BREAST CANCER CLASSIFICATION USING PYTORCH

Introduction

Breast cancer occurs when the normal cells in breast grow in an abnormal way to form a mass of cells called a tumour. There are mainly two types tumors malignant and benign.

Malignant (Cancer) : Cells invade neighbouring tissues, enter blood vessels and spread to different sites. They may not have indications at first, then there will be detection of painless lump.
Benign (Not cancer) : Tumor cells grow only locally and cannot spread by invasion. They usually push the normal tissue to the side.

Problem: Early classification of tumor whether it is malignant or benign is necessary for undertaking a proper treatment to avoid complications. Treatment check up like mammogram or ultrasound is required to do this but it may be expensive or time consuming. So it is desirable to have method which can easily classify the tumor to malignant or benign with some external measurements or features of tumor.

The entire process in classification can be broken down to following steps:

1. Loading Data and Familiarisation
2. Data PreProcessing and Quick Visualisation
3. Feature Scaling
4. Modelling Using Pytorch
5. Diagnosis

The software language used is Python and some major libraries like Numpy, Pandas, Pytorch(Modelling), Plotly(Visualisation), Sklearn.

1. Loading Data And Familiarisation

Dataset is obtained from the UCI Machine Learning Repository. It consist of 357 benign and 212 malignant tumor details.

Attribute Information:

1) ID number
2) Diagnosis (M = malignant, B = benign)

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)

The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features.

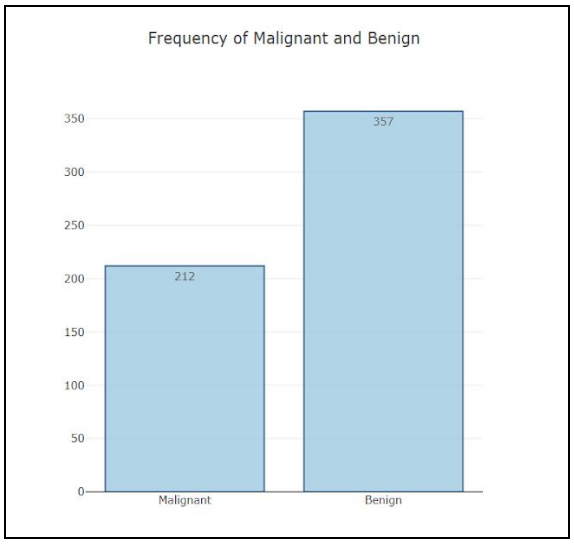
Loading Data

Data can be loaded using pandas **read_csv** function, giving PATH to the downloaded dataset.

```
import pandas as pd
pd.read_csv(PATH)
```

2. Data Preprocessing and Quick Visualisation

Frequency plot of diagnosis (target variable). From the below graph it's clear that the data consist of 212 Malignant and 357 Benign tumor.



Preprocessing

1. Label encoding for diagnosis

Diagnosis columns represented as ‘M’ for malignant and ‘B’ for Benign, so we have to encode this to 0 and 1. We can also use label encoder pre built function.

```
bin_output = {"M": 1, "B": 0}  # modifying output data to binary(0, 1)
self.y_out = self.df["diagnosis"].replace(bin_output)
```

3. Removing outliers

Outliers are data points which are very different from other data in the dataset. The major problem of having outliers are it can skew the result. It has significant effect on mean and standard deviation. So, it’s better remove outliers from data.

```
from scipy import stats
self.df = pd.DataFrame(self.df[(np.abs(stats.zscore(df_temp)) <= 4).all(axis=1)])
```

stats.zscore will calculate z score of each value in the sample relative to the sample mean and the standard deviation. Here we are removing rows having z score value greater than or equal to 4.

```
Number of Outliers: 34
Number of rows without outliers: 535
```

4. Test train split

Splitting data set available to test and train, for the purpose of validation after modelling. We can split the dataset with 33% test data and remaining train data. **DataLoader** is used here which can take in features and can create batches. Code snippet as shown below.

```
import torch.utils.data as data_utils
data_loader = data_utils.DataLoader(train_data, batch_size=128, shuffle=False)
```

3. Feature Scaling

If dataset have features which vary high in magnitudes, units and range the data should be scaled down. Here we have implemented scaling using standard scaling.

The standard score of a sample **x** is calculated as:

z = (x - u) / s

where **u** is the mean of the training samples or zero if with_mean=False, and **s** is the standard deviation of the training

```
from sklearn.preprocessing import StandardScaler
# Scaling train data
scaling_train = StandardScaler()
transformed = scaling_train.fit_transform(self.x_train)
```

The above code snippet shows scaling of the train data similarly we have to do for the testing data.

4. Modelling Using Pytorch

Pytorch is an open source library in python which is used for machine learning and it is based on Torch. It was primarily developed by facebook's machine learning research group and Uber's 'Pyro' software.The modelling process is simple and transparent in pytorch. It is used by Twitter, Salesforce, the University of Oxford and many others.

