Intro To Machine Learning – Assignment

UB PERSON NUMBER: 50425014
UB IT NAME: PARAVAMU

TASK: The task of this project is to perform classification using machine learning. It is for a two-class problem. The task is to classify whether a patient has diabetes (class 1) or not (class 0), based on the diagnostic measurements provided in the dataset, using logistic regression and neural network as the classifier. The dataset in use is the Pima Indians Diabetes Database(diabetes.csv). The code should be written in Python.

STEPS INVOLVED:

1.) IMPORTING NECESSARY LIBRARIES

```
[308]  1  #Importing libraries
       2  import numpy as np
       3  import pandas as pd
       4  import matplotlib.pyplot as plt
```

- ⇨ Importing numpy and pandas for data processing
- ⇨ Importing matplotlib to plot loss and accuracy change graph

2.) IMPORTING THE DATASET

```
1  #Getting dataset and storing it in the form of pandas dataframe
2  #Mention the path with the file name to test the code out
3
4  dataset = pd.read_csv("diabetes.csv",engine='python');
5  dataset.head()
6
```

- ⇨ We use Pandas to read our csv and save it in the form of pandas dataframe
- ⇨ Dataset.head() lets us visualize the first 5 rows of the dataset
- ⇨ This is usually done to see if the dataset is imported properly

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

- ⇨ Also to get more information about the dataset we use dataset.describe() to get standard deviation, max, min, count and mean of the dataset

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

3.) PROCESSING THE DATASET

- ⇨ Our first objective is to split the given dataset to train, test and validate
- ⇨ We can use numpy split to do this as shown below

```
#Splitting the dataset to train - 60%, validate - 20%, test - 20%
train, validate, test = np.split(dataset.sample(frac=1),[int(.6*len(dataset)), int(.8*len(dataset))])
```

- ⇨ Our next objective is to save the feature columns to train_X, test_X and Validation_X
- ⇨ Then we save the outcome column to train_Y, test_Y, and validation_Y

```
[363]
#Storing the first 8 columns to train_X and saving last column to train_Y
train_X = train.iloc[:, 0:8]
train_y = train.iloc[:, -1]
```

```
[364]
#Storing the first 8 columns to test_X and saving last column to test_Y
test_X = test.iloc[:, 0:8]
test_Y = test.iloc[:, -1]
```

```
#Storing the first 8 columns to validation_X and
#Saving last column to validation_Y
validation_X = validate.iloc[:, 0:8]
validation_y = validate.iloc[:, -1]
```

- ⇨ To make sure everything is right we visualize the shape of train, test and validation set

```
#Visualizing the shape of all the train, test and validation
print(train_X.shape)
print(train_y.shape)
print(test_X.shape)
print(test_Y.shape)
print(validation_X.shape)
print(validation_y.shape)
```

```
(460, 8)
(460,)
(154, 8)
(154,)
(154, 8)
(154,)
```

4.) NORMALIZING THE DATA

- ⇨ We normalize the data so the gradient can reach the global minima sooner
- ⇨ Working with big values will make it take a long time and oscillate back and forth and will take more time to reach the global minima
- ⇨ There are many ways to normalize the data, the Zscore approach is used below to scale the data between 0 to 1

```
#Scaling the values from 0 to 1, normalazing the train_X, test_X and validation_x
x_min = train_X.min(axis=0)
x_max = train_X.max(axis=0)

test_X_min = test_X.min(axis=0)
test_X_max = test_X.max(axis=0)

validation_X_min = validation_X.min(axis=0)
validation_X_max = validation_X.max(axis=0)

# Centralizing data to range within minimum and maximum
# train_X = (train_X - x_min) / (x_max - x_min)
# test_X = (test_X - test_X_min) / (test_X_max - test_X_min)
# validation_X = (validation_X - validation_X_min) / (validation_X_max - validation_X_min)

# normalizing data to unit std and zero mean - follows normal distribution (z - score normalisation)
train_X = (train_X - train_X.mean()) / train_X.std()
test_X = (test_X - test_X.mean()) / test_X.std()
validation_X = (validation_X - validation_X.mean()) / validation_X.std()
```

