Assignment2_rspake1

Question 1

While studying for this module, I had created the cats and dogs test/validation/train following the steps in the book, so I am simply reusing the same directories here.

```
base_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small"

train_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/train"
validation_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/validation"
test_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/test"

train_cats_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/train/cats"

train_dogs_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/train/dogs"

Validation_dogs_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/validation/dogs"

Validation_cats_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/validation/cats"

test_cats_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/test/cats"

test_dogs_dir <- "~/Cats_and_dogs_small/archive/cats_and_dogs_small/test/cats"
```

Building the base model

Following the steps in the model/book, after establishing the directories, we are now building the base model with layer_conv_2d with relu activation, max pooling, finally flattening and a final dense layer with sigmoid activation.

Lastly, we will take a look at the model to make sure we have built it properly.

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.1.2
```

```
model1 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_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 = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

```
## Loaded Tensorflow version 2.8.0
```

```
summary(model1)
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  conv2d_3 (Conv2D)                  (None, 148, 148, 32)            896
##
##  max_pooling2d_3 (MaxPooling2D)     (None, 74, 74, 32)              0
##
##  conv2d_2 (Conv2D)                  (None, 72, 72, 64)              18496
##
##  max_pooling2d_2 (MaxPooling2D)     (None, 36, 36, 64)              0
##
##  conv2d_1 (Conv2D)                  (None, 34, 34, 128)             73856
##
##  max_pooling2d_1 (MaxPooling2D)     (None, 17, 17, 128)             0
##
##  conv2d (Conv2D)                    (None, 15, 15, 128)             147584
##
##  max_pooling2d (MaxPooling2D)       (None, 7, 7, 128)               0
##
##  flatten (Flatten)                  (None, 6272)                    0
##
##  dense_1 (Dense)                    (None, 512)                     3211776
##
##  dense (Dense)                      (None, 1)                       513
##
## =========================================================================
## Total params: 3,453,121
## Trainable params: 3,453,121
## Non-trainable params: 0
## _________________________________________________________________________
```

Since our last layer in a sigmoid, we will use binary crossentropy as the loss function, as stipulated in the chart in chapter 4 of the book.

```
model1 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("acc")
)
```

Preprocessing of our data

```
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  train_datagen,
  target_size = c(150,150),
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(150,150),
  batch_size = 20,
  class_mode = "binary"
)

batch <- generator_next(train_generator)
str(batch)
```

```
## List of 2
##  $ : num [1:20, 1:150, 1:150, 1:3] 0.6039 0.9765 0.4314 0.2471 0.0627 ...
##  $ : num [1:20(1d)] 0 0 1 0 1 1 1 0 1 1 ...
```

Now it is time to fit the model the the data.

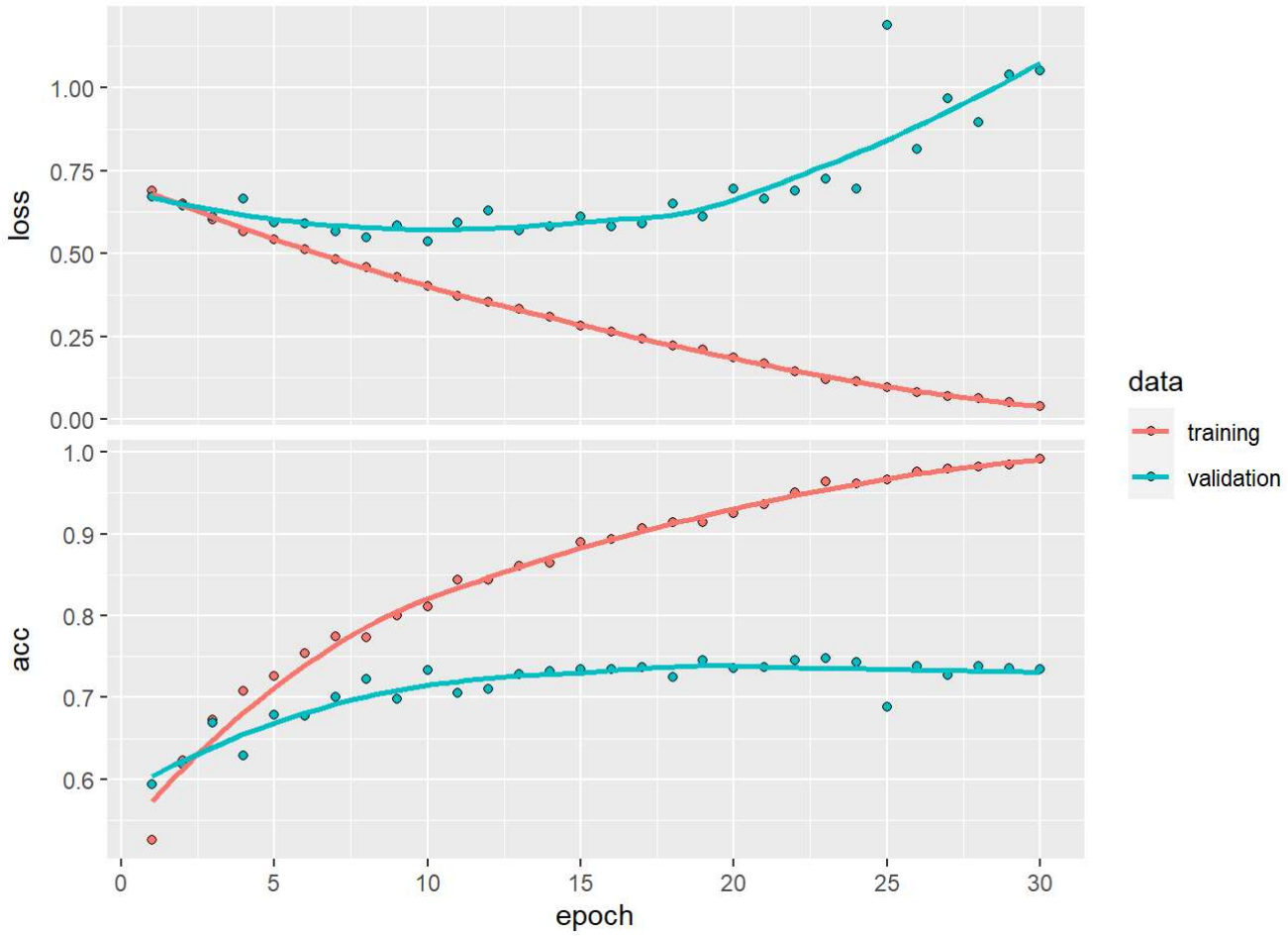
```
history1 <- model1 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```

```
## Warning in fit_generator(., train_generator, steps_per_epoch = 100, epochs
## = 30, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

As we see in the plot below of History1, the model has an “elbow” point around 18 epochs and is suffering from some heavy overfitting with a val accuracy of 7.5 as a relative high point. loss: 0.2096 - acc: 0.9255 - val_loss: 0.5929 - val_acc: 0.7500

```
plot(history1)
```

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




```
model1 %>% save_model_hdf5("cats_and_dogs_small_1.h5")
```

#Optimizing using dropout and augmentation We will optimize our model by using augmentation and dropout to combat overfitting.

