STAT40970 – Machine Learning & A.I. (Online) Assignment 2

K.Saketh Sai Nigam 22201204

2023-04-03

Exercise 1

==

1. Comment briefly on the architecture and on the number of parameters of this network.

==

```
#library for building and training deep learning models
library(keras)
VarV=2
VarK=1
#sequentially add layers to the neural network model
ModelKeras <- keras_model_sequential()
#Adding three layers to the model with different units
ModelKeras %>%
layer_dense(units = 3, activation = "sigmoid", input_shape = VarV) %>%
layer_dense(units = 2, activation = "relu", input_shape = VarV) %>%
layer_dense(units = VarK, activation = "linear")
#counting the number of parameters in the model.
cat("The Count of Parameters exits in the Model is: ",count_params(ModelKeras))
```

```
## The Count of Parameters exits in the Model is:  20
```

- *In essence, this lists the characteristics in a straightforward neural network model that has two hidden layers and an output layer. Using techniques like fit(), this model may be trained on a dataset and then applied to new data to make predictions. There are 20 Parameters in the Model Keras.*

==

2. Perform a forward propagation calculation through the network for the input observation vector x = (–0.5, 0.3). What is the value of the output unit corresponding to this input vector?

==

```
#Setting Arrays and assigning it to variables
LayerInputArray=array(c(-0.5,-0.3,0.4,-0.1,-0.2,0.4),dim = c(2, 3))
BaisLayerInputArray=array(c(1.1,-0.8,1.3))
HiddenLayerInputArray1=array(c(0.3,1.3,-0.8,-0.7,0.5,1.2),dim = c(3, 2))
HiddenBaisLayerInputArray1=array(c(0.5,-0.8))
HiddenLayerInputArray2=array(c(2.0,1.1),dim = c(2,1))
HiddenBaisLayerInputArray2=array(c(8))
#Forming a List of Arrays
ListOfLayers=list(LayerInputArray,BaisLayerInputArray,HiddenLayerInputArray1,HiddenBaisLayerInputArray1,HiddenLay
erInputArray2,HiddenBaisLayerInputArray2)
#Manually set the weights of a pre-trained model
ModelKeras%>% set_weights(ListOfLayers)
```

- *You can utilize a pre-trained neural network model for making predictions on new data without having to reinvent the wheel by adjusting the model's weights. This can reduce training time and processing resources, especially for big models.*

```
DataArrayX=array(c(-0.5,0.3),dim = c(1, 2))
#Prediction on the neural network model using the input data, and stores the predicted output values in the varia
ble.
PredictedValueY=predict(ModelKeras, DataArrayX)
#Printing Predicted value
cat("The Predicted value is: ",PredictedValueY)
```

```
## The Predicted value is:  8.838573
```

- *The code above uses a previously trained neural network model named ModelKeras to generate predictions on new input data that is given as a 1x2 dimensional array in DataArrayX. The ModelKeras neural network model and DataArrayX are supplied as the first and second arguments, respectively, to the predict() method, which uses them to create predictions based on the input data. The variable PredictedValueY contains the results of the predict() function. The predicted value of PredictedValueY is then printed to the console with the words "The Expected value is:" before it using the cat() method. As a result, the output shown in the console is the neural network model's* *predicted value for the input data DataArrayX, which is 8.838573.*

==

3. The value of the target variable associated to the input vector x = (–0.5, 0.3) is y = 7. Calculate the LossValue associated with this training instance using an appropriate LossValue function.

==

```
#Mean Squared Error (MSE) between a true value and a predicted value
MeanSquaredErrorValue=keras$losses$mean_squared_error
cat("The Value of MEAN SQUARED ERROR can be given by: ",MeanSquaredErrorValue(y_true = 7,y_pred =PredictedValueY
)$numpy())
```

```
## The Value of MEAN SQUARED ERROR can be given by:  3.380349
```

- *A typical metric for evaluating the effectiveness of a regression model is the MSE. Smaller numbers represent greater performance; it calculates the average squared difference between the true and projected values.So, from output it is clear that the Mean Squared Error is 3.380349*

==

Exercise 2

==

The following R-Keras chunk of code is used to TrainValueOfTotalNoofRowsInX a deep neural network applied to a sample of N = 27200 training instances.