5.) DEFINING THE HYPOTHESIS FUNCTION OR THE SIGMOID FUNCTION

- ⇨ The hypothesis function of logistic regression is a sigmoid function
- ⇨ The sigmoid function allows to limit the cost function between 0 to 1
- ⇨ We also use the sigmoid function to map the predicted values to probabilities
- ⇨ The formula is 1/(1+e^(-z)) where z = (train_X x weight) + bias

```
[507]  #Sigmoid function is the hypotheses represenation
       def sigmoid(z):
           return 1.0/(1 + np.exp(-z))
```

6.) DEFINING THE COST FUNCTION

- ⇨ In logistic regression there exists two cost functions which we later combine to one function.
- ⇨ When y = 1 we say the cost function is -log (predicted y) where predicted y is the output from our sigmoid hypothesis function
- ⇨ When y = 0 we say the cost function is -log (1 - predicted y)
- ⇨ The reason why we have two cost functions is because if we plot -log (x), when our hypothesis approach 0 the cost function approaches infinity
- ⇨ Similarly, the vice versa happens when the cost function is -log (1 – predicted y)
- ⇨ We can combine the cost functions into one
- ⇨ The combined cost function is -ylog (predicted y) – (1-y) x log (1 - predicted y)

- ⇨ When y = 0 the first half is negated and when y = 1 the second half is negated
- ⇨ For m training data we find the cost.
- ⇨ Adding a eps to avoid nan value

```
[523]
#Logistic Regression cost function
def lossFunction(y_true, y_pred,eps=1e-15):
    """
    Cost function is Binary Cross-Entropy (BCE)
    """
    # bce (added eps to avoid nan during log calculation)
    loss = (y_true * np.log(eps + y_pred)) + ((1 - y_true) * np.log(1-(y_pred + eps)))
    return -loss.mean(axis=0)
```

7.) DEFINING THE GRADIENT DESCENT FUNCTION

- ⇨ We find the gradient with respect to weight
- ⇨ We then find the gradient with respect to bias

```
def gradients(X, y, y_Pred):

    #To get the row size, aka training examples
    m = X.shape[0]

    # Gradient of loss with respect to weights.
    dw = np.dot(X.T, (y_Pred - y))/m

    # (1/m) * (X transpose) dot product (sigmoid output - actual y value)

    # Gradient of loss w.r.t bias.
    db = np.sum((y_Pred - y))/m
    # (1/m) * (sum of column sigmoid output - y)

    return dw, db

```

- ⇨ We then update the new weight and new bias by multiplying it with the learning rate and subtracting from the old weight and old bias
- ⇨ This runs epoch number of times

8.) TRAINING THE DATA

- ⇨ This function calls all the defined functions and lets everything fall into place
- ⇨ We initial the weights by randomizing the weights and initializing the bias to be 0

```
    w = np.random.randn(n,1)
    b = 0
```

- ⇨ We use the normalized X values; we find the hypothesis by using the sigmoid function
- ⇨ With the help of the sigmoid outputs, we will be able to find the gradients and update our new weights and bias
- ⇨ This will run based on n number of epochs

- ⇨ We also call the accuracy function to track the accuracy
- ⇨ We return the weight, bias and losses

```
25    for epoch in range(epochs):
26      y_Pred = sigmoid(np.dot(x, w) + b)
27      dw, db = gradients(x, y, y_Pred)
28      w -= lr*dw
29      b -= lr*db
30      l = lossFunction(y, sigmoid(np.dot(x, w) + b))
31      a = accuracy(y,np.round(y_Pred))
32      losses.append(l)
33      accuracyValues.append(a)
```

