Neural Network-Based Prediction of Medical Appointment No-Shows: A Comparative Study of NumPy and PyTorch Implementations

Tanishq Ahuja (24074033)

June 1, 2025

Abstract

This report presents a comparative study of **Simple Neural Network** implementations for **binary classification** task using **backpropagation** to predict medical appointment no-show rate. A real-world dataset is used and values are predicted based on multiple features. We explore **two approaches**: a Numpy implementation and a PyTorch based model. Due to the challenge of class imbalance a special loss function is implemented (**Class-weighted Binary Cross Entropy Loss**). The two differ in the methods of computation of gradients, memory management and PyTorch even gives ability to use GPU devices for computation. Data pre-processing, including feature selection and scaling is applied to increase model validity, The two methods are compared through visual plots of convergence times and performance metrics (Accuracy, F1 score, PR-AUC). Analysis and discussion is done based on the confusion matrices generated through both the methods. The PyTorch model offered slightly better generalization and performance metrics, though both models struggled with class separation highlighting the importance of algorithmic imbalance handling.

Contents

1 Implementation Overview

1.1 Goal

Given 14 associated variables (characteristics) our goal is to train a Neural Network that predicts based on these variables whether the person will show up for the appointment or not.

1.2 Dataset

The dataset used for this purpose is publicly available here

The various columns in the dataset are listed below.

- `PatientId`
- `AppointmentID`
- `Gender`
- `ScheduledDay`
- `AppointmentDay`
- `Age`
- `Neighbourhood`
- `Scholarship`
- `Hipertension`
- `Diabetes`
- `Alcoholism`
- `Handcap`
- `SMS_received`
- `No-show`

1.2.1 Challenges

The main challenge in this dataset was the class-imbalance. In any classification task, it is necessary that enough datapoints exist for each class otherwise the model may get biased towards the majority class and will majorly predict the majority class.

In this dataset, we can get a plot of the no-show count

```python
import seaborn as sns
sns.barplot(data=df["No-show"].value_counts())
```

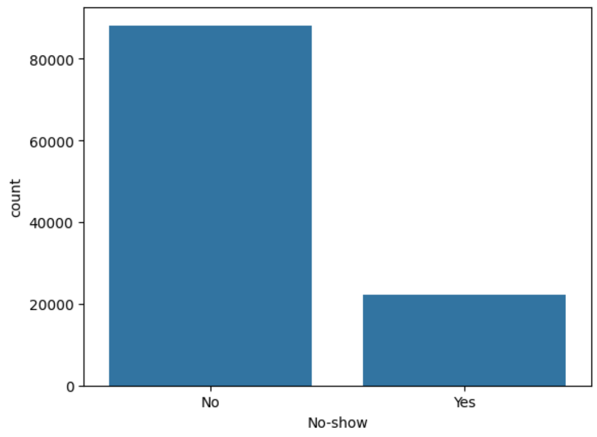


Figure 1: Count of No-show in the dataset

1.3 Solution

As it is clear from the plot above, the number of patients who showed up for their appointments significantly outweighed those who did not. In classification tasks, such imbalance often causes models to become biased toward the majority class, leading to poor generalization

As per the constraints of the project, data-level techniques such as oversampling or undersampling were not permitted. Therefore, an algorithmic solution was implemented to address this issue. It is explained in detail below.

2 Data Preprocessing

2.1 Exploratory Analysis and Data Cleaning

Mapping No-show to binary output .

```python
df["No-show"] = df["No-show"].map({"No": 0, "Yes": 1})
```

No show rate by patient's age .

```python
# dropping data-points with invalid ages
df = df[(df["Age"] > 0) & (df["Age"] < 100)].reset_index(drop=True)

no_show_age = df.groupby("Age")["No-show"].mean()
fig, ax = plt.subplots(1, figsize=(10, 5))
ax.set_title("No-show rate by age")
sns.set_context("paper", font_scale=2)
sns.lineplot(data=no_show_age, marker="o", ax=ax)
```

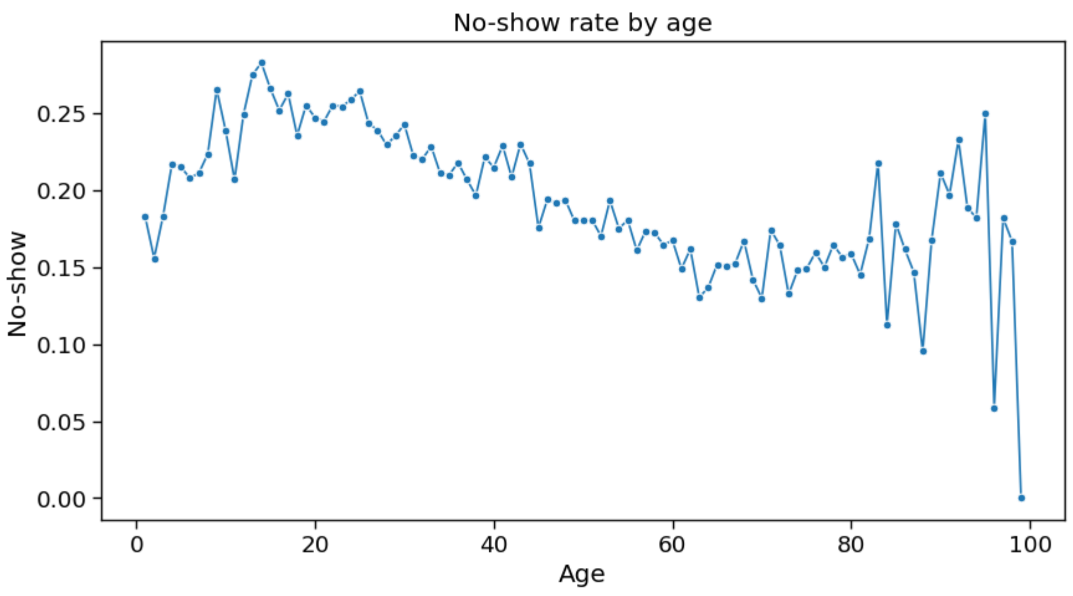


Figure 2: No-show rate by age

No show rate by patient's condition .