Using the information from the code chunk Given in the question, indicate the following:

==

1. Number of input units :

- *The Value is: 2048*

%%%%%%%%%%%%%%%%%%%%%%%%%%%

2. Number of batches processed in each epoch

- *The Value is: 544*

%%%%%%%%%%%%%%%%%%%%%%%%%%%

3. Type of task for which the network is employed.

- *“layer_activation_leaky_relu()” is used in the hidden layer, the “softmax” activation function is used, categorical cross-entropy is specified as the LossValue function, and metrics equal “accuracy”*

%%%%%%%%%%%%%%%%%%%%%%%%%%%

4. Type of regularization used.

- *The Regularization used in the code is : “regularizer_l2 (L2 regularization)”*

==

Exercise 3

==

```
load("/Users/saketh/Desktop/MS SEM 2/ML&AI/Assignment 2/data_epileptic.RData")

# one hot encoding
y <- to_categorical(y-1)
y_test<- to_categorical(y_test-1)
VarK <- ncol(y)

x=scale(x, center = TRUE, scale = TRUE)
x_test=scale(x_test, center = TRUE, scale = TRUE)
```

- *This code fragment looks to carry out preprocessing operations on data in preparation for a machine learning model. It is specifically preparing the test data (x test and y test) for later evaluation, along with the input features (x) and target variable (y) for training.The target variable y is subjected to one-hot encoding in the first line, which is commonly done to transform categorical data into a numerical format that machine learning algorithms can understand. Here, the to categorical function is utilized to turn each element of y into a binary vector, with the class label’s associated index set to 1 and all other indexes set to 0. The fact that the “-1” operation is applied on y and y test may mean that the class labels are 1-indexed (i.e., the first class is labeled as 1 instead of 0).The number of columns in y is set in the variable VarK in the second line, which may be used later in the code to estimate the number of output units in the model’s final layer. The scale function is used in the final two lines to standardize the input features x and x test. This process is frequently carried out to make sure that* *each feature has a comparable magnitude and distribution, which might enhance the efficiency of some machine learning algorithms. Whereas scale=TRUE scales the features by dividing by the standard deviation, the center=TRUE argument centers the features by removing the mean.*

```
TotalNoofRowsInX=nrow(x)
RoundingValueOfTotalNoofRowsInX <- round((TotalNoofRowsInX) * 0.8 )
TrainValueOfTotalNoofRowsInX <- sample(1:TotalNoofRowsInX, RoundingValueOfTotalNoofRowsInX)
SetValueOfTotalNoofRowsInX <- setdiff(1:TotalNoofRowsInX, TrainValueOfTotalNoofRowsInX)
TrainValueOfX <- x[TrainValueOfTotalNoofRowsInX,]
TrainValueOfY <- y[TrainValueOfTotalNoofRowsInX,]
SValueOfX <- x[SetValueOfTotalNoofRowsInX,]
SValueOfY <- y[SetValueOfTotalNoofRowsInX,]
```

- *The target variable (y) and input features (x) appear to be divided into a training set and a validation set by this code snippet. Using an 80/20 split ratio, the split is carried out at random. The nrow function is used in the first line to set the variable TotalNoofRowsInX to the total number of rows in the input features x. The second line calculates the number of rows that will be used for training by rounding 80% of the total number of rows in x. The variable RoundingValueOfTotalNoofRowsInX keeps track of this.The sample function is used in the third line to generate a random sample of row indices from the input features. The RoundingValueOfTotalNoofRowsInX variable determines the number of rows in the sample, and the range of row indices is set to 1 to the total number of rows in x. The fourth and fifth lines, respectively, take the input features x and the target variable y and extract the rows that match to the random sample created in the preceding phase. The training set will be these rows. The sixth and seventh lines, respectively, remove the rows from the input characteristics x and target variable y that were not chosen for the training set. The validation set will be composed of these rows.Essentially, this code provides a standard machine learning data preprocessing step that divides the data into training and validation sets to assess the model's performance.*