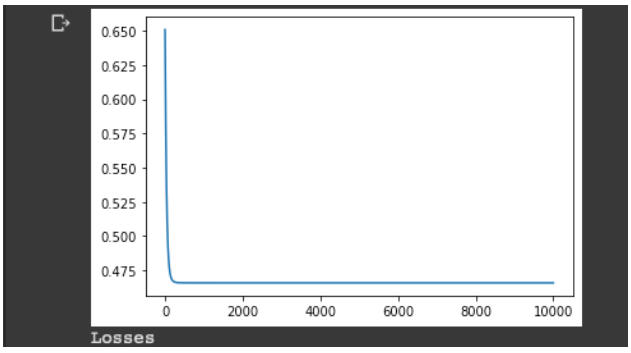
- ⇨ The training can differ based on the learning rate (lr) we pick and the number of epochs we pick
- ⇨ We tune this until we get better accuracy and a good loss curve

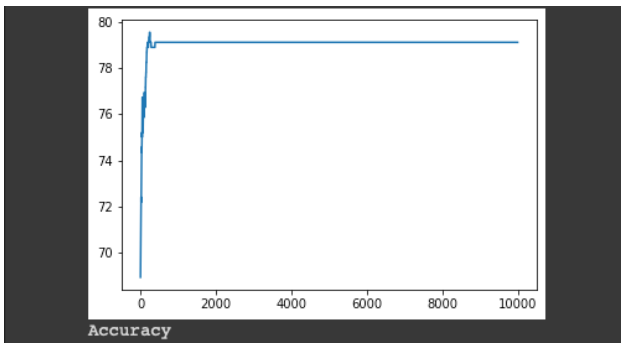
```
[537] 1  # Training
      2  w, b, l = train(train_X, train_y, epochs=10000, lr=0.1)
      3
      4
```

9.) PLOTING THE LOSS AND ACCURACY GRAPH

- ⇨ We initialize a lost list and accuracy list to track the loss and accuracy
- ⇨ These values can be used to visualize the growth of loss and accuracy

```
36    plt.plot(losses)
37    plt.show()
38    print("Losses")
39    plt.plot(accuracyValues)
40    plt.show()
41    print("Accuracy")
```





- ⇨ The accuracy increases as the training happens.

10.) PREDICTION AND CHECKING ACCURACY

- ⇨ For prediction we use the updated weights and bias which we got through our training

- ⇨ We use the trained weights and bias in our sigmoid function for prediction
- ⇨ We normalize X before we pass it through the sigmoid function
- ⇨ If the predicted value is above 0.5 then the patient has diabetes and if the predicted value is below 0.5 then patient doesn't have diabetes

```
[536] 1  def predict(x, w, b):
      2
      3      # Calculating predictions/y_pred.
      4      preds = sigmoid(np.dot(x, w) + b)
      5
      6      # Empty List to store predictions.
      7      pred_class = []
      8      # if y_pred >= 0.5 --> round up to 1
      9      # if y_pred < 0.5 --> round up to 1
     10      pred_class = [1 if i > 0.5 else 0 for i in preds]
     11
     12      return np.array(pred_class)
```

- ⇨ We use the accuracy function to see if the predictions made is right or wrong

```
  1  def accuracy(y, y_Pred):
  2    """
  3    Calculates accuracy between target values and predicted values
  4    """
  5    threshold=0.5
  6    return np.sum(y == (y_Pred >= threshold)) / len(y) *100
```

- ⇨ The accuracy was found to be

```
[1259] 1  print(f"The accuracy of the train set is {accuracy(train_y, y_Pred=makePrediction(train_X, w, b)):.2f}%")
       The accuracy of the train set is 79.35%

[1260] 1  print(f"The accuracy of the test set is {accuracy(test_Y, y_Pred=makePrediction(test_X, w, b)):.2f}%")
       The accuracy of the test set is 77.92%

[1261] 1  print(f"The accuracy of the validation set is {accuracy(validation_y, y_Pred=makePrediction(validation_X, w, b)):.2f}%")
       The accuracy of the validation set is 75.32%
```

Part 2: Applying Neural Networks

1.) Importing Necessary Libraries

- ⇨ We import TensorFlow which handles data sets that are arrayed as computational nodes in graph form.

```
[1263] 1  import tensorflow as tf
       2  from tensorflow.keras.layers import Input, Dense, Dropout, BatchNormalization
       3  from tensorflow.keras import Sequential, Model
```