The entire process of modelling can be breakdown in to following steps:

1. Model creation
2. Running Model
3. Prediction

1. **Model creation**

The model is a sequential with 3 hidden layers. The input layer takes in 30 inputs and outputs 20. Since our problem is a binary classification , the output layer consist of a single neuron.

Linear layers applies a linear transformation to incoming data. We have three linear layers:

a) Input layer (30, 20)
b) Hidden layer - 1 (20, 10)
c) Hidden layer - 2 (10, 1)

Output layer uses sigmoid, hence it will restrict the output between 0 and 1. So, if the output is greater than 0.5 we will classify it to 1 and 0 if it is less than 0.5.

Activation functions are used to introduce non linearity in data. There are different types of activation functions like Linear, Relu, Softmax, Tanh, Sigmoid etc. Here we are using Relu, Linear and Sigmoid.

Finally to initialize the weights in model for a single layer, we can use a function from `torch.nn.init` like xavier_normal.

The below snippet shows the code for model creation.

```
def create(self):
    self.model = torch.nn.Sequential()

    module = torch.nn.Linear(30, 20)
    init.xavier_normal(module.weight)
    self.model.add_module("linear - 1", module)

    module = torch.nn.Linear(20, 10)
    init.xavier_normal(module.weight)
    self.model.add_module("linear - 2", module)

    module = torch.nn.Linear(10, 1)
    init.xavier_normal(module.weight)
    self.model.add_module("linear - 3", module)

    self.model.add_module("rel", torch.nn.ReLU())
    self.model.add_module("sig", torch.nn.Sigmoid())
```

The loss functions are used to train neural network. Here loss function used is **MSEloss,** it creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

Optimization functions are used to update weights which is required to reduce error. Optimizer used here is **Adam** which stands for **Adaptive Moment Estimation.**It computes adaptive learning rates for each parameter.

```
loss_func = torch.nn.MSELoss(size_average=False) # loss function
optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate) # optimizer
```

2. Running Model

For running model following steps are done:

(i) Choose the iteration number, commonly called as epoch.
(ii) Iterate over the data from the data loader by batch.
(iii) Using **Variable** we predict for a specific batch

Variable wraps the tensor. It support almost all the API's provided by the tensor.

```
from torch.autograd import Variable
y_pred = self.model(Variable(batch))
```

(iv) Calculate loss using loss function (MSEloss)

(v) Classify output probability to 0 or 1(malignant or benign).

(vi) Accuracy and loss is calculated for each epoch.

(vii) Back propagation

- Sets gradients of all model parameters to zero. It is to avoid accumulation of gradients on subsequent backward passes.
- Performing back propagation for the gradients.
- Parameter update based on current gradient.

```
optimizer.zero_grad() # Making gradient zero
loss.backward() # Performing back propagation
optimizer.step() # Updating parameter
```

(viii) Repeat above steps for the given epoch.

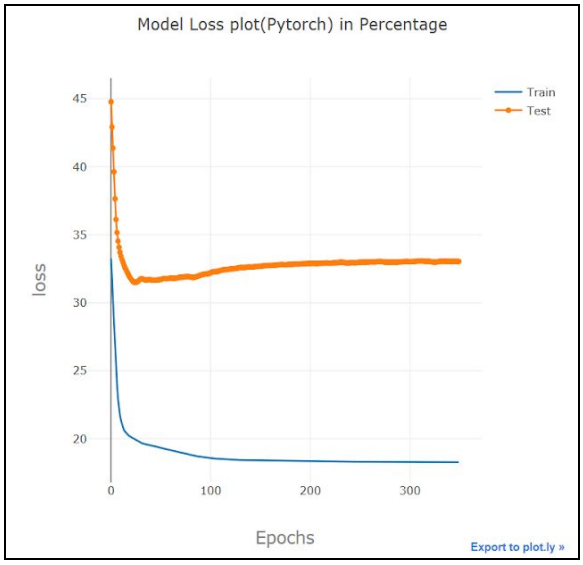
3. **Prediction**

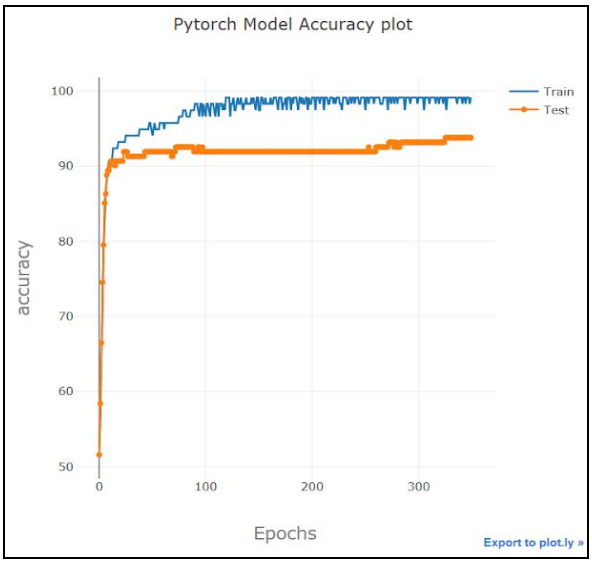
Giving test data to the model we will get the probability output, and then we will classify it to 0 and 1.

```
def predict(self):
        y_pred = self.model(Variable(self.test_data))
```