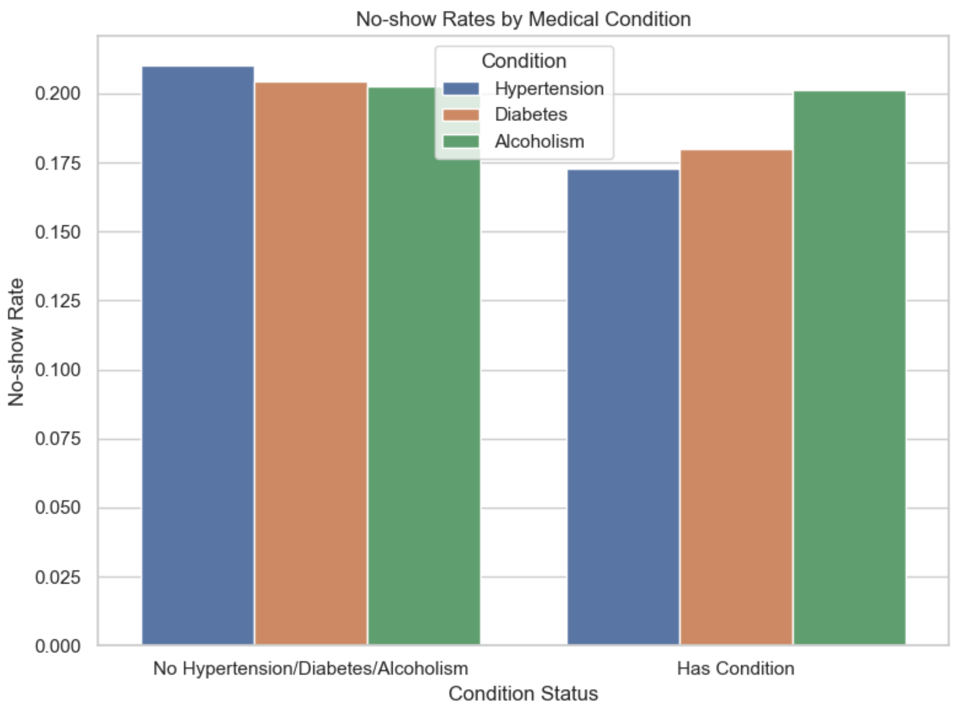


Figure 3: No-show rate by patient's condition

No show rate by Handicap level .

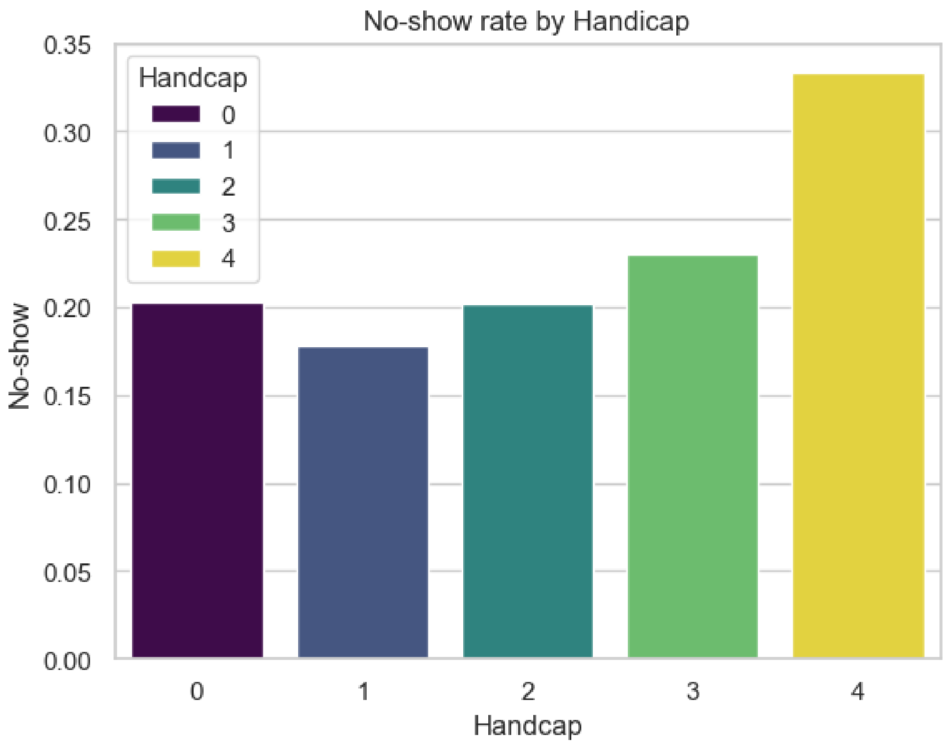


Figure 4: No-show rate by Handicap level

2.2 Preprocessing

Mapping Gender values to binary output .

```
df["Gender"] = df["Gender"].map({"M": 0, "F": 1})
```