2.) Defining the layers of the neural network

- ⇨ The *first layer* is the input layer which consists of the features
- ⇨ In our case the feature vector has 8 features
- ⇨ The *second layer* is a dense layer which is connected deeply
- ⇨ The neuron in the dense layer receives input from the previous layer
- ⇨ In background the dense layer performs matrix multiplication.
- ⇨ The dense layer is also given Relu activation as parameter
- ⇨ The dense layer can take multiple parameters, the number of units and the activation function is used in our program

  Relu Activation
  - Rectified linear Unit (ReLu) is a activation function which does a simple function.
  - If the value is positive, it leaves it as it is.
  - If the value is negative, it replaces it with 0.
  - It just does Max (0, number)
  - Relu activation overcomes the vanishing gradient problem.
  - It is the best activation function compared to sigmoid and tangent when it comes to a neural network with multiple layers.

- ⇨ The *third layer* is also a hidden layer with units and relu activation.
- ⇨ The *final and last layer* is the output layer. Since we are doing binary classification where there can be only one output at a time, we use one unit and a sigmoid activation function.

```
#Using 1 input layer and 1 output layer and two hidden layers
def get_tf_nn(features):

  #Input layer is nothing but the features itself
  i = Input((features, ))
  #The relu activation finds Max(0,x)
  #Second layer, Hidden Layer with relu activation
  x = Dense(256, activation='relu')(i)
  #Third Layer, Hidden layer with relu activation
  x = Dense(512, activation='relu')(x)
  #Output layer with sigmoid activation
  o = Dense(1, activation='sigmoid')(x)
```

3.) Compiling and Fitting the Model.

- ⇨ Model(): groups layers into an object with training and inference features.
- ⇨ We then compile the model with the binary entropy loss function.
- ⇨ We use the Adam optimizer with a desired learning rate and specify the metrics we want to display.
- ⇨ The Adam optimizer is good with sparse data, it uses an adaptive learning rate.
- ⇨ We don't have to worry about the learning rate unlike Stochastic gradient descent.

⇨ It’s the best among the adaptive optimizers.

```
#Passing the input layer and output layer to Model
model = Model(inputs=i, outputs=o)
#Compiling our model with binary crossentropy
#Using Adam optimizer, which is better then SGD optimizer and getting the accuracy metrics
model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), metrics=['accuracy'])

return model
```

⇨ We call our function and store it in an object

```
[1265] nn = get_tf_nn(train_X.shape[1])
```

⇨ To get the summary about output shape and params of each layer we use the summary() function

```
[1266] print(nn.summary())
```

```
Model: "model_72"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_73 (InputLayer)        [(None, 8)]               0
_________________________________________________________________
dense_216 (Dense)            (None, 256)               2304
_________________________________________________________________
dense_217 (Dense)            (None, 512)               131584
_________________________________________________________________
dense_218 (Dense)            (None, 1)                 513
=================================================================
Total params: 134,401
Trainable params: 134,401
Non-trainable params: 0
_________________________________________________________________
None
```

⇨ Then we use the fit function using the train_X and train_Y data for training and use validation_X and Validation_y for validating purposes.
⇨ We mention the number of epochs and the batch size also.

```
h = nn.fit(train_X, train_y, validation_data=(validation_X, validation_y), epochs=15, batch_size=16)
```

