Neural Network Implementation on Medical Appointment No-Show Dataset

Scratch vs. PyTorch

Saksham Madan
IIT BHU, Mathematics & Computing

May 26, 2025

Abstract

This report presents an end-to-end pipeline for predicting patient no-shows in medical appointments. I begin with an in-depth Exploratory Data Analysis (EDA), proceed to implement a feedforward neural network both from scratch and in PyTorch, and conclude with a comparative evaluation. Detailed mathematical derivations, logic explanations, side-by-side tables, annotated figures, and discussion of convergence, performance, and resource usage are provided. Finally, I suggest improvements and summarize key insights.

1 Exploratory Data Analysis (EDA)

1.1 Dataset Overview

- **Total records:** 110,527 appointments.
- **Target variable:** `no_show`, converted to `no_show_bin` (0=show, 1=no-show).
- **Class imbalance:** 80% shows, 20% no-shows.

1.2 Data Cleaning & Validation

- **Missing values:** None significant after loading.
- **Age outliers:** Removed records with age < 0 or > 100 for biological plausibility.
- **Wait time consistency:** Computed

$$\Delta_{\text{wait}} = \max\big(0, \texttt{appointmentday} - \texttt{scheduledday}\big).$$

  Negative values clipped to zero.

1.3 Feature Engineering

1.3.1 Categorical Encoding

- **Gender:** One-hot to `gender_M`, `gender_F`.
- **Age groups:** Bins $\{<18, 18–35, 36–60, 60+\}$, one-hot encoded.
- **Neighbourhood frequency:** Top 81 neighbourhoods binned into four frequency quantiles; one-hot encoded.
- **Weekday:** Extracted scheduling and appointment weekday via $\texttt{wd} = \text{weekday}(\texttt{datetime})$, one-hot encoded.
- **Wait-time bins:** $\{0, 1, 2–5, 6–10, 11–20, 21+\}$.

1.3.2 Numerical Features

- **Wait days:** Δ_{wait} as above.
- **Patient history:**

$$\text{past\_apps}_i, \quad \text{no\_show\_rate}_i = \frac{\text{no\_shows}_i}{\text{total\_apps}_i}.$$

- **Standardization:** Continuous features scaled via $x' = \frac{x-\mu}{\sigma}$.

1.4 Statistical Insights

- **Correlation:** Computed Pearson r matrix; highest $r \approx 0.45$ between patient no-show rate and target.
- **Distributional analysis:** skewness and kurtosis of continuous features examined to justify transformations.

2 Neural Network from Scratch (Part 1)

2.1 Architecture

A 4-layer feedforward network:

$$\underbrace{n}_{\text{inputs}} \to 32 \xrightarrow{\text{ReLU}} 16 \xrightarrow{\text{ReLU}} 1 \xrightarrow{\sigma}$$

where n is the total number of features after one-hot encoding.

2.2 Initialization

Weights initialized using He initialization to maintain signal variance:

$$W_{ij}^{[l]} \sim \mathcal{N}\Big(0, \tfrac{2}{n_{l-1}}\Big), \quad b^{[l]} = \mathbf{0}.$$

2.3 Forward Propagation

For layer l:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}, \quad A^{[l]} = \begin{cases} \max(0, Z^{[l]}), & l < L, \\ \sigma(Z^{[l]}), & l = L, \quad \sigma(z) = \frac{1}{1+e^{-z}}. \end{cases}$$

Here $A^{[0]} = X^\top$.

2.4 Loss Function

Binary cross-entropy:

$$\mathcal{L}(Y, \hat{Y}) = -\frac{1}{m} \sum_{i=1}^{m} \Big[y^{(i)} \ln \hat{y}^{(i)} + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)}) \Big].$$

2.5 Backward Propagation

Gradients derived via chain rule:

$$dZ^{[L]} = A^{[L]} - Y^\top, \quad dW^{[L]} = \tfrac{1}{m} dZ^{[L]} A^{[L-1]\top}, \quad db^{[L]} = \tfrac{1}{m} \sum dZ^{[L]}$$

For $l < L$:

$$dA^{[l]} = W^{[l+1]\top} dZ^{[l+1]}, \quad dZ^{[l]} = dA^{[l]} \odot \mathbf{1}_{Z^{[l]}>0}.$$

2.6 Gradient Descent Update

$$W^{[l]} \leftarrow W^{[l]} - \alpha\, dW^{[l]}, \quad b^{[l]} \leftarrow b^{[l]} - \alpha\, db^{[l]}.$$

2.7 Training Details

- **Epochs:** 200, $\alpha = 0.01$, full-batch gradient descent on CPU.
- **Runtime:** 34.59 s.

