Neural Network Implementation on Medical Appointment No-Show Dataset

COPS Summer of Code 2025

Intelligence Guild

Club of Programmers, IIT (BHU) Varanasi

By

Tanishq Singh

24045164

hello@tanishqsingh.com

Official IG Website: https://cops-iitbhu.github.io/IG-website

Contents

1 Implementation Overview

The project involves building two distinct neural network implementations (given below) to analyze their performance differences on a real-world healthcare dataset.

1. One from scratch using basic `Python` libraries, and
2. Another using `PyTorch` framework.

we compare two neural networks trained on the same dataset, with different performance metrics such as `F1 Score`, `PR AUC`, `Accuracy`, and the `Confusion Matrix`. We also take a closer look at how efficient each approach is, both in terms of prediction quality and computational effort.

2 Data Analysis

We use the `joniarroba/noshowappointments` dataset to analyze the different implementations of neural network mentioned above.

2.1 Import Data

From the kaggle Data Dictionary, there are 14 columns, which are :-

Table 1: Data Properties

Index	Column Name	Description
0	PatientId	Identification of a patient.
1	AppointmentID	Identification of each appointment.
2	Gender	Male or Female.
3	ScheduledDay	The day someone called or registered the appointment, this is before appointment of course.
4	AppointmentDay	The day of the actuall appointment, when they have to visit the doctor.
5	Age	How old is the patient.
6	Neighbourhood	Where the appointment takes place.
7	Scholarship	True or False.
8	Hypertension	True or False.
9	Diabetes	True or False.
10	Alcoholism	True or False.
11	Handicap	True or False.
12	SMS_received	True or False.
13	No-Show	Yes or No.

As from the table we can see than `AppointmentDay` and `ScheduledDay` are date, and there are some spelling mistakes also, which we take care of.

```python
import pandas as pd

# Import Data from csv file.
df = pd.read_csv("medical-no-show.csv",
                 parse_dates=["ScheduledDay", "AppointmentDay"])

# Correct spelling mistakes.
df.rename(columns={"Hipertension": "Hypertension",
                   "Handcap": "Handicap",
                   "No-show": "no_show"}, inplace=True)
```

2.2 Missing Data

As there are only 20 rows where data is missing from more than 1 lakh rows, we drop those rows. As it'll not make a significant difference in our predection.

```python
print(len(df)) # output: 110527
print(df.isna().sum().sum()) # output: 20

df.dropna(inplace=True) # Rows dropped with missing data
```

3 Exploratory Data Analysis

3.1 Creating New Features

3.1.1 Waiting Period

Waiting period is referred here as number of days a patient waited from booking day to appointment day.

```python
df["days_waited"] = (df["AppointmentDay"] - df["ScheduledDay"]).dt.days
```

3.1.2 Appointment Weekday

The weekday on which the appointment is made.

```python
# sunday(0), monday(1), tuesday(2), and so on.
df["appointment_weekday"] = df["AppointmentDay"].dt.weekday
```

3.1.3 Patients missed appointment before

Number of patients who missed appointment before, and if they did then number of times they missed appointment before.

```python
# Dict with key as PatientId and value as no. of missed appointments.
patient_missed_before = df.groupby("PatientId")["no_show"].sum()
patient_missed_before = patient_missed_before.to_dict()

df["missed_appointment_before"] = df["PatientId"].map(lambda x:
                                  patient_missed_before.get(x) > 0)
df["times_missed_before"] = df["PatientId"].map(lambda x:
                                  patient_missed_before.get(x))
```

3.1.4 Track Neighbour-hood with Maximum Missed Appointment

Here we sort neighbourhood based on maximum number of missed appointments and then assign each patients with it's index.

```python
import seaborn as sns
import matplotlib.pyplot as plt

no_shows_df = sorted_neighbourhoods.reset_index()

plt.figure(figsize=(12, 6))
sns.barplot(
    data=no_shows_df,
    x="Neighbourhood",
    y="no_show",
    hue="no_show",
    palette="viridis"
)

plt.xticks(rotation=90)
plt.title("Total No-Shows by Neighbourhood")
plt.xlabel("Neighbourhood")
plt.ylabel("Number of No-Shows")
plt.tight_layout()
plt.show()
```