Feature Creation: `LeadTime` .
An important temporal feature engineered for this dataset was `LeadTime`, which represents the number of days between the date a patient scheduled their appointment and the actual appointment date.

```
df["LeadTime"] = (pd.to_datetime(df["AppointmentDay"]) -
↪   pd.to_datetime(df["ScheduledDay"])).dt.days
```

Since, lead-time cannot be negative, I removed all rows which had invalid lead-times.

```
df = df[df['LeadTime'] >= 0].reset_index(drop=True)
```

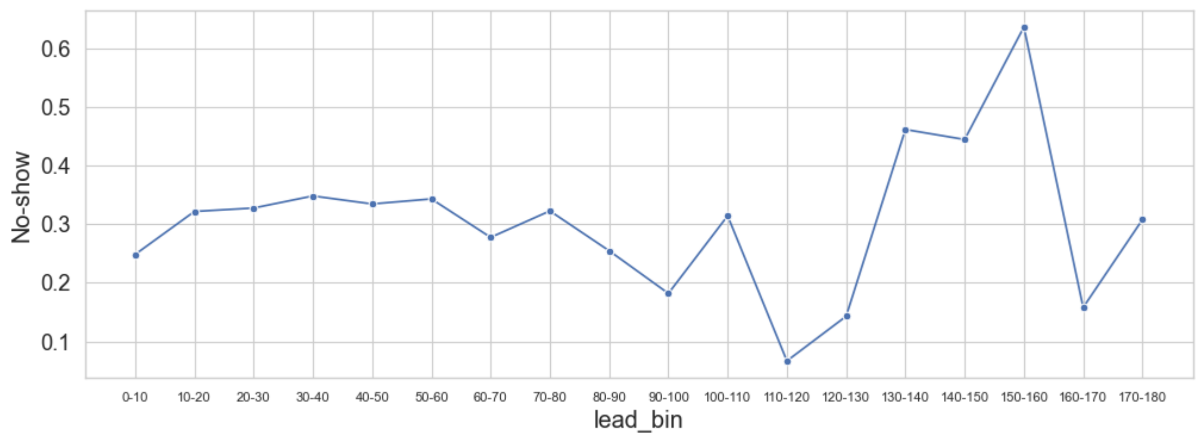


Figure 5: No-show rate by LeadTime ranges

Feature Selection and Normalization .

```
features = ["Gender", "Age", "Scholarship", "Hipertension", "Diabetes",
↪   "Alcoholism", "Handcap", "SMS_received", "LeadTime"]
label = "No-show"
X = df.loc[:, features].to_numpy()
y = df[label].to_numpy().reshape(-1, 1)
X_norm = normalize(X)
```

3 Neural Networks

A neural network is an algorithmic approach towards machine learning tasks. In this project, we aim to build an algorithm that can learn to classify data from the given datapoints, specifically for a binary classification task.

But what is it?

- A neural network is made up of multiple layers. All of these layers are in turn built up of neurons.
- Each neuron can be seen as a light bulb with varying brightness (0 to 1). This real number in the circle is called the activation of the neuron.
- We start with the first layer that contains all of the input features, then we use **feed-forward** mechanism to advance to the next layers and finally make our way to the output layer.
- Each layer is connected to the previous layer by a set of weights (colored line segments). Each neuron in the current layer gets its value (brightness) from the previous layer.

A visualization of neurons, layers and neural networks .

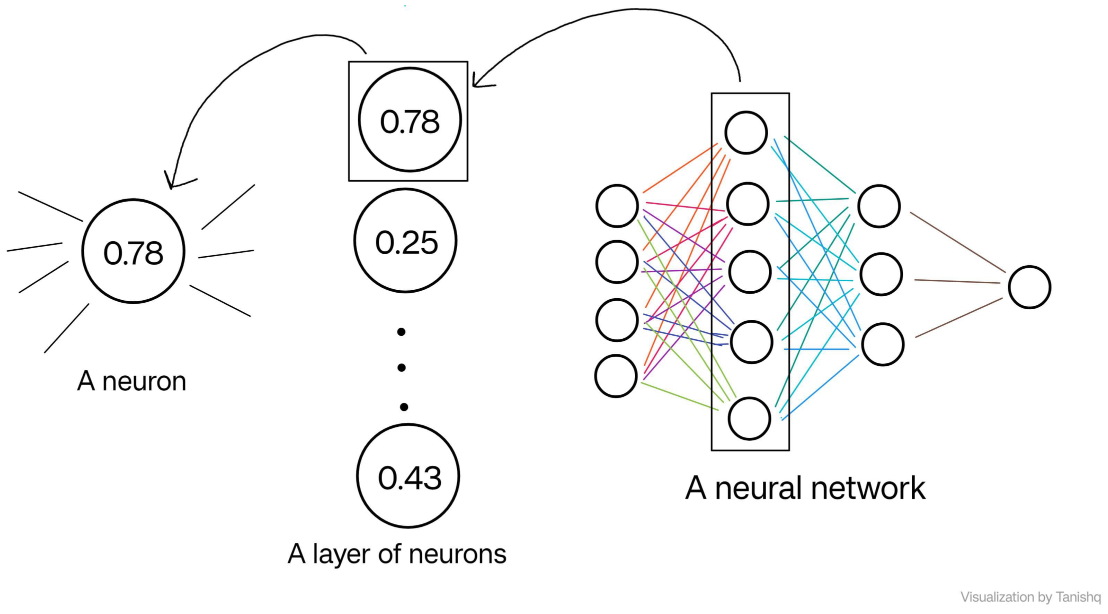


Figure 6: Neurons, layers and neural networks

How does it learn? Like any machine learning algorithm, the neural network is initialized with a random set of weights, which it uses to make predictions. These predictions are compared to the actual label through a **loss function**. The gradient of this loss is calculated with respect to each parameter involved (weights and biases) using **bakpropagation** and we try to minimize this loss using algorithms like **gradient descent** and its variations. The mathematical equations and functions are explained in the following section.

