R Lab: Training Dense Feed-Forward Neural Networks

In this lab, we train a dense feed-forward neural network in Keras and TensorFlow, using the raw data that was pre-processed in the previous lab.

The deliverable for this lab is a Word or PDF file containing responses to the exercises at the end of the lab. The deliverable should be submitted through Canvas.

Data Pre-Processing

We first run all of the pre-processing code from the previous lab.

```
library(dplyr)
library(caret)

data <- read.csv("lab_4_data.csv")
training_ind <- createDataPartition(data$lodgepole_pine,
                                    p = 0.75,
                                    list = FALSE,
                                    times = 1)
training_set <- data[training_ind, ]
test_set <- data[-training_ind, ]

top_20_soil_types <- training_set %>%
  group_by(soil_type) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  select(soil_type) %>%
  top_n(20)

training_set$soil_type <- ifelse(training_set$soil_type %in% top_20_soil_types$soil_type,
                                 training_set$soil_type,
                                 "other")

training_set$wilderness_area <- factor(training_set$wilderness_area)
training_set$soil_type <- factor(training_set$soil_type)

onehot_encoder <- dummyVars(~ wilderness_area + soil_type,
                            training_set[, c("wilderness_area", "soil_type")],
                            levelsOnly = TRUE,
                            fullRank = TRUE)

onehot_enc_training <- predict(onehot_encoder,
                               training_set[, c("wilderness_area", "soil_type")])

training_set <- cbind(training_set, onehot_enc_training)

test_set$soil_type <- ifelse(test_set$soil_type %in% top_20_soil_types$soil_type,
                             test_set$soil_type,
```

```
                              "other")

test_set$wilderness_area <- factor(test_set$wilderness_area)
test_set$soil_type <- factor(test_set$soil_type)

onehot_enc_test <- predict(onehot_encoder, test_set[, c("wilderness_area", "soil_type")])
test_set <- cbind(test_set, onehot_enc_test)

test_set[, -c(11:13)] <- scale(test_set[, -c(11:13)],
                         center = apply(training_set[, -c(11:13)], 2, mean),
                         scale = apply(training_set[, -c(11:13)], 2, sd))
training_set[, -c(11:13)] <- scale(training_set[, -c(11:13)])

training_features <- array(data = unlist(training_set[, -c(11:13)]),
                  dim = c(nrow(training_set), 33))
training_labels <- array(data = unlist(training_set[, 13]),
                  dim = c(nrow(training_set)))

test_features <- array(data = unlist(test_set[, -c(11:13)]),
                  dim = c(nrow(test_set), 33))
test_labels <- array(data = unlist(test_set[, 13]),
                  dim = c(nrow(test_set)))
```

Training Dense Feed-Forward Neural Networks

Now, we are ready to build a dense feed-forward neural network on the training set. We load the Keras and TensorFlow libraries, as well as the Python virtual environment.

```
library(reticulate)
library(tensorflow)
library(keras)

use_virtualenv("my_tf_workspace")
```

The initial architecture we use is a dense feed-forward neural network with two hidden layers. The first hidden layer contains 20 units (nodes) and the second hidden layer contains 10 units (nodes), with each using the "relu" activation function. Since the target (i.e., lodgepole_pine) is binary, the output layer has one node and uses the "sigmoid" activation function. The output from this node corresponds to the probability that the observation is of the positive class (i.e., class 1). The "sigmoid" activation function ensures that the output is indeed between 0 and 1, and thus, can be interpreted as a probability.

```
model <- keras_model_sequential(list(
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 10, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))
```

The default optimizer for a binary classification problem, such as in our example here, is "rmsprop". Furthermore, the default loss function is "binary_crossentropy". We also specify the metrics we would like to observe throughout the training process; for classification, a common metric is accuracy, which is the proportion of predictions that the model gets correct.

