Neural Network Implementation
Medical Appointment No-Show Prediction

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

Submitted by:
Abhyudays Singh
Roll Number: 24065008

June 25, 2025

Contents

Chapter 1

Introduction

This report presents a comparative analysis of two neural network implementations for predicting patient no-shows in medical appointments. As part of the COPS Summer of Code 2025 Intelligence Guild track, I implemented and compared two approaches:

1. A neural network built from scratch using only NumPy and core Python libraries
2. An equivalent implementation using the PyTorch deep learning framework

The primary objective was to understand the fundamental differences between building neural networks from scratch versus using established frameworks, particularly in the context of handling imbalanced datasets where the minority class (no-shows) represents only 20.19% of the data.

1.1 Dataset Overview

The Medical Appointment No-Show Dataset contains 110,527 appointments with the following characteristics:

- **Total samples:** 22,106 (test set)
- **Class distribution:**
  - Shows: 17,642 (79.81%)
  - No-shows: 4,464 (20.19%)
- **Imbalance ratio:** 3.95:1
- **Features:** 7 input features (Age, Scholarship, Hypertension, Diabetes, Alcoholism, Handicap, SMS received)

This significant class imbalance posed a key challenge that both implementations needed to address effectively.

Chapter 2

Methodology

2.1 Model Architecture

Both implementations used identical neural network architectures to ensure fair comparison:

- **Layer configuration:** $7 \to 128 \to 256 \to 128 \to 64 \to 32 \to 1$
- **Activation functions:** ReLU for hidden layers, Sigmoid for output
- **Regularization:** Batch normalization and dropout
- **Weight initialization:** He initialization for ReLU networks
- **Total parameters:** 89,153

2.2 Scratch Implementation

The from-scratch implementation (`FocalMLP`) included:

- Custom batch normalization implementation
- Focal loss with alpha parameter used to upweight the minority class and gamma to reduce the loss contribution from easy examples
- Custom SGD optimizer with gradient clipping
- Early stopping prevented overfitting by monitoring validation performance and halting training when no further improvements were observed
- Dropout implementation with custom mask generation

2.3 PyTorch Implementation

The PyTorch implementation (`ImbalancedMLP`) featured:

- PyTorch's built-in `nn.Linear`, `nn.BatchNorm1d`, and `nn.Dropout` layers
- Weighted Binary Cross Entropy loss with computed class weights

- Adam optimizer with learning rate scheduling
- Early stopping with patience mechanism
- Post-training threshold tuning was conducted using the validation set to select a probability threshold (0.49) that maximized the F1 score, rather than using the default 0.5

2.4 Handling Class Imbalance

Both implementations addressed the class imbalance problem but used different approaches:

2.4.1 Scratch Implementation

- Implemented focal loss with alpha-balancing
- Alpha parameter calculated based on class frequencies
- Gamma parameter set to 2.0 for focal weighting

2.4.2 PyTorch Implementation

- Used `compute_class_weight` to calculate balanced class weights
- Applied a class weight of 3.952 for the positive class (no-show) based on inverse class frequency to counteract the imbalance
- Optimized classification threshold (0.49 instead of default 0.5)

Chapter 3

Results and Analysis

3.1 Performance Comparison

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 implementation consistently outperformed the scratch implementation across all metrics, with particularly significant improvements in recall (+40.8%) and PR-AUC (+36.3%).

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

Key insights from the confusion matrices:

- **PyTorch model** identified 915 more true positives (no-shows caught)
- **PyTorch model** reduced false negatives by 915 cases (missed no-shows)
- The PyTorch model's superior recall is crucial for the medical appointment domain where identifying potential no-shows is more important than minimizing false alarms

3.3 Computational Efficiency

Table 3.3: Resource Utilization Comparison

Metric	**Scratch**	**PyTorch**
Training Time	823.6s (13.71 min)	711.68s (11.86 min)
Convergence Epochs	174 epochs	27 epochs
Memory Usage	0.0932 MB	1426.3 MB
Model Size	0.60 MB	0.34 MB
Hardware	CPU	CUDA (GPU)

The PyTorch implementation converged significantly faster—requiring only 27 epochs compared to 174 for the scratch model—thanks to its optimized training routines and GPU acceleration. Despite the scratch model using minimal memory (0.0932 MB vs. 1426.3 MB), its reliance on CPU-based NumPy operations resulted in longer overall training time. The difference in computational efficiency is thus attributable more to hardware utilization and framework-level optimizations than to architectural differences between the models.