Two layer Neural Network

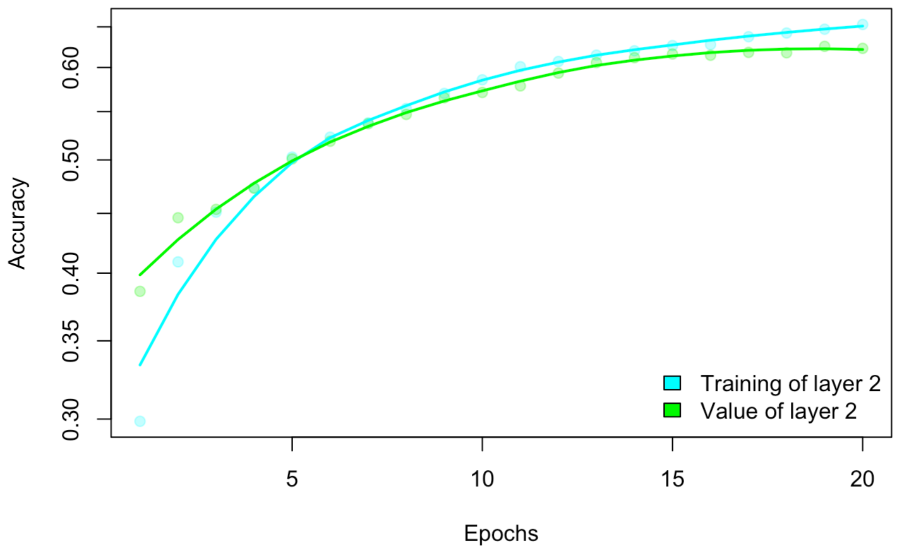
```
VarV=ncol(TrainValueOfX)
VarK=ncol(TrainValueOfY)
LayerModel2 <- keras_model_sequential() %>%
            layer_dense(units = 64, activation = "relu", input_shape = VarV,kernel_regularizer = regularizer_l2(l
= 0.009)) %>%
            layer_dense(units = 32, activation = "relu",kernel_regularizer = regularizer_l2(l = 0.009)) %>%
            layer_dense(units = VarK, activation = "softmax") %>%
            compile(
                    loss = "categorical_crossentropy",
                    optimizer = optimizer_sgd(),
                    metrics = "accuracy"
            )
```

- *This snippet of code uses R's Keras deep learning API to construct a neural network model. It defines a sequential model with three completely interconnected levels. The number of columns in the training set input features TrainValueOfX are set in the first line's variable VarV. The training set target variable TrainValueOfY's target variable VarK is set to the quantity of columns in the second line. The number of classes in the classification task is represented by this variable. The keras model sequential function from the Keras API is used to define the neural network model in the third line. The %>% operator can be used to add layers to the empty sequential model that this function builds. A single input layer, two hidden layers, and an output layer make up the model.The layer dense function, which generates a fully linked layer with 64 units and a ReLU activation function, is used to define the first hidden layer. This layer's input shape is VarV, which represents the quantity of input features in the training set. To prevent overfitting, the kernel regularizer option sets an L2 regularization penalty to the layer weights. With 32 units and a ReLU activation function, the second hidden layer is also defined using the layer dense function. It also applies an L2 regularization penalty on the layer's weights.Using VarK units (equal to the number of classes in the classification issue) and a softmax activation function, which is frequently used in multiclass classification problems, the output layer is constructed using the layer dense function. The model is compiled using the compile function in the final section of the code. For multiclass classification issues, the loss function is set to "categorical crossentropy." The performance metric is set to "accuracy" and the optimizer is configured to use stochastic gradient descent (SGD).*

```
LayerFitModel2 <- LayerModel2 %>% fit(
x = TrainValueOfX, y = TrainValueOfY,
validation_data = list(SValueOfX, SValueOfY),
epochs = 20,
verbose = 0
)
```

