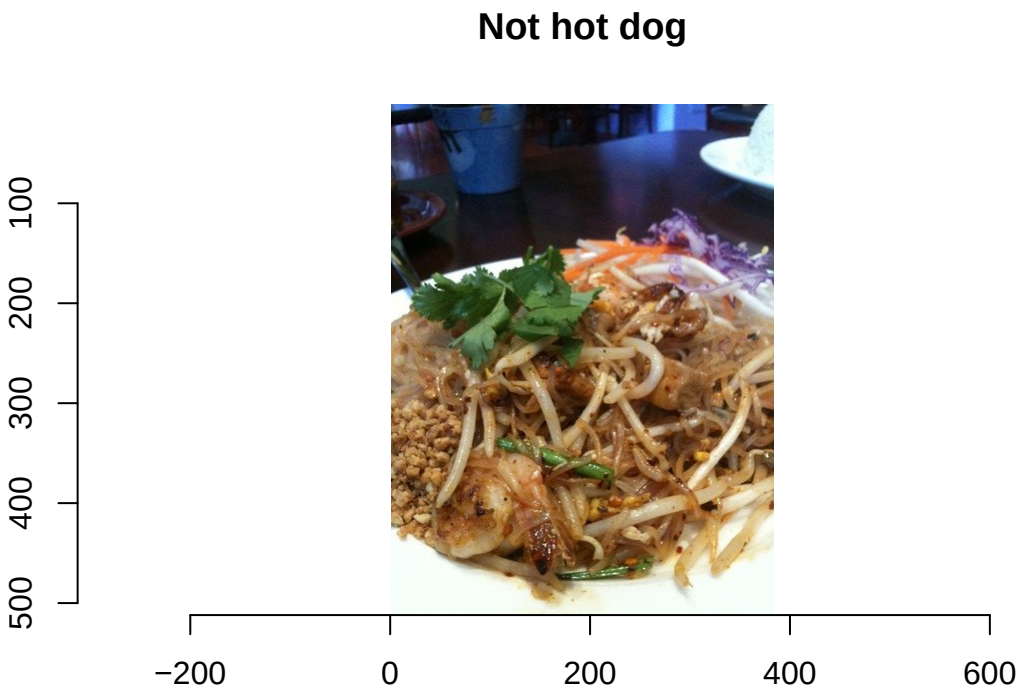
Show two images: one hot dog and one not hot dog. (Hint: You may use knitr::include_graphics() or install the imager package to easily accomplish this.)

Showing a hot dog and a not hot dog:

```
#Show hot dog
path<-"C:/CEU/DS1/ceu-ds1/data/train/"
hot_dog<-load.image(paste0(path,"hot_dog/7896.jpg"))
plot(hot_dog, main="Hot dog")
```



```r
# Show not hot dog
not_hot_dog<-load.image(paste0(path,"not_hot_dog/4770.jpg"))
plot(not_hot_dog, main="Not hot dog")
```



B)

What would be a good metric to evaluate such a prediction?

Accuracy would be also a good metric here, because we want to maximize the number of correctly classified images and we are indifferent between false negative and false positive cases. So classifying a hot dog as not hot dog is equally bad than classifying a not hot dog as hot dog in this case, only we case about the ratio of correctly classified images.

C)

To be able to train a neural network for prediction, let's first build data batch generator functions as we did in class for data augmentation (train_generator and valid_generator).

