Neural Network Implementation on Medical Appointment No-Show Dataset: From-Scratch and PyTorch Comparison

Vaishnavi
Roll No: 24124048
Department of Mathematics and Computing
IIT BHU

Abstract

This report presents two implementations of a neural network model for predicting patient no-shows in medical appointments. The first model is built from scratch using NumPy, and the second is implemented using PyTorch. We evaluate both models on accuracy, F1-score, PR-AUC, convergence time, and memory usage, and provide an in-depth analysis of their performance, convergence behavior, and architectural efficiency. The study highlights the trade-offs between manual implementation and framework-based development in applied machine learning.

1 Exploratory Data Analysis

A detailed EDA was conducted to clean the data, engineer meaningful features, and prepare the dataset for modeling. The key steps and motivations behind them are summarized below:

Data Cleaning and Filtering

- **Removal of Invalid Ages:** Records with negative ages and those exceeding 100 years were identified using distribution plots and descriptive statistics. These entries were considered either erroneous or outliers, and were therefore removed to maintain data integrity.
- **Dropping Redundant Columns:** Columns such as `AppointmentID`, `ScheduledDay`, and `AppointmentDay` were dropped, as they were either irrelevant or replaced by more meaningful engineered features.

Feature Engineering

- **Waiting Time:** A new feature named `WaitingTime` was created by computing the difference (in days) between the scheduled date and the appointment date. This variable captures how long a patient had to wait for their appointment — a potential behavioral factor in determining attendance.
- **Day of the Week:** A new categorical feature named `AppointmentWeekday` was created by extracting the weekday from the appointment date. Analysis revealed trends such as increased no-shows on certain weekdays (e.g., Mondays), suggesting behavioral or logistical patterns. This feature was used as an input to the model after encoding.
- **Neighbourhood Filtering:** The `Neighbourhood` feature was analyzed in terms of the no-show percentage per area. To avoid noise and unreliable statistics, neighborhoods with very few total appointments were removed from the dataset. This ensured that neighborhood-level no-show trends were statistically significant.

Encoding and Target Processing

- **Frequency Encoding:** The `Neighbourhood` column, being high-cardinality and categorical, was frequency-encoded. This method preserved the relative occurrence of each category without adding dimensionality.
- **Binary Target Encoding:** The `No-show` target variable was transformed into binary form: 1 for 'Yes' (no-show) and 0 for 'No' (showed up), aligning with the binary classification task.

Scaling and Final Preparation

- **Numerical Scaling:** Continuous features like `Age`, `Neighbourhood_freq_enc`, and `WaitingTime` were scaled using normalization techniques to improve neural network convergence and numerical stability.
- **Train-Test Split:** The dataset was split into training and validation sets, preserving the natural imbalance of the target variable as per assignment constraints. No oversampling, undersampling, or augmentation techniques were applied.

Additional EDA Insights

- **Target Distribution:** A strong class imbalance was observed in the `No-show` variable, with only about 20% representing no-shows. This was visualized using bar and pie charts.

- **Age and Waiting Time Distributions:** Univariate analyses using histograms, boxplots, and KDE plots revealed natural age distributions and outliers. KDE plots showed that patients with longer waiting times had a higher probability of missing appointments.
- **Categorical Feature Distributions:** Features such as `Gender` and `Scholarship` were examined via bar and pie plots. These helped in understanding the data balance across categories.
- **Weekday vs. No-show Trends:** Countplots indicated that no-shows were more common on Mondays. Additionally, very few appointments were scheduled for Saturdays and none for Sundays, which influenced scheduling patterns.
- **Patient Behavior Consideration:** The `PatientId` column was initially considered to extract behavioral patterns, such as frequent no-shows by specific individuals. However, incorporating such longitudinal behavior required temporal sequencing and patient history modeling, which was beyond the scope of this assignment. Hence, the column was ultimately dropped.

