Logistic Regression & Neural Networks Implementation
Dataset: Pima Indians Diabetes Database

Abstract

Phase 1:
This phase of project is an effort to use Logistic Regression, written from the ground up, to learn from the Pima Indians Diabetes Database dataset and predict if a patient has Diabetes or not based on the diagnostic measures supplied in the dataset. Gradient Descent is used to optimize weights and bias. On the test set, an accuracy of 80% is obtained.

Phase 2:
The goal of this phase is to predict if a patient has Diabetes or not based on the diagnostic measures supplied in the dataset using Neural Networks. Binary cross entropy loss function is used to calculate the loss and Optimizer Adam is being used to minimize the loss function.

1.Introduction

Pima Indians with type 2 diabetes are **metabolically characterized by obesity, insulin resistance**, insulin secretory dysfunction, and increased rates of endogenous glucose production, which are the clinical characteristics that define this disease across most populations. In this study, variables such as glucose levels, skin thickness, and insulin levels are used to determine whether a person has diabetes.

2.Dataset

Pima Indians Diabetes Database dataset will be used for training, and testing. The dataset contains medical data of female patients above the age of 21 and 768 instances with the diagnostic measurements of 8 features. The 8 features are as follows:

1	Glucose (Blood Glucose level)
2	Pregnancies (The number of pregnancies the patient has had)
3	Blood Pressure (mm Hg)
4	Skin Thickness (Triceps skin fold thickness (mm))
5	Insulin level
6	BMI (Body Mass Index: weight in kg/(height in m)^2)
7	Diabetes Pedigree Function
8	Age (In years)

3.Data Preprocessing

- The given dataset contains 768 instances, splitting this data into training, test, and validation sets.

Testing data	60% of Total data
Validation data	20% of Total data
Test data	20% of Total data

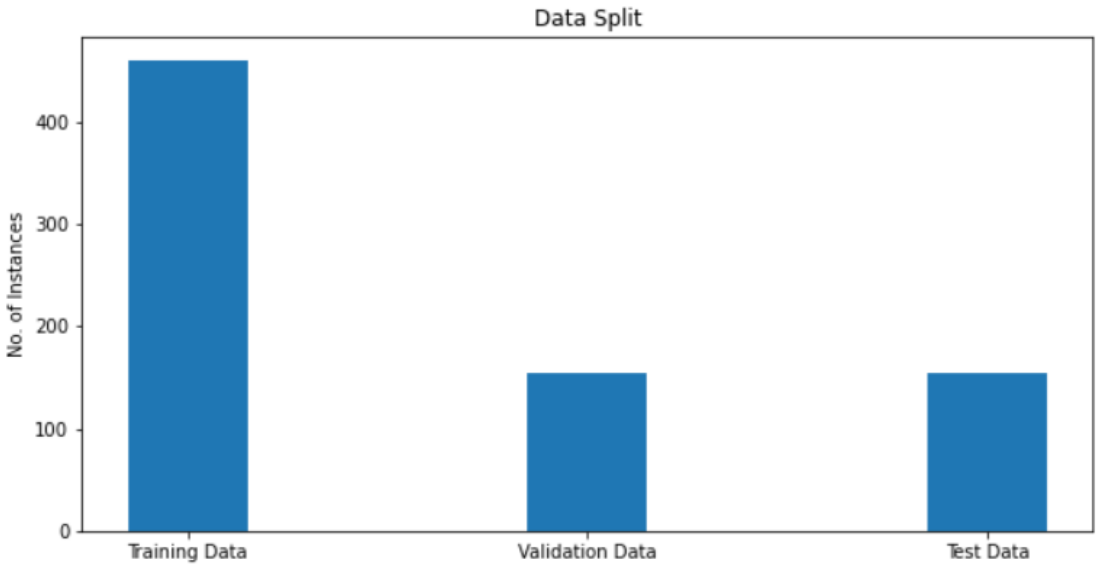


Figure 1

3.1 Imputation and Correlation

- There are many instances with zero values for some features, which is practically not possible.

Risk Factors / Features	**Zero Count**
Pregnancies	111
Blood Pressure	35
Glucose	5
Skin Thickness	227
Insulin	374
BMI	11
Diabetes Pedigree Function	0
Age	0

- To avoid the data leakage and improve model performance these zero values are imputed with mean of the respective column.
- The correlation between these features improves after imputation, represented in Figure2.

Implementation

Calculated the mean of each column and replaced the 0 values with the mean.

```
mean_insulin=train_data_mean['Insulin'].mean()
mean_Glucose=train_data_mean['Glucose'].mean()
mean_BloodPressure=train_data_mean['BloodPressure'].mean()
mean_SkinThickness=train_data_mean['SkinThickness'].mean()
mean_BMI=train_data_mean['BMI'].mean()
mean_Pregnancies=math.floor(train_data_mean['Pregnancies'].mean())

train_data_mean.loc[train_data_mean['Insulin'] == 0, 'Insulin'] = mean_insulin
train_data_mean.loc[train_data_mean['Glucose'] == 0, 'Glucose'] = mean_Glucose
train_data_mean.loc[train_data_mean['BloodPressure'] == 0, 'BloodPressure'] = mean_BloodPressur
train_data_mean.loc[train_data_mean['SkinThickness'] == 0, 'SkinThickness'] = mean_SkinThickne
train_data_mean.loc[train_data_mean['BMI'] == 0, 'BMI'] = mean_BMI
train_data_mean.loc[train_data_mean['Pregnancies'] == 0, 'Pregnancies'] = mean_Pregnancies
```