I did the necessary steps to build data batch generator function from directory as you can see below:

```r
batch_size <- 32

train_datagen <- image_data_generator(rescale = 1/255)

valid_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
    directory="C:/CEU/DS1/ceu-ds1/data/train/",
    class_mode = "binary",
    generator = train_datagen,
```

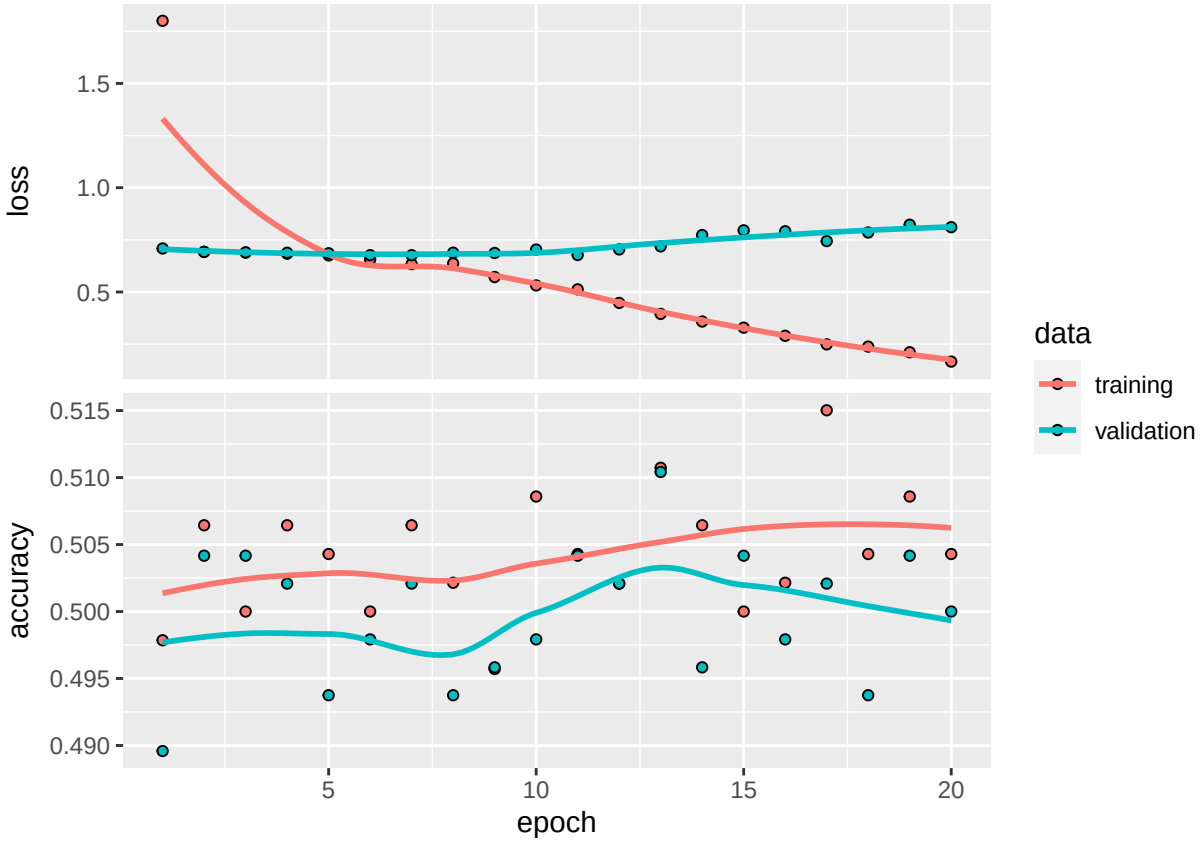
```
    batch_size = batch_size,
    target_size = c(128, 128)
)

valid_generator <- flow_images_from_directory(
    directory="C:/CEU/DS1/ceu-ds1/data/test/",
    class_mode = "binary",
    generator = valid_datagen,
    batch_size = batch_size,
    target_size = c(128, 128)
)
```

D)

Build a simple convolutional neural network (CNN) to predict if an image is a hot dog or not. Evaluate your model on the test set. (Hint: Account for the fact that you work with colored images in the input_shape parameter: set the third dimension to 3 instead of 1.)

