Deep Learning with Iris Dataset in R

What is IRIS Flower

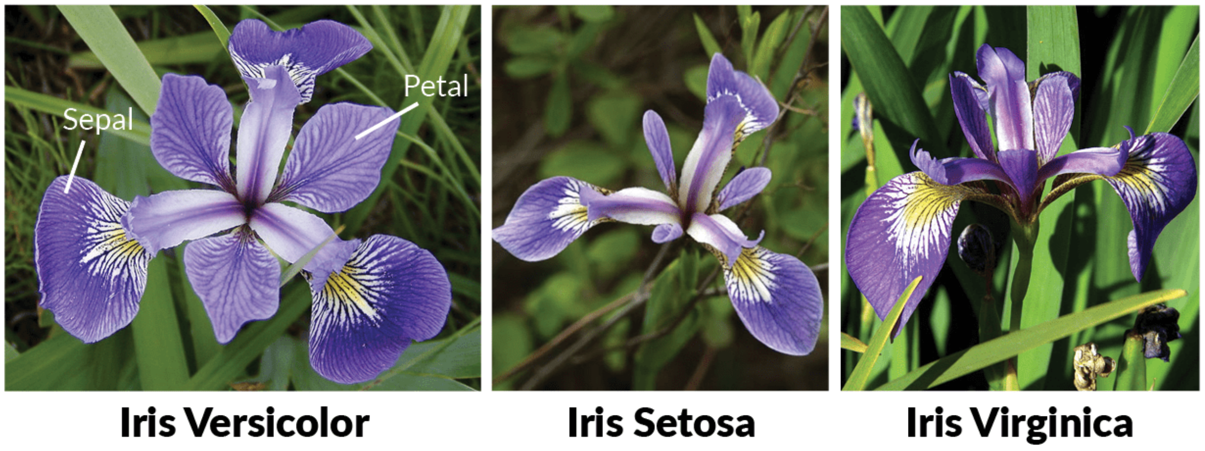


Figure 1: Iris Flowers

. . .

- Image Source: UCI Machine Learning Repository - Iris Data Set*

Load the dataset

```
data(iris)
```

This line loads the built-in Iris dataset into the R environment. The Iris dataset is a popular dataset often used for classification tasks.

Scale the features

```
iris[,1:4] = scale(iris[,1:4])
```

Here, the `scale()` function is applied to the first four columns of the Iris dataset, which correspond to the numeric features (sepal length, sepal width, petal length, and petal width). Scaling the features ensures that they have zero mean and unit variance, which can be beneficial for some machine learning algorithms.

Prepare the target variable

```
iris$Species = as.numeric(iris$Species) - 1
```

In this step, the species column is converted from categorical labels (setosa, versicolor, and virginica) to numeric labels (0, 1, and 2) by subtracting 1 from the numeric representation. This conversion is often done in classification tasks to enable numerical computations.

Split the dataset into train and test sets

```
library(keras)
set.seed(123)
ind <- sample(2, nrow(iris), replace = TRUE, prob = c(0.7, 0.3))
trainData <- iris[ind == 1, 1:4]
testData <- iris[ind == 2, 1:4]
trainLabels <- to_categorical(iris[ind == 1, 5])
testLabels <- to_categorical(iris[ind == 2, 5])
```

Here, the dataset is split randomly into training and testing sets using a 70:30 ratio. The `sample()` function is used to generate random indices for the split. The `trainData` and `testData` variables store the feature data, while `trainLabels` and `testLabels` store the corresponding categorical labels in one-hot encoded format using the `to_categorical()` function. One-hot encoding converts categorical labels into binary vectors, which is commonly used in multi-class classification problems.