3.1 Neural Network Primer

Mathematical Structure A neural network defines a mapping of all the input features x to a prediction y.

$$f(x;\theta) = y$$

where θ represents weights and biases

Feed-Forward As previously mentioned, we start with the input layer and proceed to the further layers. For each hidden layer of the Neural Network,

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}, \qquad a^{[l]} = \sigma(z^{[l]})$$

where $z^{[l]}$ is the pre-activation, $a^{[l]}$ is the activation, σ is the activation function

Learning The real *learning* of the network happens when the weights and biases are updated, based on the cost function. The most common technique for that is gradient descent, we use the equations explained below to make the updates to weights and biases. For more information, refer this book

But the question remains, where do we get these derivatives from?

Backpropagation This algorithm is the heart of any neural network as it helps us to calculate the derivatives of the cost function at each layer of the neural network (This task is complicated compared to linear regression but is essentially the same). The actual mathematical formulation can be found here. Simplifying it, here is what I understand

- We start at the last layer and make our way to the input layer going through each hidden layer in the process.
- At each layer, we calculate something called the error signal of that layer ($\delta^{[l]}$). It represents the sensitivity of the cost with respect to the activation of that layer. I will come at how to calculate $\delta^{[l]}$ in a while.
- After, we have the deltas, we can compute the partial derivatives of the cost function with respect to the weights and biases. Keep in mind, the $W^{[l]}$ is the weight matrix connecting layer $l-1$ to l. (colored line segments from the previous page)

$$\frac{\partial \mathcal{C}}{\partial W^{[l]}} = \delta^{[l]}(a^{[l-1]})^\top, \quad \frac{\partial \mathcal{C}}{\partial b^{[l]}} = \delta^{[l]}$$

- Then, the weights and biases are updated using the update rule

$$W^{[l]} := W^{[l]} - \eta \cdot \frac{\partial \mathcal{C}}{\partial W^{[l]}}, \quad b^{[l]} := b^{[l]} - \eta \cdot \frac{\partial \mathcal{C}}{\partial b^{[l]}}$$

- **How is $\delta^{[l]}$ calculated?** The value of $\delta^{[l]}$ for the last layer is fairly simple, but the further layers have a recursive definition with $\delta^{[l]}$ depending on $\delta^{[l+1]}$ and so on.

$$\delta^{[l]} = \begin{cases} \nabla_a C \odot \sigma'(z^{[l]}) & l = L \\ ({W^{[l+1]}}^\top \delta^{[l+1]}) \odot \sigma'(z^{[l]}) & l \neq L \end{cases}$$

3.2 Architecture

Now, I will describe the architecture of the neural network that I have implemented for this project.

- I have implemented a **fully connected** NN.
- It uses **ReLU** (Rectified Linear Unit) activation for hidden layers and **sigmoid** activation for the output layer.
- The initialization is **He** initialization for the layers using **ReLU** and **Xavier** initialization for the output layer since it uses **sigmoid**.
- The loss function implemented is **Class-Sensitive Binary Cross Entropy** loss. More on that in the next sections.

3.2.1 Activations

ReLU activation It is defined as follows

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

- Faster to compute and simpler derivative
- Removes saturation or squashing of gradients for large values.

Sigmoid activation for output layer It is defined as follows

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Output range is $[0, 1]$, so can be interpreted as probability.
- With BCE, $\delta^{[L]}$ (error signal for the last layer) becomes simple i.e. $\delta^{[L]} = \hat{y} - y$

3.2.2 Weight Initialization

According to this paper, **Xavier** initialization works well for **sigmoid** activated layers, while **He** initialization works well for **ReLU** activated layers.

3.2.3 Special Loss Function

Class-Weighted Binary Cross-Entropy Loss To reduce the impact of class imbalance, I used **class-weighted binary cross-entropy loss** function. This approach alters the standard binary cross-entropy loss (used for binary classification tasks) by multiplying a higher weight to the minority class during training. It ensures that misclassifications of minority class examples penalizes the model more and higher impact is observed on parameters.

The weighting method was inspired by Chapter 4 of *Designing Machine Learning Systems*, which tells about algorithmic methods to handle class imbalance without altering the data distribution (resampling). Specifically, the approach resembles *class-balanced loss*, which uses inverse class frequencies to penalize minority classes.

Let w_0 and w_1 represent the weights for the negative and positive classes, respectively. The loss function used is:

$$\mathcal{L}(y, \hat{y}) = -[w_1 \cdot y \cdot \log(\hat{y}) + w_0 \cdot (1 - y) \cdot \log(1 - \hat{y})]$$

where:

- y is the true label (0 or 1),
- $\hat{y}$ is the predicted probability,
- $w_1 = \frac{N}{2N_1}$, $w_0 = \frac{N}{2N_0}$, with N_1 and N_0 being the number of samples in each class, and $N = N_1 + N_0$.