- First we will apply dropout, by adding a dropout layer in the model itself.

```
model1.2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model1.2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("acc")
)
```

Now we will apply augmentation to the train images.

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

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

Now lets check the history with these changes and see how this new model holds up.

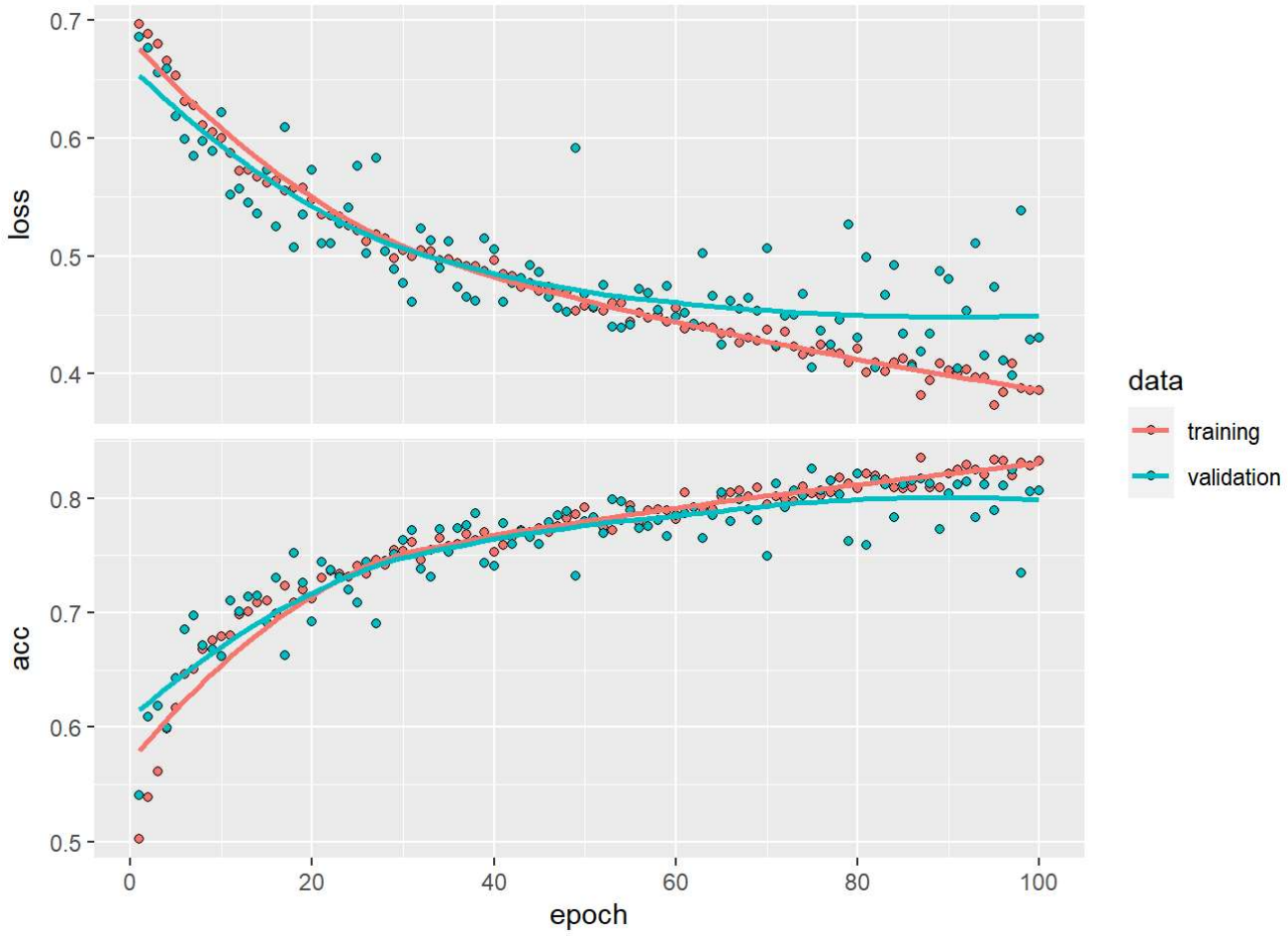
```
history1.2 <- model1.2 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 100,
  validation_data = validation_generator,
  validation_steps = 50
)
```

```
## Warning in fit_generator(., train_generator, steps_per_epoch = 100, epochs
## = 100, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

At epoche 90, we have our best validation outcome (loss: 0.2861 - acc: 0.8820 - val_loss: 0.3810 - val_acc: 0.8470)

```
plot(history1.2)
```

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



Though this output is very hectic, its output is much more accurate than previously.

```
model1.2 %>% save_model_hdf5("cats_and_dogs_small_1.2.h5")
```

Question 2

Now we will increase the training data set so that we have more samples to train with.

```
train_dir2 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_big/train"

train_cats_dir2 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_big/train/cats"

train_dogs_dir2 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_big/train/dogs"
```

with the above code, we have called new train data for both cats and dogs that consists of 1500 units of training data for each. Now we will apply this new train set to our model.

```
train_generator2 <- flow_images_from_directory(
  train_dir2,
  train_datagen,
  target_size = c(150,150),
  batch_size = 20,
  class_mode = "binary"
)

batch <- generator_next(train_generator2)
str(batch)
```

```
## List of 2
##  $ : num [1:20, 1:150, 1:150, 1:3] 0.0784 0.3961 0.7961 0.7529 0.502 ...
##  $ : num [1:20(1d)] 0 0 1 1 0 0 0 1 0 1 ...
```

Now we will apply the new data to the base model.