Building the model

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 3,
              activation ='softmax',
              input_shape = ncol(trainData))
```

In this step, a sequential model is created using the `keras_model_sequential()` function from the Keras library. The model consists of a single dense (fully connected) layer with 3 units/neurons. The `activation ='softmax'` argument specifies the activation function for the output layer, which is the softmax function. The `input_shape` parameter is set to the number of features in the training data.

Compile the model

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

The `compile()` function is used to configure the model for training. The loss function is set to "categorical_crossentropy", which is commonly used for multi-class classification problems. The optimizer is set to "optimizer_rmsprop()", and the metric used to evaluate the model is accuracy.

Convert dataframe into a matrix

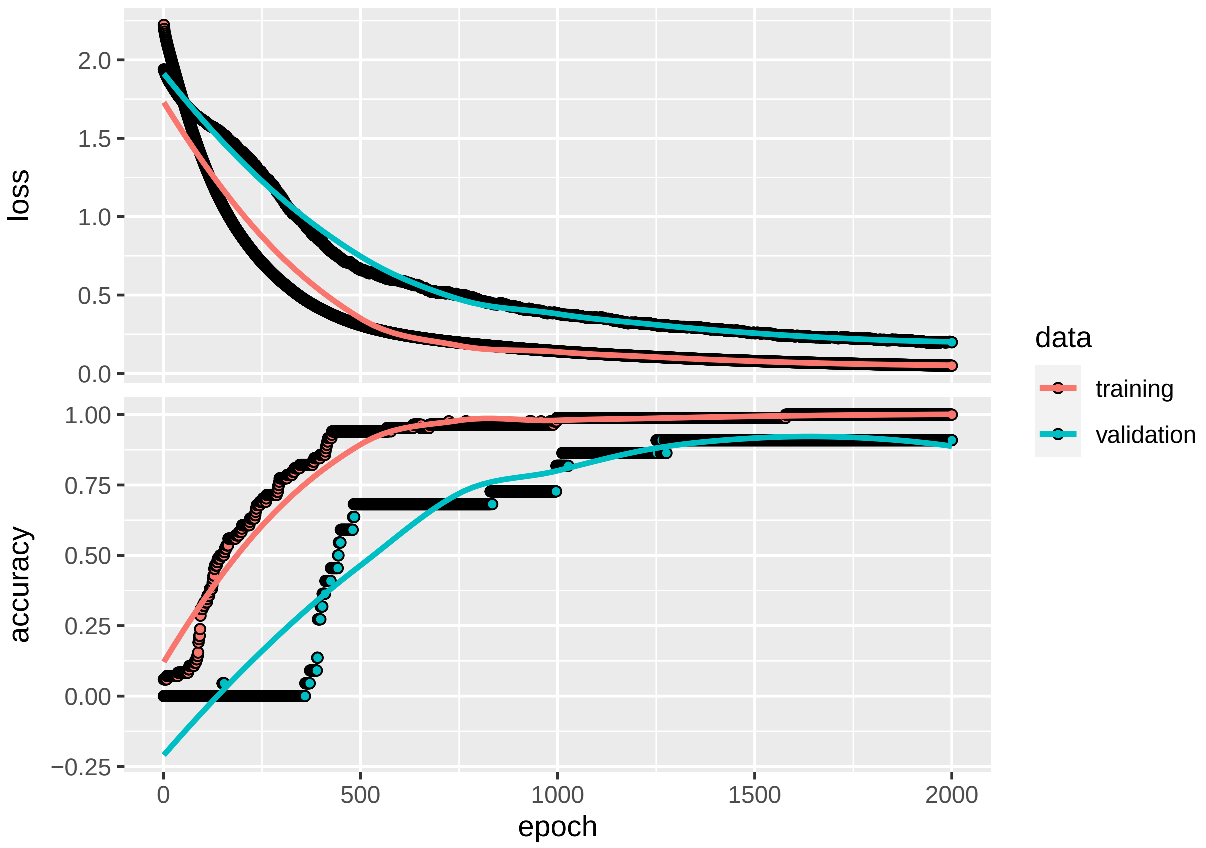
```
trainData <- as.matrix(iris[ind == 1, 1:4])
testData <- as.matrix(iris[ind == 2, 1:4])
```

The code snippet converts the selected subsets of the Iris dataset into matrix format. This conversion is done to ensure compatibility with certain functions or libraries that require matrix inputs, such as the **`fit()`** function for training the model.

Now we will train the model

```
history <- model %>% fit(
  trainData,
  trainLabels,
  epochs = 2000,
  batch_size = 64,
  validation_split = 0.2
)

# Plot the training history
plot(history)
```



Now, the model is trained using the fit() function. The trainData and trainLabels are used as the training data and labels, respectively. The epochs parameter specifies the number of training iterations, batch_size defines the number of samples per gradient update, and validation_split indicates the proportion of training data to use for validation.

#Model Tarining Results

The model appears to be performing well during the training process. Here are some observations:

accuracy: 1.0000: The accuracy achieved during the training process is 1.0000, indicating that the model correctly predicted the target variable for all instances in the training set. This perfect accuracy suggests that the model has learned the patterns and relationships present in the training data.

val_accuracy: 0.9091: The validation accuracy achieved during the training process is 0.9091. This indicates that the model performs well on unseen data, as it achieved a high accuracy on the validation set. A validation accuracy close to the training accuracy suggests that the model is generalizing well to new, unseen examples.

loss: 0.0454 and val_loss: 0.1894: The loss values obtained during the training and validation steps are relatively low. A lower loss value indicates that the model's predictions are closer to the actual values. The low loss values suggest that the model has learned the underlying patterns in the training data and is generalizing well to the validation set.

Overall, the model seems to be performing effectively during the training process, achieving high accuracy and low loss values.

Evaluate the model in Test Data

```
score <- model %>% evaluate(testData, testLabels)
cat('Test loss:', score[1], '\n')
```

```
## Test loss: 0.1821503
```

```
cat('Test accuracy:', score[2], '\n')
```

```
## Test accuracy: 0.9318182
```

The trained model is evaluated on the test data using the evaluate() function. The testData and testLabels are provided as input, and the resulting test loss and accuracy are displayed using the cat() function.

The model appears to perform well, achieving a high accuracy and relatively low loss on the test data. This suggests that the model has learned to make accurate predictions on unseen instances, demonstrating its effectiveness in classifying the data it was trained on.

Save The Model

```
keras::save_model_hdf5(model, "my_model.h5")
```

Finally, the trained model is saved in the HDF5 format using the save_model_hdf5() function from the Keras library. The model is saved with the filename "my_model.h5".