Artificial Neural Networks
Medical Appointment No-Show Prediction

CSOC Summer of Code 2025 – Intelligence Guild, COPS IIT (BHU) Varanasi

Shreshth Vishwakarma
24095105

June 1, 2025

1 Introduction

This project explores two approaches to the problem: a vanilla neural network implemented from scratch and a PyTorch-based neural network. The goal is to compare their performance and resource usage on the Medical Appointment No-Show Dataset.

2 Dataset and Preprocessing

2.1 Dataset Description

The dataset contains 110,527 records with 14 features, including patient demographics, appointment details, and the target variable (`No-show`).

2.2 Data Preprocessing

- Converting the date columns to datetime objects.
- Feature engineering: DaysAdvance feature tells the number of days prior appointment was booked, ScheduledDayOfWeek, AppointmentDayOfWeek, ScheduledHour.
- Label encoding for binary categorical variables like gender and neighbourhood.
- One hot encoding for ScheduledDayOfWeek, AppointmentDayOfWeek, ScheduledHour features. One hot encoding is important as neural networks incorrectly interpret higher numerical values as greater weighted than the smaller ones. This increases the number of features.
- No class balancing or data augmentation, as per the constraints of assignment.
- Data is distributed into 80% training and 20% cross validation or test set. Also ensuring that the random sampling of the data is same for both the implementation for fair comparison using random_state.

- I standardized the Age and DaysAdvance features using scikit learns StandardScaler().

3 Methodology

3.1 Neural Network from Scratch

- Architecture: Input layer (38 features), one hidden layer (8 units, ReLU), output layer (1 unit, Sigmoid).
- Forward and backward propagation implemented using NumPy.
- Loss: Binary Cross Entropy Loss (with cost sensitive learning)
- Optimization: Gradient descent with optimization strategy of mini batch GD.

3.2 PyTorch Implementation

- Same architecture as above, implemented using `torch.nn.Module`.
- Loss: Binary Cross Entropy Loss (with cost sensitive learning)
- Optimizer: Adam. But no use of Mini batch.

4 Experiments

4.1 Train/Test Split

- Training set: 80%
- Cross Validation set: 20%

4.2 Training Details

- Number of iterations: 3000
- Learning rate: 0.01
- Regularization constant: 0.01

5 Results

5.1 Convergence Time

Model	Training Time (s)	Memory Usage (MB)
From Scratch	613.70	0.56 MB
PyTorch	21.70	13.72 MB

Table 1: Convergence time and memory usage for both models.

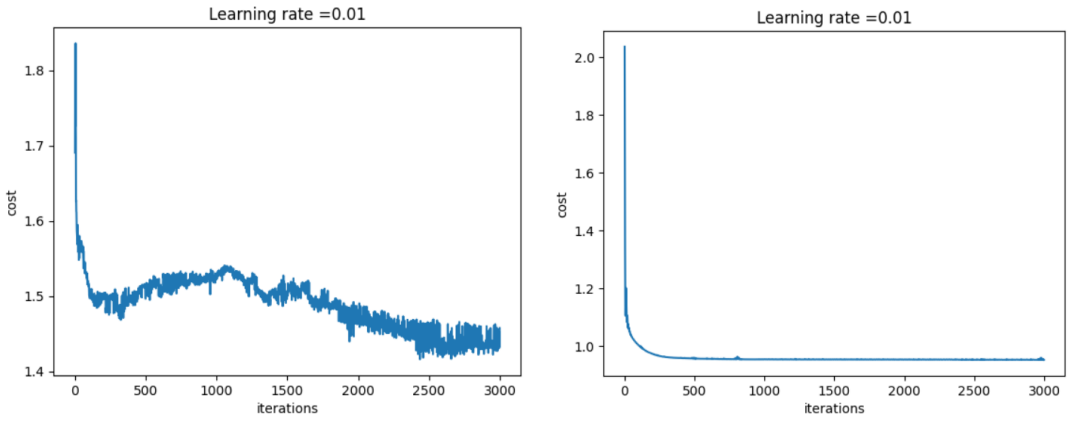

Figure 1: cost vs iteration curve (I used mini batch GD for numpy hence the curve is noisy)

The convergence speed is much faster for PyTorch implementation mainly because of GPU usage, Autograd, compute optimization. There was a bug in memory usage in NumPy. But usually PyTorch takes less memory than Numpy due to efficient tensor computation and memory reuse. And scratch implementation may not reuse memory.