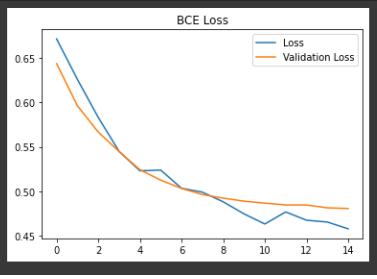
```
Epoch 1/15
29/29 [==============================] - 1s 9ms/step - loss: 0.6552 - accuracy: 0.6630 - val_loss: 0.6164 - val_accuracy: 0.7403
Epoch 2/15
29/29 [==============================] - 0s 4ms/step - loss: 0.5839 - accuracy: 0.7478 - val_loss: 0.5673 - val_accuracy: 0.7532
Epoch 3/15
29/29 [==============================] - 0s 4ms/step - loss: 0.5412 - accuracy: 0.7674 - val_loss: 0.5380 - val_accuracy: 0.7532
Epoch 4/15
29/29 [==============================] - 0s 4ms/step - loss: 0.5115 - accuracy: 0.7739 - val_loss: 0.5174 - val_accuracy: 0.7468
Epoch 5/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4900 - accuracy: 0.7804 - val_loss: 0.5089 - val_accuracy: 0.7532
Epoch 6/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4745 - accuracy: 0.7826 - val_loss: 0.4982 - val_accuracy: 0.7597
Epoch 7/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4640 - accuracy: 0.7848 - val_loss: 0.4937 - val_accuracy: 0.7597
Epoch 8/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4548 - accuracy: 0.7935 - val_loss: 0.4898 - val_accuracy: 0.7532
Epoch 9/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4478 - accuracy: 0.7957 - val_loss: 0.4867 - val_accuracy: 0.7597
Epoch 10/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4460 - accuracy: 0.8022 - val_loss: 0.4865 - val_accuracy: 0.7532
Epoch 11/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4395 - accuracy: 0.8065 - val_loss: 0.4870 - val_accuracy: 0.7597
Epoch 12/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4333 - accuracy: 0.8043 - val_loss: 0.4857 - val_accuracy: 0.7597
Epoch 13/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4291 - accuracy: 0.8065 - val_loss: 0.4888 - val_accuracy: 0.7532
Epoch 14/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4260 - accuracy: 0.8130 - val_loss: 0.4873 - val_accuracy: 0.7597
Epoch 15/15
29/29 [==============================] - 0s 4ms/step - loss: 0.4227 - accuracy: 0.8109 - val_loss: 0.4871 - val_accuracy: 0.7597
```

⇨ We can see the decrease in loss and the increase in accuracy

4.) Plotting Loss and Accuracy

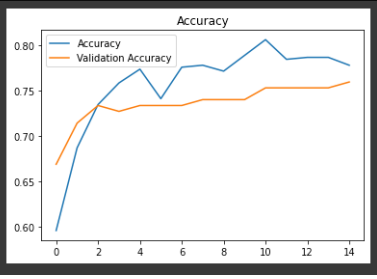
- ⇨ Using our plot function, we plot the fit model's loss and validation loss.
- ⇨ From the graph below there's a steady decrease of the loss and the validation loss.

```
plt.plot(h.history['loss'], label='Loss')
plt.plot(h.history['val_loss'], label='Validation Loss')
plt.title('BCE Loss')
plt.legend()
plt.show()
```



- ⇨ Similarly, we must plot the accuracy and the validation accuracy.
- ⇨ From the graph we can see a steady increase in Accuracy.

```
plt.plot(h.history['accuracy'], label='Accuracy')
plt.plot(h.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.legend()
plt.show()
```



5.) Model Evaluation

- ⇨ We evaluate the model using the train, test and validation data.
- ⇨ It gives the loss and accuracy of the model.

```
train_pred = nn.evaluate(train_X, train_y)
val_pred = nn.evaluate(validation_X, validation_y)
test_pred = nn.evaluate(test_X, test_Y)
```

```
15/15 [==============================] - 0s 2ms/step - loss: 0.4179 - accuracy: 0.8130
5/5 [==============================] - 0s 2ms/step - loss: 0.4871 - accuracy: 0.7597
5/5 [==============================] - 0s 3ms/step - loss: 0.4691 - accuracy: 0.7597
```