To implement this in code, I used only the positive weight (just divide the above loss function by w_0). then, `self.class_weight` or $w_1 = \frac{Number\ of\ positive\ samples}{Number\ of\ negative\ samples}$

```
delta = (1 - y) * f - self.class_weight * y * (1 - f)
```

This method allowed the model to remain sensitive to the minority class without violating the constraint of not resampling the data. It proved effective in improving the F1-score and PR-AUC. A plot is shown below for F1-scores of two model using weighted and non-weighted loss function respectively with other parameters of the architecture same.

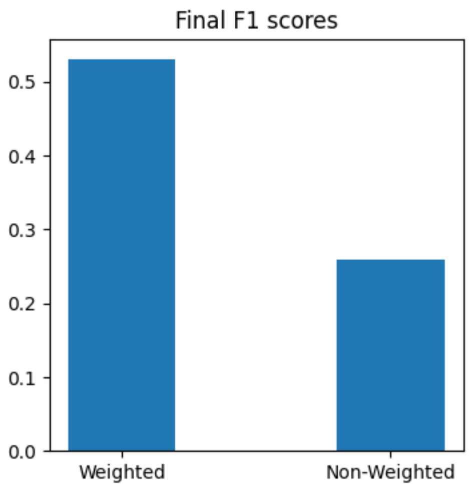


Figure 7: Final F1 Scores for weighted and vanilla BCE Loss

4 Part 1: Scratch Implementation

4.1 Implementation Details

My implementation of the Neural Network is object-oriented, with a focus on modularity and flexibility. The code supports a dynamic number of layers and neurons, making it easy to experiment with different architectures without structural changes. This design promotes organization and scalability.

Inspired from the book Deep Learning and Neural Networks by Michael Nielsen
Let's walkthrough the code step-by-step

```python
class NeuralNetworkBinary:
    def __init__(self, layers):
        self.num_layers = len(layers) - 1
        self.weights = [np.random.randn(layers[i], layers[i + 1]) *
        ↪  np.sqrt(2 / layers[i]) for i in range(len(layers) - 2)]
        self.weights.append(np.random.randn(layers[-2], layers[-1]) /
        ↪  np.sqrt(layers[-2]))
        # shape = (n_in, n_out)
        self.biases = [np.random.randn(1, layers[i]) for i in range(1,
        ↪  len(layers))]
        # shape = (1, n_out)
...
```

This shows the initialization of the network, `layers` is a list containing the number of neurons in each layer. For example: `layers = [X_train.shape[1], 64, 16, 8, 1]` The weights and biases are initialized with the methods described earlier.

4.1.1 Forward Pass

```python
...
    def forward_pass(self, x):
        z = []
        a = [x]
        for i in range(self.num_layers):
            z_new = a[i] @ self.weights[i] + self.biases[i]
            z.append(z_new)
            if i == self.num_layers - 1:
                a_new = self.sigmoid(z_new)
            else:
                a_new = self.relu(z_new)
            a.append(a_new)
        return (z, a)
...
```

The pre-activations (z) and activations (a) are calculated according to the previous mentioned formuale and returned.

4.1.2 Gradient Descent

```python
...
    def gradient_descent(self, training_data, batch_size, epochs, alpha, threshold=0.5, weighted=False, val_data=None):
        # training_data = (X, y)
        # X = (m, n)     y = (m, 1)
        X = training_data[0]
        y = training_data[1]
        self.class_weight = self.compute_class_weight(y)

        self.accuracies = np.zeros(epochs)
        self.f1s = np.zeros(epochs)
        self.costs = np.zeros(epochs)

        for e in range(epochs):
            # mini batches creation
            perm = np.random.permutation(X.shape[0])
            X_shuffled = X[perm]
            y_shuffled = y[perm]
            batches = [(X_shuffled[i:i+batch_size], y_shuffled[i:i+batch_size]) for i in range(0, X.shape[0], batch_size)]
            for batch in batches:
                dws, dbs = self.compute_derivatives(batch, weighted)
                self.weights = [self.weights[i] - alpha*dws[i] for i in range(self.num_layers)]
                self.biases = [self.biases[i] - alpha*dbs[i] for i in range(self.num_layers)]

            epsilon = 1e-8
            self.costs[e] = self.compute_cost(np.clip(self.forward_pass(X)[1][-1], epsilon, 1 - epsilon), y)

            if val_data:
                X_val = val_data[0]
                y_val = val_data[1]
                # self.accuracies[e] = self.compute_accuracy(y_hat, y_test)

                confusion_matrix = self.confusion_matrix(X_val, y_val, threshold=threshold)
                self.f1s[e] = self.compute_f1(confusion_matrix)
        if val_data:
            self.pr_roc_data = self.pr_roc(X_val, y_val)
...
```

This uses mini-batch gradient descent to minimize the loss. Most of the heavylifting is done by `compute_derivatives` function which is essentially the backpropagation algorithm. This continuously stores f1 scores after each epoch too.

