CSoC 2025 Intelligence Guild Assignment

Artificial Neural Network on Medical Appointment No-Show Prediction

Suyash Ranjan
Mining(IDD) / 24154022

June 2, 2025

Abstract

This report presents the implementation and comparison of binary classification using artificial neural networks (ANNs) with two approaches: (i) a neural network from scratch using only NumPy, and (ii) a PyTorch-based implementation. The models aim to predict whether a patient will show up for their medical appointment using the Medical Appointment No-Show Dataset. We evaluate the models based on convergence speed, predictive performance (accuracy, F1, PR-AUC), memory usage, and the insights from confusion matrices.

1 Introduction

The aim of this project is to build an intuitive and practical understanding of artificial neural networks (ANNs) by implementing a binary classifier using two different paradigms — first from scratch with NumPy, and second with PyTorch. This comparative study helps us appreciate the underlying mechanics as well as the benefits of using deep learning frameworks for larger and more complex tasks.

2 Dataset Description

The dataset used is the **Medical Appointment No-Show Dataset**, which contains various features including gender, age, presence of medical conditions (like hypertension, diabetes), SMS received, and scheduling times. The target variable is whether the patient showed up for their appointment (`No-show`: 'No' = 0, 'Yes' = 1).

3 Data Preprocessing

- Dropped irrelevant identifiers (`PatientID, Neighbourhood, Alcoholism, AppointmentID`). I decided based on what is having a good impact on my model predictions using the metrices.

- Encoded categorical features like `Gender` using binary encoding. To work with numbers.
- Converted scheduling timestamps to datetime objects and created a `WaitingTime` feature.
- Filtered out negative waiting times.
- Standardized the numerical features using Z-score normalization.This is done to ensure there is not much fluctuations in the convergence because of different ranges in features.
- Final feature set was chosen after experimentation for best validation performance.
- Split the dataset into 80% training and 20% validation sets.

4 Model Implementations

4.1 (NumPy-based) Neural Network

Implemented a simple feedforward neural network using NumPy. This included manual implementation of:

- Forward propagation using matrix operations.
- Sigmoid activation at the output layer since it is classification problem.
- Weighted-Binary cross-entropy loss function to handle the minority class in the class imbalance.
- Backpropagation using gradient descent.
- Weight initialization (instead of random or Zero initialization to avoid vanishing gradient and convergence issues, i used proper initialization methods like "He initialization" for Relu activation and "Xavier Initialization" for Sigmoid activation.
- To tackle the convergence problem as the loss gets closer to the minimum loss, i have used a decaying learning rate instead of a absolute learning rate as the steps should get smaller as time goes to avoid the delay in convergence.

4.2 PyTorch-based Neural Network

Re-implemented the same architecture using PyTorch:

- Defined the model as a subclass of `nn.Module`.
- Used `BCEWithLogitsLoss` for numerical stability and i have not applied sigmoid in the last layer as this function internally applies sigmoid and then BCE.
- Handled class imbalance via `pos_weight`.

- Trained the model using `SGD` optimizer and DataLoader API for batching, I used dataloader so that i can break the dataset in batches of 1024(since the size of dataset is really large).
- Here again, i applied a decaying learning rate to tackle the same problem as in part 1.

5 Evaluation and Results

Evaluation Metrics

- **Accuracy:** Defined as $\frac{\text{Total Correct Predictions}}{\text{Total Predictions}}$. While it gives a general sense of correctness, it performs poorly with imbalanced datasets. For example, a naive model that always predicts 0 can still achieve over 90% accuracy depending on the dataset. Hence, I do not consider it a reliable metric in this context.
- **F1-score:** Defined as $\frac{\text{2 x Precision x Recall}}{\text{Precision + Recall}}$. This shows how good is your model at balancing both precision and recall.
- **Precision-Recall AUC (PR-AUC):** PR-AUC stands for the Area Under the Precision-Recall Curve.