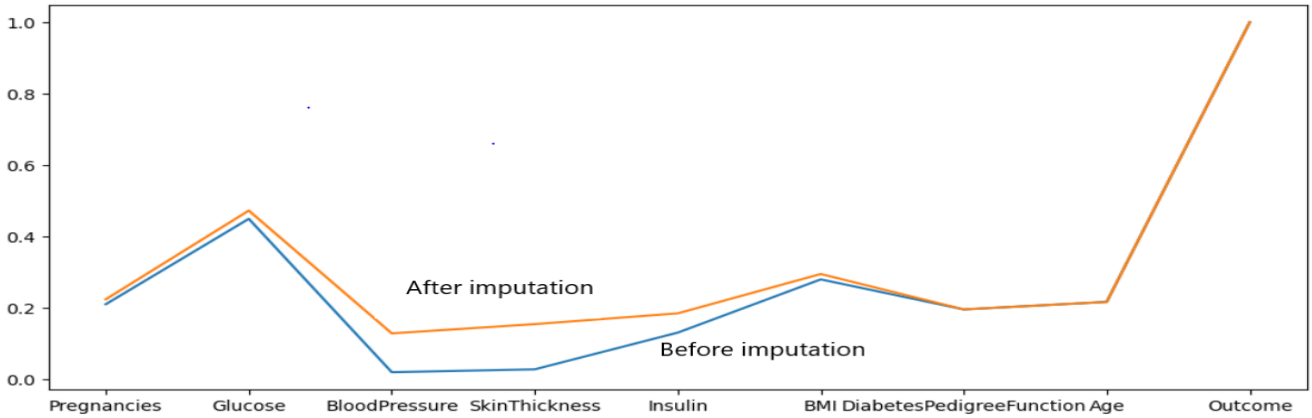


Figure 2

3.2 Normalization

- The values of each feature are in different ranges

```
In [4]: diabetes_data.head()
```

Out[4]:

	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

Figure 3

```
(diabetes_data[:]==0).sum()
```

```
Pregnancies                 111
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
```

From the above data it is observed that each feature has different ranges, so normalizing the data and bringing every value in the range of 0-1 would help the model to process the data accurately.

$$y = (x - \min(x))/(\max(x) - \min(x))$$

y: cell value of each column

Every cell value will be normalized, and the range of every column will be between 0 to1

- The normalization is done after the data split, the train data min max values are used to normalize test and validation data.

Implementation

Calculated the mean of each column and replaced the 0 values with the mean.

```
#Normalize
train_data_norm=train_data_mean.copy()
validation_data_norm=validation_data_mean.copy()
test_data_norm=test_data_mean.copy()
for i in train_data_norm.columns[:-1]:
    validation_data_norm[i]=(validation_data_norm[i]-train_data_norm[i].min())/(train_data_norm[i].max()-train_data_norm[i].min())
    test_data_norm[i]=(test_data_norm[i]-train_data_norm[i].min())/(train_data_norm[i].max()-train_data_norm[i].min())
    train_data_norm[i]=(train_data_norm[i]-train_data_norm[i].min())/(train_data_norm[i].max()-train_data_norm[i].min())
```

4.Model Architecture – Logistic Regression

4.1 Logistic Regression

Logistic regression is one of the most common machine learning algorithms used for binary classification. It predicts the probability of occurrence of a binary outcome using a logit function.
It is a special case of linear regression as it predicts the probabilities of outcome using log function.

We use the activation function (sigmoid) to convert the outcome into categorical value.

Sigmoid Function

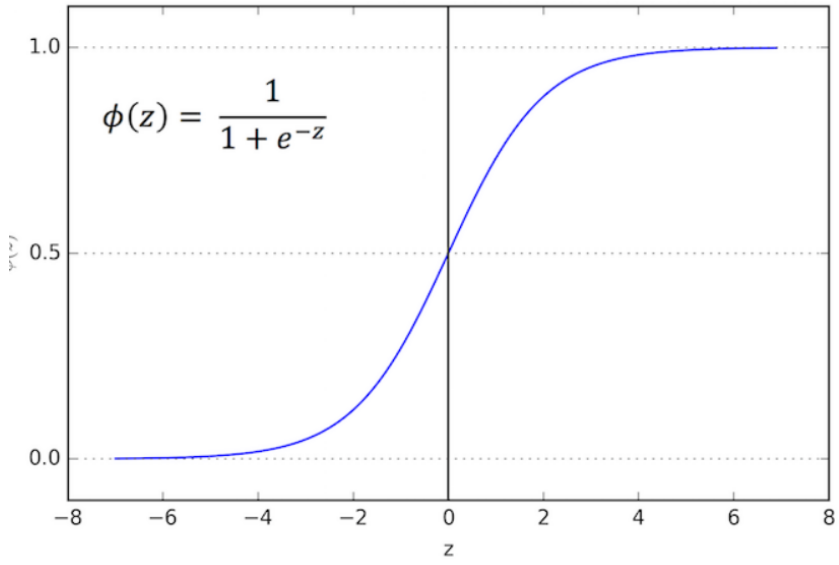


Figure 4

4.2 Cross entropy log loss

The cross-entropy loss function is an optimization function used for training machine learning classification models that classify data by predicting the likelihood (value between 0 and 1) of belonging to one class or another. When the projected probability of class differs significantly from the actual class label (0 or 1), the cross-entropy loss is large.

It is a single value representation of performance of the model, The below cost function is used in this model by gradient decent to optimize the weights and bias.