4.1.3 Backpropagation

```python
...
    def compute_derivatives(self, batch, weighted):
        X = batch[0]
        y = batch[1]
        batch_size = X.shape[0]

        z, a = self.forward_pass(X)

        # for numerical stability, to avoid division by zero (suggested by
         ↪ GPT)
        f = a[-1]
        epsilon = 1e-8
        f = np.clip(f, epsilon, 1 - epsilon)

        dws_batch = [np.zeros_like(w) for w in self.weights]
        dbs_batch = [np.zeros_like(b) for b in self.biases]

        # this is the derivative of class weighted binary cross entropy loss
         ↪ multiplied by the derivative of sigmoid
        if weighted:
            delta = (1 - y) * f - self.class_weight * y * (1 - f)
        else:
            delta = f - y

        for i in range(self.num_layers - 1, -1, -1):
            dws_batch[i] = a[i].T @ delta / batch_size
            dbs_batch[i] = np.mean(delta, axis=0).reshape(1, -1)
            if i > 0:
                delta = (delta @ self.weights[i].T) * self.relu_prime(z[i -
                 ↪ 1])

        return dws_batch, dbs_batch
...
```

The main goal of this function is to compute $\frac{\partial \mathcal{C}}{\partial W^{[l]}}$ and $\frac{\partial \mathcal{C}}{\partial b^{[l]}}$ for every layer. It implements the mathematical formulations for error signal, loss function and its gradients as derived earlier. The function handles both class-weighted and non-weighted binary cross-entropy loss, allowing flexibility in handling imbalanced datasets as required by the project constraints.

Numerical stability is ensured by clipping activations in the final layer.

4.1.4 Helper functions

Other helper functions for this implementation are present at the end of the document.

4.2 Training Procedure

4.2.1 Vanilla Training

The modular code makes sure restructuring is not required for tuning the number of neurons, or tuning hyperparameters. For simple training and validation, the neural network can be initialized and trained as follows:

```
nn_np = NeuralNetworkBinary([X_train.shape[1], 64, 16, 8, 1])
nn_np.gradient_descent((X_train, y_train), 100, 5000, 0.0005, threshold=0.4,
↪  weighted=True, val_data=(X_val, y_val))
```

4.2.2 Cross Validation (Stratified K-Fold)

For datasets showing imbalance, validation and training must be done in a special way.

- We will first divide the data into k parts and have k training sessions.
- Before each training k-1 parts are selected and the model is trained on it. The remaining part becomes the validation set.
- After each training session, performance metrics are computed on the validation set.
- This ensures fair division of all datapoints for both validation and training. It also makes sure that the model has seen all data points.

The code to implement it is as follows:

```
from sklearn.model_selection import StratifiedKFold

f1_scores = []
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

for fold, (train_index, val_index) in enumerate(skf.split(X_norm, y)):
    X_train, X_val = X_norm[train_index], X_norm[val_index]
    y_train, y_val = y[train_index], y[val_index]

    model = NeuralNetworkBinary([X_train.shape[1], 64, 16, 8, 1])
    model.gradient_descent((X_train, y_train), 100, 5000, 0.0005,
    ↪  threshold=0.4, weighted=True, val_data=(X_val, y_val)

    f1_scores.append(model.f1s[-1])
    print(f"fold {fold} completed")

print(np.mean(f1_scores))
```

5 Part 2: PyTorch Implementation

5.1 Implementation Details

```python
class NeuralNetwork(nn.Module):
    def __init__(self, layers_dim):
        super(NeuralNetwork, self).__init__()
        self.num_layers = len(layers_dim) - 1
        self.layers = nn.ModuleList()

        for i in range(len(layers_dim) - 1):
            layer = nn.Linear(layers_dim[i], layers_dim[i + 1])
            if i < self.num_layers - 1:
                nn.init.kaiming_normal_(layer.weight)
            else:
                nn.init.xavier_normal_(layer.weight)

            self.layers.append(layer)

    def forward(self, x):
        for i in range(self.num_layers):
            x = self.layers[i](x)
            if i < self.num_layers - 1:
                x = nn.functional.relu(x)

        return x

    def predict_class(self, x, threshold=0.5):
        x = torch.tensor(x, dtype=torch.float32)
        logits = self(x)
        probs = torch.sigmoid(logits)
        preds = (probs > threshold).float()
        return preds

def train_model(model, training_data, batch_size, epochs, alpha,
↪   threshold=0.5, weighted=False, val_data=None):
    X = training_data[0]
    y = training_data[1]
    tensor_x = torch.tensor(X, dtype=torch.float32)
    tensor_y = torch.tensor(y, dtype=torch.float32)
    dataset = TensorDataset(tensor_x, tensor_y)
    training_loader = DataLoader(dataset, batch_size=batch_size,
    ↪   shuffle=True)

    model.costs = np.zeros(epochs)
    model.f1s = np.zeros(epochs)

    optimizer = torch.optim.SGD(model.parameters(), lr=alpha)

    if weighted:
        weight_tensor = torch.tensor(np.sum(y == 0) / np.sum(y == 1),
        ↪  dtype=torch.float32)
        loss_fn = nn.BCEWithLogitsLoss(pos_weight=weight_tensor)
    else:
        loss_fn = nn.BCEWithLogitsLoss()

    for e in range(epochs):
        model.costs[e] = train(model, optimizer, loss_fn, threshold,
        ↪  training_loader, "cpu")
        logits = model(tensor_x)
        preds = (torch.sigmoid(logits) > threshold).float()
        model.f1s[e] = f1_score(y, preds)

def train(model, optimizer, loss_fn, threshold, training_loader,
↪   device="cpu"):
    m = len(training_loader)
    model.train()

    epoch_loss = 0

    for X_batch, y_batch in training_loader:
        X_batch = X_batch.to(device)
        y_batch = y_batch.to(device)

        optimizer.zero_grad()

        logits_batch = model(X_batch)

        loss = loss_fn(logits_batch, y_batch)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
    return epoch_loss / m

def evaluate(model, validation_data, threshold=0.5):
    X_val = validation_data[0]
    tensor_x = torch.Tensor(X_val)
    y_val = validation_data[1]

    y_prob = torch.sigmoid(model(tensor_x)).detach().numpy()
    y_pred = model.predict_class(tensor_x, threshold).numpy()

    f1 = f1_score(y_val, y_pred)
    pr = precision_recall_curve(y_val, y_prob)
    roc = roc_curve(y_val, y_prob)
    return f1, pr, roc
```