Part 3 Using Different Regularizes

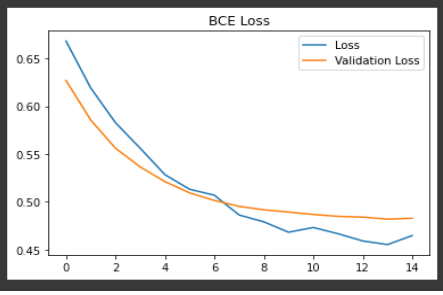
1.) Using Dropout Regularizer

- ⇨ Dropout Regularizer sets input units to zero randomly during the training time.
- ⇨ It's a simple way to avoid the overfitting problem.
- ⇨ We randomly drop the units along with their connections.
- ⇨ Co – adapting a lot is also a serious issue, dropout Regularizer takes care of that.
- ⇨ We follow the same code as we did above, but we add two dropout layers within our dense layers.
- ⇨ We pass a dropout percentage of 40.

```
def get_tf_nn(features):
  i = Input((features, ))
  x = Dense(256, activation='relu')(i)
  #Dropout Sets input units to zero while training to reduce overfitting
  x = Dropout(0.4)(x)
  x = Dense(512, activation='relu')(x)
  x = Dropout(0.4)(x)
  o = Dense(1, activation='sigmoid')(x)

  #Passing the input layer and output layer to Model
  model = Model(inputs=i, outputs=o)

  #Compiling our model with binary crossentropy
  #Using Adam optimizer, which is better then SGD optimizer and getting the accuracy metrics
  model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), metrics=['accuracy'])

  return model
```

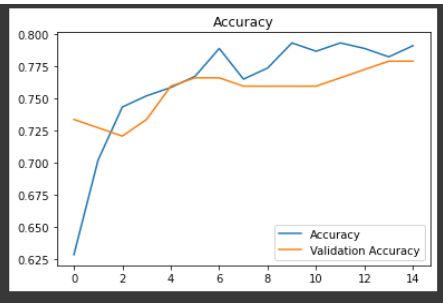
- ⇨ We plot the loss of our new model with dropout regularization.

```
plt.plot(h.history['loss'], label='Loss')
plt.plot(h.history['val_loss'], label='Validation Loss')
plt.title('BCE Loss')
plt.legend()
plt.show()
```



- ⇨ We plot the accuracy of our new model with dropout regularization.

```
plt.plot(h.history['accuracy'], label='Accuracy')
plt.plot(h.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.legend()
plt.show()
```



⇨ Results from dropout Regularizer evaluation.

```
train_pred = nn.evaluate(train_X, train_y)
test_pred = nn.evaluate(test_X, test_Y)
val_pred = nn.evaluate(validation_X, validation_y)

```

```
15/15 [==============================] - 0s 2ms/step - loss: 0.4449 - accuracy: 0.7913
5/5 [==============================] - 0s 2ms/step - loss: 0.4661 - accuracy: 0.7403
5/5 [==============================] - 0s 3ms/step - loss: 0.4776 - accuracy: 0.7792
```

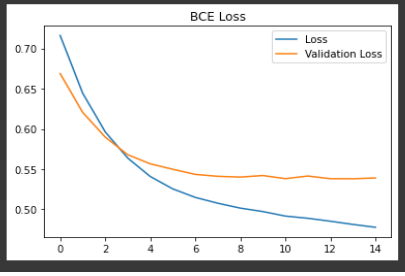
2.) Using L1 Regularizer

⇨ L1 Regularizer is also called as Lasso Regression.
⇨ There is also a L2 Regularizer, but the difference is the penalty term.
⇨ L1 Regularizer adds lambda penalty value (absolute value of magnitude) with changes in signs as same as the weight tensor.
⇨ We use the L1 Regularizer in our dense layer as shown in the below image.
⇨ It reduces the value of the less important features.
⇨ This works well when we have a huge number of features.