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

4.3 Gradient Descent

Gradient Descent is used in this project to optimize weights and bias. The gradient of the loss function is computed, and each weight is reduced by the product of the gradient and the learning rate. The learning rate is assumed to be 0.1.

$$\frac{\partial}{\partial w} J(w) = \nabla_w J \qquad w = w - \alpha \nabla_w J$$

$$\frac{\partial}{\partial b} J(w) = \nabla_b J \qquad b = b - \alpha \nabla_b J$$

Implementation

```
[102] #Sigmoid function
      def sigmoid(z):
          return (1/(1+np.exp(-z)))
```

```
[103] #cost function: log loss
      def cost_func(y,z):
          return -sum((y*np.log(z))+((1-y)*np.log(1-z)))/len(y)
```

Logistic Regression before hyperparameter tuning

```
[141] # LR
      loss_stored=[]
      loss_stored_val = []
      learning_rate =0.01
      weights=np.zeros(8)
      cnst=0.1
      for i in range(2000):
          z=np.dot(train_data_norm_X,weights)+cnst
          target_pred=sigmoid(z)
          loss=cost_func(train_data_norm_y,target_pred)
          gradient_weights=np.dot(train_data_norm_X.T,(target_pred-train_data_norm_y))/train_data_norm_X.shape[0]
          gradient_cnst=np.mean(target_pred-train_data_norm_y)
          weights=weights-learning_rate*gradient_weights
          cnst=cnst-learning_rate*gradient_cnst
          loss_stored.append(loss)

          z_val=np.dot(validation_data_norm_X,weights)+cnst
          target_pred_val=sigmoid(z_val)
          loss_val=cost_func(validation_data_norm_y,target_pred_val)
          loss_stored_val.append(loss_val)
      plt.plot(loss_stored, label='training_loss')
      plt.plot(loss_stored_val,label='validation_loss')
      plt.legend()
      plt.show()
      print('Plot to represent Training and Validation loss')
      print(" Training_loss  ",loss_stored[-1],"\n Validation_loss  ",loss_stored_val[-1])
```

the training loss and the validation loss is more than 0.6 so to find the optimal parameters the hyper parameter tuning is done.

5. Hyper Parameters Tuning

In this project batch size and the learning rate are the hyper parameters, trying to find the weights and bias for different batch size and learning rates would help us understand what the optimal hyper parameters are.

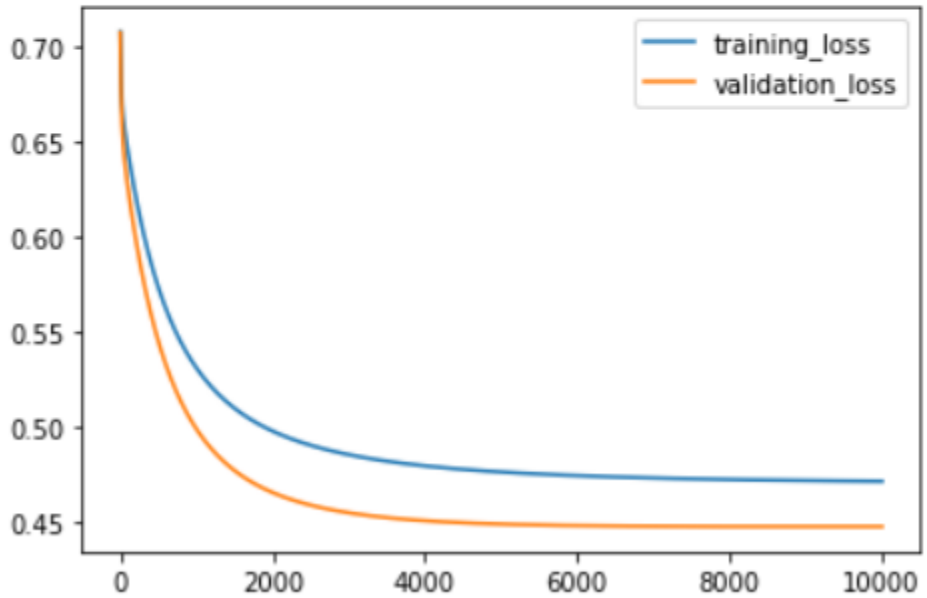
iterations = [3000,10000,15000]
learning rates = [0.1, 0.05, 0.06]

Validation and train loss are calculated with different batch sizes and learning rates, it would give 9 such graphs for the above-mentioned combinations.

Implementation

```
[105] iterations = [3000,10000,15000]
      learning_rates = [0.1, 0.05, 0.06]
      for l in learning_rates:
          for itr in iterations:
              loss_stored = []
              loss_stored_val = []
              Logistic_reg(itr, l)
```

```
iterations:  10000
 learning rate:  0.1
```




```
Training_loss   0.4715777075852962
Validation_loss   0.4477331117210225
```

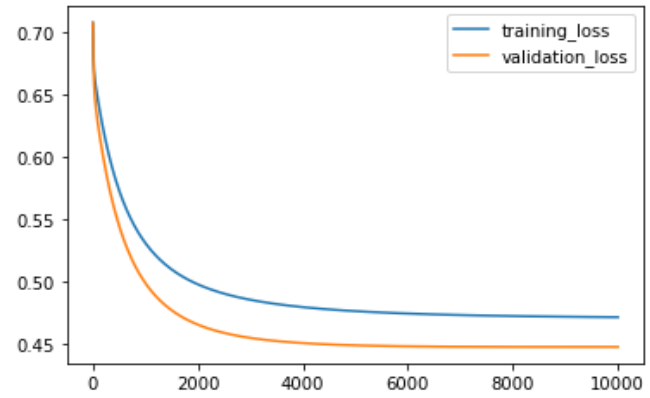
Figure 5

After examining the training and validation loss graphs for various batch sizes and learning rates, batch size =10000 and learning rate =0.1 would result in less loss.

6.Results

6.1 Training Data & Validation Data

Batch size =10000 and learning rate =0.1. Initial weights are a zero matrix, and the bias is 0.1




```
Plot to represent Training and Validation loss
 Training_loss    0.4715777075852963
 Validation_loss    0.44773311172102254
```

Figure 6

6.2 Testing

Batch size =10000 and learning rate =0.1. Initial weights are a zero matrix, and the bias is 0.1
The model gave an accuracy of 80% over the test data, considering any probable value over 0.5 as 1.