The code logic is functionally the same, but it uses the `PyTorch` library for more efficient and faster computations. One neat thing about the PyTorch implementation is that I did not need to derive the gradient by hand. It uses something called **computational graphs** to automatically calculate the gradients. This is a very efficient and modular feature.

The `train_model` is a wrapper function over the main `train` function which does most of the heavy-lifting including backpropagation and parameter updates.

5.2 Training Procedure

The model can be trained as follows

```
pt_nn = NeuralNetwork([X_train.shape[1], 64, 16, 8, 1])
train_model(pt_nn, (X_train, y_train), 1000, 5000, 0.0005, threshold=0.4,
↪   weighted=True)
```

6 Comparative Evaluation and Analysis

6.1 Convergence Time and Performance Metrics plot

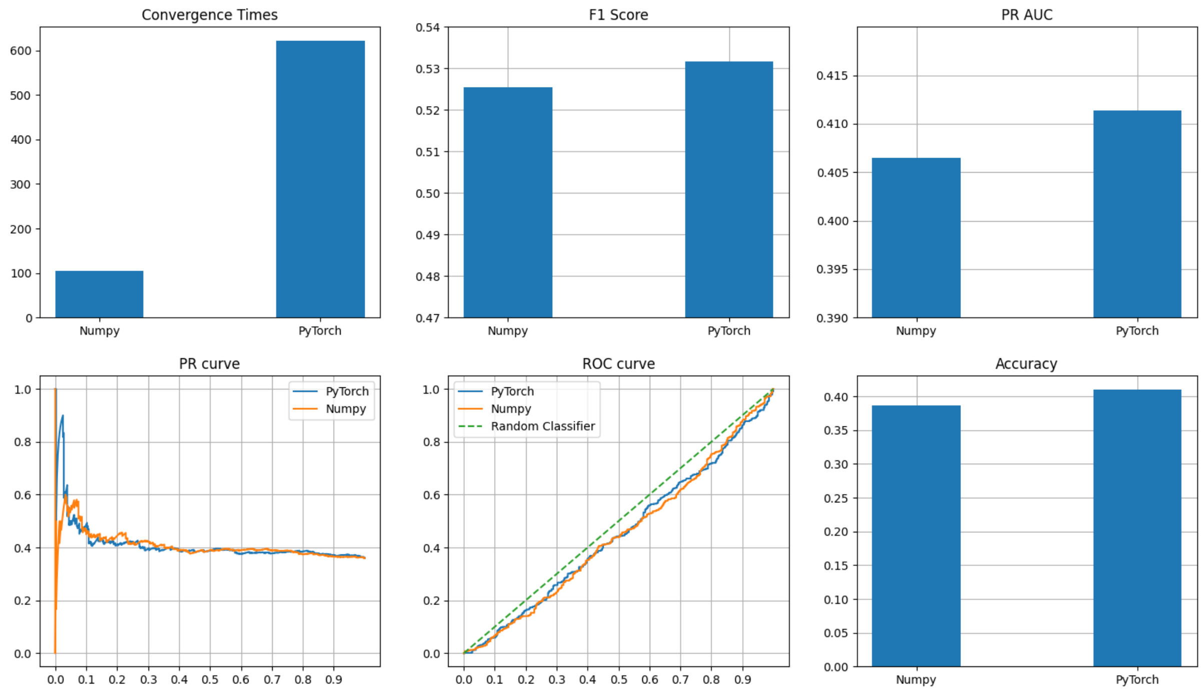
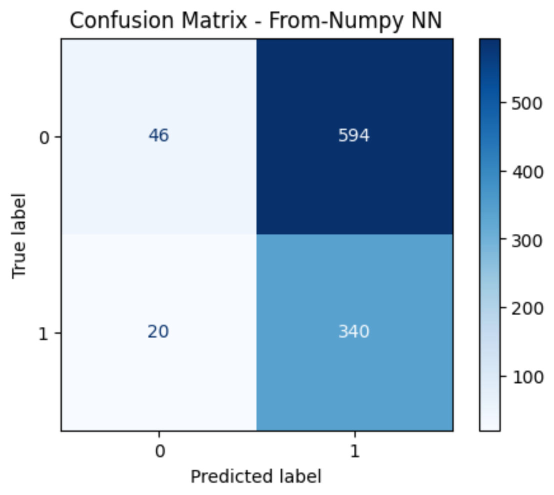


Figure 8: Comparative plot

PyTorch performs slightly better in all of the performance metrics but it takes a lot more time to converge (probably because I was using CPU). For industrial purposes, with larger datasets and better GPUs, PyTorch will outperform the numpy implementation in convergence times too.

Both the models show poor class separation (from the ROC curve) even after the implementation of class-weighted loss function. In the future, loss functions like **focal loss** can be tried out to improve metrics.