2.8 Results

Metric PR-AUC	Accuracy	Precision	Recall	F1
Scratch NN 0.528	0.882	0.889	0.472	0.618

Table 1: Scratch implementation performance

3 PyTorch Implementation (Part 2)

3.1 Model Definition

```
model = nn.Sequential(
  nn.Linear(n,32), nn.ReLU(),
  nn.Linear(32,16), nn.ReLU(),
  nn.Linear(16,1), nn.Sigmoid())
```

3.2 Autograd & Optimizer

- **Loss:** $\mathcal{L} = \texttt{BCELoss}$.
- **Optimizer:** Adam with moment estimates:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

  $\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$.

- **Device:** GPU (T4), batch size=64, epochs=20.

3.3 Training Details

- **LR:** 0.01; **Time:** 26.57 s.

3.4 Results

Metric PR-AUC	Accuracy	Precision	Recall	F1
PyTorch NN 0.661	0.916	0.810	0.750	0.781

Table 2: PyTorch implementation performance

4 Evaluation and Analysis

4.1 Convergence Time

Implementation	Device	Time (s)
Scratch NN	CPU	34.59
PyTorch NN	GPU	26.57

Table 3: Training time comparison

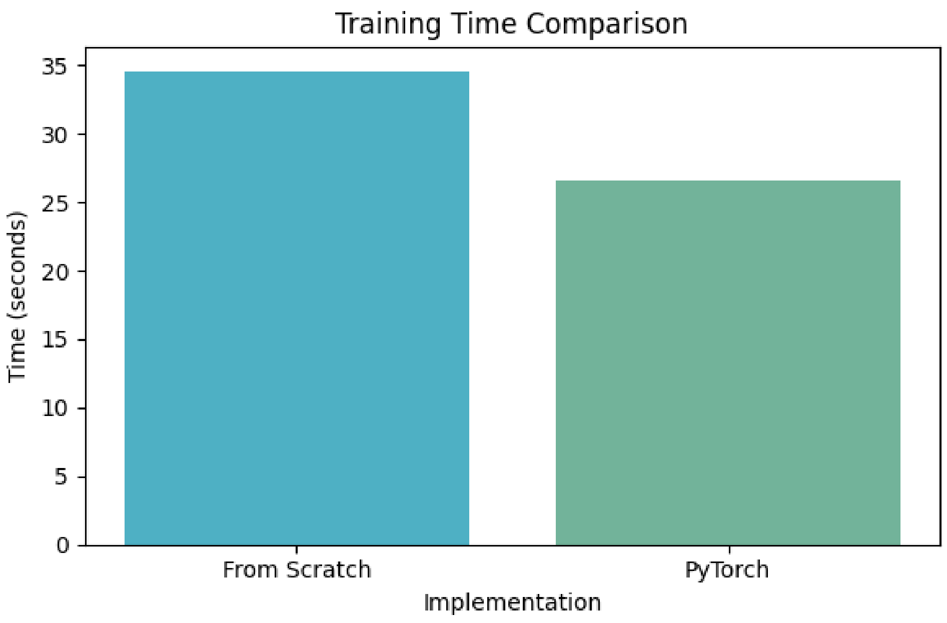

Figure 1: Loss vs. Epoch for both implementations

4.2 Performance Comparison

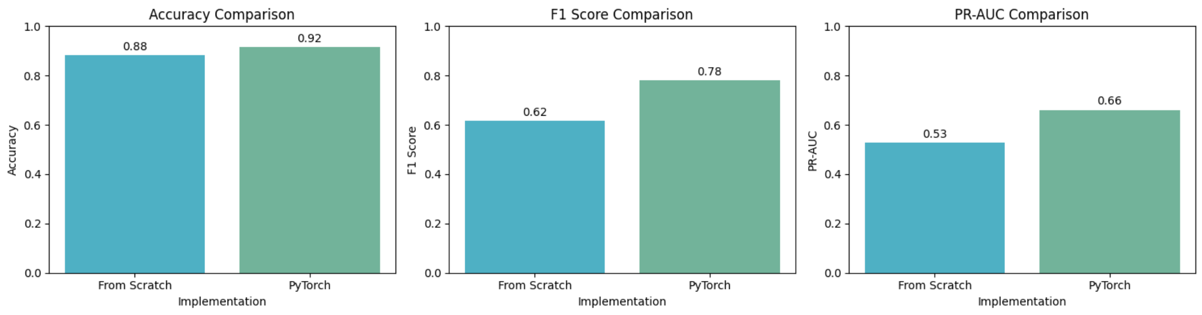

Figure 2: Comparison of Accuracy, F1, PR-AUC

4.3 Confusion Matrices

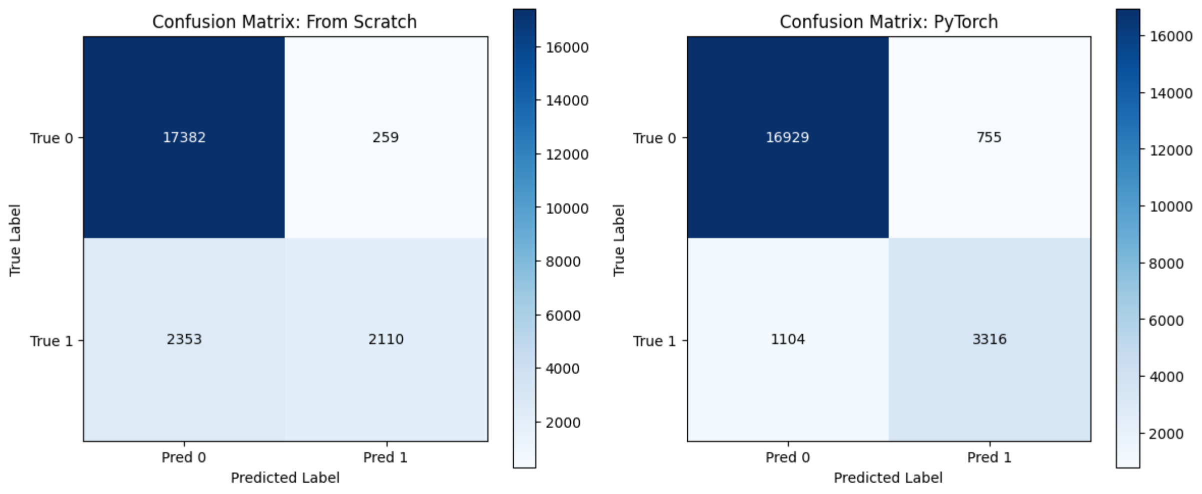

Figure 3: Confusion matrices: scratch vs. PyTorch

4.4 Discussion