Calculating Training, validation accuracy.

```
#Train data accuracy
y_predicted=predict(train_data_norm_X,weights,cnst)
train_accuracy=predicted_accuracy(train_data_norm_y,y_predicted)*100
print("Taining data accuracy: ",train_accuracy)

#Validation data accuracy
y_predicted=predict(validation_data_norm_X,weights,cnst)
validation_accuracy=predicted_accuracy(validation_data_norm_y,y_predicte
print("Validation data accuracy: ",validation_accuracy)
```

```
Taining data accuracy:  77.39130434782608
Validation data accuracy:  77.92207792207793
```

Testing the model with Test data

```
#Test data accuracy
y_predicted=predict(test_data_norm_X,weights,cnst)
test_accuracy=predicted_accuracy(test_data_norm_y,y_predicted)*100
print("Test data accuracy: ",test_accuracy)
```

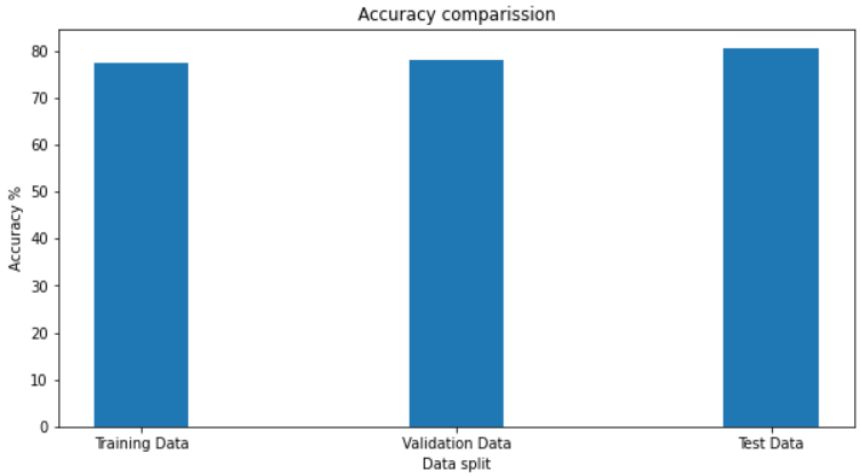
```
Test data accuracy:  80.51948051948052
```

7. Evaluation Metrics

Accuracy is measured by calculating the sum of predicted instances that matches the actual instances(outcomes) and dividing it by number of outcomes.

```
def predicted_accuracy(actual_data_y,pred_data_y):
    prob=np.sum(actual_data_y==pred_data_y)/len(actual_data_y)
    return prob
```

Graph between Training, validation, and test data Accuracy



8. Model Architecture – Neural Networks

8.1 Neural Networks

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning that provide the foundation of deep learning techniques. Their name and form are inspired by the human brain, and they imitate the way real neurons communicate with one another.

Neural Network is a layered architecture, it contains an input layer, optional number of hidden layers and an output layer.

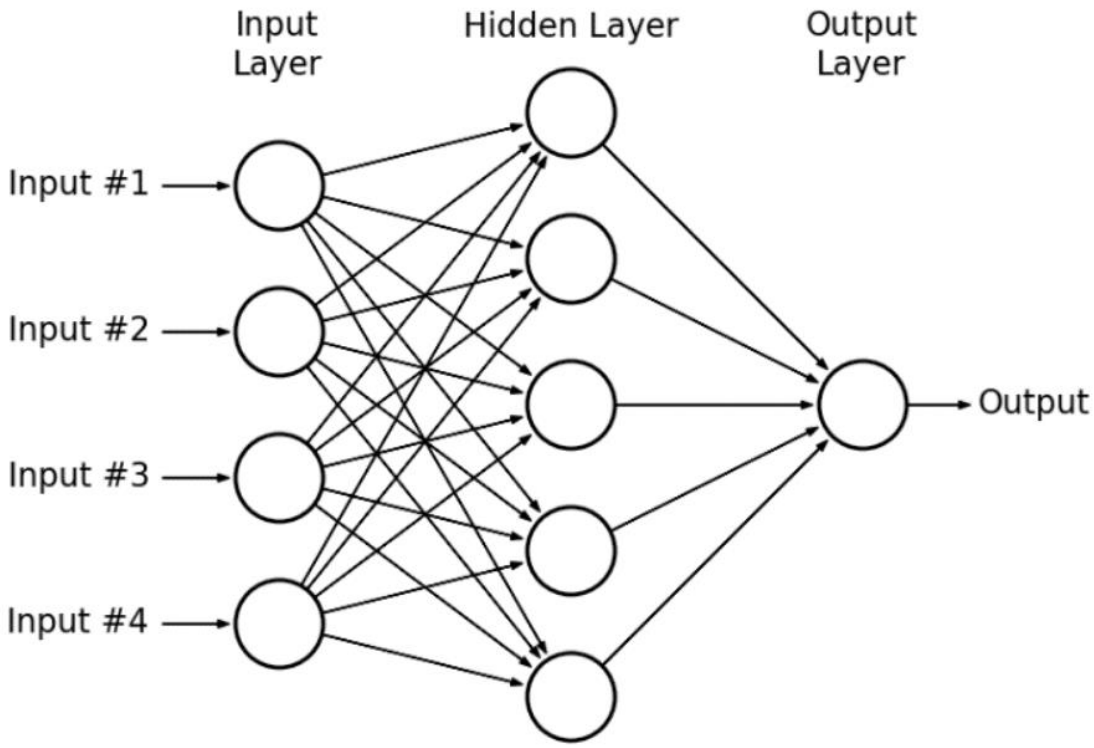


Figure 7

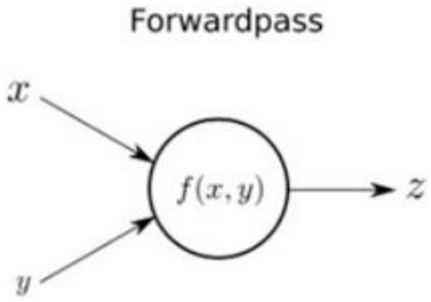
The above figure is a visual representation of the Neural network for binary classification, where n number of neurons map to a single value in the output.