```
history2 <- model1 %>% fit_generator(
  train_generator2,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```

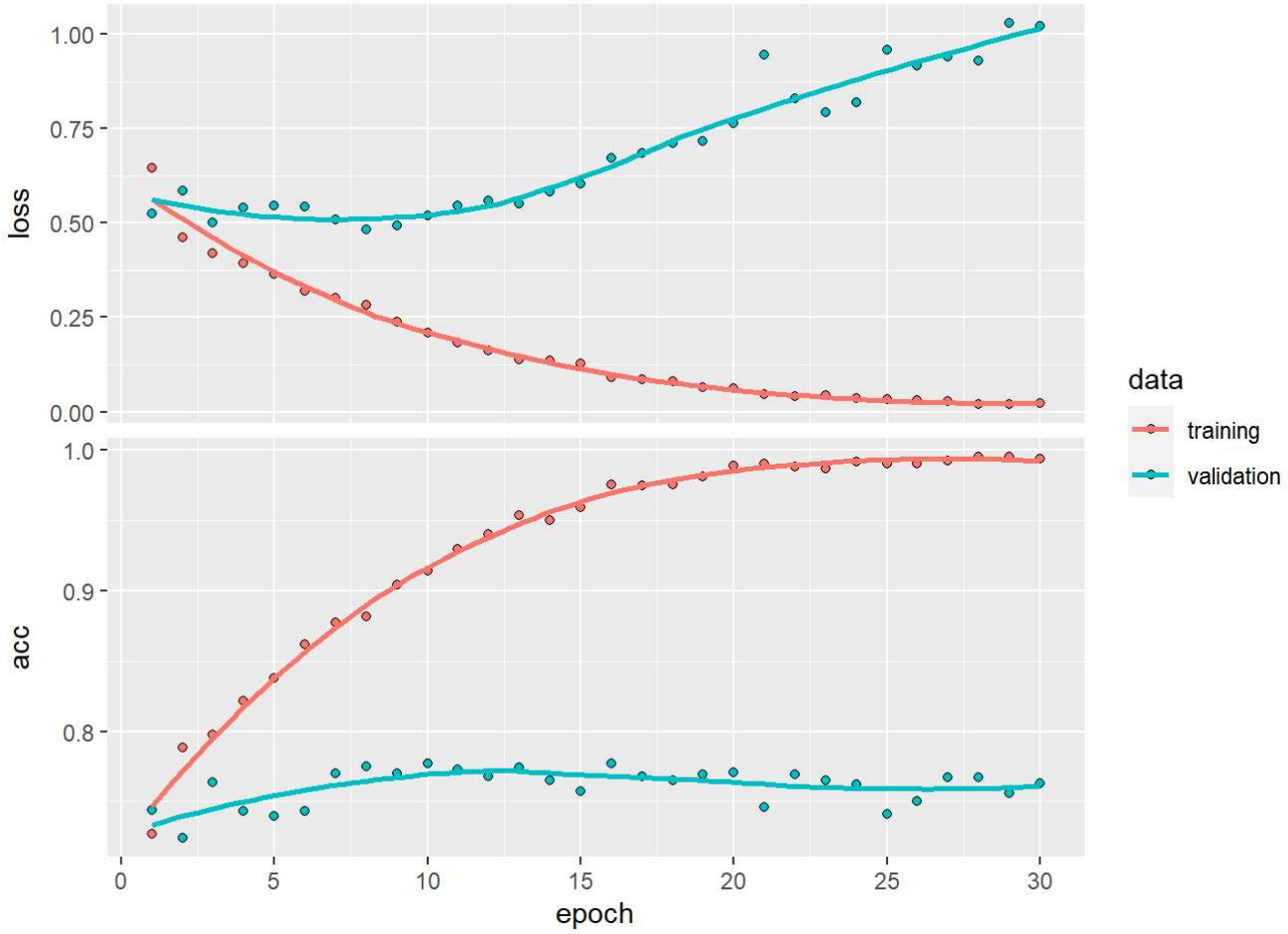
```
## Warning in fit_generator(., train_generator2, steps_per_epoch = 100, epochs
## = 30, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

Looking at the plot bellow we can see how increasing the sample size effects the output.

With the best output coming at 14 epochs at ~ loss of 0.4739 and accuracy of 0.7530. Compared to base model1, there is an improvement in validation loss.

```
plot(history2)
```

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



#Optimizing the new model using dropout and augmentation We will optimize our model by using augmentation and dropout to combat overfitting.

- First we will apply dropout, by adding a dropout layer in the model itself.

```
model2.2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model2.2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("acc")
)
```

Now we will apply augmentation to the train images of the new trainign sample.

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

test_datagen <- image_data_generator(rescale = 1/255)

train_generator2.2 <- flow_images_from_directory(
  train_dir2,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

Now that we have augmented the images, we will apply them to the model.

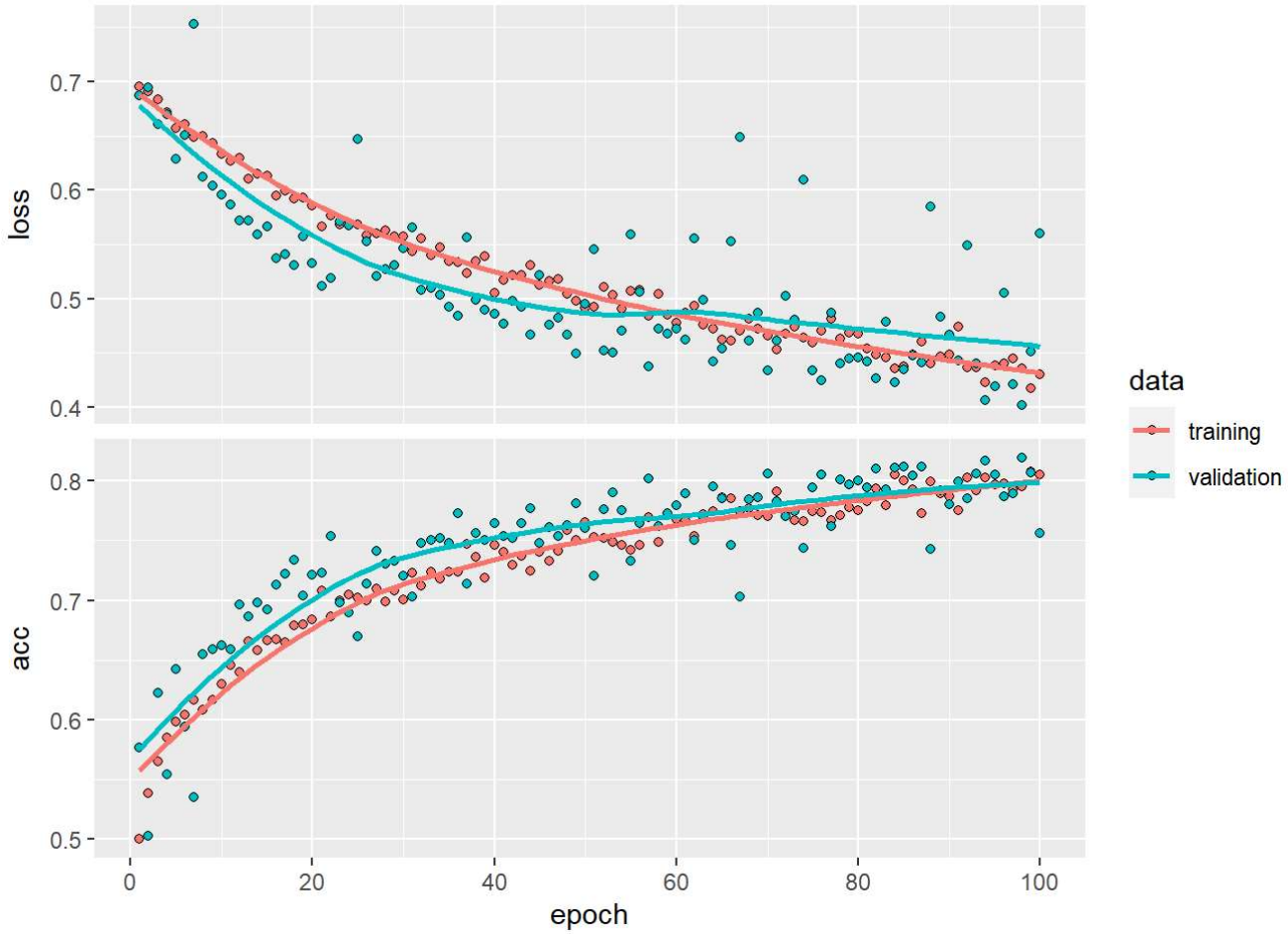
```
history2.2 <- model2.2 %>% fit_generator(
  train_generator2.2,
  steps_per_epoch = 100,
  epochs = 100,
  validation_data = validation_generator,
  validation_steps = 50
)
```