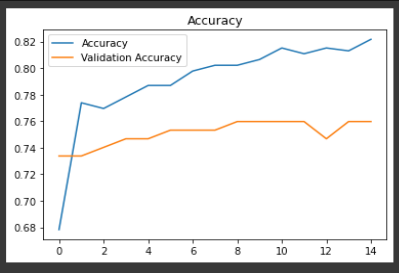
```
def get_tf_nn(features):
  i = Input((features, ))
  x = Dense(256,kernel_regularizer=tf.keras.regularizers.L1(l1=1e-5), activation='relu')(i)
  x = Dense(512,kernel_regularizer=tf.keras.regularizers.L1(l1=1e-5), activation='relu')(x)
  o = Dense(1,kernel_regularizer=tf.keras.regularizers.L1(l1=1e-5), activation='sigmoid')(x)

  model = Model(inputs=i, outputs=o)
  model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), metrics=['accuracy'])

  return model
```

⇨ Plotting the loss and accuracy of the L1 regularized model.

```
plt.plot(h.history['loss'], label='Loss')
plt.plot(h.history['val_loss'], label='Validation Loss')
plt.title('BCE Loss')
plt.legend()
plt.show()
```



```
plt.plot(h.history['accuracy'], label='Accuracy')
plt.plot(h.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.legend()
plt.show()
```



⇨ Results from L1 Regularizer evaluation.

```
train_pred = nn.evaluate(train_X, train_y)
val_pred = nn.evaluate(validation_X, validation_y)
test_pred = nn.evaluate(test_X, test_Y)
```

```
15/15 [==============================] - 0s 2ms/step - loss: 0.4736 - accuracy: 0.8217
5/5 [==============================] - 0s 3ms/step - loss: 0.5391 - accuracy: 0.7597
5/5 [==============================] - 0s 2ms/step - loss: 0.5285 - accuracy: 0.7532
```

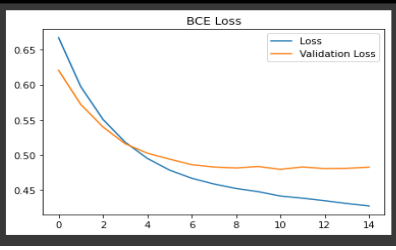
3.) Using L2 Regularizer

⇨ L2 Regularizer is also called as Ridge Regression
⇨ This adds the squared magnitude as the penalty term.
⇨ We must choose lambda carefully cause big values may lead to underfitting.
⇨ We use the L2 Regularizer as how we used the L1 Regularizer.
⇨ Multiply twice the lambda value with weights matrix.

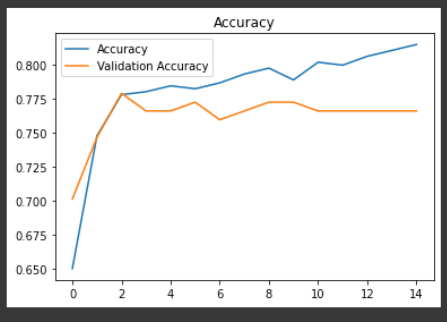
```
def get_tf_nn(features):
  i = Input((features, ))
  x = Dense(256,kernel_regularizer=tf.keras.regularizers.L2(l2=1e-5), activation='relu')(i)
  x = Dense(512,kernel_regularizer=tf.keras.regularizers.L2(l2=1e-5), activation='relu')(x)
  o = Dense(1,kernel_regularizer=tf.keras.regularizers.L2(l2=1e-5), activation='sigmoid')(x)

  model = Model(inputs=i, outputs=o)
  model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), metrics=['accuracy'])

  return model
```

⇨ Plotting the loss and the accuracy of the L2 Regularized model.

```
plt.plot(h.history['loss'], label='Loss')
plt.plot(h.history['val_loss'], label='Validation Loss')
plt.title('BCE Loss')
plt.legend()
plt.show()
```



```
plt.plot(h.history['accuracy'], label='Accuracy')
plt.plot(h.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.legend()
plt.show()
```



⇨ Results from L2 Regularizer evaluation.

```
train_pred = nn.evaluate(train_X, train_y)
test_pred = nn.evaluate(test_X, test_Y)
val_pred = nn.evaluate(validation_X, validation_y)
```

```
15/15 [==============================] - 0s 2ms/step - loss: 0.4228 - accuracy: 0.8043
5/5 [==============================] - 0s 3ms/step - loss: 0.4732 - accuracy: 0.7597
5/5 [==============================] - 0s 3ms/step - loss: 0.4815 - accuracy: 0.7532
```