6.2 Confusion Matrix Analysis



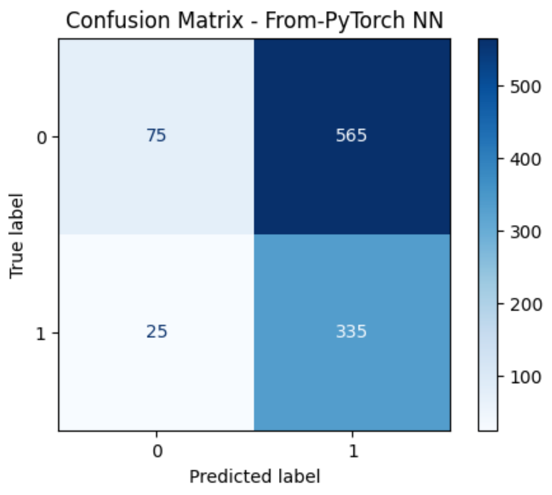


Figure 9: Confusion Matrices

The focus for this project was to capture the most **True Positives**. Both the models do decent in that but the number of false positives are extremely high which shows poor class separation. The models are highly biased towards the label 1.

PyTorch shows slightly more balanced output due to the comparatively higher number of **True Negatives**.

6.3 Insights and Discussion

Framework optimizations When we write the numpy implementation, we only take help of the framework for computation, we need to hard-code the gradients and need to explicitly tell what to tell. While, PyTorch offers automatic gradient computation using the `.backward()` method. This results in better gradient calculations which results in slightly better performance metrics. But, due to the overhead time of initialization and abstraction per epoch, PyTorch takes more time to converge.

Hardware acceleration Numpy primarily runs on CPU but PyTorch can utilize GPU acceleration if available (not used by me). This can lead to faster convergence in practice if optimized (proper batch size, number of epochs etc.).

Numerical stability In the Numpy implementation, care had to be taken to avoid mathematical errors like division by zero, $log(0)$ etc. but all of this was automatically taken care of by PyTorch's abstractions.

Code efficiency As mentioned before, Numpy implementation required manual implementations for every step be it activation, loss functions or gradient calculations. This is feasible if the Neural Network is simple and iterative (like this project) but when we have more complicated NNs where each layer requires different loss functions, activations etc. Numpy implementation might become difficult to maintain and debug but PyTorch will shine there due to its autograd feature.

7 Conclusion

The plots and performance metrics show both NumPy and PyTorch implementation achieve comparable results. The NumPy implementation while less resource intensive and quicker can be dangerous and difficult to debug in bigger projects because of its manual nature.

PyTorch consistently performs better and offers more balanced results due to its mathematical stability and code abstractions. The autograd feature is revolutionary and allows quick prototyping without the need of calculating complex gradients by hand.

Although both implementations do not perform upto the level expected, some of the blame can be put on the class imbalance of the dataset. A better approach for this type of problem would be algorithms like **Random Forest Classifiers** or **XGBoost** rather than ANNs.

Helper Functions .

```python
...
    def relu(self, x):
        return np.maximum(0, x)

    def relu_prime(self, x):
        return (x > 0).astype(float)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def compute_cost(self, y_hat, y):
        return -np.mean(self.class_weight * y * np.log(y_hat) + (1 - y) * np.log(1
        ↪  - y_hat))

    def compute_accuracy(self, y_hat, y, threshold=0.5):
        p = (y_hat > threshold).astype(float)
        return np.mean(p == y)

    def predict_class(self, X, threshold=0.5):
        z, a = self.forward_pass(X)
        epsilon = 1e-8
        y_hat = np.clip(a[-1], epsilon, 1 - epsilon)
        return (y_hat > threshold).astype(float)

    def confusion_matrix(self, X, y, threshold=0.5):
        p = self.predict_class(X, threshold)
        confusion_matrix = np.zeros(shape=(2, 2))
        confusion_matrix[0, 0] = np.sum(np.logical_and(p == 1, y == 1)) # tp
        confusion_matrix[0, 1] = np.sum(np.logical_and(p == 1, y == 0)) # fp
        confusion_matrix[1, 0] = np.sum(np.logical_and(p == 0, y == 1)) # fn
        confusion_matrix[1, 1] = np.sum(np.logical_and(p == 0, y == 0)) # tn
        return confusion_matrix

    def compute_f1(self, cm):
        tp = cm[0, 0]
        fp = cm[0, 1]
        fn = cm[1, 0]
        tn = cm[1, 1]

        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * precision * recall / (precision + recall) if (precision + recall)
        ↪  > 0 else 0
        return f1

    def pr_roc(self, X, y):
        precision = []
        recall = []
        fpr = []
        f1_sweep = []
        # tp  fp
        # fn  tn
        for threshold in np.linspace(0.01, 0.99, 101):
            cm = self.confusion_matrix(X, y, threshold)
            tp = cm[0, 0]
            fp = cm[0, 1]
            fn = cm[1, 0]
            tn = cm[1, 1]

            p = tp / (tp + fp) if (tp + fp) > 0 else 0
            r = tp / (tp + fn) if (tp + fn) > 0 else 0
            f = fp / (fp + tn) if (fp + tn) > 0 else 0
            f1 = 2 * p * r / (p + r) if (p + r) > 0 else 0
            precision.append(p)
            recall.append(r)
            fpr.append(f)
            f1_sweep.append(f)
        return (precision, recall, fpr, f1_sweep)
...
```