```
## Warning in fit_generator(., train_generator2.2, steps_per_epoch = 100, epochs
## = 100, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

In this new Augmented model, the best performance happens at 95 with ~ val_loss: 0.3936 and val_acc: 0.8240

```
plot(history2.2)
```

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



This augmented model is less hectic than the first augmented model, and performs better on average. As this model has higher sample training data, it will perform better on average.

Question 3

This time we will use the entirety of the data that is not in test or validation to train.

```
train_dir3 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_all/train"

train_cats_dir3 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_all/train/cats"

train_dogs_dir3 <- "~/Cats_and_dogs_small/archive/cats_and_dogs_all/train/dogs"
```

with the above code, we have called new train data for both cats and dogs that consists of 1500 units of training data for each. Now we will apply this new train set to our model.

```
train_generator3 <- flow_images_from_directory(
  train_dir3,
  train_datagen,
  target_size = c(150,150),
  batch_size = 20,
  class_mode = "binary"
)

batch <- generator_next(train_generator3)
str(batch)
```

```
## List of 2
##  $ : num [1:20, 1:150, 1:150, 1:3] 0.502 0.933 0.298 0.769 0.792 ...
##  $ : num [1:20(1d)] 0 0 1 1 1 1 0 0 1 0 ...
```

Now we will apply the new data to the base model.

```
history3 <- model1 %>% fit_generator(
  train_generator3,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```

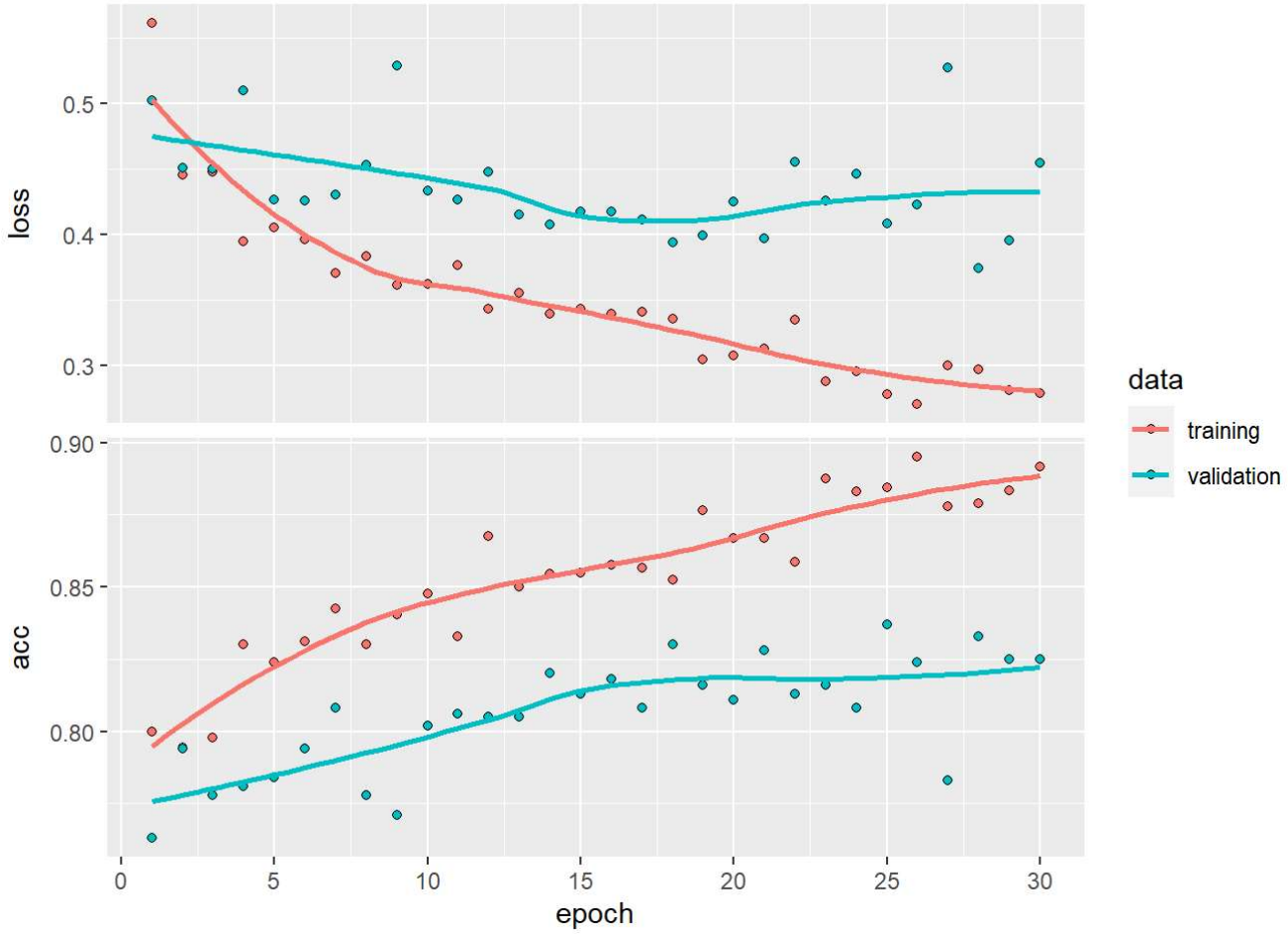
```
## Warning in fit_generator(., train_generator3, steps_per_epoch = 100, epochs
## = 30, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