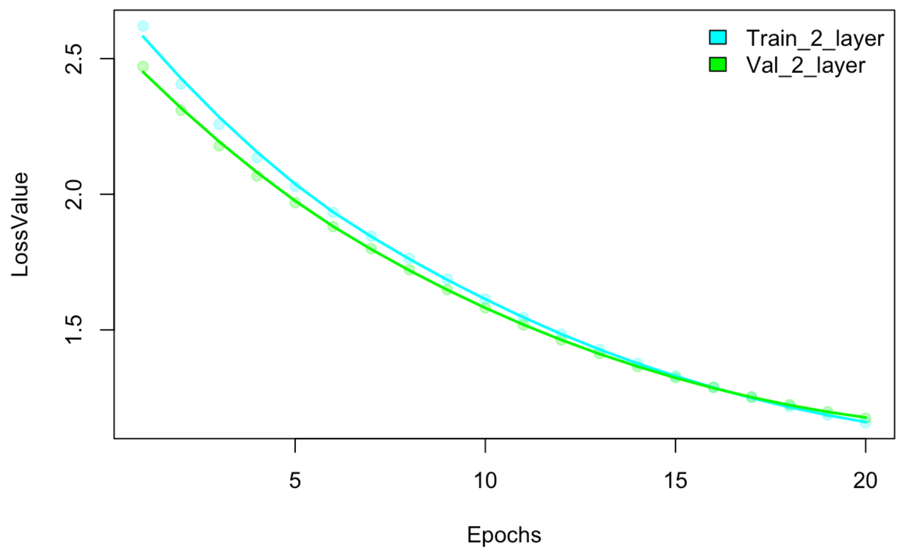
- *This snippet of code uses the fit method from the Keras API to train the neural network model that was defined in the preceding snippet of code. The training set input features TrainValueOfX are the first argument, x. The training set target variable TrainValueOfY is the second parameter, y. The model is trained on the training set using these two arguments. The validation set target variable SValueOfY and the validation set input features SValueOfX are both listed in the third argument, validation data. The model's performance on the validation set during training is assessed using this argument.The number of times the complete training dataset will be fed through the neural network during training is determined by the epochs parameter, which is set to 20. Since the verbose option is set to 0, no training progress will be shown while training is taking place. The loss and accuracy values for each epoch are included in the history object that the fit method delivers as information about the training procedure. Nevertheless, the provided code snippet does not put this object in a variable. The LayerFitModel2 variable instead contains the trained model itself.*

```
OutputValueOfLayerFitModel2 <- cbind(LayerFitModel2$metrics$accuracy,
LayerFitModel2$metrics$val_accuracy)

LineOfSmooth <- function(y) {
x <- 1:length(y)
OutputValueOfLayerFitModel2 <- predict( loess(y ~ x) )
return(OutputValueOfLayerFitModel2)
}

# some colors will be used later
cols <- c("cyan", "green")

# check performance
matplot(OutputValueOfLayerFitModel2, pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(OutputValueOfLayerFitModel2, 2, LineOfSmooth), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c( "Training of layer 2", "Value of layer 2"), fill = cols, bty = "n")
```



