Neural Network Implementation Report

Predicting Medical Appointment No-Shows

Club Of Programmers, IIT (BHU) Varanasi
Intelligence Guild - CSOC 2025

Submitted by: Abhyudaya Singh
Roll Number: 24065008

June 28, 2025

Contents

Chapter 1

Introduction

As part of the COPS Summer of Code 2025 Intelligence Guild track, I tackled the challenge of predicting patient no-shows for medical appointments using the Medical Appointment No-Show Dataset. The dataset, with 110,527 appointments and a 20.2% no-show rate, presented a significant class imbalance. My task was to implement two neural networksone from scratch using only NumPy and core Python, and another using PyTorchand compare their performance. The goal was not just to achieve high accuracy but to deeply understand neural network mechanics and effectively handle the imbalanced dataset without using oversampling or undersampling, as specified in the assignment guidelines.

This project was a journey of balancing theoretical learning with practical implementation. I aimed to craft models that could reliably identify no-shows, which is critical for optimizing healthcare resources, while gaining insights into the trade-offs between custom-built and framework-based solutions.

Chapter 2

Methodology

My approach centered on designing two neural networks with identical architectures to ensure a fair comparison, focusing on handling the datasets 3.95:1 class imbalance (79.8% shows vs. 20.2% no-shows).

2.1 Data Preparation

The dataset included 7 features: Age, Scholarship, Hypertension, Diabetes, Alcoholism, Handicap, and SMS received. I split the data into training (60%), validation (20%), and test (20%) sets, maintaining the class distribution via stratification. Features were standardized using StandardScaler to ensure numerical stability. Per the assignments constraints, I avoided any data augmentation techniques to balance classes.

2.2 Model Architecture

Both models used a 6-layer Multi-Layer Perceptron (MLP) with the configuration 7 128 256 128 64 32 1, totaling 89,153 parameters. Key components included:

- **Activations**: ReLU for hidden layers, Sigmoid for the output to produce probabilities.
- **Regularization**: Batch normalization and dropout (rates: 0.3, 0.4, 0.3, 0.2, 0.1) to prevent overfitting.
- **Initialization**: He initialization to stabilize training with ReLU activations.

2.3 Scratch Implementation (FocalMLP)

Building from scratch was like assembling a car from raw partsit demanded a deep understanding of every component. I implemented:

- **Custom Layers**: Fully connected layers using NumPy, with manual forward and backward passes.
- **Focal Loss**: To address class imbalance, I used focal loss with α calculated from class frequencies (0.798 for shows, 0.202 for no-shows) and $\gamma = 2$ to focus on hard examples.

- **Optimization**: Stochastic Gradient Descent (SGD) with gradient clipping (max norm 5.0) to prevent exploding gradients.
- **Early Stopping**: Training halted if validation loss didnt improve after 15 epochs.

This approach required meticulous coding of backpropagation and batch normalization, offering unparalleled insight into neural network mechanics.

2.4 PyTorch Implementation (ImbalancedMLP)

Using PyTorch felt like driving a pre-built sports carfast and efficient. I leveraged:

- **PyTorch Modules**: `nn.Linear`, `nn.BatchNorm1d`, and `nn.Dropout` for layers.
- **Weighted BCE Loss**: Class weights (3.952 for no-shows) were computed using `compute_class_we` to penalize misclassifications of the minority class.
- **Optimization**: Adam optimizer with a learning rate of 0.001, weight decay of 0.0001, and a ReduceLROnPlateau scheduler (patience=8, factor=0.5).
- **Threshold Tuning**: Post-training, I optimized the classification threshold (0.49) on the validation set to maximize the F1-score.
- **GPU Acceleration**: Training on CUDA for faster computation.

2.5 Handling Class Imbalance

Class imbalance was the heart of the challenge. For the scratch model, focal loss down-weighted easy examples, emphasizing no-shows. For PyTorch, weighted BCE increased the penalty for misclassifying no-shows. Both approaches avoided data augmentation, relying solely on loss function modifications to address the 3.95:1 imbalance ratio.

Chapter 3

Results and Analysis

I evaluated both models on the test set (22,106 samples) using accuracy, F1-score, precision-recall AUC (PR-AUC), precision, recall, and confusion matrices, as required.