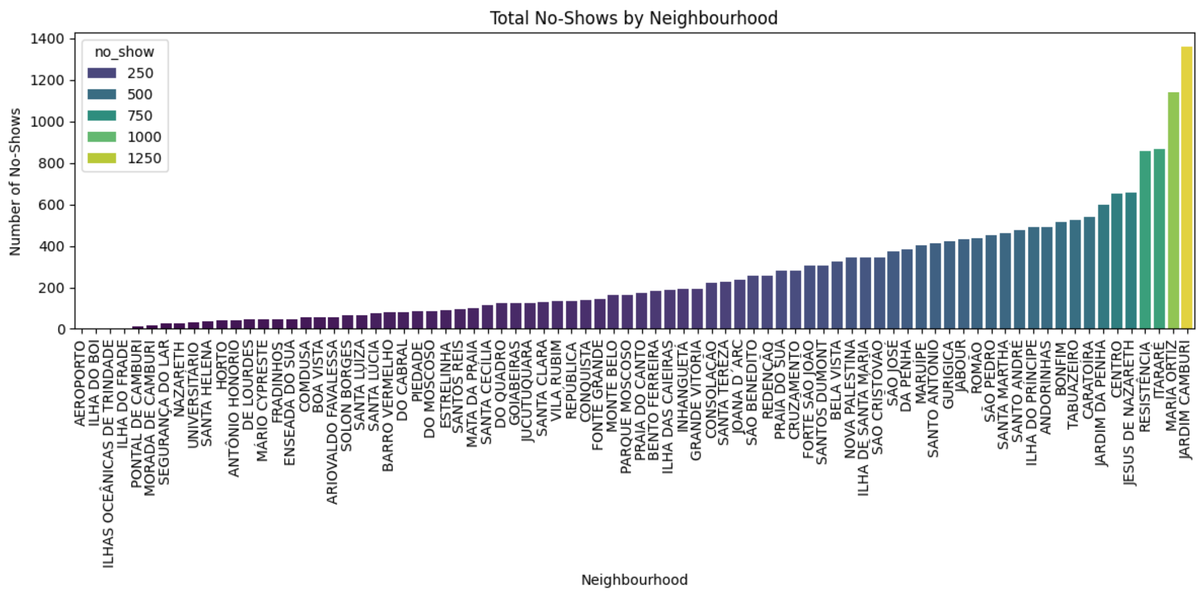


Figure 1: Total No-Shows by each Neighbourhood

```python
sorted_neighbourhoods = df.groupby("Neighbourhood")["no_show"]
                          .sum()
                          .sort_values()

df["neighbourhood_weight"] = df["Neighbourhood"].map(lambda x:
                                     sorted_neighbourhoods.index
                                     .to_list().index(x))
```

3.2 Features Relation

We can plot a simple heatmap of features correlations with `No-Show`, to find out which features are highly correlated with our needs.

```
correlation = df.drop(columns=["PatientId", "AppointmentID",
                               "ScheduledDay", "AppointmentDay",
                               "Neighbourhood"])
                .corrwith(df["no_show"])
                .drop("no_show")

plt.figure(figsize=(10, 6))
sns.barplot(
    x=correlation.values,
    y=correlation.index,
    hue=correlation.index,
    palette="coolwarm"
)

plt.title("Correlation with no_show")
plt.ylabel("Features")
plt.xlabel("Correlation Coefficient")

plt.grid(axis="x", linestyle="--", alpha=0.7)
plt.tight_layout()
plt.show()
```

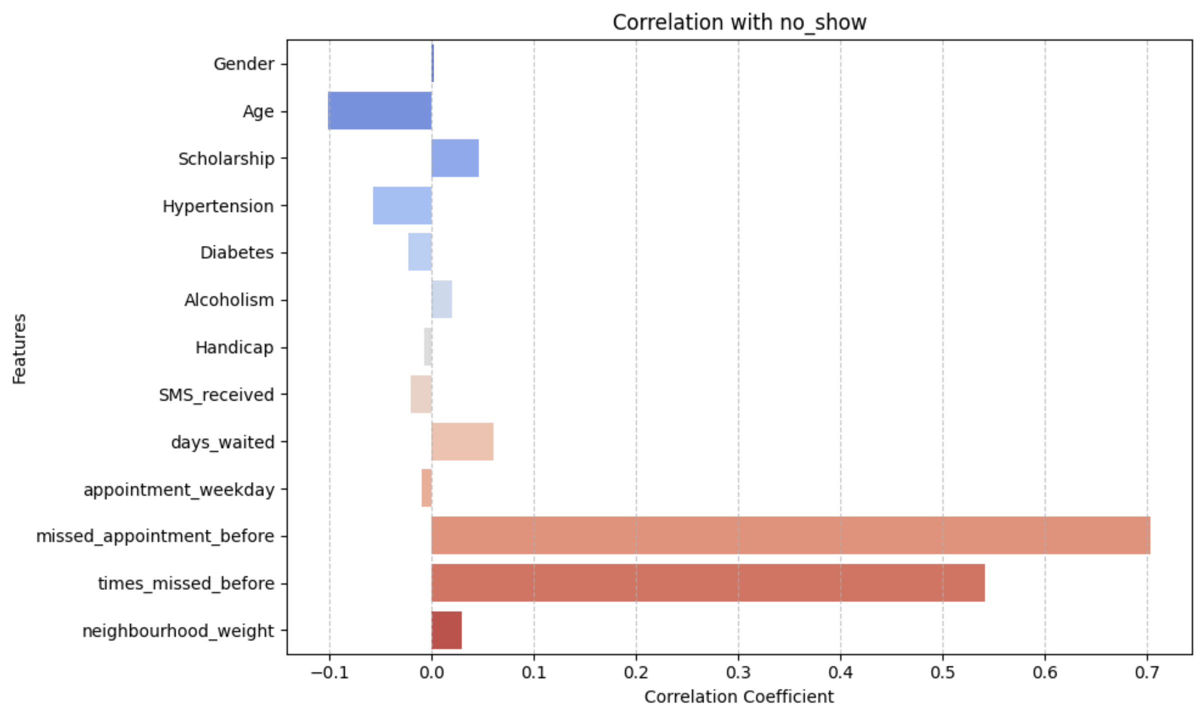


Figure 2: Correlation with No-Show

4 Data Preprocessing

Here we'll process the data to make fit for the model to train, like encoding 'yes' / 'no' to 1 and 0, splitting train, test set, and related things.

4.1 LabelEncoding

Rows like 'no_show', and 'Gender' contains value 'Yes' / 'No', and 'Female', 'Male' respectiely. Which should be encoded to 0 and 1.

```
from sklearn.preprocessing import LabelEncoder

# Label Encode `no_show` to 0's and 1's
le_noshow = LabelEncoder()
df["no_show"] = le_noshow.fit_transform(df["no_show"])

# Label Encode `Gender` to 0's and 1's
le_gender = LabelEncoder()
df["Gender"] = le_gender.fit_transform(df["Gender"])
```

4.2 Removing False Data

There may be some data which are not true like age to be negative, or having appointment date to be before than schedule date. So we'll remove those rows.

```
df = df[df["Age"] >= 0]
df = df[df["days_waited"] >= 0]
```