Building a simple CNN:

```
cnn_model_wo_augmentation <- keras_model_sequential()
cnn_model_wo_augmentation |>
    layer_conv_2d(
        filters = 32,
        kernel_size = c(3, 3),
        activation = 'relu',
        input_shape = c(128, 128, 3)
    ) |>
    layer_max_pooling_2d(pool_size = c(2, 2)) |>
    layer_dropout(rate = 0.2) |>
    layer_flatten() |>
    layer_dense(units = 32, activation = 'relu') |>
    layer_dense(units = 1, activation = 'softmax')
set.seed(my_seed)
compile(
    cnn_model_wo_augmentation,
    loss = 'binary_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
)

history1 <-fit(
    cnn_model_wo_augmentation,
    train_generator,
    epochs = 20,
    steps_per_epoch =floor(length(train_generator$labels)/batch_size), #8, #nrow(data_train_x) / batch_siz
    validation_data = valid_generator,
    validation_steps =floor(length(valid_generator$labels)/batch_size) #8#nrow(data_valid_x) / batch_size
)

cnn_model_wo_augmentation %>% save_model_hdf5("hotdog_cnn1.h5")

plot(history1)
```




```
history1
```

```
##
## Final epoch (plot to see history):
##         loss: 0.1663
##     accuracy: 0.5043
##     val_loss: 0.8106
## val_accuracy: 0.5
```

```
evaluate(cnn_model_wo_augmentation,valid_generator , valid_generator$labels)
```

```
##      loss  accuracy
## 0.8268489 0.5000000
```

```
keras_predictions <- predict(cnn_model_wo_augmentation, valid_generator)
predicted_labels <- k_argmax(keras_predictions) |> as.numeric()
table(valid_generator$labels, predicted_labels)
```

```
##    predicted_labels
##       0
##   0 250
##   1 250
```

The train and test accuracy are both about 50% which means, that this binary classification model is not better than random :(. (We have the same number of images in both classes.)

E)

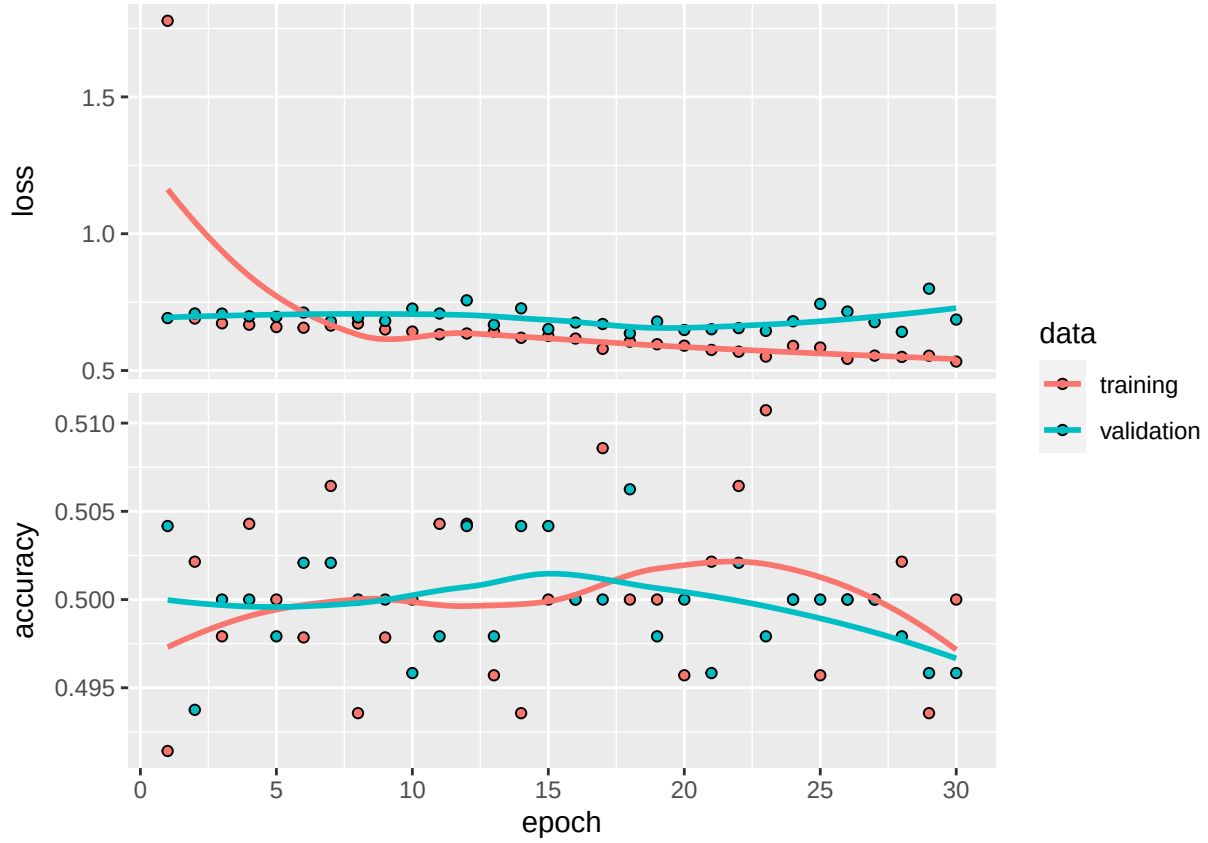
Could data augmentation techniques help with achieving higher predictive accuracy? Try some augmentations that you think make sense and compare to your previous model.

