Neural Network Implementation on Medical Appointment No-Show Dataset

Hardik Prabhakar

24095038

Introduction

Neural networks are computational models inspired by the structure and functions of biological neural networks. In this report, we will compare two different implementation of an ANN.

The implementations that we will use are:

1. Neural Network from Scratch
2. PyTorch Implementation

The dataset we're using is the Medical Appointment No-Show Dataset, predicting the `No-show` column.

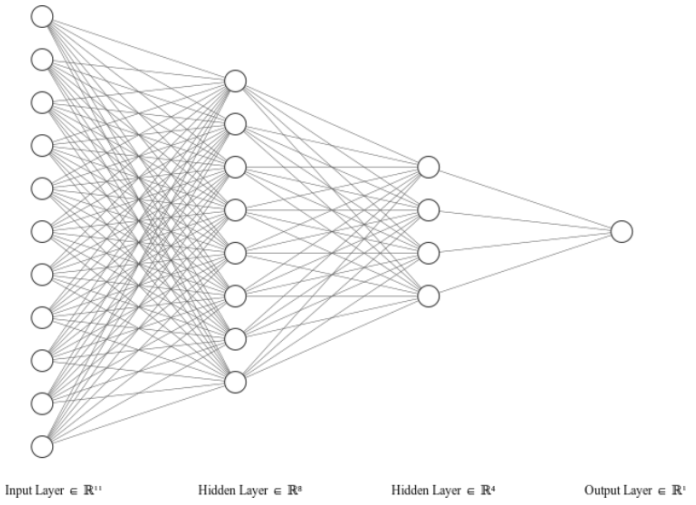
Data Pre-Processing

- First, we load the dataset using `pandas` and check for null records. As there are none, no dropping or imputing of data is needed.
- We converted the day columns from `object` dtype to `datetime64[ns]` for convenience.
- Then, we rename some misspelled columns and drop the ID columns as they are not relevant in predicting the No-Show probability.
- After that, we cap the values of the `Handicap` column to 1 as having a disability or not is more informative about the No-Show probability than the number of disabilities.
- We check for records with negative ages and drop them.

- We create two new columns `WaitingDays` and `AppointmentDOW`, representing the difference between the appointment and scheduling day and the day of week of the appointment, respectively. We drop the day columns as they are not relevant in predicting No-Show.
- We check for negative `WaitingDays` entries and drop them.
- Lastly, we check the percentage of `Show` and `NoShow` records and find that approximately 80% of the records belong to the `Show` category, which signifies that we have an imbalanced dataset.
- Using `scikit-learn`'s `LabelEncoder`, we convert all non-numerical columns like `Neighbourhood` into numerical entries.
- Finally, we split the dataset into target and features, and then further split them into training and validation sets, using a 80/20 split. We apply *Z-Score Normalization* on every feature with a range of values.

Implementation

We input our 11 features into a feedforward neural network with 2 hidden layers consisting of 8 and 4 neurons respectively and a single neuron for the output layer.



We initialize our weights using He Initialization, which draws values from a normal distribution with $\mu = 0$ and $\sigma = \sqrt{\frac{2}{no.\ of\ input\ neurons}}$, and our biases to 0.1. This initialization technique prevents gradients from exploding as the training proceeds.

The forward propagation of each layer is defined as:

$$X_{\mathrm{n}} = X_{\mathrm{n}-1} w_{\mathrm{n}} + b_{\mathrm{n}}$$

where X_{n}, w_{n} & b_{n} are the output, weight and bias matrix of the nth layer respectively.

For the first two hidden layers, we use **ReLU** as the activation function. It is defined as:

$$\mathrm{ReLU}(x) = \max(0, x)$$

For the output layer, we apply the **sigmoid** function on the output of the single neuron to convert it into a probability of getting a `No-Show`. The function is defined as:

$$p = \frac{1}{1 + e^{-z}}$$

As we have a imbalanced dataset, we have to use an appropriate loss function that accounts for this imbalance, or else our model will generalize poorly, especially for the minority class.

To measure the performance of our model, we use a modified version of **Binary Cross Entropy Loss**, defined as:

$$J(p) = -\alpha_0 (1 - y) \log(1 - p) - \alpha_1 y \log(p)$$

where p is the probability of belonging to `No-Show`, y is the ground truth and α_i are defined as:

$$\alpha_i = \frac{N}{\text{total number of samples of i}^{\text{th}}\text{ class}}$$

This loss function penalizes the model more for misclassifying the minority class than the majority class, driving the training in the right direction.