4.3 Train, Test Split

Splitting all the data we've to `train` (90%) and `test` (10%).

```
from sklearn.model_selection import train_test_split

x = df.drop(columns=["PatientId", "AppointmentID", "ScheduledDay",
                     "AppointmentDay", "Neighbourhood", "no_show"])
y = df["no_show"]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=.1)

# Output: ((64763, 13), (64763,), (7196, 13), (7196,))
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

4.4 Standardizing Data Values

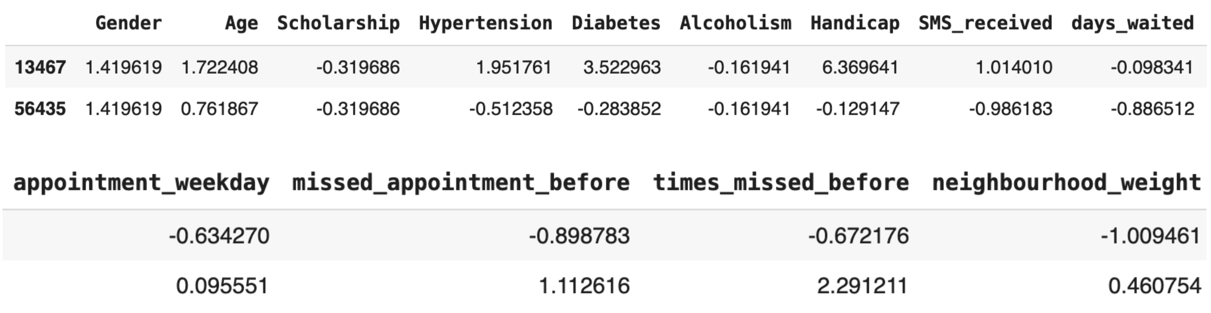
While training a `neural network` we should make sure all our data are standardized before. Because of this required time to reach convergence is reduced.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

4.5 Processed Data

After performing `EDA`, and `pre-processing`, our final data looks somthing like this.