Trying to improve accuracy, I tried the following augmentation steps: - rotation range=50 : rotating the images

within 50 degrees, to randomly rotate the pictures. I suppose the 50 degrees is appropriate here, bigger rotation might just cause confusion between images. - whidth_shift and height_shift 0.1 : fraction of width and height within which randomly translate pictures horizontally and vertically. - zoom_range=0.1 : randomly zooming in pictures. The food is usually at the center of the picture, so minimal zooming cannot cause harm - shear_range=0.1 randomly applying shearing transformation - horizontal_flip : randomly flipping half the images horizontally. In this "food case" horizontal flip do not change the classification/meaning of the image - fill_mode: is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

```
train_datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 50,
  width_shift_range = 0.1,
  height_shift_range = 0.1,
  shear_range = 0.1,
  zoom_range = 0.1,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)



train_generator <- flow_images_from_directory(
    directory="C:/CEU/DS1/ceu-ds1/data/train/",
    class_mode = "binary",
    generator = train_datagen,
    batch_size = batch_size,
    target_size = c(128, 128)
)

valid_generator <- flow_images_from_directory(
    directory="C:/CEU/DS1/ceu-ds1/data/test/",
    class_mode = "binary",
    generator = valid_datagen,
    batch_size = batch_size,
    target_size = c(128, 128)
)

# try the same model, just with augmentation
cnn_model_w_augmentation <- keras_model_sequential()
cnn_model_w_augmentation |>
    layer_conv_2d(
        filters = 32,
        kernel_size = c(3, 3),
        activation = 'relu',
        input_shape = c(128, 128, 3)
    ) |>
    layer_average_pooling_2d(pool_size = c(2, 2)) |>
    layer_dropout(rate = 0.1) |>
    layer_flatten() |>
    layer_dense(units = 64, activation = 'relu') |>
    layer_dense(units = 1, activation = 'softmax')
set.seed(my_seed)
compile(
    cnn_model_w_augmentation,
    loss = 'binary_crossentropy',
    optimizer = optimizer_adam(),
    metrics = c('accuracy')
```

```
)

history2 <-fit(
    cnn_model_w_augmentation,
    train_generator,
    epochs = 30,
    steps_per_epoch =floor(length(train_generator$labels)/batch_size), #8, #nrow(data_train_x) / batch_siz
    validation_data = valid_generator,
    validation_steps =floor(length(valid_generator$labels)/batch_size)# 8#nrow(data_valid_x) / batch_size
)

cnn_model_w_augmentation %>% save_model_hdf5("hotdog_cnn2.h5")

plot(history2)
```




```
history2
```

```
##
## Final epoch (plot to see history):
##         loss: 0.5329
##     accuracy: 0.5
##     val_loss: 0.6863
## val_accuracy: 0.4958
```

```
evaluate(cnn_model_w_augmentation,valid_generator,valid_generator$labels)
```

```
##     loss accuracy
## 0.680106 0.500000
```