Conclusion of EDA

The EDA process enabled us to identify and correct data inconsistencies (e.g., negative ages), engineer informative features (e.g., `WaitingTime`, `AppointmentWeekday`), and prepare the dataset for modeling through encoding and scaling. Behavioral trends like weekday effects and waiting time sensitivity were uncovered. Although `PatientId` had potential to reveal individual-level behavior, it was excluded due to the complexity of modeling longitudinal data within the given scope.

2 Neural Network Implementation and Evaluation

2.1 Overview

This report compares two implementations of a neural network model trained on the Medical Appointment No-Show dataset:

1. A neural network built from scratch using core Python and NumPy
2. An equivalent neural network built using PyTorch

Both models were trained without any data balancing techniques such as oversampling or undersampling.

2.2 Architecture

Both models share the same architecture:

- Input layer: Matching the number of preprocessed features
- Hidden layer: 2 hidden layer with 32, 16 neurons, ReLU activation
- Output layer: 1 neuron with Sigmoid activation for binary classification

2.3 Training Configuration

- Loss Function: Binary Cross Entropy
- Optimizer: Gradient Descent (manual in scratch, SGD in PyTorch)
- Learning Rate: Tuned around 0.01 for scratch, 0.01 for PyTorch
- Epochs: 1000
- Batch size: Full-batch in scratch; mini-batch in PyTorch

2.4 Evaluation Metrics

From-Scratch Implementation

- Accuracy: 57.80%
- F1-Score: 0.37
- **Confusion Matrix:**

9997	7685
1642	2778

PyTorch Implementation

- Accuracy: 59.69%
- F1-Score: 0.43
- PR-AUC: 0.35
- **Confusion Matrix:**

9708	7974
935	3485

2.5 Convergence Time

- From-scratch model: Training took approx. 130 seconds
- PyTorch model: Significantly faster due to GPU-acceleration and optimized backend 25 seconds

2.6 Detailed Observations and Analysis

Class Imbalance Handling and Threshold Selection

The Medical No-Show dataset suffers from significant class imbalance: the majority of samples correspond to patients who showed up (label = 0), while the minority represent no-shows (label = 1). This imbalance posed a challenge for training both the from-scratch and PyTorch models.

From-Scratch Model:

- An attempt was made to handle class imbalance by manually modifying the binary cross-entropy loss function. Specifically, a weighting parameter $\alpha > 1$ was introduced to assign more penalty to misclassified class 1 (no-show) samples.
- However, due to the lack of native support for automatic differentiation and fine-grained monitoring of gradient flow, tuning this parameter was experimentally difficult. As a result, although the option to use weighted loss was implemented (commented in the code), the default unweighted version was used for final results.

PyTorch Model:

- PyTorch's `BCEWithLogitsLoss` was used along with the `pos_weight` parameter to handle class imbalance. This allows weighting the loss contribution of positive (minority) samples, encouraging the model to give more attention to the underrepresented class.
- The use of weighted loss in PyTorch significantly improved the recall and F1-score for class 1, as shown in the confusion matrix. This resulted in a more balanced classification performance.

Threshold Tuning:

- The default decision threshold of 0.5 (used to convert predicted probabilities to class labels) was suboptimal, especially in the presence of class imbalance.
- To improve recall for class 1, the threshold was manually reduced to 0.2 in both models. This decision was based on empirical observation that many valid positive predictions had probability scores between 0.2 and 0.5.
- Lowering the threshold increased true positive detection (reducing false negatives), which is critical for imbalanced classification. This adjustment notably improved the F1-score in the PyTorch implementation, without drastically lowering overall accuracy.

Overall, both weighted loss and threshold tuning proved effective strategies for improving classification on imbalanced data, especially in the PyTorch model due to its flexible and robust loss API.