```python
print(pd.DataFrame(x_train, columns=x.columns).sample(2))
```



	Gender	Age	Scholarship	Hypertension	Diabetes	Alcoholism	Handicap	SMS_received	days_waited
13467	1.419619	1.722408	-0.319686	1.951761	3.522963	-0.161941	6.369641	1.014010	-0.098341
56435	1.419619	0.761867	-0.319686	-0.512358	-0.283852	-0.161941	-0.129147	-0.986183	-0.886512

	appointment_weekday	missed_appointment_before	times_missed_before	neighbourhood_weight
13467	-0.634270	-0.898783	-0.672176	-1.009461
56435	0.095551	1.112616	2.291211	0.460754

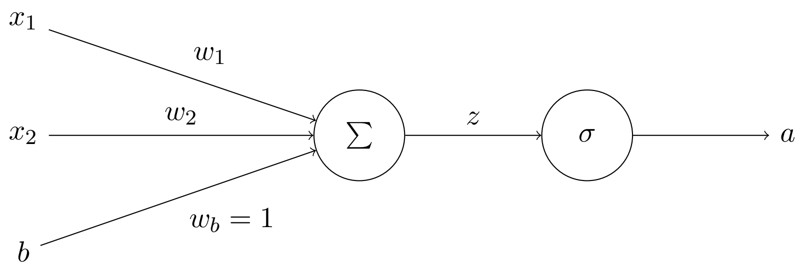
Figure 3: Final Processed Data

5 Model Implementation

5.1 Mathematics Behind Neural Network

5.1.1 What are nodes?

A node takes in inputs, processes them (usually by applying a weighted sum and an activation function), and passes the result to the next layer. It's responsible for learning patterns in data.



$$z = w_1x_1 + w_2x_2 + 1 \cdot b$$
$$a = \sigma(z)$$

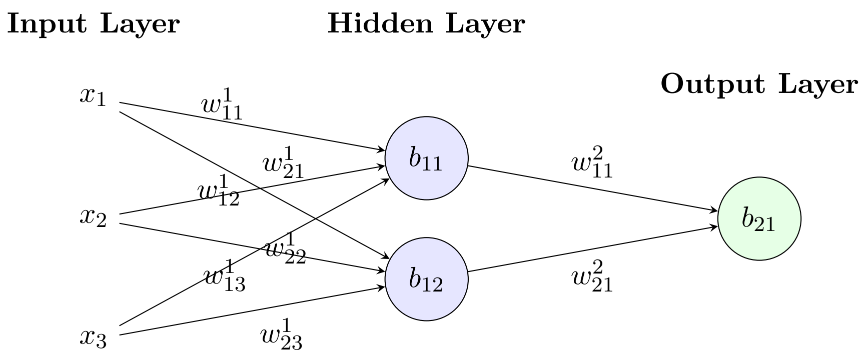
Figure 4: A single ANN node with two inputs, bias, and activation function

There can be n-number of `inputs` (features).

$$z = w_1x_1 + w_2x_2 + ... + W_nx_n + 1 \cdot b$$
$$a = \sigma(z)$$

5.1.2 Artificial Neural Network

An `ANN` is a network of these nodes connected to each other which are organized in layers. An `Artificial Neural Network` is a computational model inspired by the way biological brains process information. It's made up of layers of simple processing units called `neurons` or `nodes`, connected in a network.



$$\begin{aligned} o_{11} &= \sigma(w^1_{11}x_1 + w^1_{21}x_2 + w^1_{31}x_3 + b_{11}) \\ o_{12} &= \sigma(w^1_{12}x_1 + w^1_{22}x_2 + w^1_{32}x_3 + b_{12}) \\ \hat{y} = o_{21} &= \sigma(w^2_{11}o_{11} + w^1_{21}o_{12} + b_{21}) \end{aligned}$$

Figure 5: A ANN three inputs, one hidden, and single output layer

5.1.3 Mathematics of Complex ANN

This contains multiple inputs (features), multiple hidden layers, with different number of nodes in each layer, and there can be single output, or even multiple.

$$\mathbf{A}^{[0]} = \mathbf{X} = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix} \quad \mathbf{A}^{[l]} = \begin{bmatrix} o_{l1} \\ o_{l2} \\ \vdots \\ o_{lm} \end{bmatrix} \quad \mathbf{W}^{[l]} = \begin{bmatrix} w^1_{11} & w^1_{22} & \cdots & w^1_{mq} \\ w^2_{11} & w^2_{22} & \cdots & w^2_{mq} \\ \vdots & \vdots & \vdots & \vdots \\ w^l_{11} & w^l_{22} & \cdots & w^l_{mq} \end{bmatrix} \quad \mathbf{B}^{[l]} = \begin{bmatrix} b_{l1} \\ b_{l2} \\ \vdots \\ b_{lm} \end{bmatrix}$$

Then for each layer $l = 1, 2, \ldots, L$:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$

$$\mathbf{A}^{[l]} = f^{[l]}(\mathbf{Z}^{[l]})$$

$f^{[l]}(Z)$ (activation function) can be different for each layer.

$$\text{ReLU:} \quad f(x) = \max(0, x) \qquad \text{Sigmoid:} \quad \sigma(x) = \frac{1}{1 + e^{-x}} \qquad \text{Tanh:} \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Some of the Activation Functions

5.1.4 Forward Propagation in our Implementation

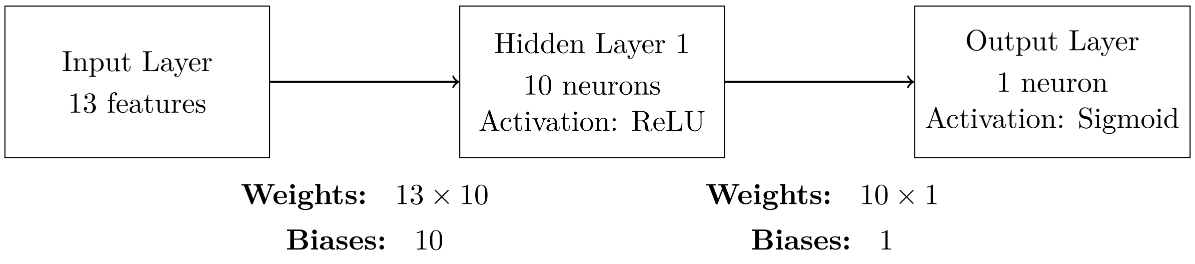


Figure 6: Fully Connected Feedforward Neural Network Diagram

$$A^{[1]} = \text{ReLU}(W_{10\times13}{}^{[1]} X_{13\times1} + B_{10\times1}{}^{[1]})$$
$$\hat{Y} = A^{[2]} = \sigma(W_{1\times10}{}^{[2]} A_{10\times1}{}^{[1]} + B_{1\times1}{}^{[2]})$$

As we are performing binary classification, the `loss function` will be `binary classentropy`

$$L(y, \hat{y}) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

Values of `weights`, and `biases` are found using a method called `BackPropagation`.

5.1.5 Backpropagation: Theory and Detailed Steps

Backpropagation is the key algorithm used to train artificial neural networks. It computes the gradient of the loss function with respect to each weight by the chain rule, enabling gradient descent optimization.

Gradient Descent: is an iterative optimization algorithm. In each iteration, we update the weights and bias using the partial derivatives of the loss function with respect to the weights and bias.

$$w_{new,i} := w_{old,i} - \alpha \cdot \frac{\partial L}{\partial w_i}$$

$$b_{new,i} := b_{old,i} - \alpha \cdot \frac{\partial L}{\partial b_i}$$

Where α is the `learning rate`, and L is `loss function`. It controls the step-size of each update. Repeated application of these updates gradually make the model parameters (i.e weights and bias) converge. Since, the loss function used has only one minima, for this application the convergence will always be at the global minima.

ANN Equations for Our Model

$$A^{[1]} = \text{ReLU}(Z^{[1]}) \quad , \quad Z^{[1]} = W_{10\times13}{}^{[1]} X_{13\times1} + B_{10\times1}{}^{[1]}$$
$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]}) \quad , \quad Z^{[2]} = W_{1\times10}{}^{[2]} A_{10\times1}{}^{[1]} + B_{1\times1}{}^{[2]}$$

Step 1: Compute Gradient for Output Layer $(2^{nd} - layer)$

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$
$$\frac{\partial L}{\partial B^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial B^{[2]}}$$
$$\frac{\partial L}{\partial \hat{y}} = (\frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}}) \quad , \quad \frac{\partial \hat{y}}{\partial Z^{[2]}} = \hat{y}\cdot(1-\hat{y}) \quad , \quad \frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]} \quad , \quad \frac{\partial Z^{[2]}}{\partial B^{[2]}} = 1$$

Step 2: Compute Gradient for Hidden Layer $(1^{st} - layer)$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}}$$
$$\frac{\partial L}{\partial B^{[1]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial B^{[1]}}$$
$$\frac{\partial Z^{[2]}}{\partial B^{[1]}} = \begin{cases} W^{[2]} & \text{if } Z^{[1]} > 0 \\ 0 & \text{otherwise} \end{cases} \quad , \quad \frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot \frac{\partial A^{[1]}}{\partial W^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$
$$\frac{\partial A^{[1]}}{\partial W^{[1]}} = \begin{cases} 1 & \text{if } Z^{[1]} > 0 \\ 0 & \text{otherwise} \end{cases} \quad , \quad \frac{\partial Z^{[1]}}{\partial W^{[1]}} = X_{[i]}$$

Step 3: Updates the `weights`, and `biases`

$$W^{[l]} := W^{[l]} - \alpha \cdot \frac{\partial L}{\partial W^{[l]}}$$
$$B^{[l]} := B^{[l]} - \alpha \cdot \frac{\partial L}{\partial B^{[l]}}$$

Then for each layer $l = 1, 2$.
Where α is learning rate.

5.2 Neural Network Implementation from Scratch

5.2.1 Assumptions In Our Neural Network

We'll implement `Stochastic Gradient Descent` which updates parameters after each row, because of this in `PyTorch NN` we'll not use any `DataLoaders` as then the performance will differ.

5.2.2 Defining Layer Class

This `Layer` class defines a single neural network layer with support for **ReLU** and **Sigmoid** activations. It stores weights, biases, pre-activation (`z`), and activated outputs (`a`), and includes methods for the forward pass and activation function derivatives which are essential for backpropagation.