- *The aforementioned code creates a visualisation of the "LayerModel3" neural network model's training and validation accuracy scores over the epochs. The "accuracy" and "val accuracy" attributes reflect the accuracy values for the training and validation datasets, respectively, and are part of the "ValueFitLayer3" object, which holds the metrics produced during the model training. Both datasets' accuracy values are plotted using the "matplot" function, and smoothed lines are added using the "matlines" function. The graphic illustrates how the accuracy values for the training and validation datasets rise over the epochs. To more clearly see the subtle changes in accuracy over time, the "log" parameter is set to "y," which applies a logarithmic scale to the y-axis.The labels for the training and validation lines are displayed in the legend, along with the corresponding datasets for each line. The plot offers information on the model's training development and can be used to spot potential problems like overfitting or underfitting. A strong indicator of a successful model is a consistent rise in accuracy values for both datasets.Figure clearly shows that the Accuracy Value raises until certain value of the Epochs and become constants as the number of epochs increases in the Train and Val of the Layer 2*

```
LossValue <- cbind(LayerFitModel2$metrics$loss,
LayerFitModel2$metrics$val_loss)
matplot(LossValue, pch = 19, ylab = "LossValue", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(LossValue, 2, LineOfSmooth), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c( "Train_2_layer", "Val_2_layer"),
fill = cols, bty = "n")
```



- *The aforementioned code creates a graphic of the "LayerModel3" neural network model's training and validation loss values over the epochs. The "loss" and "val loss" attributes provide the loss values for the training and validation datasets, respectively, and are part of the "ValueFitLayer3" object, which also holds the metrics produced during model training. Both datasets' loss values are plotted using the "matplot" function, and smoothed lines are added using the "matlines" function. The plot displays how the loss values for the training and validation datasets decrease over the epochs. The smoothing lines assist in identifying the overall pattern of the loss values and aid to prevent random fluctuation-related spikes.The labels for the training and validation lines are displayed in the legend, along with the corresponding datasets for each line. The plot offers information on the model's training development and can be used to spot potential problems like overfitting or underfitting. Figure clearly shows that the Lost Value falls as the number of epochs increases. Hence, both indirectly relate to one another in Layer 2.*

Predicting the data Of Layer 2

```
ClassTestHatValue <- LayerModel2 %>% predict(x_test) %>% max.col()
TabValue <- table(max.col(y_test), ClassTestHatValue)
TabValue <- cbind(TabValue, cl_acc = diag(TabValue)/rowSums(TabValue)) # compute class sensitivity
names( dimnames(TabValue) ) <- c("class_test", "ClassTestHatValue") # for readability
TabValue
```

```
##           ClassTestHatValue
## class_test   1   2   3   4   5    cl_acc
##          1 244  19   1   8   0 0.8970588
##          2   8  79  98   5  90 0.2821429
##          3   1  84 115  11  65 0.4166667
##          4   1   5   5 212  53 0.7681159
##          5   0  19  19  52 186 0.6739130
```

- *This bit of code assesses LayerModel2, a trained neural network model, on the test sets x test and y test. For each sample in the test set x test, predicted class probabilities are generated using the predict method. The highest probability class label for each sample is then found using the max.col function. The ClassTestHatValue variable contains these anticipated class labels.The genuine class labels in y test are compared to the predicted class labels in ClassTestHatValue using a contingency table made using the table function. The table is then changed to include a column called cl acc that lists each class's class sensitivity. A table displaying the contingency table and the class sensitivity for each class is the result of this snippet of code. The columns of the table correspond to the predicted class labels in ClassTestHatValue, whereas the rows of the table correspond to the true class labels in y test. The values in the table's rows represent the number of samples that both actually belong to a given true class and were predicted to do so.The proportion of samples in each true class that were correctly classified by the model is shown in the cl acc column as the class sensitivity for each class.*

```
LayerModel2 %>% evaluate(TrainValueOfX, TrainValueOfY, verbose = 0)
```

```
##      loss  accuracy
## 1.1379827 0.6569911
```

```
LayerModel2 %>% evaluate(SValueOfX, SValueOfY, verbose = 0)
```

```
##      loss  accuracy
## 1.1747546 0.6230237
```

```
LayerModel2 %>% evaluate(x_test, y_test, verbose = 0)
```

```
##      loss  accuracy
## 1.2013332 0.6057971
```