8.2 Computation of Neural networks

Computation of Neural Network is done through forward propagation and computation of gradients is done by backward propagation

8.2.1 Forward Propagation

Forward Propagation is a process of calculation and storage of intermediate values from input to output layers.



$$\boldsymbol{Z = W^T x + b}$$

Z is a weighted sum of input neuros with some bias.

The value of each neuron in the hidden layer is represented as an activation function applied on Z (weighted sum of inputs).

Y= Activation Function(Z)

8.2.2 Activation Functions

Activation functions help in transforming the linear inputs to nonlinear outputs.
There are many activation functions like sigmoid, Relu, Tanh, softmax. Relu has been used in this project.

ReLU (Rectified Linear Unit)

Rectified Linear activation function is a piece wise linear function that will output the input directly if it is positive, otherwise it will output zero (Figure 8). The function solves the problem of vanishing gradients, letting models to train quicker and perform better.

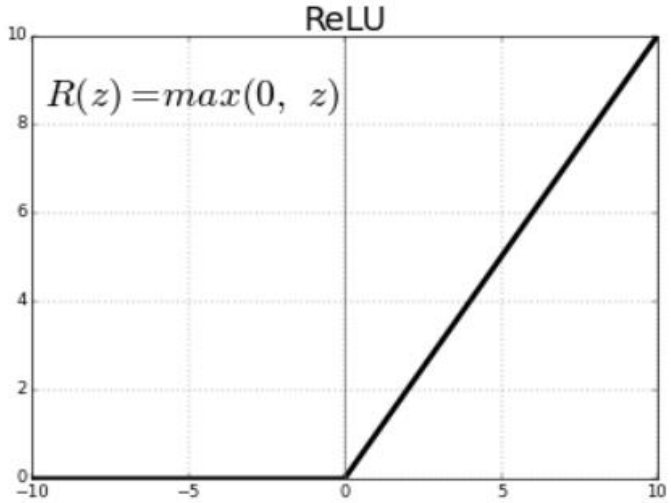
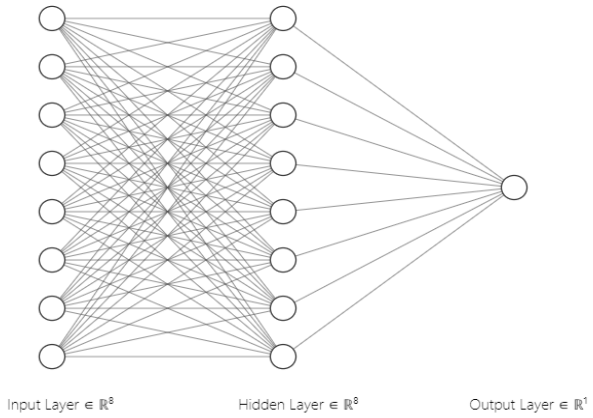


Figure 8

8.2.3 Loss function

As the given project is a binary classification problem (The output values would be either 0 or 1) , Binary Cross Entropy can be used as a loss function.

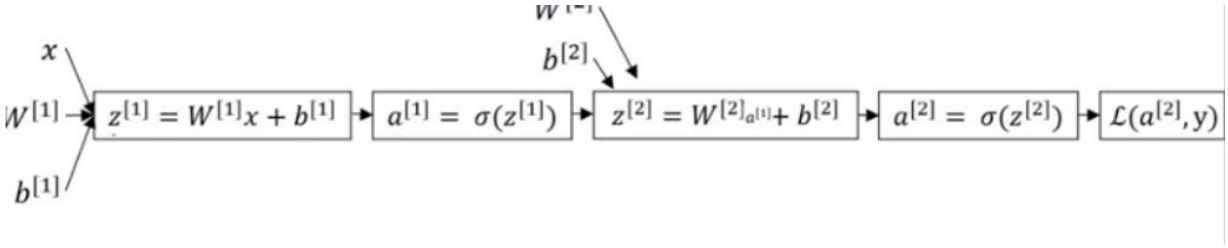
$$BCE = -\frac{1}{N}\sum_{i=0}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$$

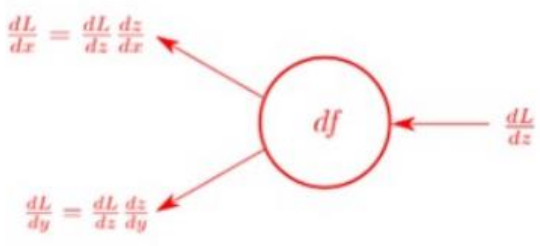


Value of every neuron in the second layer is computed as the weighted sum of the inputs from the input layer.

Back Propagation

Backpropagation is a method used to train specific types of neural networks - it is simply a principle that allows the machine learning software to adapt itself based on its previous function.





8.2.4 Optimizer

Adam

Adam is characterized as "a technique for efficient stochastic optimization using just first-order gradients and requiring little memory."