```python
class Layer:
    def __init__(self, activation, units):
        self.activation = activation
        self.units = units # Number of nodes

        self.z = None # Pre-Activation
        self.a = None # Activated

        self.weights = None
        self.bias = None

    def activation_function(self, xi):
        if self.activation == "relu":
            return np.maximum(0, xi)

        elif self.activation == "sigmoid":
            return 1 / (1 + np.exp(-xi))

    def activation_derivative(self, xi):
        if self.activation == "relu":
            return (xi > 0).astype(float)

        elif self.activation == "sigmoid":
            sig = self.activation_function(xi)
            return sig * (1 - sig)

    def forward(self, xi):
        self.z = np.dot(self.weights, xi) + self.bias
        self.a = self.activation_function(self.z)

        return self.a
```

5.2.3 Initializing Layer Class

Were are defining two layers:

1. One hidden with `ten nodes` and `relu` as activation function, and
2. Another one output layer with `one node` and `sigmoid` as activation function

```python
layer_1 = Layer(units=10, activation="relu")   # Hidden Layer
layer_2 = Layer(units=1, activation="sigmoid") # Output Layer
```

5.2.4 Defining Model Class

The `Model` class defines a simple feedforward neural network framework with customizable layers. It includes support for weight initialization `(He/Xavier)`, forward propagation, binary cross-entropy loss, and backpropagation with gradient descent. It also provides methods to compile, train (fit), and predict using the model.

```python
class Model:
    def __init__(self, layers, input_features):
        self.layers = layers # List of layers in the model
        self.input_features = input_features # Number of inputs

        self.compiled = False
        self.metrics = False # To show metrics after each epochs

        self.learning_rate = 0.01 # Learning rate of the model

    """
        We'll define these functions later
    """
    def compile(self, metrics=False): pass

    def summary(self): pass

    def predict(self, x_test): pass

    def loss_calculate(self, yi_true, yi_pred): pass

    def fit(self,x_train,y_train,val_data,epochs=1,learning_rate=0.01):
        pass
```

Summary Function Initialize

This function tells about the units, activation, trainable parameters in each layer, and also the total trainable parameters in the model.

```python
def summary(self):
    total_trainable_parameters = 0

    for layer_i in range(len(self.layers)):
        layer = self.layers[layer_i]
        layer_trainable_parameters = self.input_features * layer.units
        total_trainable_parameters += layer_trainable_parameters

        print(" " * 16, f"Layer: {layer_i + 1}")
        print(f"Units: {layer.units}")
        print(f"Activation: {layer.activation}")
        print(f"Trainable Parameters: {layer_trainable_parameters}")
        print("=" * 40)

    print(f"Total Trainable Parameters: {total_trainable_parameters}")
    print("=" * 40, "\n")
```

Loss Function Initialize

Binary cross-entropy loss measures the difference between predicted probabilities and actual binary labels. It penalizes incorrect predictions by calculating the -ve log-likelihood of the true class, encouraging the model to output probabilities close to 0 or 1.

```python
def loss_calculate(self, yi_true, yi_pred):
    yi_pred = yi_pred + (1e-8) # Handling 0 case

    return -1 * np.mean(
        (yi_true * np.log(yi_pred)) +
        ((1 - yi_true) * np.log(1 - yi_pred))
    )
```

Weights, and Biases Initialize

1. Weights are initialized differently based on the activation function to maintain stable gradients and avoid problems like `vanishing or exploding gradients` during training.
2. For layers with `ReLU activation`, weights are initialized using `He initialization`, which scales the weights by $\sqrt{\frac{2}{\text{fan\_in}}}$ to keep variance consistent through the network.
3. For `sigmoid activations, Xavier (Glorot) initialization` is used, scaling weights by $\sqrt{\frac{1}{\text{fan\_in}}}$ to help activations avoid saturation and maintain gradient flow.
4. Bias terms for all layers are initialized as `zero vectors` to provide a baseline offset without introducing bias in the initial output.
5. Once weights and biases are set, the network is marked as compiled and ready for training, with an option to track additional metrics during the process.

```python
def compile(self, metrics=False):
    for layer_i in range(len(self.layers)):
        layer = self.layers[layer_i]

        fan_in = self.input_features if layer_i == 0 else self.layers[
                                                layer_i - 1].units

        # He initialization
        if layer.activation == "relu":
            layer.weights = np.random.randn(layer.units, fan_in) *
                                           np.sqrt(2. / fan_in)

        # Xavier initialization
        elif layer.activation == "sigmoid":
            layer.weights = np.random.randn(layer.units, fan_in) *
                                           np.sqrt(1. / fan_in)

        # Default small random values
        else:
            layer.weights = np.random.randn(layer.units, fan_in) * 0.01


        # Biases initialized to zero
        layer.bias = np.zeros((layer.units, 1))

    self.compiled = True
    self.metrics = metrics
```

Fit Function Initialize

The `fit` function in an ANN trains the model by iteratively updating its weights and biases using different inputs. It performs forward propagation, calculates loss, backpropagates errors, and optimizes parameters to minimize the loss over multiple epochs.