3.1 Performance Metrics

Table 3.1: Performance Metrics Comparison

Metric	Scratch	PyTorch	Improvement	% Improvement
Accuracy	0.6197	0.6359	+0.0162	+2.6%
F1-Score	0.3477	0.4395	+0.0918	+26.4%
PR-AUC	0.2619	0.3570	+0.0951	+36.3%
Precision	0.2660	0.3189	+0.0529	+19.9%
Recall	0.5020	0.7070	+0.2050	+40.8%

The PyTorch model outperformed the scratch model across all metrics, with significant gains in recall (+40.8%) and PR-AUC (+36.3%), critical for imbalanced datasets where identifying no-shows is paramount.

3.2 Confusion Matrix Analysis

Table 3.2: Confusion Matrix Comparison

Implementation	TN	FP	FN	TP	Total
Scratch	11,458	6,184	2,223	2,241	22,106
PyTorch	10,901	6,741	1,308	3,156	22,106

The PyTorch model caught 915 more true positives (3,156 vs. 2,241) and missed fewer no-shows (1,308 vs. 2,223 false negatives), making it more effective for clinical applications where missing no-shows is costly.

Table 3.3: Resource Utilization Comparison

Metric	Scratch	PyTorch
Training Time	823.6s (13.7 min)	711.7s (11.9 min)
Convergence Epochs	174	27
Memory Usage	93.2 MB	1,426.3 MB
Model Size	0.60 MB	0.34 MB
Hardware	CPU	CUDA (GPU)

3.3 Computational Efficiency

PyTorch converged in fewer epochs (27 vs. 174) and trained faster, likely due to the Adam optimizer and GPU acceleration. However, it consumed significantly more memory, reflecting framework overhead and GPU usage. The scratch models lower memory footprint (93.2 MB) is due to its minimalistic NumPy-based design, but CPU limitations slowed training.

3.4 Why PyTorch Outperformed

- **Optimization**: Adams adaptive learning rates likely accelerated convergence compared to SGD.
- **GPU Acceleration**: CUDA enabled faster matrix operations.
- **Numerical Stability**: PyTorchs optimized implementations reduced numerical errors.
- **Threshold Tuning**: The 0.49 threshold improved F1-score and recall.

Chapter 4

Challenges and Insights

4.1 Implementation Challenges

The scratch model required careful debugging of backpropagation and batch normalization, with numerical stability issues posing hurdles. PyTorch demanded learning framework conventions and GPU memory management but was easier to debug due to built-in tools.

4.2 Key Insights

- **Scratch Learning**: Coding from scratch deepened my understanding of backpropagation, loss functions, and normalization, but it was time-intensive.
- **PyTorch Efficiency**: PyTorchs tools simplified development and improved performance, especially for imbalanced data.
- **Imbalance Handling**: Focal loss and weighted BCE effectively addressed the 3.95:1 imbalance, with PyTorchs threshold tuning boosting recall.
- **Metric Choice**: F1-score and PR-AUC were more informative than accuracy for this imbalanced problem.

Chapter 5

Confession

In the spirit of transparency, I acknowledge that I used AI assistance during the development of this project. Specifically, I utilized ChatGPT to generate a code snippet for implementing the weighted Binary Cross Entropy (BCE) loss logic in the PyTorch model. This snippet helped me efficiently set up the loss function to handle the class imbalance. For the scratch implementation, I used PerplexityPro to generate the code logic for updating weights, including the implementation of focal loss and backpropagation, to ensure correctness and efficiency. The use of AI was limited to these specific components, while the core learning, model design, and implementation efforts were my own.

Chapter 6

Conclusion

This project was a profound learning experience, blending hands-on coding with theoretical depth. The PyTorch models superior performance (70.7% recall vs. 50.2%) highlights the power of optimized frameworks for real-world applications like no-show prediction, where catching more no-shows can optimize healthcare delivery. The scratch model, while less efficient, provided invaluable insights into neural network mechanics.

6.1 Reflection

Balancing the scratch models educational value with PyTorchs practical advantages was enlightening. The 26.4% F1-score improvement and 40.8% recall boost in PyTorch underscore the importance of framework optimizations, while the scratch implementation solidified my understanding of deep learning fundamentals. This dual approach, combined with selective AI assistance, has equipped me to tackle complex machine learning challenges with both insight and efficiency.

Repository: 24065008-CSOC-IG