8.3 Implementation

Neural Networks is implemented using keras and Neural Networks. Keras an open-source library is used for the implementation of this Artificial Neural Network Model.

```
import tensorflow
tensorflow.random.set_seed(0)
model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(15,input_dim=8, activation=tf.nn.relu,kernel_regularizer=keras.regularizers.l1_l2(0.0001,0.0001)),
      tf.keras.layers.Dense(1, activation=tf.nn.sigmoid),])
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy'],
)
```

The Sequential model API is a method of creating learning models in which a Sequential class representation is established, and model layers are built and added to it. The dense layer is a neural network layer that is highly linked, which implies that each neuron in the dense layer receives input from all neurons in the preceding layer.

```
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy'],
)
```

Compilation is the final stage in generating a model before training. While compiling, various parameters such as optimizers, loss, and metrics must be considered.

```
Old_data=model.fit(train_data_norm_X,train_data_norm_y,validation_data=(validation_data_norm_X,validation_data_norm_y),epochs=10,batch_size=100)
tensorflow.random.set_seed(0)
loss_train,accuracy_train = model.evaluate(train_data_norm_X, train_data_norm_y)
loss_valid,accuracy_valid = model.evaluate(validation_data_norm_X,validation_data_norm_y)
loss_test,accuracy_test = model.evaluate(test_data_norm_X,test_data_norm_y)
```

As the parameters are not tuned the accuracy is very low.

Hyper parameter tuning

List of Hyper parameters

1. Number of Neurons
2. Learning Rate
3. Penalty (Lambda)
4. Epochs (No. of iterations)
5. Batch Size

Implementation

```
import tensorflow as tf
from tensorflow import keras
from tensorboard.plugins.hparams import api as hp

HP_LR = hp.HParam('learning_rate', hp.Discrete([0.01,0.001]))
HP_Lambda = hp.HParam('lambda', hp.Discrete([0.01,0.001]))
HP_NUM_UNITS = hp.HParam('num_units', hp.Discrete([6,8,12]))
HP_BATCH = hp.HParam('batch', hp.Discrete([20,32]))
HP_EPOCH = hp.HParam('epochs', hp.Discrete([80,100]))

METRIC_ACCURACY = 'accuracy'

with tf.summary.create_file_writer('logs/hparam_tuning').as_default():
  hp.hparams_config(
    hparams=[HP_NUM_UNITS, HP_LR, HP_BATCH,HP_EPOCH,HP_Lambda],
    metrics=[hp.Metric(METRIC_ACCURACY, display_name='Accuracy')],
  )
```

A Set of hyper parameters are used as shown in the above figure.

```
from tensorflow import keras
tensorflow.random.set_seed(0)
def train_test_model(hparams):
  model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(hparams[HP_NUM_UNITS],input_dim=8, activation=tf.nn.relu,kernel_regularizer=keras.regularizers.l1_l2(hparams[HP_Lambda],hparams[HP_Lambda])),
    tf.keras.layers.Dense(1, activation=tf.nn.sigmoid),
  ])

  model.compile(
      optimizer=keras.optimizers.Adam(learning_rate=hparams[HP_LR]),
      loss='binary_crossentropy',
      metrics=['accuracy'],
  )

  model.fit(train_data_norm_X,train_data_norm_y,validation_data=(validation_data_norm_X,validation_data_norm_y),epochs=hparams[HP_EPOCH],batch_size=hparams[HP_BATCH])
  _,accuracy = model.evaluate(validation_data_norm_X, validation_data_norm_y)
  return accuracy

def run(run_dir, hparams):
  with tf.summary.create_file_writer(run_dir).as_default():
    hp.hparams(hparams)  # record the values used in this trial
    accuracy = train_test_model(hparams)
    tf.summary.scalar(METRIC_ACCURACY, accuracy, step=1)
```

Repetitively calling the above functions for all the hyperparameters

```
    session_num = 0
    for num_units in HP_NUM_UNITS.domain.values:
      for lr in HP_LR.domain.values:
        for batch_size in HP_BATCH.domain.values:
          for epoch in HP_EPOCH.domain.values:
            for llambda in HP_Lambda.domain.values:
              hparams = {
                HP_NUM_UNITS: num_units,
                HP_BATCH: batch_size,
                HP_LR: lr,
                HP_EPOCH: epoch,
                HP_Lambda: llambda
              }
              run_name = "run-%d" % session_num
              print('--- Starting trial: %s' % run_name)
              print({h.name: hparams[h] for h in hparams})
              run('logs/hparam_tuning/' + run_name, hparams)
              session_num += 1
```