  It tells you how well your model balances precision and recall at all possible thresholds. A higher PR-AUC means the model performs well even when the decision threshold changes.
- **Recall:** This shows how much percentage of the minority class were actually predicted correct although this may seem a good metrices but i don't feel the same because if you predict all 1s, your recall will be 100% so i don't rely on this metric much.
- **Precision:** This shows how much of predicted 1s were actually 1s and this is the best metric for measuring the strength of your model according to me.

  We can handle some false positives but missing even a few false positives would affect a lot (suppose in the case of predicting tumors in patients, again handling an imabalanced dataset).

Evaluation Metric	**Numpy NN**	**PyTorch NN**
Accuracy	0.5691	0.5736
F1-score	0.4312	0.4360
Precision	0.3495	0.3539
Recall	0.5627	0.5677
PR-AUC	0.3642	0.3704

Table 1: Comparison of Evaluation Metrics between NumPy and PyTorch Neural Networks

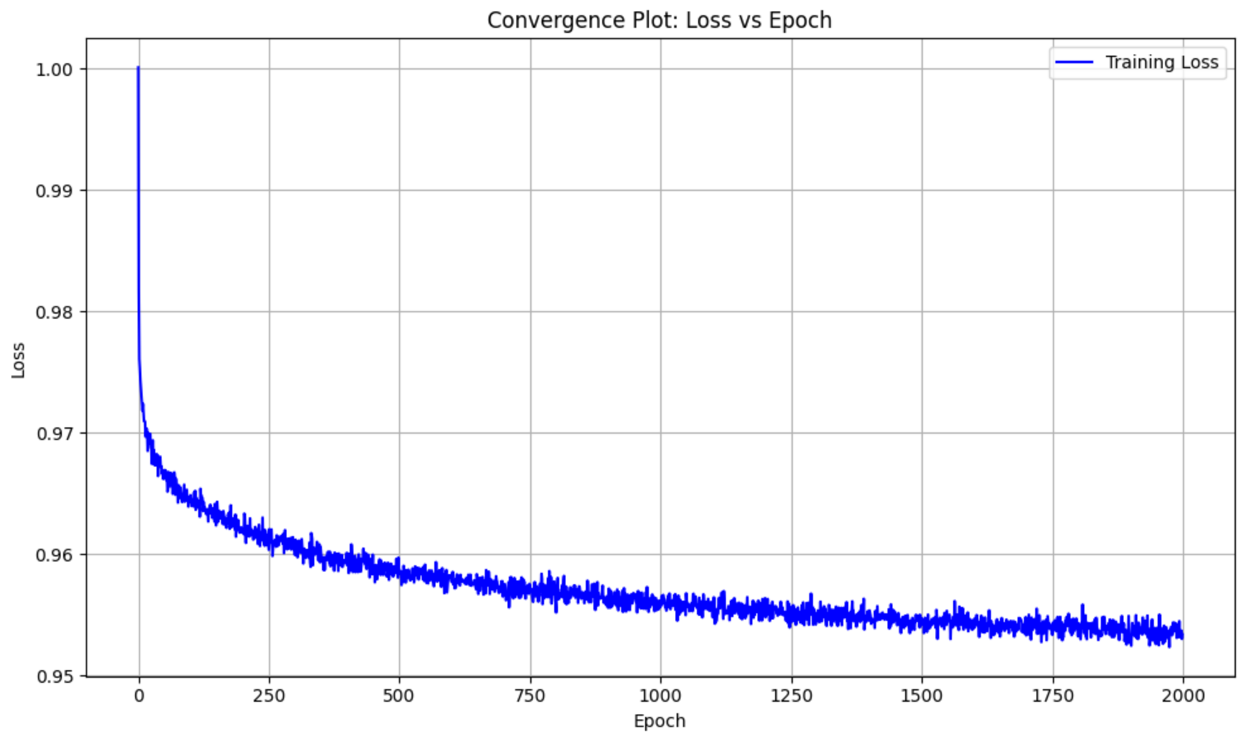

Figure 1: Convergence Pure NN

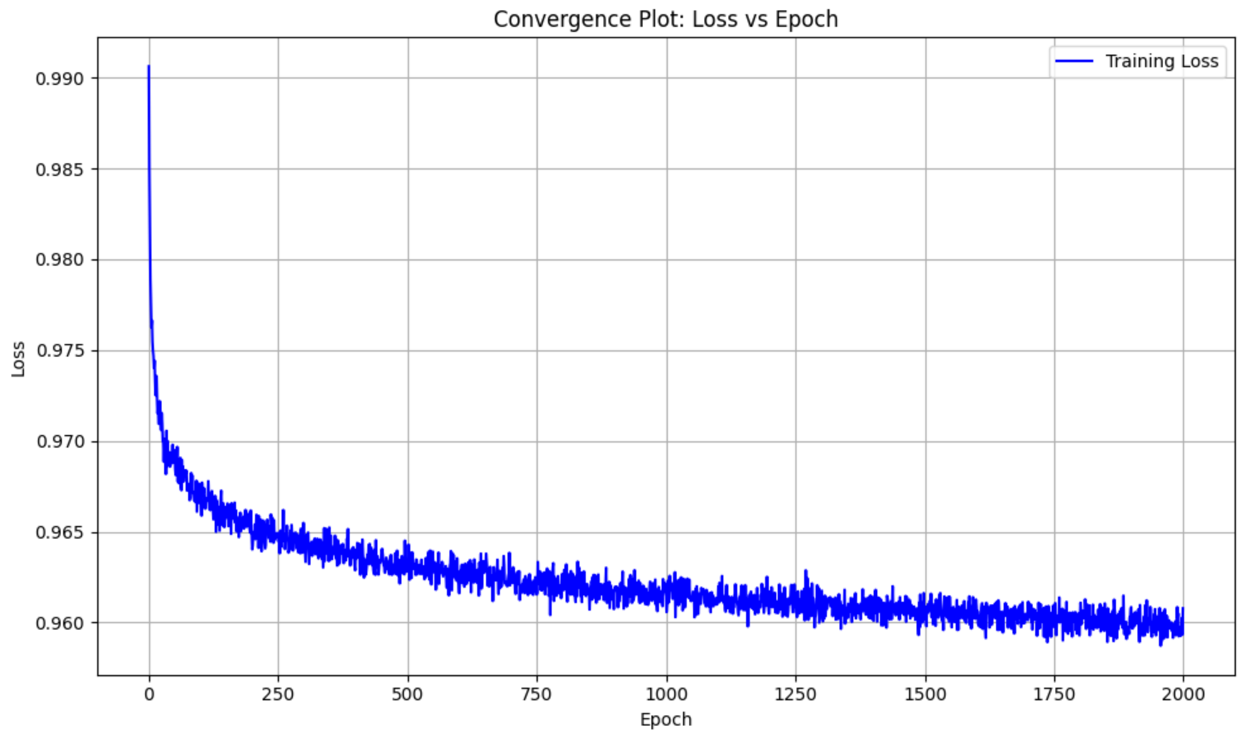

Figure 2: Pytorch NN

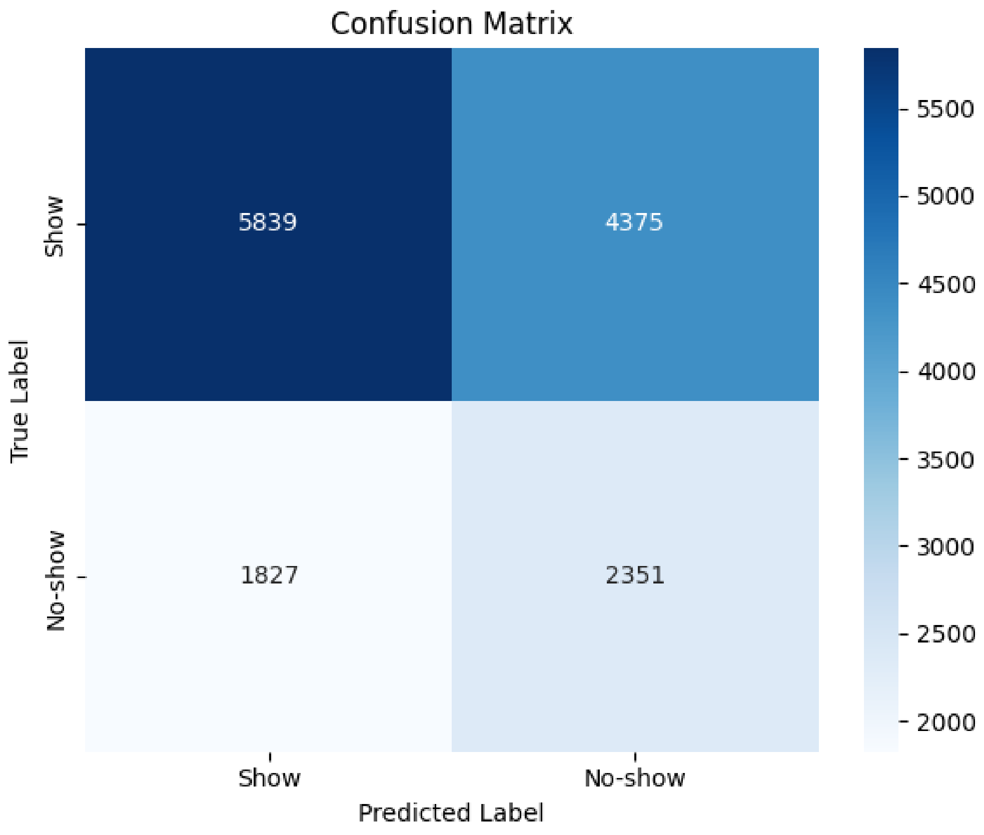

Figure 3: Confusion-Matrix(Pure NN)

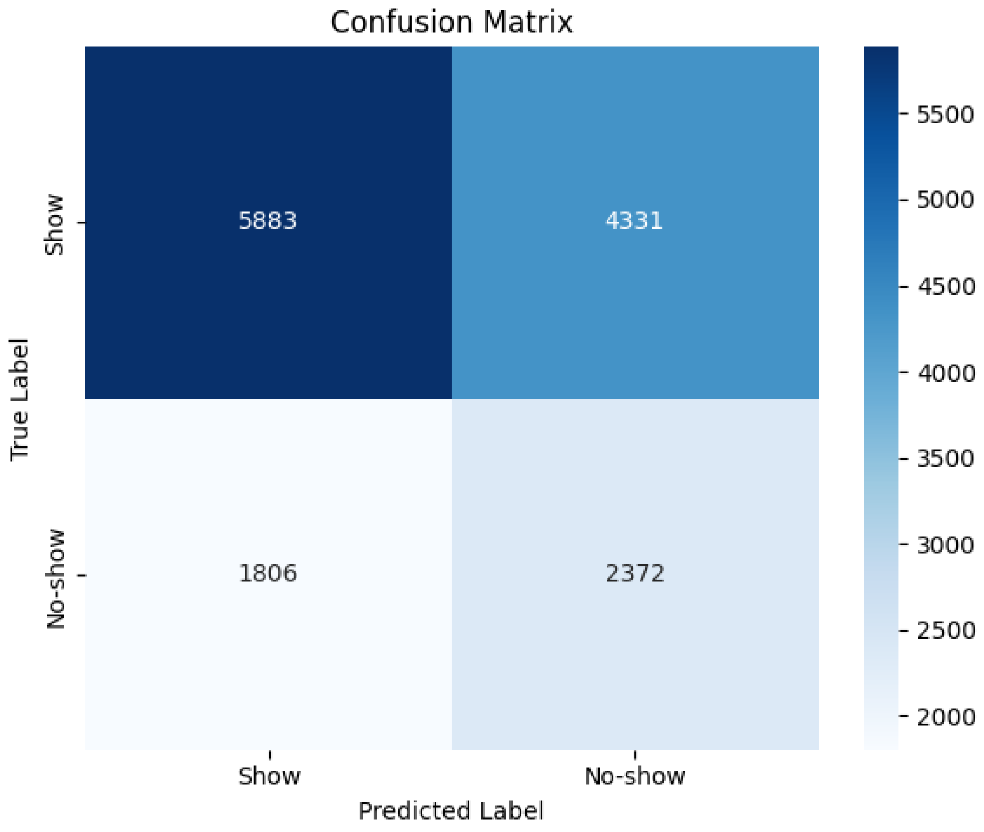

Figure 4: Confusion-Matrix(Pytorch NN)