```python
def fit(self, x_train, y_train, val_data, epochs=1,learning_rate=0.01):
    self.learning_rate = learning_rate
    history_core = []
    counter_core = 0
    time_start = time.time()

    for epoch in range(epochs):
        for xi, yi in zip(x_train, y_train):
            if counter_core % 200 == 0:
                cost = self.loss_calculate(val_data[1],
                                           self.predict(val_data[0]))
                history_core.append([counter_core / 200, cost,
                                     time.time()-time_start])
            counter_core = counter_core + 1

            yi_pred = (self.predict([xi]))[0]
            loss = self.loss_calculate(yi, yi_pred)

            z1, a1 = self.layers[0].z, self.layers[0].a
            yi_pred = yi_pred + (1e-8) # Handling 0 case

            # Calculating derivatives (discussed in 5.1.5)
            dl_dy_hat = -1 * (yi/yi_pred - (1 - yi)/(1 - yi_pred))
            dy_hat_dz2 = yi_pred * (1 - yi_pred)
            dz2_dw2 = a1.T
            dz2_db2 = 1
            dz2_dw1 = self.layers[1].weights.T * xi *
                      self.layers[0].activation_derivative(z1)
            dz2_db1 = self.layers[0].activation_derivative(z1)

            """
            Calculating partial derivatives of loss wrt weight/bias
            Layer 1, and 2
            """
            dl_dw1 = np.array(dl_dy_hat * dy_hat_dz2 * dz2_dw1)
            dl_db1 = np.array(dl_dy_hat * dy_hat_dz2 * dz2_db1)
            dl_dw2 = np.array(dl_dy_hat * dy_hat_dz2 * dz2_dw2)
            dl_db2 = np.array(dl_dy_hat * dy_hat_dz2 * dz2_db2)

            # Updating weights and biases
            self.layers[1].weights -= self.learning_rate * dl_dw2
            self.layers[1].bias -= self.learning_rate * dl_db2
            self.layers[0].weights -= self.learning_rate * dl_dw1
            self.layers[0].bias -= self.learning_rate * dl_db1

        metrics_core = metrics(val_data[1], self.predict(val_data[0]))

        if self.metrics: # Print metrics for each epoch
            print(f"Epoch: {epoch+1}, metrics: {metrics_core}")

    return (metrics_core, history_core)
```

Predict Function Initialize

The `pretict` function in an ANN models predicts the output for any given input, by passing the input through all the layera in the neural network called `feed forwawrd`.

```python
def predict(self, x_test):
    if self.compiled:
        y_pred = []

        for xi in x_test: # different cases
            for i in range(len(self.layers)):
                xi = self.layers[i].forward(xi.reshape(-1, 1))

            y_pred.append(xi[0][0])
        return np.array(y_pred)
    else: print("Model is not yet compiled.")
```

5.2.5 Initializing Model Class

Define custom model with layers, and input features.

```python
custom_model = Model(layers=[layer_1, layer_2],
                     input_features=x_train.shape[1])

custom_model.summary() # show trainable parameters, layers, nodes, etc
custom_model.compile(metrics=True) # show metrics after each epoch
```

5.2.6 Defining Metrics Function

```python
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
from sklearn.metrics import precision_recall_curve, auc

def metrics(y_true, y_hat):
    # Convert y_hat (probability) to 0(<0.5) to 1(>0.5)
    y_pred = np.where(y_hat > .5, 1, 0)

    accuracy = accuracy_score(y_true, y_pred) # Accuracy Score
    f1 = f1_score(y_true, y_pred)             # F1 Score
    cm = confusion_matrix(y_true, y_pred)     # Confusion Matrix

    # Precision-Recall AUC
    precision, recall, _ = precision_recall_curve(y_true, y_hat)
    pr_auc = auc(recall, precision)

    return {
        "accuracy": accuracy, "pr_auc": pr_auc,
        "f1_score": f1, "confusion_matrix": cm
    }
```

5.2.7 Training Custom ANN

```python
metrics_core, history_core = custom_model.fit(x_train, y_train,
                                              val_data=(x_test, y_test))
```

5.3 Neural Network Implementation using PyTorch

5.3.1 Assumptions

Already covered (please refer 5.2.1).

5.3.2 Importing Libraries and Converting Numpy to tensor

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score

# Convert NumPy arrays to PyTorch tensors
x_train_tensor = torch.tensor(x_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.to_numpy(), dtype=torch.float32)
                                    .view(-1, 1)

x_test_tensor = torch.tensor(x_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.to_numpy(), dtype=torch.float32)
                                    .view(-1, 1)
```

5.3.3 Defining Model and Layers

Here we are making the exact same model we made using core python, this consist of sequential layers:

1. One hidden with `ten nodes` and `relu` as activation function, and
2. Another one output layer with `one node` and `sigmoid` as activation function

```python
class BinaryClassifier(nn.Module):
    def __init__(self, input_dim):
        super(BinaryClassifier, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(input_dim, 10), # 10 nodes in first layer
            nn.ReLU(),

            nn.Linear(10, 1), # one node in last (output) layer
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)
```

5.3.4 Initializing Model

```python
# Initialize model
model = BinaryClassifier(input_dim=x_train.shape[1])

criterion = nn.BCELoss() # Binary Cross Entropy as Loss Function
optimizer = optim.SGD(model.parameters(), lr=0.01) # Gradient Descent
```

5.3.5 Training PyTorch Model

```python
history_torch = []
counter_torch = 0
time_start = time.time()

# Training loop
for epoch in range(1): # Single Epochs
    model.train()
    for xi, yi in zip(x_train_tensor, y_train_tensor):
        if counter_torch % 200 == 0:
            cost = criterion(model(x_test_tensor), y_test_tensor).item
                                                         ()
            history_torch.append([counter_torch / 200, cost,
                                  time.time()-time_start])
        counter_torch = counter_torch + 1

        optimizer.zero_grad() # Making gradient zero

        loss = criterion(model(xi), yi)
        loss.backward() # Calculate gradient

        optimizer.step()
```

5.3.6 Model Evaluation

```python
model.eval()
with torch.no_grad():
    y_preds = model(x_test_tensor).detach().numpy()

metrics_pytorch = metrics(y_test, y_preds)
```

6 Evaluation and Analysis

6.1 Convergence Time

6.1.1 Convergence of PyTorch is faster than core?

The time it takes for the model to learn enough to predict well is called **convergence time**. After this time the loss reaches minimum, and the graph flattens.