Results after hyperparameter tuning

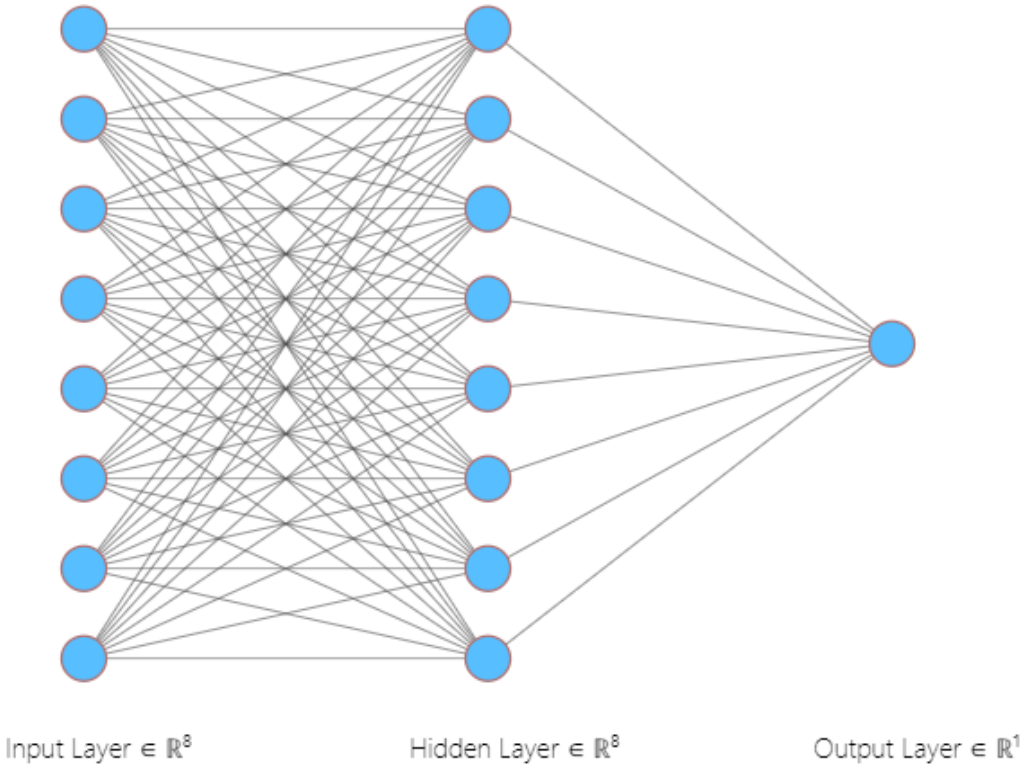
TABLE VIEW | PARALLEL COORDINATES VIEW | SCATTER PLOT MATRIX VIEW

s	num_units	learning_rate	batch	epochs	lambda	Accuracy
	8.0000	0.0010000	32.000	100.00	0.0010000	0.80519
	12.000	0.010000	20.000	100.00	0.010000	0.79221
	6.0000	0.0010000	32.000	100.00	0.0010000	0.79221
	6.0000	0.0010000	20.000	80.000	0.0010000	0.79221
	6.0000	0.0010000	32.000	100.00	0.010000	0.78571
	6.0000	0.0010000	32.000	80.000	0.0010000	0.78571
	12.000	0.010000	20.000	80.000	0.0010000	0.78571
	12.000	0.0010000	32.000	100.00	0.010000	0.78571
	8.0000	0.0010000	20.000	80.000	0.0010000	0.78571
	8.0000	0.0010000	20.000	100.00	0.0010000	0.77922
	6.0000	0.010000	20.000	100.00	0.010000	0.77922
	12.000	0.010000	20.000	100.00	0.0010000	0.77922
	12.000	0.010000	32.000	80.000	0.0010000	0.77922
	8.0000	0.0010000	20.000	80.000	0.010000	0.77922

The first row in the table shows the hyperparameters obtained from tuning.

Hyperparameters obtained from tuning

No. of Neurons	8
Learning Rate	0.001
Batch Size	20
Epochs	100
Lambda	0.001



Neural Networks Implementation with Regularization (L1, L2)

Using the hyperparameters obtained from tuning the Neural network model is being implemented.

```
import tensorflow
tensorflow.random.set_seed(0)
model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(8,input_dim=8, activation=tf.nn.relu,kernel_regularizer=keras.regularizers.l1_l2(0.001,0.001)),
      tf.keras.layers.Dense(1, activation=tf.nn.sigmoid),])
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy'],
)
Old_data=model.fit(train_data_norm_X,train_data_norm_y,validation_data=(validation_data_norm_X,validation_data_norm_y),epochs=100,batch_size=32)
```

Calculating loss and accuracy of the model

```
tensorflow.random.set_seed(0)
loss_train,accuracy_train = model.evaluate(train_data_norm_X, train_data_norm_y)
loss_valid,accuracy_valid = model.evaluate(validation_data_norm_X,validation_data_norm_y)
```

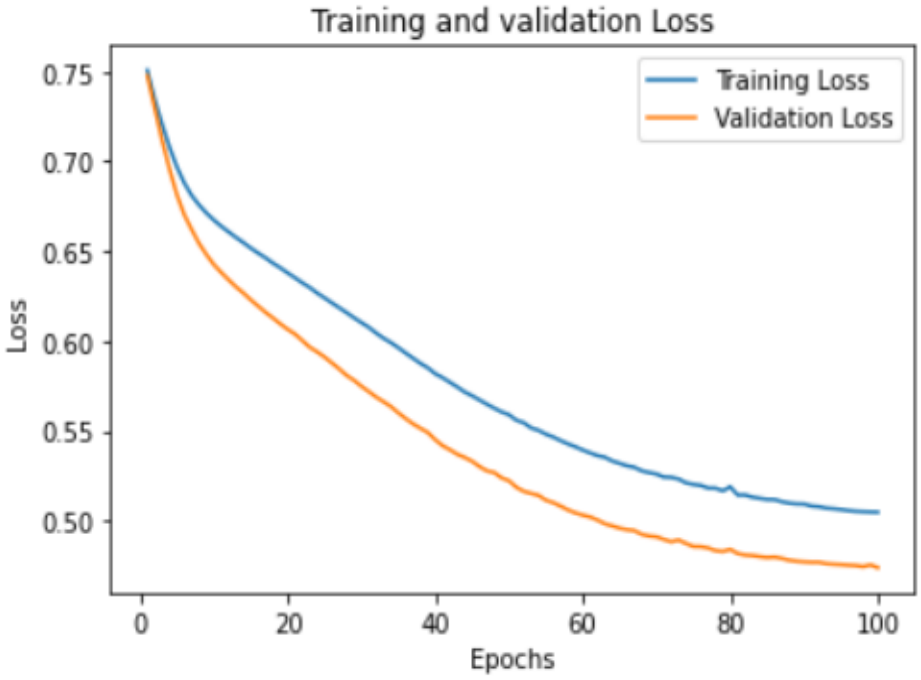
```
15/15 [==============================] - 0s 1ms/step - loss: 0.5035 - accuracy: 0.7739
5/5 [==============================] - 0s 2ms/step - loss: 0.4740 - accuracy: 0.7662
```