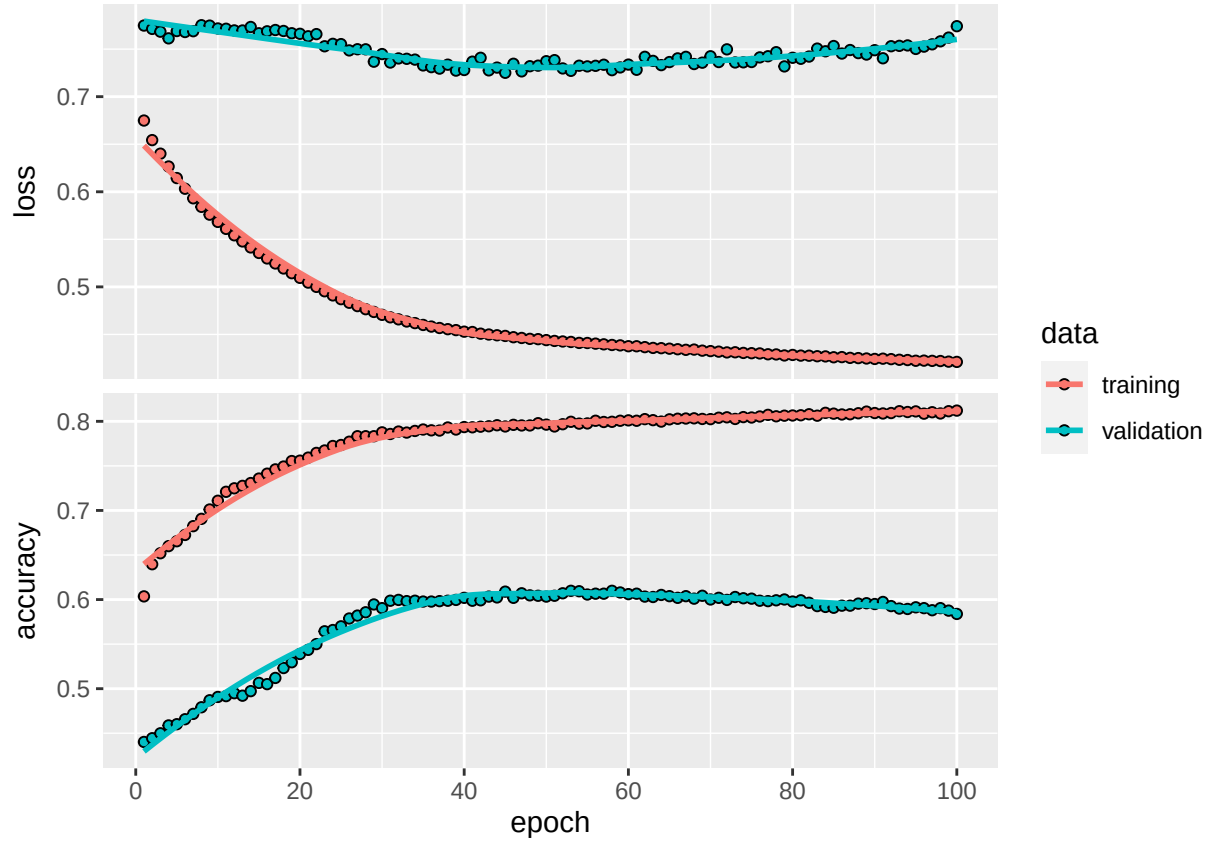
```
compile(model,
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
```

```
        metrics = "accuracy")
```

Now that we have specified the architecture of the model as well as the optimizer, loss, and metrics for the training process, we are ready to train the model using the "fit" function. The model will be trained on the training set, including the training_features and training_labels. We use 100 epochs and a batch size of 512. Lastly, we use a validation split of 0.33. The training process is captured in the variable "history", so that we can plot the loss and accuracy throughout the process. Note, the loss and accuracy curves throughout the training process, also called the learning curves, may be different than shown here: this is due to randomness in the training process.

```
history <- fit(model, training_features, training_labels,
    epochs = 100, batch_size = 512, validation_split = 0.33)

plot(history)
```



We now have a trained dense feed-forward neural network model that has been trained on the training data. This model can be used to predict the observations in the test set.

```
predictions <- predict(model, test_features)
head(predictions, 10)
```

```
##              [,1]
##  [1,] 0.947726607
##  [2,] 0.823752403
##  [3,] 0.947215378
##  [4,] 0.630638778
##  [5,] 0.548763156
```

```
##  [6,] 0.437513649
##  [7,] 0.005109702
##  [8,] 0.048081025
##  [9,] 0.841043472
## [10,] 0.033703975
```

These probabilities are converted into predictions using the cutoff of 0.5: if the output probability is greater than or equal to 0.5, then we predict the positive class (i.e., class 1). Otherwise, we predict the negative class (i.e., class 0).

```
predicted_class <- (predictions[, 1] >= 0.5) * 1
head(predicted_class, 10)
```

```
##  [1] 1 1 1 1 1 0 0 0 1 0
```

In the next lab, we will evaluate the predictions made on the test set, as well as vary the threshold for converting predicted probabilities to the class predictions (i.e., instead of using 0.5).

Exercises

1) Copy and paste the loss and accuracy curves obtained from running the code above (note, the curves will be slightly different than those shown in this lab).
2) Change the hidden layers to have 50 units and 25 units, respectively, and re-run the code. Copy and paste the new loss and accuracy curves.
3) Compare the curves from 1) and 2) and discuss which architecture (i.e., number of nodes in the hidden layers) results in better performance.
4) Calculate the accuracy on the test set for the models in 1) and 2). Which accuracy is better?