To update the weights and biases of the layers, we use the method of **backpropagation**. It is an efficient application of the chain rule to neural networks. **Backpropagation** computes the gradient of a loss function with respect to the weights of the network, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

For a single training example, The derivative of the loss with respect to the probability is:

$$\frac{\partial J(p)}{\partial p} \;=\; \frac{\alpha_0(1 \;-\; y)}{1 \;-\; p} - \;\frac{\alpha_1 y}{p}$$

The derivative of the probability with respect to the layer output is:

$$\frac{\partial p}{\partial z} \;=\; p(1 \;-\; p)$$

Finally, the combined derivative of the loss with respect to the layer output, calculated with the chain rule, is:

$$\frac{\partial J(p)}{\partial z} \;=\; \frac{\partial J(p)}{\partial p} \cdot \frac{\partial p}{\partial z} \;=\; \alpha_0(1 \;-\; y)p \;-\; \alpha_1 y(1 \;-\; p)$$

The derivative of ReLU function is given as:

$$\frac{\partial \mathrm{ReLU}(x)}{\partial x} = \begin{cases} 0 & x \;\leq\; 0 \\ 1 & x \;>\; 0 \end{cases}$$

The gradient matrix for the cost function with respect to the weights, biases and previous layer's output is given as:

$$\frac{\partial X_{\mathrm{n}}}{\partial w_{\mathrm{n}}} \;=\; X_{\mathrm{n}-1}^{T} D$$

$$\frac{\partial X_{\mathrm{n}}}{\partial b_{\mathrm{n}}} \;=\; D$$

$$\frac{\partial X_{\mathrm{n}}}{\partial X_{\mathrm{n}-1}} \;=\; D w_{\mathrm{n}}^{T}$$

where D is the derivative propagating back from the next layer.

The overall gradient is the average of the gradients over every training example.

For updating the weights and biases, we use the **Adam** optimizer, which adapts the learning rate of each parameter as the training progresses using the exponentially weighted moving averages of past gradients and squared gradients. This incorporation smoothens the learning curve of the model.

The update rules of this optimizer are:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial L}{\partial w_t} \right)^2$$

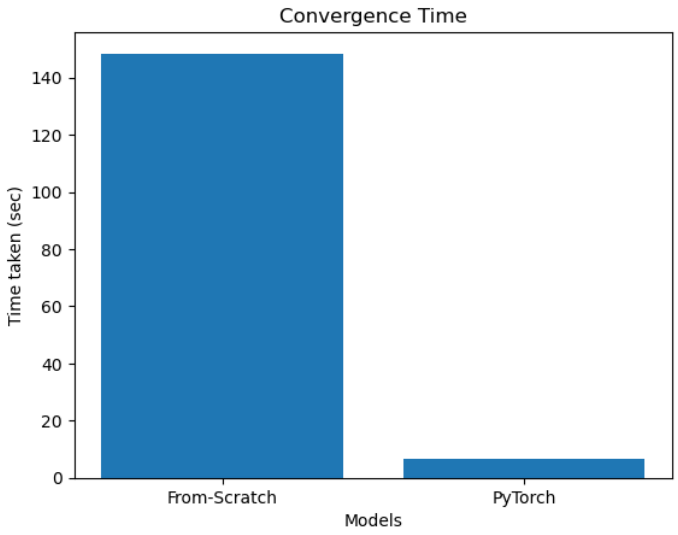
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

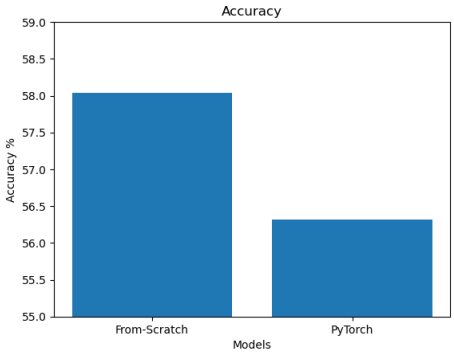
where t is the number of the current iteration and β_0, β_1, ϵ & α are hyperparameters set at 0.9, 0.999, 10^-7 and 0.003 respectively.