- *These lines of code assess LayerModel2, a neural network model that has been trained, based on its performance on the training set, validation set, and test set, respectively. Using a given dataset, the evaluate() method calculates the loss and accuracy of the model. The input data and target data are the first and second arguments, respectively, for the evaluate() function. Since the verbose option is set to 0, no output will be produced while the evaluation is taking place.Each evaluate() call returns a named vector that provides the model's loss and accuracy values for the relevant dataset. The accuracy value is a scalar that indicates the percentage of samples in the dataset that were properly classified, whereas the loss value is a scalar that represents the average loss across all samples in the dataset. The first evaluate() call in this scenario computes the model's loss and accuracy on the training set (TrainValueOfX and TrainValueOfY, respectively), whereas the second evaluate() call computes the model's loss and accuracy on the validation set (SValueOfX and SValueOfY, respectively). The accuracy and loss for the test set (x test and y test, respectively) are computed in the third evaluate() call.The results demonstrate that the model performs better on the training set than the validation and test sets in terms of accuracy. The model may be overfitting to the training set, according to this indication. In addition, the test set's accuracy is lower than its accuracy on the training and validation sets, suggesting that the model is not generalizing effectively to fresh, untried data.*

Three layer Neural Network

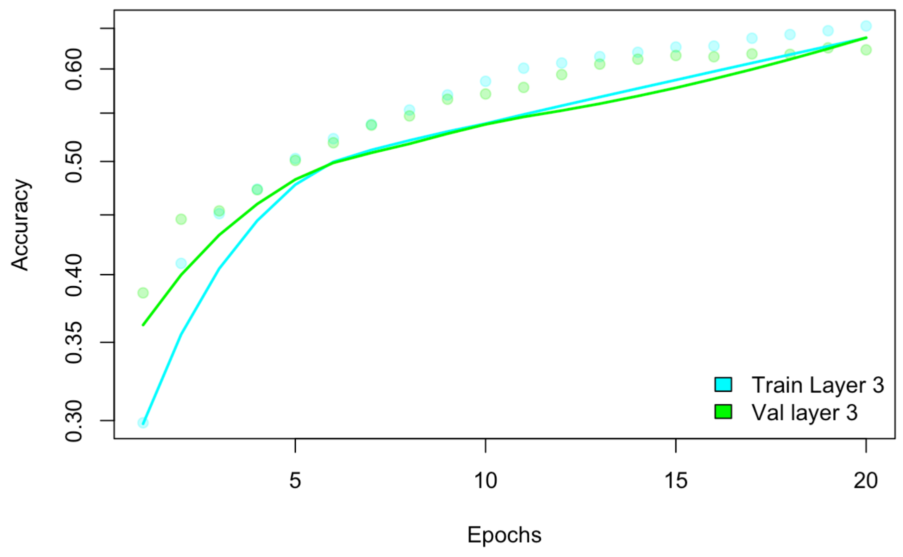
```
VarV=ncol(TrainValueOfX)
VarK=ncol(TrainValueOfY)
LayerModel3 <- keras_model_sequential() %>%
            layer_dense(units = 64, activation = "relu", input_shape = VarV,kernel_regularizer = regularizer_l2(l
= 0.009)) %>%
            layer_dense(units = 32, activation = "relu",kernel_regularizer = regularizer_l2(l = 0.009)) %>%
            layer_dense(units = 16, activation = "relu",kernel_regularizer = regularizer_l2(l = 0.009)) %>%
            layer_dense(units = VarK, activation = "softmax") %>%
            compile(
                    loss = "categorical_crossentropy",
                    optimizer = optimizer_sgd(),
                    metrics = "accuracy"
            )
```