The `PyTorch` model converged slightly faster than the `core Python` implementation, both in terms of iterations and training time. This difference wasn't very large (check figure 7 mentioned below), some of the possible reasons can be:

1. I didnt́ use `DataLoader` in `PyTorch`.
2. Updated `weight/biases` after each row, that means in each epoch parameters are updated more than `60k times`.
3. Both models were trained on `CPU`.

6.1.2 Comparision of training time and cost

```python
sns.set(style="whitegrid") # style to show grid

# convert to numpy array
history_core = np.array(history_core)
history_torch = np.array(history_torch)

# set figure size
plt.figure(figsize=(10, 4))

# Red Line for Core Implementation
sns.lineplot(x=history_core[:117, 2], y=history_core[:117, 1],
             color="red", label="Core Implementation")

# Blue Line for PyTorch Implementation
sns.lineplot(x=history_torch[:100, 2], y=history_torch[:100, 1],
             color="blue", label="PyTorch Implementation")

plt.xlabel("Time (seconds)")
plt.ylabel("Cost")
plt.title("Time(second) v/s Cost: for PyTorch and Core Implementation")

plt.tight_layout()
plt.show()
```

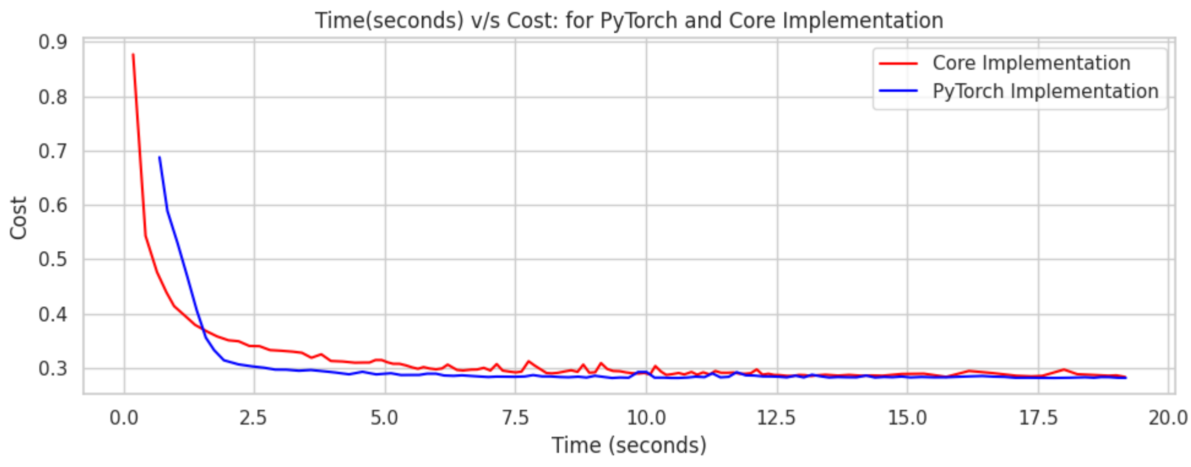


Figure 7: Time (seconds) v/s Loss: for PyTorch and Core Implementation

Conclusion:

1. Here you can see that the convergence time for `PyTorch` was comparatively less than `Core Implementation`.
2. The cost for `PyTorch` got more and more stable with time,
3. But with the other one i.e `Core Implementation` it was not the case, we can still some sharp pitches in the graph.

6.2 Performance Metrics

We compare the two models built above using the following metrics:

6.2.1 Accuracy

The ratio of correct predictions out of all the predictions is called `accuracy` for the given model. Higher the accuracy more better out predictions.

$$Accuracy = \frac{\text{total correct predictions}}{\text{total predictions}}$$

Implementation	**Accuracy**
PyTorch Implementations	83.94%
Core Implementations	83.86%

Table 2: Accuracy comparison between `Core` and `PyTorch Implementations`

here you can see, the difference in accuracy is not quite large between both the implementations. Both the models are predicting around $\approx 84\% correctly$.

6.2.2 F1 Score

The harmonic mean of the precision and recall is called `F1 Score` for the given model. Where as `precision` if the ratio of true predicted positives by total predicted positives, and `recall` is the ratio of true predicted positives by total true positives.

Higher the `F1 Score` means both `precision`, and `recall` is high, which identify that our model is correctly classifying right positives and not misclassifying too many negatives as positives.

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Precision} = \frac{\text{true predicted positives}}{\text{total predicted positives}} \quad , \quad \text{Recall} = \frac{\text{true predicted positives}}{\text{total positives}}$$

Implementation	**F1 Score**
PyTorch Implementations	76.71%
Core Implementations	75.65%

Table 3: F1-Score comparison between `Core` and `PyTorch Implementations`

The difference in the `F1 Score` is nearly $\approx 1\%$, which signifies that `PyTorch` model is more nicely predicting compared to `Core Implementation` model.

6.2.3 Precision-Recall Area Under Curve (PR AUC)

It measures the model's ability to distinguish between categories, especially when dealing with imbalanced datasets (e.g., 90% negatives, 10% positives).

Higher the `PR AUC` means higher the ability of the model to correctly classify between positives and negatives.