Chapter 4

Key Learnings and Insights

4.1 Framework Benefits

4.1.1 PyTorch Advantages

- **Optimized backend:** Highly optimized C++ and CUDA implementations
- **GPU acceleration:** Seamless GPU utilization for faster training
- **Built-in components:** Robust, tested implementations of common layers
- **Advanced optimizers:** Access to state-of-the-art optimization algorithms
- **Automatic differentiation:** Efficient gradient computation

4.1.2 Scratch Implementation Benefits

- **Deep understanding:** Forces comprehension of underlying mathematics
- **Flexibility:** Complete control over implementation details
- **Custom optimizations:** Ability to implement domain-specific improvements
- **Educational value:** Better understanding of neural network mechanics

4.2 Handling Imbalanced Data

Both implementations successfully addressed class imbalance, but with different strategies:

- **Loss function modification:** Both used weighted approaches (focal loss vs. weighted BCE)
- **Threshold optimization:** PyTorch implementation benefited from threshold tuning
- **Metric selection:** Focus on F1 score and PR-AUC rather than accuracy
- **Class weight calculation:** Automatic computation based on class frequencies

4.3 Performance Insights

- **Recall improvement:** PyTorch's 40.8% improvement in recall is clinically significant
- **Framework optimization:** PyTorch's optimized implementations provide measurable benefits
- **Threshold tuning:** Simple threshold optimization significantly improved performance
- **Early stopping:** Both implementations benefited from preventing overfitting

4.4 Technical Trade-offs

Table 4.1: Implementation Trade-offs

Aspect	**Scratch Implementation**	**PyTorch Implementation**
Development Time	High (custom everything)	Low (leverage existing components)
Performance	Lower (NumPy limitations)	Higher (optimized backend)
Understanding	Deep mathematical insight	Framework abstraction
Flexibility	Complete control	Limited by framework design
Debugging	Challenging (custom code)	Easier (established tools)
Scalability	Limited	Production-ready
Maintenance	High effort	Lower effort

Chapter 5

Challenges and Solutions

5.1 Class Imbalance Challenges

- **Imbalance ratio:** Approximately 3.95:1 (i.e., for every no-show, there are roughly four show-up instances)
- **Solution:** Implemented class weighting and focal loss mechanisms
- **Result:** Both models achieved reasonable recall for minority class

5.2 Implementation Challenges

5.2.1 Scratch Implementation

- Custom batch normalization implementation complexity
- Gradient computation and backpropagation debugging
- Numerical stability issues with custom implementations

5.2.2 PyTorch Implementation

- Understanding framework-specific conventions
- Proper GPU memory management
- Learning rate scheduling configuration

Chapter 6

Conclusion

This comparative study demonstrates the significant advantages of using established deep learning frameworks like PyTorch while highlighting the educational value of implementing neural networks from scratch.

6.1 Key Findings

1. **Performance:** PyTorch implementation achieved superior results across all metrics, with particularly notable improvements in recall (40.8%) and PR-AUC (36.3%)
2. **Efficiency:** Framework optimizations and GPU acceleration provide substantial computational benefits
3. **Development:** While PyTorch accelerated development and improved performance, the scratch implementation was invaluable for learning core concepts such as backpropagation, normalization, and loss computation.
4. **Understanding:** Scratch implementation provides invaluable insights into neural network mechanics and mathematics

6.2 Practical Implications

For medical appointment no-show prediction:

- The PyTorch model's higher recall (70.7% vs 50.2%) means it would catch 915 more potential no-shows
- This improvement could significantly impact healthcare resource allocation and patient care
- The model's ability to identify no-shows enables proactive interventions

6.3 Future Work

- Implement ensemble methods combining both approaches
- Explore advanced architectures (attention mechanisms, residual connections)

- Investigate other imbalanced learning techniques (SMOTE, cost-sensitive learning)
- Deploy models for real-world medical appointment systems

6.4 Final Reflection

Building neural networks from scratch provided deep insights into the mathematical foundations of deep learning, while the PyTorch implementation demonstrated the power of modern frameworks. The 26.4% improvement in F1 score achieved by the PyTorch implementation validates the benefits of using optimized frameworks for practical applications, especially when handling challenging problems like imbalanced datasets.

This project reinforced the importance of:

- Choosing appropriate metrics for imbalanced problems
- Understanding the trade-offs between custom implementations and frameworks
- The critical role of proper handling of class imbalance in real-world applications
- The value of both theoretical understanding and practical implementation skills

The complete implementation code and detailed analysis are available in the project repository: 24065008-CSOC-IG