5. Diagnosis

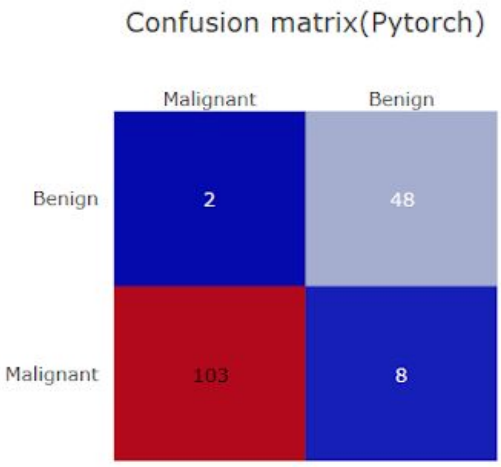
1. **Loss and Accuracy Plot**





From the loss and accuracy it is evident that test data is not performing well as the training data. Even though model have an overall accuracy above 93 percentage.

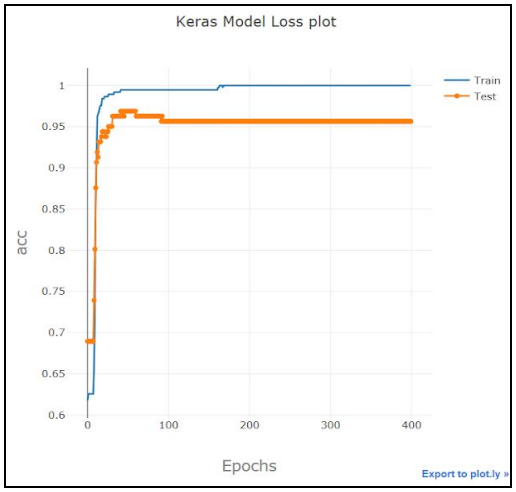
2. **Confusion matrix**

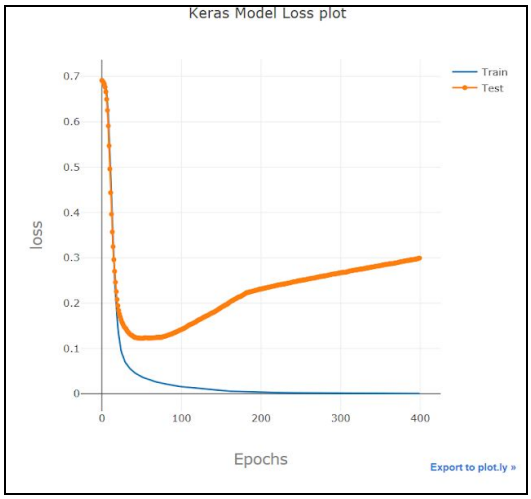


Confusion matrix indicates for a test sample of 161 data points 103 Malignant data was predicted correctly, where as 8 gone wrong also 48 Benign was predicted correctly with 2 wrong ones.

Comparison with Keras Model

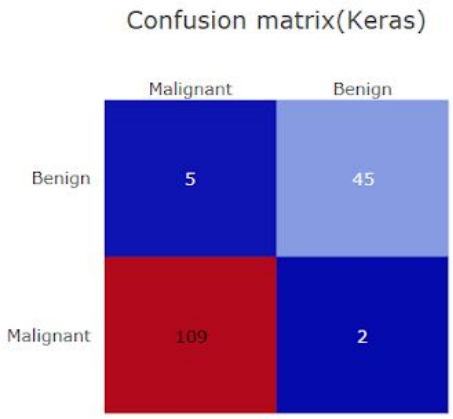
1. **Loss and Accuracy Plot**





From the above plots we can conclude that the keras model also have an overall accuracy above 90 percentage.

2. **Confusion Matrix**



It indicates for a test sample of 161 data points 109 Malignant data was predicted correctly, where as 8 gone wrong also 45 Benign was predicted correctly with 5 wrong ones.

So, both keras and pytorch worked equally good, both having an overall predicting accuracy over 90 percentage. When coming to coding keras has little upper hand because its more mature.

Conclusion

The model was capable of classifying tumor by taking in its features with an accuracy over 90 percentage. So we have a easy method of classifying tumors which can help in early identification and can provide proper treatment.

Reference

The dataset was taken from UCI,
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)