5.2 Performance Metrics

Model	Accuracy	F1-score	PR-AUC
From Scratch	62.467203%	0.661662096087028	0.3500898039318638
PyTorch	62.942187%	0.665935129122393	0.3436586700418471

Table 2: Validation set performance metrics.

Training set was used to train the model, and cross validation set was used solely for evaluating performance metrics.

5.3 Confusion Matrix

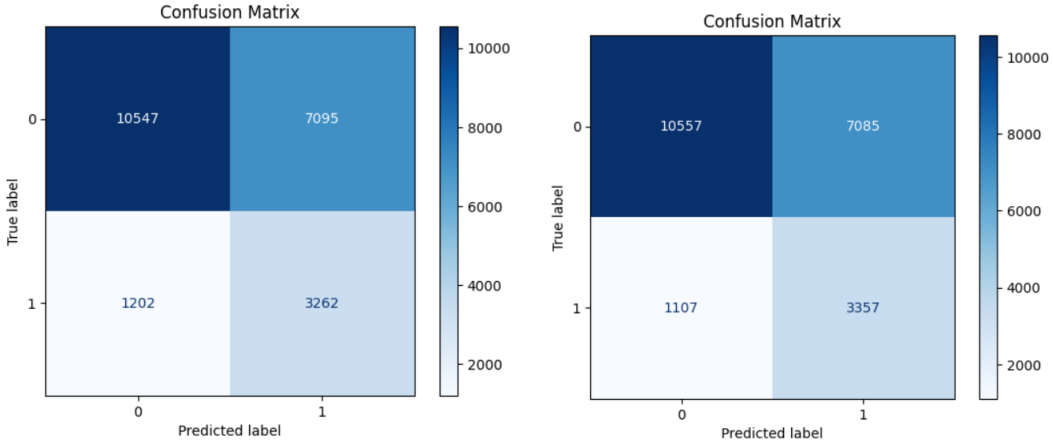

Figure 2: Confusion matrices for (left) Numpy from scratch and (right) PyTorch models.

6 Analysis and Discussion

IMPORTANT IMPLEMENTATIONS USED :

- I used Mini batch gradient descent in Numpy from scratch implementation because my training time was very high, andd implementing MBGD, it reduced significantly [ChatGPT was used for proper implementation of MBGD].
- Used stratified train test split as the dataset is imbalanced. Stratifying makes the Train and CV set in similar proportion of no-show cases, to degrade the misfunctionality caused by imbalance.
- Both the models were regularized. Also cost sensitive learning technique was used in both models to put weight to the no-show(minority) cases, to ensure their proper weight in the cost function.
- Tuned decision threshold method was used to ensure tune the threshold, rather than simply hardcoding the threshold to 0.5, as hardcoding threshold is not effective for imbalanced dataset. The tuning was meant to increase the F1 score, which resulted in degrading the accuracy of the model. The code of this part was referenced from ChatGPT.

6.1 Convergence Speed

The PyTorch model has much greater convergence speed due to several reasons like use of GPU acceleration (my implementation was on CPU), use of Autograd, and optimizers.

- GPU acceleration speeds up the computation using Nvidia CUDA.
- Autograd automatically calculates the gradients which are necessary for backpropagation, and hence makes convergence very fast.
- Optimizers: PyTorch has several optimizers. I used the Adam optimizer which updates the model's weights using optimization techniques like momentum and RMSProp.

6.2 Confusion Matrix Insights

The confusion metrics indicates that the models are not very good at performance, but seem to follow the general thing. But more proper techniques and hyperparameter tuning is needed for improved results.

7 Conclusion

- The convergence speed is much faster for PyTorch implementation mainly because of GPU usage, Autograd, compute optimization.
- The PyTorch takes less memory than Numpy due to efficient tensor computation and memory reuse. And scratch implementation may not reuse memory.
- GPU acceleration speeds up. Autograd automatically calculates the gradients. PyTorch has many options for several optimizers.

References

ChatGPT was used to implement the parts of mini batching, coding the memory usage part, using tuned decision threshold for both parts, and writing the pos_weight part of Pytorch.
Also references were taken from Andrew Ng's coursera courses.