Looking at the plot bellow we can see how increasing the sample size effects the output.

With the best output coming at 26 epochs at ~ loss of 0.3575 and accuracy of 0.8390. Compared to all of the models so far, there is an improvement in validation loss using this method.

```
plot(history3)
```

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



#Optimizing the new model using dropout and augmentation We will optimize our model by using augmentation and dropout to combat overfitting.

- First we will apply dropout, by adding a dropout layer in the model itself.

```
model3.2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model3.2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("acc")
)
```

Now we will apply augmentation to the train images of the new trainign sample.

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

test_datagen <- image_data_generator(rescale = 1/255)

train_generator3.2 <- flow_images_from_directory(
  train_dir3,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

Now that we have augmented the images, we will apply them to the model.

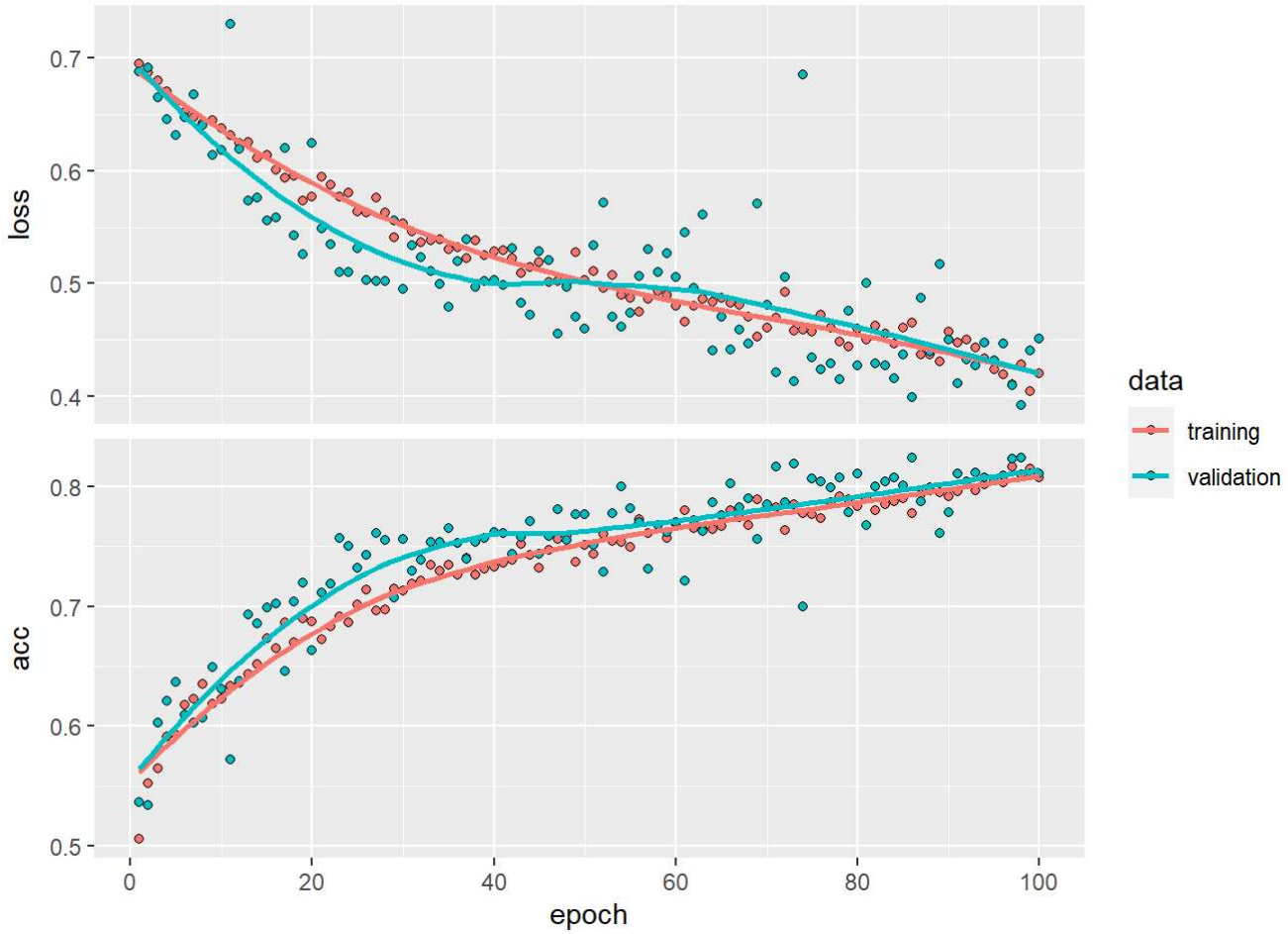
```
history3.2 <- model3.2 %>% fit_generator(
  train_generator2.2,
  steps_per_epoch = 100,
  epochs = 100,
  validation_data = validation_generator,
  validation_steps = 50
)
```

```
## Warning in fit_generator(., train_generator2.2, steps_per_epoch = 100, epochs
## = 100, : `fit_generator` is deprecated. Use `fit` instead, it now accept
## generators.
```

In this new Augmented model, the best performance happens at 96 with ~ val_loss: 0.4017 - val_acc: 0.8160. What we find when we compare the plots of the three different models that use three different trainign sizes: is that while model 3 has MUCH larger trainign data, its best performer is beaten by the best performer in model 2. However, the big take away is the general performance of the model itself. It can be hard to see in the tables, but the plots make it very clear that as we add more units to the training data, less overfittign takes place and the model performs better on average and can make better generalizations.

```
plot(history3.2)
```

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



This augmented model is less hectic than the first augmented model, and performs better on average. As this model has higher sample training data, it will perform better on average.