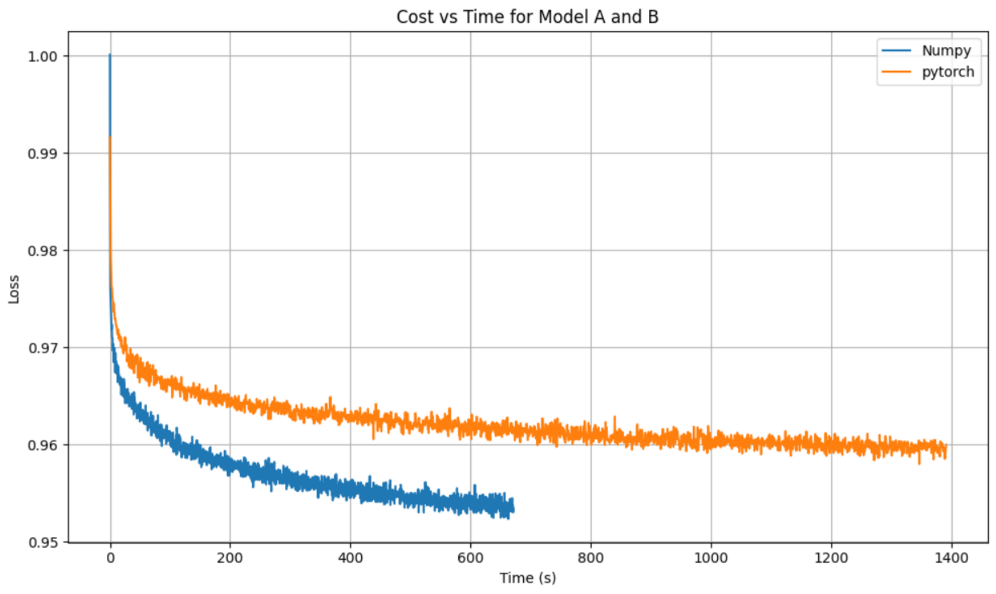

Figure 5: Convergence-Speed

6 Analysis

To summarize, this project compared the ANN implementation using NumPy and PyTorch on the Medical Appointment No-Show dataset. Below are the key observations:

Convergence Time: PyTorch outperformed the NumPy model in terms of results (metrics), although just by a slight margin.

Memory Usage: Both implementations used batch training, which helped manage memory efficiently and made the comparison fairer. The NumPy-based model had a relatively low memory footprint and gave full control over memory allocation, with minimal framework overhead. However, manual implementation of batching and gradient computation in NumPy required careful handling to avoid unnecessary memory consumption, especially during backpropagation.

The PyTorch model introduced some additional memory overhead due to automatic differentiation (autograd), computational graph tracking, and internal framework structures. However, it also managed memory more intelligently — automatically freeing up unused tensors and handling backpropagation efficiently. Allowing to run batches parrallely by using num_workers. Additionally, PyTorch is better equipped to handle larger models and datasets due to its dynamic computation graph and integration with hardware-accelerated backends.

Overall, while the NumPy model was slightly more memory-efficient in raw usage, PyTorch offered superior memory management and scalability, especially when moving to more complex architectures or larger data.

Performance Metrics: The PyTorch model generally achieved better PR-AUC and F1-score, likely due to its numerical stability, gradient flow management, and use of a more stable loss function (`BCEWithLogitsLoss`).

Confusion Matrix Analysis: Both models showed decent precision but struggled with recall, which is expected due to the class imbalance. Incorporating `pos_weight` in the PyTorch model improved recall slightly and helped balance the precision-recall trade-off.