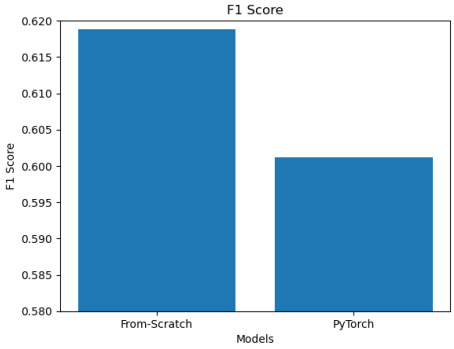
Evaluation

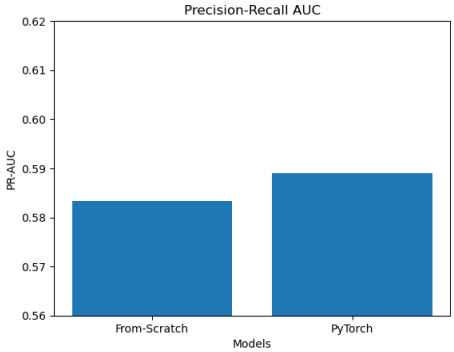
Convergence Time



Performance Metrics







Confusion Matrix

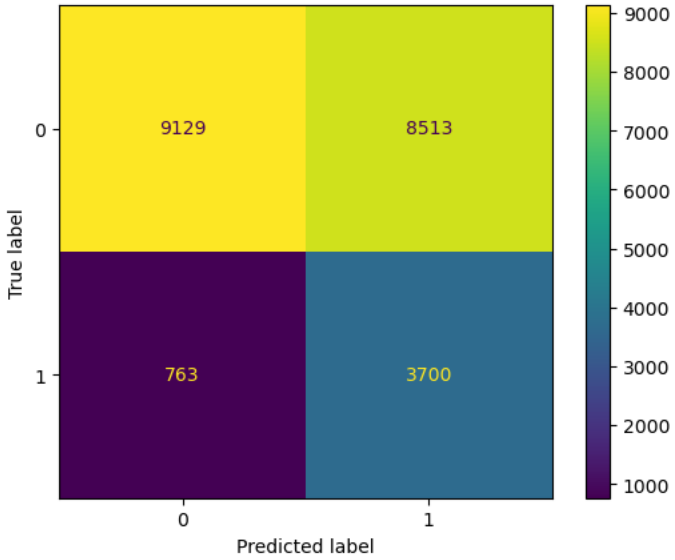


Figure 1: From-Scratch Model

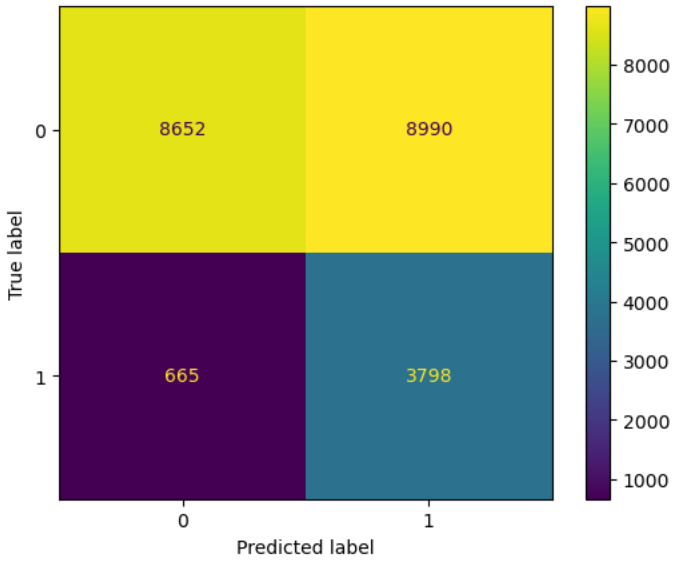


Figure 2: PyTorch Model

Analysis & Discussion

- Comparing the convergence time of the two models, we can clearly see that the PyTorch implementation is far superior. This is because PyTorch utilizes the GPU for parallel computation, which greatly optimizes the training process.
- Both the models perform about the same in regards to evaluation metrics like accuracy, F1 score and PR AUC.
- The PyTorch model has more memory usage than the from-scratch model due to its `autograd` component which stores gradients and constructs a computational graph that allows it to calculate gradients with ease.
- From the confusion matrix, we can infer that the 0 class has high precision but a low recall. In contrast, the 1 class has low precision but a high recall. This is the result of training our model on an imbalanced dataset.
- We can conclude that using PyTorch to implement neural networks is far better than implementing it from scratch as it is both more time efficient and provides convenience for gradient calculation and updates through its `autograd` feature.