Question 4

```
base_conv <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150,150,3)
)
```

With this we have created the base_conv from the pretrained network that we will now apply to our model. Since I am not using a GPU, I will have to use the first method mentioned in the book to do this.

```
datagen4 <- image_data_generator(rescale= 1/255)
batch_size <- 20

extract_features <- function(directory, sample_count) {
  features <- array(0, dim = c(sample_count,4,4,512))
  labels <- array(0, dim = c(sample_count))

  generator <- flow_images_from_directory(
    directory = directory,
    generator = datagen4,
    target_size = c(150,150),
    batch_size = batch_size,
    class_mode = "binary"
  )
  i <- 0
  while (TRUE) {
    batch <- generator_next(generator)
    inputs_batch <- batch[[1]]
    labels_batch <- batch[[2]]
    features_batch <- base_conv %>% predict(inputs_batch)

    index_range <- ((i * batch_size)+1):((i+1)*batch_size)
    features[index_range,,,] <- features_batch
    labels[index_range] <- labels_batch

    i <- i+1
    if (i * batch_size >= sample_count)
      break
  }
  list(
    features = features,
    labels = labels
  )
}

train <- extract_features(train_dir,2000)
validation <- extract_features(validation_dir,1000)
test <- extract_features(test_dir,1000)
```

After following the instructions in the book to apply a pretrained network, we now have to flatten the output.

```
reshape_features <- function(features) {
  array_reshape(features, dim = c(nrow(features), 4*4*512))
}
train$features <- reshape_features(train$features)
validation$features <- reshape_features(validation$features)
test$features <- reshape_features(test$features)
```

Just like before, we will now use dropout to help fight overfitting.

```
model4 <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu",
              input_shape = 4*4*512) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")

model4 %>% compile(
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

Now that we have created our new model with a pretrained network, we can now print a “history” and determine how many epochs we should use.

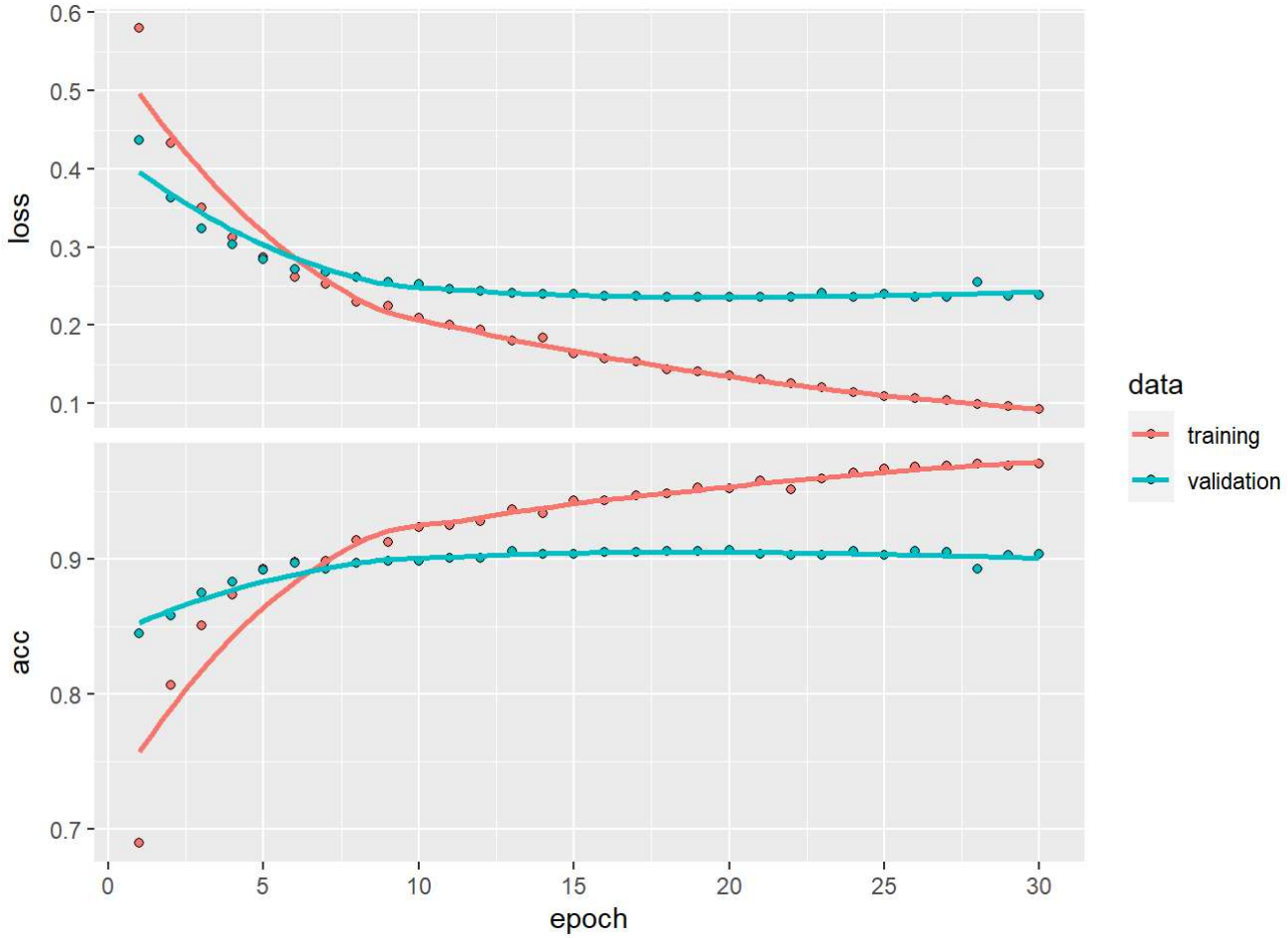
```
history4 <- model4 %>% fit(
  train$features, train$labels,
  epochs = 30,
  batch_size = 20,
  validation_data = list(validation$features, validation$labels)
)
```

As seen in the plot bellow, the model very quickly overfits. Now we know that small data sets contribute to overfitting, so in the next two sections we will try to combat that.

Even though this model overfits very quickly, the validation loss and accuracy had really good outputs at the best epoch of 7 (val_loss: 0.2604 - val_acc: 0.8950).