Implementation	**PR AUC**
PyTorch Implementations	73.75%
Core Implementations	73.51%

Table 4: PR-AUC comparison between `Core` and `PyTorch Implementations`

Although the margin is not quite large, but `PyTorch` model is classifying more correctly positives and negatives as that compared to `Core Implementation` model.

6.3 Memory Usage

1. The `PyTorch` implementation consumed more memory than the `Core Implementation` model
2. As `PyTorch` includes additional overhead for managing tensors, autograd (automatic differentiation), and internal buffers.
3. On the other hand, the `Core Implementation` model only stores the bare minimum: weights, inputs, and gradients manually.

6.4 Confusion Matrix and Inference

A `Confusion Matrix` is a performance measurement tool for classification models. It tells how well a model is performing by showing the correct and incorrect predictions made by the model compared to the actual outcomes.

Actual / Predicted	**Positive**	**Negative**
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Table 5: Typical Confusion Matrix for Binary Classification

With the help of this, we can easily calculate other performance metrics such as:

1. Accuracy
2. F1 Score
3. Precision-Recall Area Under Curve

```python
# Plot 2 graphs on same axis, with required size
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Confusion Matrix for both implementation
cm_core = metrics_core["confusion_matrix"]
cm_torch = metrics_pytorch["confusion_matrix"]

# Plotting graph for Core Implementation
sns.heatmap(cm_core, annot=True, fmt="d", cmap="Reds", ax=axs[0])
axs[0].set_title("Confusion Matrix: Core Implementation")
axs[0].set_xlabel("Predicted")
axs[0].set_ylabel("Actual")

# Plotting graph for PyTorch Implementation
sns.heatmap(cm_torch, annot=True, fmt="d", cmap="Blues", ax=axs[1])
axs[1].set_title("Confusion Matrix: PyTorch")
axs[1].set_xlabel("Predicted")
axs[1].set_ylabel("Actual")

plt.tight_layout()
plt.show()
```

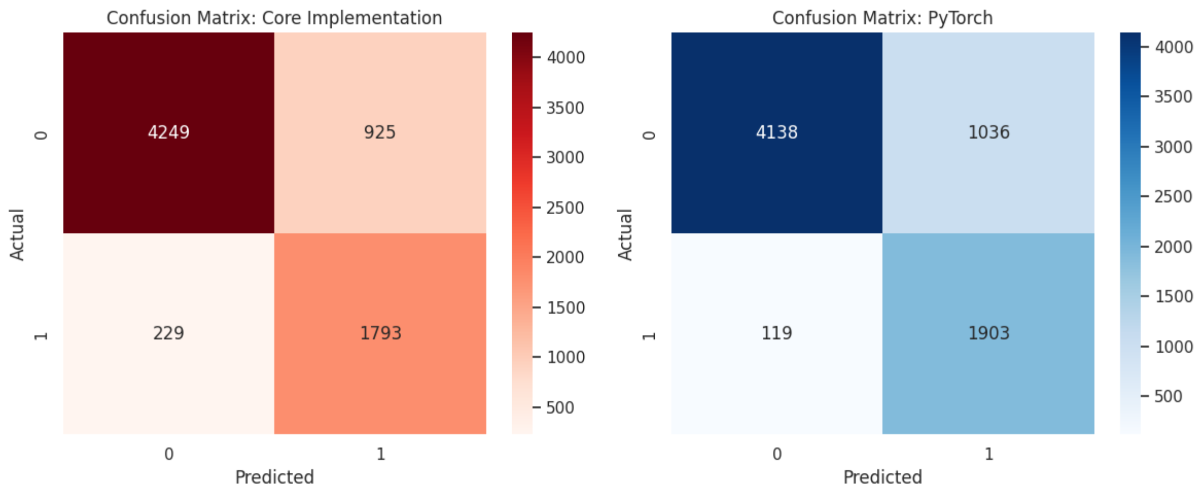


Figure 8: Confusion Matrix: for Both Implementation

Conclusions

1. Overall accuracy is almost same in both implementations.
2. `PyTorch` model is better at identifying true positives compared to `Core Implementation` model.
3. `Core Implementation` model is better at identifying true negatives compared to `PyTorch` model.
4. `PyTorch` model makes more mistake identifying false positives, and `Core Implementation` model makes more mistakes identifying false negatives compared to each other.
5. If you want higher true positives then go with `PyTorch`, else `Core Implementation` model will be better for higher true negatives.

6.5 Analysis and Discussion

6.5.1 Convergence Speed, Performance, and Memory Usage

Discussion for these topics have been already done above in their respective domains:

1. Convergence Speed 6.1
2. Performance Metrics 6.2
3. Memory Usage 6.3

6.5.2 Some Other Factors

Optimizations

1. You can use `DataLoaders` in `PyTorch` model which significantly improves the performance.
2. Instead of updating `weights/biases` after each rows which is like if a data set contains `60 k` rows, then in two epochs parameters are updates nearly `120 k` times. Instead this you can use batch gradient descent.
3. Change in `learning rate` can help reaching convergence quickly.

Hardware Acceleration

1. Here we used `CPU` in both the model which is slow for these type of tasks, In `PyTorch` model we could upgrade the tensors to use `GPU` which are significantly faster.
2. Distributing load to multiple `GPU`, or if possible multiples machines can reduce training time.

Numerical Stability

Standardizing all the values in the dataset, can help reduce the training time and help reach convergence fast. Example in a dataset containing fields `IQ`, `CPI` and `salary`, predicting salary can be not efficient as it can be in lakhs and other features will be in hundreds.

Code Efficiency

1. Instead of writing all code in single go, better approach will be to define `function` of some general tasks.
2. If some functions share related task, then define `class` for it.
3. Efficiently writing loops, Correctly naming `variables/functions/classes`, and more.

Thank You