Model 2 by Christine Pallon

2022-12-14

Loading and pre-processing data

For this model, we will use a **network-based model** for classifying the **radiomics** dataset. First, let's load and prepare the dataset, check for any nulls, and scale our data.

```
radiomics <- read.csv("radiomics_completedata.csv")
radiomics <- radiomics %>% select(-Institution, -Failure)

is.null(radiomics)
```

```
## [1] FALSE
```

```
radiomics[, -1] <- scale(radiomics[, -1])
```

Data splitting

Next, we split the data into an **80/20 training/testing split**.

```
set.seed(123)
radiomics_split <- initial_split(radiomics, prop = 0.8, strata = "Failure.binary")
radiomics_train <- training(radiomics_split)
radiomics_test  <- testing(radiomics_split)
```

Then, once we have our testing and training splits, we subset those into our x_train, x_test, y_train, and y_test divisions and then pipe them into the **as.matrix()** function so they become matrices.

```
x_train <- radiomics_train %>% select(-Failure.binary) %>% as.matrix()
x_test <- radiomics_test %>% select(-Failure.binary) %>% as.matrix()
y_train <- radiomics_train$Failure.binary %>% as.matrix()
y_test <- radiomics_test$Failure.binary %>% as.matrix()
```

Since Failure.binary only has two possible outcomes, 0 or 1, we use the keras function **to_categorical()** with the second argument set to 2.

```
y_train <- to_categorical(y_train, num_classes = 2)
y_test <- to_categorical(y_test, num_classes = 2)
```

Build the model!

Now it's time to build our model! We create five layers with 256 (our input layer), 128, 128, 64 and 64 neurons, each with activation functions of **sigmoid**. Each layer is followed by a dropout to avoid overfitting.

The **input_shape** in the first layer is defined as the number of features in our data, which is 428.

The output layer's units are set to 2 since we are working with binary classification, and the activation function is **softmax**. Once we've built the model, we also implement a backpropagation compiler approach.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "sigmoid", input_shape = 428) %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = "softmax") %>%


  # Backpropagation
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy")
  )
```

Then, we compile the model using the following approach:

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(),
  metrics = c("accuracy")
)
```

Model training and evaluation

Now that our model is built, we use our training data to train the model. We've set the **epochs** to 10, the **batch_size** to 128, and the **validation split** to 0.15.

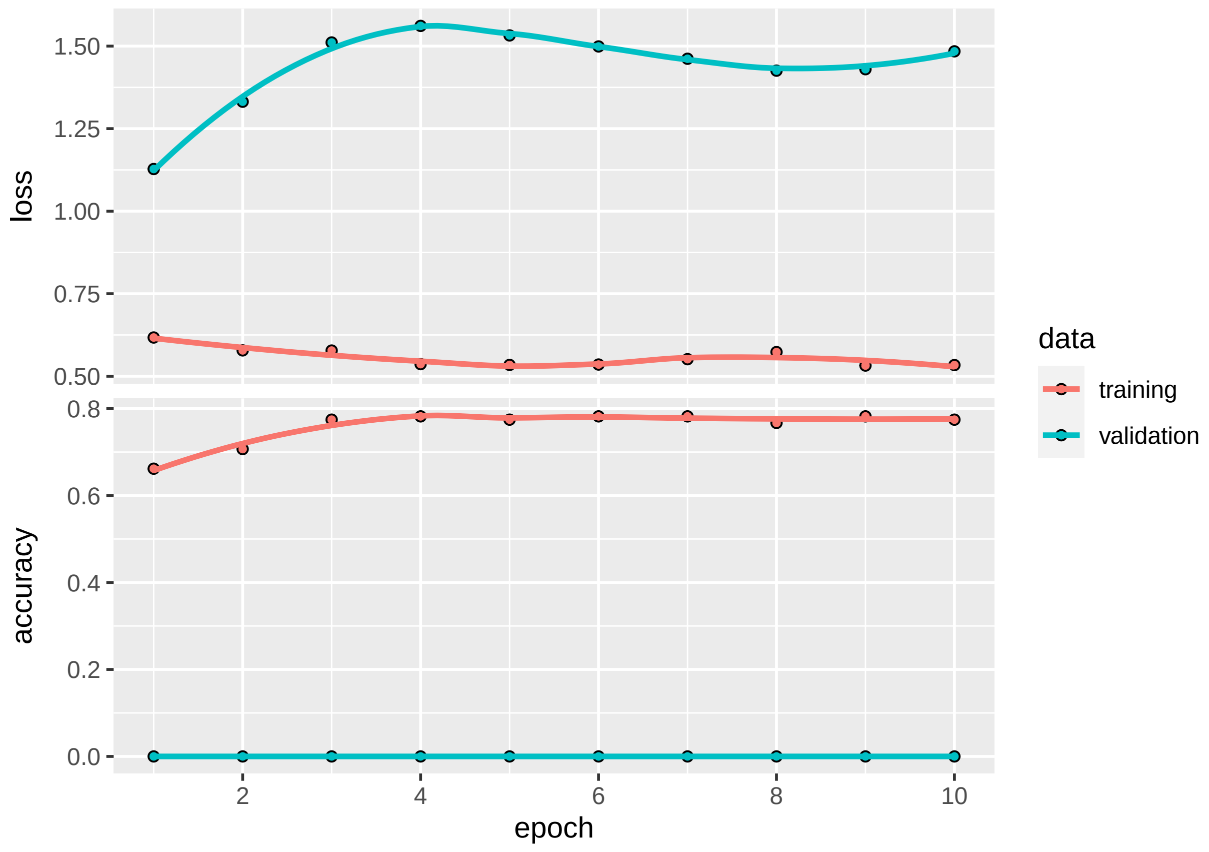
We then call the **history** object to get some details on our model and visualize that performance with a plot.

```
history <- model %>%
  fit(x_train, y_train, epochs = 10, batch_size = 128, validation_split = 0.15)

history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.5338
##     accuracy: 0.7744
##     val_loss: 1.484
## val_accuracy: 0
```

```
plot(history)
```



Let's evaluate our model using the testing dataset!

```
model %>%
  evaluate(x_test, y_test)
```

```
##      loss  accuracy
## 0.6862294 0.6500000
```

Our model's accuracy using the testing dataset was only **0.65**, which is noticeably lower than what it was with the training dataset.

Considering that we're working with binary classification, 0.65 isn't that much better than randomly guessing the outcome. In practice, we would want to revisit our methods and see if improvements can be made to our model.

Model prediction using testing data

Finally, we can get the model's prediction using our testing dataset. Please note that the method demonstrated in class and in the exercises, **predict_classes()**, is deprecated and and was removed from tensorflow in version 2.6.

The below code is what the error message suggested I use instead for predicting using the testing data.

```
model %>% predict(x_test) %>% `>`(0.5) %>% k_cast("int32")
```

```
## tf.Tensor(
## [[1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]
##  [1 0]], shape=(40, 2), dtype=int32)
```