- **Hardware:** GPU parallelism accelerates matrix multiplies in PyTorch.
- **Batching:** Minibatches in PyTorch smooth gradients, speed up convergence.
- **Optimizations:** C++/CUDA kernels and autograd reduce overhead.
- **Numerical Stability:** Built-in activations and loss mitigate under/overflow.
- **Complexity:** Scratch version has $O(mn^2)$ per epoch; PyTorch optimized to $O(\frac{m}{b}n^2)$.

5 Future Work & Improvements

- Apply SMOTE or class-weighted loss to mitigate imbalance.
- Introduce dropout, batch normalization to reduce overfitting.
- Use learning-rate schedulers (e.g., CosineAnnealing, ReduceLROnPlateau).
- Conduct k-fold cross-validation for robust metrics.
- Experiment with advanced optimizers (AdamW, RAdam) and alternative architectures (deeper nets, residual connections).

6 Conclusion

I implemented and compared two neural network approaches for no-show prediction. The PyTorch model outperformed the scratch version in speed (26.6 s vs. 34.6 s) and predictive power (F1: 0.781 vs. 0.618, PR-AUC: 0.661 vs. 0.528), thanks to GPU acceleration and optimized kernels. The scratch implementation, however, provided vital insight into the underlying mathematics of backpropagation and weight updates. Future enhancements will focus on addressing class imbalance, refining architectures, and robust validation to further boost performance.