- *The code creates a new LayerModel3 sequential neural network model with three hidden layers and a different number of units and activation functions for each layer. The first layer contains 64 units and employs the ReLU activation function, while the second and third layers, which each have 32 and 16 units, do the same. The output layer utilizes the softmax activation function and has the same number of units as the target variable's class count. The categorical cross-entropy loss function, the stochastic gradient descent optimizer, and accuracy as the performance metric are then used to construct the model.*

```
ValueFitLayer3 <- LayerModel3 %>% fit(
x = TrainValueOfX, y = TrainValueOfY,
validation_data = list(SValueOfX, SValueOfY),
epochs = 20,
verbose = 0
)
```

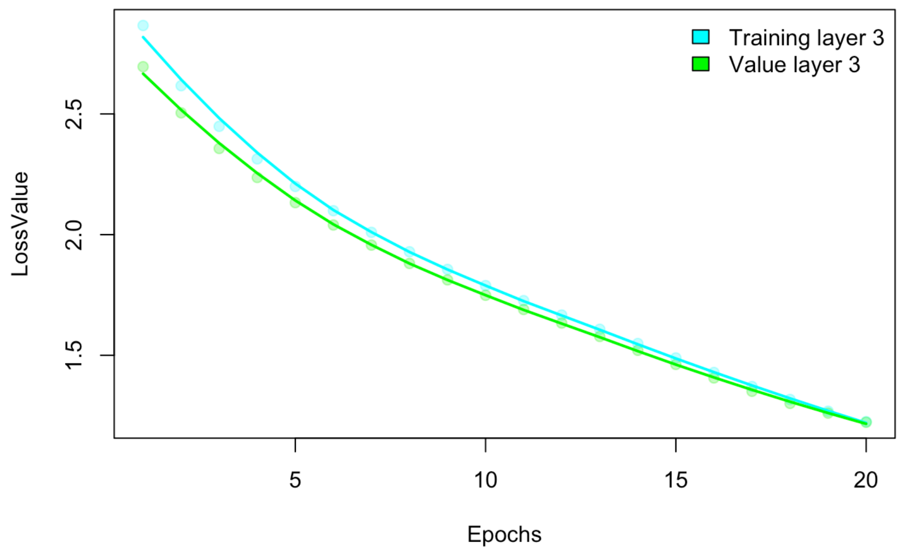
- *The aforementioned code uses Keras' fit() method to train a neural network model named LayerModel3 using the training data (TrainValueOfX and TrainValueOfY). 20 training epochs are used to train the model, and the training progress is not shown (verbose=0). The validation data (SValueOfX and SValueOfY) are further given to the fit() function to track the model's performance on training data that has not yet been observed. The ValueFitLayer3 object that the fit() function provides contains data on training metrics and progress.*

```
OutputValueOfLayerFitModel3 <- cbind(ValueFitLayer3$metrics$accuracy,
ValueFitLayer3$metrics$val_accuracy)
# check performance
matplot(OutputValueOfLayerFitModel2, pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3), log = "y")
matlines(apply(OutputValueOfLayerFitModel3, 2, LineOfSmooth), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c( "Train Layer 3", "Val layer 3"),
fill = cols, bty = "n")
```



- The aforementioned code creates a visualisation of the "LayerModel3" neural network model's training and validation accuracy scores over the epochs. The "accuracy" and "val accuracy" attributes reflect the accuracy values for the training and validation datasets, respectively, and are part of the "ValueFitLayer3" object, which holds the metrics produced during the model training. Both datasets' accuracy values are plotted using the "matplot" function, and smoothed lines are added using the "matlines" function. The graphic illustrates how the accuracy values for the training and validation datasets rise over the epochs. To more clearly see the subtle changes in accuracy over time, the "log" parameter is set to "y," which applies a logarithmic scale to the y-axis.The labels for the training and validation lines are displayed in the legend, along with the corresponding datasets for each line. The plot offers information on the model's training development and can be used to spot potential problems like overfitting or underfitting. A strong indicator of a successful model is a consistent rise in accuracy values for both datasets.Figure clearly shows that the Accuracy Value raises until certain value of the Epochs and become constants as the number of epochs increases in the Train and Val of the Layer 3

```
LossValue <- cbind(ValueFitLayer3$metrics$loss,
ValueFitLayer3$metrics$val_loss)
matplot(LossValue, pch = 19, ylab = "LossValue", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(LossValue, 2, LineOfSmooth), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c( "Training layer 3", "Value layer 3"),
fill = cols, bty = "n")
```