Graph to represent Training and Validation loss, Accuracy

```
import matplotlib.pyplot as pt

loss_values = Old_data.history['loss']
val_loss_values = Old_data.history['val_loss']
epochs = range(1, len(loss_values) + 1)

# plot
pt.plot(epochs, loss_values, label='Training Accuracy')
pt.plot(epochs, val_loss_values, label='Validation Accuracy')
pt.title('Training and validation Accuracy')
pt.xlabel('Epochs')
pt.ylabel('Accuracy')
pt.legend()
pt.show()
```

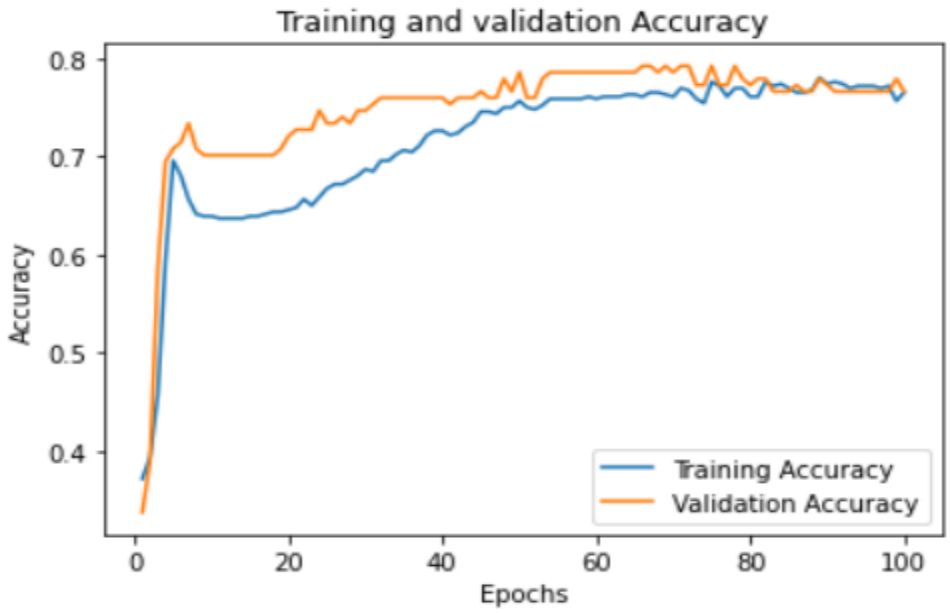


Accuracy Graph (Valdation Accuracy : 76.62%, Training Accuracy : 77.39)

```
import matplotlib.pyplot as pt

acc_values = Old_data.history['accuracy']
val_acc_values = Old_data.history['val_accuracy']
epochs = range(1, len(acc_values) + 1)

# plot
pt.plot(epochs, acc_values, label='Training Accuracy')
pt.plot(epochs, val_acc_values, label='Validation Accuracy')
pt.title('Training and validation Accuracy')
pt.xlabel('Epochs')
pt.ylabel('Accuracy')
pt.legend()
pt.show()
```



Results:

Test Accuracy

```
[177] loss_test,accuracy_test = model.evaluate(test_data_norm_X,test_data_norm_y)

      5/5 [==============================] - 0s 2ms/step - loss: 0.4793 - accuracy: 0.8052
```

By passing the test data set values to the above trained model, it is giving an accuracy of 80.52%.

Bonus Part

Applying Dropout regularization

```
tensorflow.random.set_seed(0)
drop_model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(8,input_dim=8, activation=tf.nn.relu,kernel_regularizer=keras.regularizers.l1_l2(0.001,0.001)),
      tf.keras.layers.Dropout(.2),
      tf.keras.layers.Dense(1, activation=tf.nn.sigmoid),])
drop_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.01),
    loss='binary_crossentropy',
    metrics=['accuracy'],
)

Old_data_drop=drop_model.fit(train_data_norm_X,train_data_norm_y,validation_data=(validation_data_norm_X,validation_data_norm_y),epochs=100,batch_size=32)
```

Accuracy comparison after dropout

```
tensorflow.random.set_seed(0)
drop_model.evaluate(train_data_norm_X, train_data_norm_y)
drop_model.evaluate(validation_data_norm_X,validation_data_norm_y)
```

```
15/15 [==============================] - 0s 1ms/step - loss: 0.4307 - accuracy: 0.7957
5/5 [==============================] - 0s 3ms/step - loss: 0.4250 - accuracy: 0.7597
[0.42498448491096497, 0.7597402334213257]
```

```
drop_model.evaluate(test_data_norm_X,test_data_norm_y)
```

```
5/5 [==============================] - 0s 3ms/step - loss: 0.4542 - accuracy: 0.7727
[0.45417627692222595, 0.7727272510528564]
```

Dropout regularization can be used when the model is overfitting, in this case as the model is not overfitting so there is no positive effect of applying dropout regularization.

Conclusion

We can confidently state that the model is neither overfitting nor underfitting because the validation loss was lower than the training loss. Furthermore, on unseen data from the testing set, the model had an accuracy of 80% in both Logistic Regression and Neural Networks. As a result, if the model is used in the real world, it can be confidently stated that it will perform well.

References:

https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

https://financial-engineering.medium.com/logistic-regression-without-sklearn-107e9ea9a9b6

https://www.analyticssteps.com/blogs/introduction-logistic-regression-sigmoid-function-code-explanation

https://towardsdatascience.com/end-to-end-data-science-example-predicting-diabetes-with-logistic-regression-db9bc88b4d16

https://machinelearningmastery.com/difference-test-validation-datasets/

https://www.kaggle.com/karthik7395/binary-classification-using-neural-networks