I tried the same model with them mentioned augmentation steps, and the valuation accuracy changed from 0.5 to

0.4958. So the model with augmentation is still around the performance of random classification.

It not a surprising result, we have really low number of observation (~500 images) and the images are really different even within class. Not surprising ,that the model cannot recognize the hot dog patter in just 250 images which are really different. It would be interesting to train the same model on larger dataset, like on 10 GB of images, bit it would probably destroy an average PC.

Instead, we can try a pre-trained model in the next subtask.

optional

Try to rely on some pre-built neural networks to aid prediction. Can you achieve a better performance using transfer learning for this problem?

ResNet-50 model is a convolutional neural network (CNN) that is 50 layers deep. A Residual Neural Network (ResNet) is an Artificial Neural Network (ANN) of a kind that stacks residual blocks on top of each other to form a network. Using Resnet50 pre-trained model from keras, I try an image of a hot dog and show the top 3 predictions:

```
library(keras)

# instantiate the model
model <- application_resnet50(weights = 'imagenet')

# load the image
img_path <- "../data/train/hot_dog/7896.jpg"
img <- image_load(img_path, target_size = c(224,224))
x <- image_to_array(img)

# ensure we have a 4d tensor with single element in the batch dimension,
# the preprocess the input for prediction using resnet50
x <- array_reshape(x, c(1, dim(x)))
x <- imagenet_preprocess_input(x)

# make predictions then decode and print them
preds <- model %>% predict(x)
pred<-imagenet_decode_predictions(preds, top = 1)[[1]]
pred$class_description
```

```
## [1] "hotdog"
```

We can see that at #1 it is correctly classified as hot dog.

Do classification based on this model with the top1 class.

```
hot_dogs<-list.files("C:/CEU/DS1/ceu-ds1/data/test/hot_dog")

path<-"C:/CEU/DS1/ceu-ds1/data/test/hot_dog/"

hot_dog_pred<-c()
for(i in hot_dogs){
  img <- image_load(paste0(path,i), target_size = c(224,224))
  x <- image_to_array(img)

# ensure we have a 4d tensor with single element in the batch dimension,
# the preprocess the input for prediction using resnet50
  x <- array_reshape(x, c(1, dim(x)))
  x <- imagenet_preprocess_input(x)

# make predictions then decode and print them
  preds <- model %>% predict(x)
```

```
  hot_dog_pred[i]<-imagenet_decode_predictions(preds, top = 1)[[1]]$class_description
}

sum(hot_dog_pred=="hotdog")
```

```
## [1] 197
```

```
not_hot_dogs<-list.files("C:/CEU/DS1/ceu-ds1/data/test/not_hot_dog")

path<-"C:/CEU/DS1/ceu-ds1/data/test/not_hot_dog/"

not_hot_dog_pred<-c()
for(i in not_hot_dogs){
  img <- image_load(paste0(path,i), target_size = c(224,224))
  x <- image_to_array(img)

# ensure we have a 4d tensor with single element in the batch dimension,
# the preprocess the input for prediction using resnet50
  x <- array_reshape(x, c(1, dim(x)))
  x <- imagenet_preprocess_input(x)

# make predictions then decode and print them
  preds <- model %>% predict(x)
  not_hot_dog_pred[i]<-imagenet_decode_predictions(preds, top = 1)[[1]]$class_description
}

sum(not_hot_dog_pred!="hotdog")
```

```
## [1] 230
```

The resnet50 model correctly classified 427 out of 500 images in the test set, and we can see that the model is much better classifying not hot dog. This actually means 0.854 accuracy, which is much better than my previously trained model with accuracy around 0.5. This problem is a good example for the benefit of using transfer learning, instead of trying to train a "better than random" model.