Performance Comparison

- The **from-scratch model** achieved an accuracy of 57.80% and an F1-score of 0.37. Although it had a lower accuracy, it demonstrated a reasonably balanced tradeoff between precision and recall as seen in its confusion matrix.
- The **PyTorch model** performed slightly better with an accuracy of 59.69% and a higher F1-score of 0.43. Its confusion matrix shows better recall for the positive class due to the use of the weighted loss function and adjusted threshold.
- The PR-AUC for the PyTorch model (0.35) also reflects a more informative precision-recall tradeoff compared to earlier unweighted runs where PR-AUC was negligible due to complete class imbalance in predictions.

Confusion Matrix Interpretation

- In the **from-scratch implementation**, the model predicted:
  - 9997 true negatives
  - 7685 false positives
  - 1642 false negatives
  - 2778 true positives

  Despite lower overall accuracy, this indicates a moderate ability to detect no-show cases (class 1).
- In the **PyTorch model**, the confusion matrix shows:
  - 9708 true negatives
  - 7974 false positives
  - **only 935 false negatives**
  - 3485 true positives

  This shows that the PyTorch model was better at capturing true positives, indicating improved recall.

Convergence and Efficiency

- The **from-scratch model** took approximately 130 seconds to train for 1000 epochs. This slower training is expected due to the absence of GPU acceleration, and lack of optimized numerical computation.
- The **PyTorch model** trained in approximately 25 seconds for the same number of epochs. This reflects PyTorch's performance optimizations through efficient tensor operations and GPU acceleration.

- Additionally, the PyTorch model required fewer lines of code for gradient computation and weight updates due to its automatic differentiation engine, resulting in clearer, more maintainable code.

2.7 Memory Usage

- **From-Scratch Model:** Utilizes memory efficiently, with explicit memory control via NumPy. However, the absence of hardware acceleration and limited vectorization in backpropagation results in slower training.
- **PyTorch Model:** Exhibits higher memory usage due to dynamic computation graphs and mini-batching but benefits from optimized tensor operations and automatic differentiation for faster computation.

2.8 Confusion Matrix Interpretation

From-Scratch Implementation

- The model demonstrates relatively balanced detection across both classes.
- It achieves **higher recall for the minority class** (No-Show), indicating it is more sensitive to identifying actual no-shows, though at the cost of false positives.

PyTorch Implementation

- After applying **threshold tuning (0.2 instead of 0.5)** and a **weighted loss function**, the model achieved significantly better F1-score and PR-AUC.
- The confusion matrix reflects improved minority class detection compared to default thresholding.

2.9 Discussion

Performance Comparison:

- **From-Scratch Model:** Delivered more balanced performance in terms of F1-score (0.37) and robustness to class imbalance, albeit with lower accuracy (57.80%). Offers full transparency in how forward and backward passes operate.
- **PyTorch Model:** Slightly better accuracy (59.69%) and significantly faster training (25 seconds vs. 130 seconds). Initially underperformed due to default thresholding and lack of class weighting. However, applying:
  - Weighted binary cross-entropy
  - Custom threshold (0.2) for classification

improved its effectiveness in identifying no-shows, as evidenced by a better F1-score (0.43) and PR-AUC (0.35).

Advantages of PyTorch:

- Rapid prototyping with modular API
- GPU acceleration and optimized tensor operations
- Easier experimentation with loss functions, optimizers, and thresholds

Advantages of From-Scratch Implementation:

- Provides in-depth understanding of neural network internals
- Total control over training logic, gradient flow, and numerical precision
- Useful for learning purposes and algorithmic transparency

2.10 Conclusion

Both models exhibit different strengths:

- The **from-scratch model** performed well in identifying minority class instances due to careful threshold tuning and control over loss weighting.
- The **PyTorch model**, while initially underperforming, demonstrated that correct use of class-weighted loss and tuned thresholds can significantly enhance performance.

Overall, the scratch implementation achieved better balance across F1 and PR-AUC metrics, while the PyTorch model showed better convergence speed and scalability. These observations emphasize the trade-off between model transparency and engineering efficiency.