```
plot(history4)
```

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



Now during the next two steps in question 4, we will be taking advantage of the directories I created for question 2 and 3 to see how increased sample size effect the model's performance.

##Question 4.2 The code bellow recodes the extract features section and all that follows and adjusts for the new model. This section should copy Question 2 with the new pretrained network.

```
datagen4 <- image_data_generator(rescale= 1/255)
batch_size <- 20

extract_features <- function(directory, sample_count) {
  features <- array(0, dim = c(sample_count,4,4,512))
  labels <- array(0, dim = c(sample_count))

  generator <- flow_images_from_directory(
    directory = directory,
    generator = datagen4,
    target_size = c(150,150),
    batch_size = batch_size,
    class_mode = "binary"
  )
  i <- 0
  while (TRUE) {
    batch <- generator_next(generator)
    inputs_batch <- batch[[1]]
    labels_batch <- batch[[2]]
    features_batch <- base_conv %>% predict(inputs_batch)

    index_range <- ((i * batch_size)+1):((i+1)*batch_size)
    features[index_range,,,] <- features_batch
    labels[index_range] <- labels_batch

    i <- i+1
    if (i * batch_size >= sample_count)
      break
  }
  list(
    features = features,
    labels = labels
  )
}

train <- extract_features(train_dir2,3000)
validation <- extract_features(validation_dir,1000)
test <- extract_features(test_dir,1000)
```

After following the instructions in the book to apply a pretrained network, we now have to flatten the output.

```
reshape_features <- function(features) {
  array_reshape(features, dim = c(nrow(features), 4*4*512))
}
train$features <- reshape_features(train$features)
validation$features <- reshape_features(validation$features)
test$features <- reshape_features(test$features)
```

Just like before, we will now use dropout to help fight overfitting.

```
model4.2 <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu",
              input_shape = 4*4*512) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")

model4.2 %>% compile(
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

Now that we have created our new model with a pretrained network, we can now print a “history” and determine how many epochs we should use.

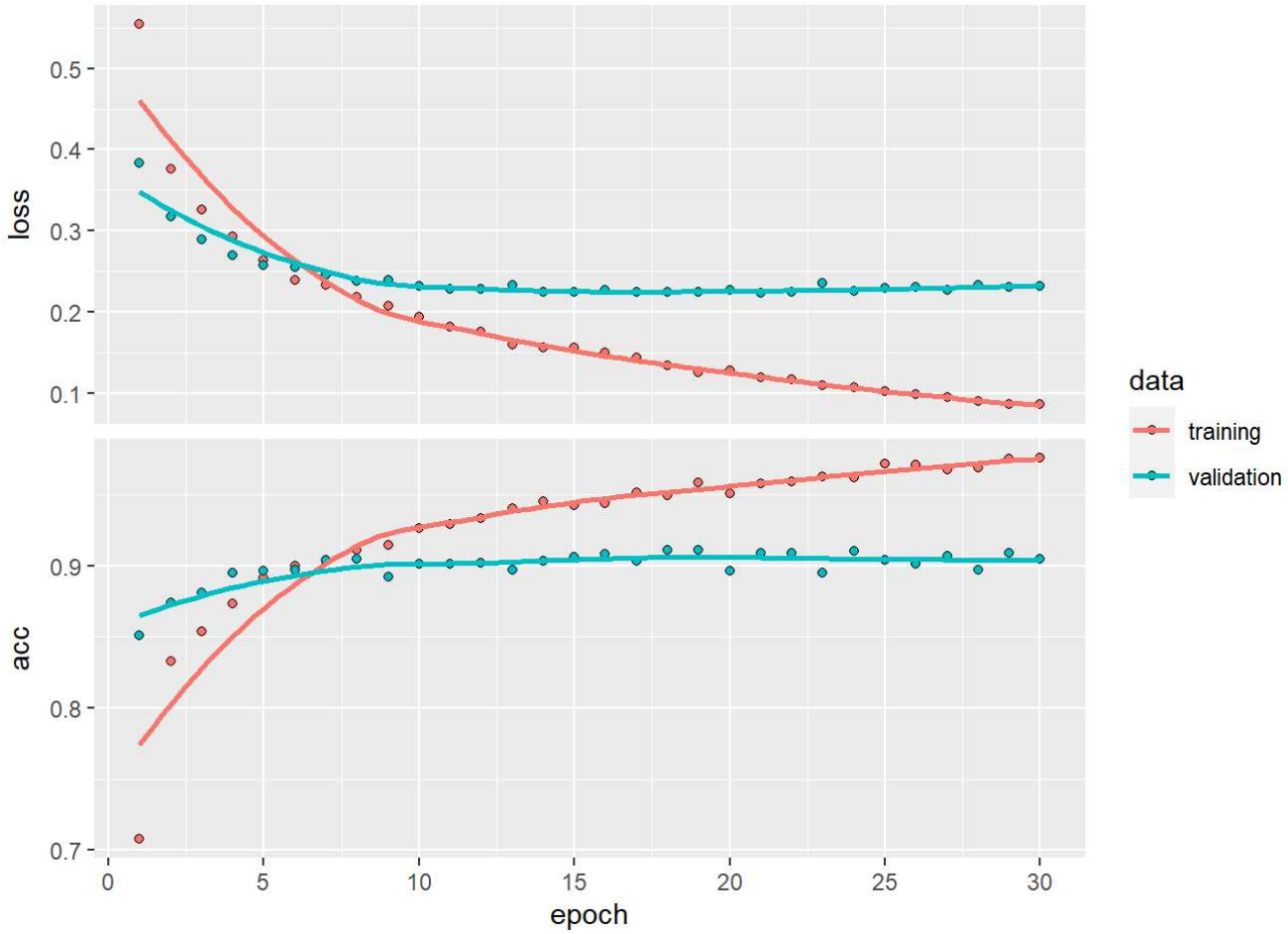
```
history4.2 <- model4.2 %>% fit(
  train$features, train$labels,
  epochs = 30,
  batch_size = 20,
  validation_data = list(validation$features, validation$labels)
)
```

The best epoche in this model is epoch 8 (val_loss: 0.2364 - val_acc: 0.9060), but we still are struggling with the model quickly overfitting. As we move onto the last model, we want to see it last a little longer before over fitting, but without augmentation, fighting overfitting is difficult.