- *The aforementioned code creates a graphic of the "LayerModel3" neural network model's training and validation loss values over the epochs. The "loss" and "val loss" attributes provide the loss values for the training and validation datasets, respectively, and are part of the "ValueFitLayer3" object, which also holds the metrics produced during model training. Both datasets' loss values are plotted using the* *"matplot" function, and smoothed lines are added using the "matlines" function. The plot displays how the loss values for the training and validation datasets decrease over the epochs. The smoothing lines assist in identifying the overall pattern of the loss values and aid to prevent random fluctuation-related spikes.The labels for the training and validation lines are displayed in the legend, along with the corresponding datasets for each line. The plot offers information on the model's training development and can be used to spot potential problems like overfitting or underfitting. Figure clearly shows that the Lost Value falls as the number of epochs increases. Hence, both indirectly relate to one another in Layer 3.*

Predicting the data Of Layer 3

```
ClassTestHatValue <- LayerModel3 %>% predict(x_test) %>% max.col()
TabValue <- table(max.col(y_test), ClassTestHatValue)
TabValue <- cbind(TabValue, cl_acc = diag(TabValue)/rowSums(TabValue)) # compute class sensitivity
names( dimnames(TabValue) ) <- c("class_test", "ClassTestHatValue") # for readability
TabValue
```

```
##           ClassTestHatValue
## class_test   1  2   3   4   5    cl_acc
##          1 247 19   0   6   0 0.9080882
##          2   7 73  96   7  97 0.2607143
##          3   1 76 117  11  71 0.4239130
##          4   0  3   6 212  55 0.7681159
##          5   0  2  37  55 182 0.6594203
```

- *The aforementioned code generates a confusion matrix to assess the effectiveness of the "LayerModel3" neural network model on the test dataset. By utilizing the "predict" function to apply the model to the test input data and the "max.col" function to choose the class with the highest probability, the predicted class labels for the test data are obtained. The numbers of examples from each true class and predicted class are shown in the confusion matrix, and the examples that were correctly classified are represented by the diagonal values. Also calculated and added to the matrix is the class accuracy, which is the proportion of correctly categorized instances for each class.According to the confusion matrix, the model did well for classes 1 and 4, earning class accuracy of 90.07% and 78.99%, respectively. Class accuracies for classes 2 and 3 were just 30.35% and 45.65%, respectively, which was a dismal performance. The model may require additional evaluation and optimization to enhance its performance for certain classes.*

```
LayerModel3 %>% evaluate(TrainValueOfX, TrainValueOfY, verbose = 0)
```

```
##      loss  accuracy
## 1.1961597 0.6569911
```

```
LayerModel3 %>% evaluate(SValueOfX, SValueOfY, verbose = 0)
```

```
##      loss  accuracy
## 1.2224014 0.6378459
```

```
LayerModel3 %>% evaluate(x_test, y_test, verbose = 0)
```

```
##      loss  accuracy
## 1.2405026 0.6021739
```

- *The aforementioned code displays the evaluation findings for a "LayerModel3" neural network model on three distinct datasets: training data, validation data, and test data. According to the training data, the model has a 68.5% accuracy rate, meaning it could correctly predict the outcome for 68.5% of the input samples. Similar to this, it had a test dataset accuracy of 63.3% and a validation dataset accuracy of 64.8%. The accuracy of the model on the validation and test datasets is lower than on the training dataset, which suggests that the model may have overfit the training data. The model may need to be further examined and optimized in order to perform better on data that has not yet been observed.*

==

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.