Although we are struggling with overfitting, the model does have a very goo doutput with high accuracy (over 90%) and low loss (bellow 25%).

```
plot(history4.2)
```

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



##Question 4.3 The code bellow recodes the extract features section and all that follows and adjusts for the new model. This section should copy Question 3 with the new pretrained network.

```
datagen4 <- image_data_generator(rescale= 1/255)
batch_size <- 20

extract_features <- function(directory, sample_count) {
  features <- array(0, dim = c(sample_count,4,4,512))
  labels <- array(0, dim = c(sample_count))

  generator <- flow_images_from_directory(
    directory = directory,
    generator = datagen4,
    target_size = c(150,150),
    batch_size = batch_size,
    class_mode = "binary"
  )
  i <- 0
  while (TRUE) {
    batch <- generator_next(generator)
    inputs_batch <- batch[[1]]
    labels_batch <- batch[[2]]
    features_batch <- base_conv %>% predict(inputs_batch)

    index_range <- ((i * batch_size)+1):((i+1)*batch_size)
    features[index_range,,,] <- features_batch
    labels[index_range] <- labels_batch

    i <- i+1
    if (i * batch_size >= sample_count)
      break
  }
  list(
    features = features,
    labels = labels
  )
}

train4.3 <- extract_features(train_dir3,20000)
validation <- extract_features(validation_dir,1000)
test <- extract_features(test_dir,1000)
```

After following the instructions in the book to apply a pretrained network, we now have to flatten the output.

```
reshape_features <- function(features) {
  array_reshape(features, dim = c(nrow(features), 4*4*512))
}
train$features <- reshape_features(train$features)
validation$features <- reshape_features(validation$features)
test$features <- reshape_features(test$features)
```

Just like before, we will now use dropout to help fight overfitting.

```
model4.3 <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu",
              input_shape = 4*4*512) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")

model4.3 %>% compile(
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

Now that we have created our new model with a pretrained network, we can now print a “history” and determine how many epochs we should use.

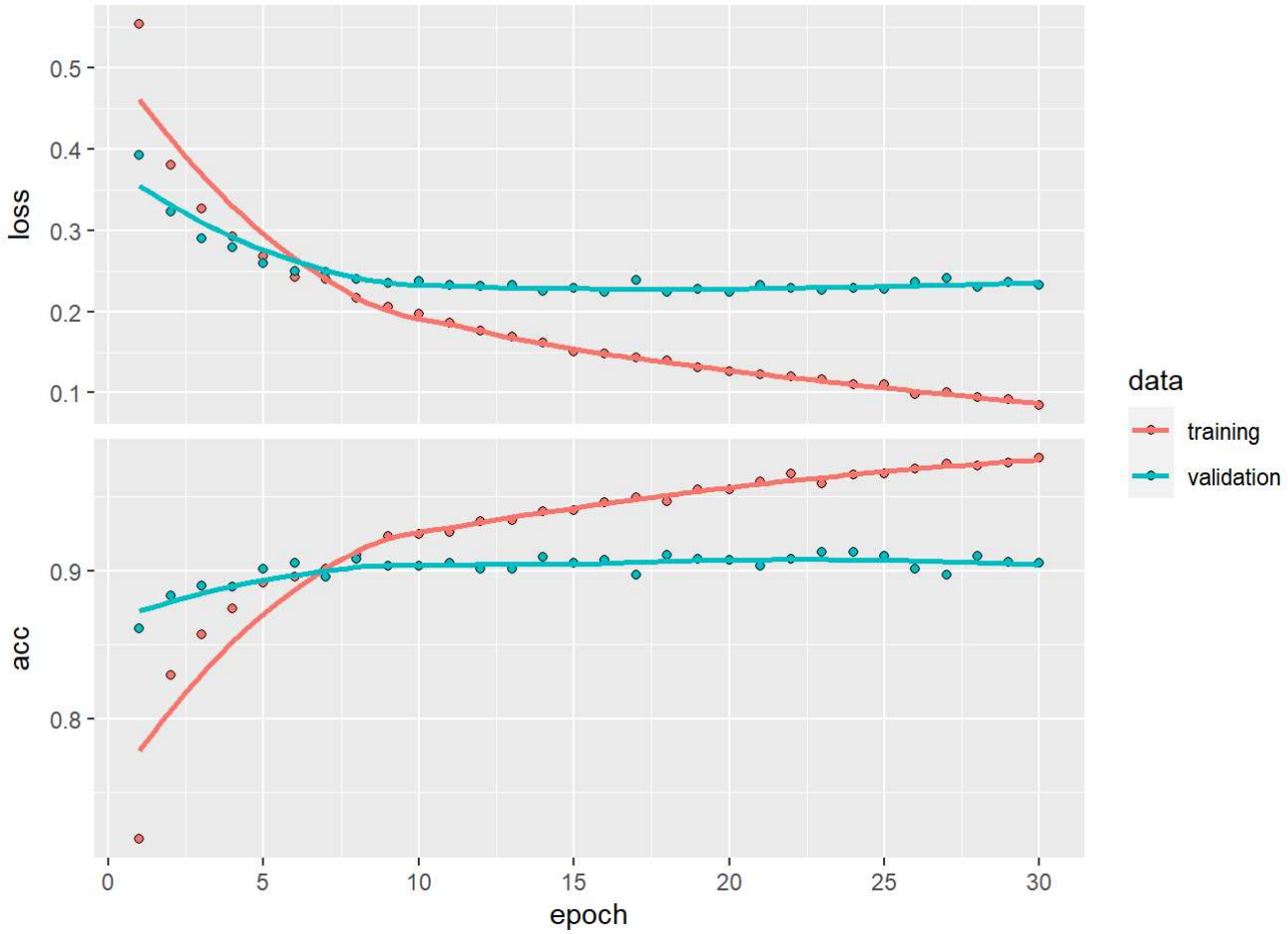
```
history4.3 <- model4.3 %>% fit(
  train$features, train$labels,
  epochs = 30,
  batch_size = 20,
  validation_data = list(validation$features, validation$labels)
)
```

In this final model, we have our best epoch at 8 (val_loss: 0.2352 - val_acc: 0.9040). This model is only slightly better than the previous model (model4.2) and is only slightly more resistant to overfitting.

With that said, this model still performs well despite the overfitting. This just goes to show how important the augmentation was that we used in questions 1-3.

```
plot(history4.3)
```

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



In the end, while using pretrained networks is super useful, the biggest downside is that I am unable to use augmentation (like I was using in the previous questions) which, evidently, is very important in fighting overfitting. Even though, we are getting good accuracy results, the lack of augmentation makes this method less useful, in future efforts to optimize this model, I should set up a GPU that can handle the second way to use pretrained networks so that augmentation can be implemented.