PyTorch Neural Network for Predicting Medical Appointment No-Shows

Abhishek Kumar Chaubey
Roll Number: 24144001
IIT (BHU) Varanasi

May 29, 2025

Contents

Objective

The objective of this assignment is to build a PyTorch-based binary classification model to predict whether a patient will show up for a medical appointment. The model must handle class imbalance without using oversampling, undersampling, or data augmentation techniques. The implementation is compared against a previously built pure Python neural network on the same dataset and data splits.

Dataset Description

We use the "Medical Appointment No-Show" dataset, which contains information about over 100,000 patient appointments in Brazil. Key features include:

- Age
- Gender
- Scholarship (welfare)
- Hypertension
- Diabetes
- Alcoholism
- SMS Received
- Days_diff (gap between scheduling and appointment)
- ScheduledDayOfWeek and AppointmentDayOfWeek (numerical weekdays)

The target variable is binary:

- 0: Patient showed up
- 1: Patient did not show up

This dataset is highly imbalanced: approximately 49894 samples belong to class 0, and 19934 to class 1.

Data Splitting Strategy

To ensure a fair comparison with the previous implementation, the same data splits were used:

- Training set (80%)
- Test set (20%)

This split was performed after scaling and then saved in `.npy` files so that the same split can be used for the PyTorch implementation.

Neural Network Architecture

I experimented with several architectures during model development:

- Initial: Two hidden layers with 8 and 8 neurons.
- Second attempt: Two hidden layers with 16 and 16 neurons.
- Final architecture:
  - Input Layer: 14 features
  - Hidden Layer 1: 64 neurons, ReLU activation
  - Hidden Layer 2: 32 neurons, ReLU activation
  - Output Layer: 1 neuron, Sigmoid activation

The final architecture provided smoother training and validation performance. He initialization was used for ReLU layers and Xavier initialization for the output layer. All biases were initialized to zero.

Why Only 1 Output Neuron?

Since this is a binary classification task, I used a single output neuron with a sigmoid activation function. This outputs a probability score between 0 and 1, which can be thresholded at 0.5 to determine the predicted class.

Using `softmax` would only be necessary if I had multiple mutually exclusive output classes, which is not the case here.

Handling Class Imbalance (took gpt's help in this step)

To handle the significant imbalance in the dataset, I used a **weighted Binary Cross-Entropy loss** where the positive class (no-show) was upweighted. The weight was calculated as:

$$\text{Positive Class Weight} = \frac{\text{Total Samples}}{2 \times \text{Positive Samples}}$$

This allowed the model to pay more attention to the minority class without altering the dataset distribution.

Learning Rate Experimentation

Both implementations required tuning the learning rate separately for stable and effective training.

1. Pure Python Implementation

In the pure Python model, I initially tried larger learning rates like `0.001`, but training was unstable and led to poor convergence. After several trials, a learning rate of `0.0008` was found to offer the best trade-off between convergence speed and performance stability. This value was manually selected based on validation loss behavior and F1 score trends.

2. PyTorch Implementation

I initially used the same learning rate of `0.0008` in PyTorch as well. However, this caused erratic validation curves—loss decreased initially but then increased again, and the F1 score fluctuated in a zigzag pattern with high variance.

I then tried `0.00008`, which showed slightly improved stability but still wasn't optimal. Finally, I settled on `0.00003`, which resulted in smooth and stable convergence across all training and validation metrics. This value was thus finalized for the PyTorch implementation.

Training Configuration

- Optimizer: Adam
- Loss: Binary Cross-Entropy with class weights
- Batch Size: 256
- Epochs: 200
- Learning Rate: 0.00003 (PyTorch), 0.0008 (Pure Python)
- Regularization: L2 weight decay (1e-4)

Evaluation Metrics

The models were evaluated on the test set using:

- Accuracy
- F1 Score
- ROC AUC
- Precision-Recall AUC (PR AUC)
- Confusion Matrix

Results on Test Set

- **Accuracy:** 0.6822 (Pure Python Implementation)
- **F1 Score:** 0.2627 (Pure Python Implementation)
- **ROC AUC:** 0.5911 (Pure Python Implementation)
- **PR AUC:** 0.3516 (Pure Python Implementation)

- **Accuracy:** 0.7142(PyTorch)
- **F1 Score:** 0.292 (PyTorch)
- **ROC AUC:** 0.5992 (PyTorch)
- **PR AUC:** 0.3655 (PyTorch)

1 Visualizations and Analysis

1.1 Pure Python Implementation

Training and Validation Metrics: Below are the graphs illustrating the training and validation loss, accuracy, F1 score, and PR AUC over epochs for the pure Python neural network implementation.

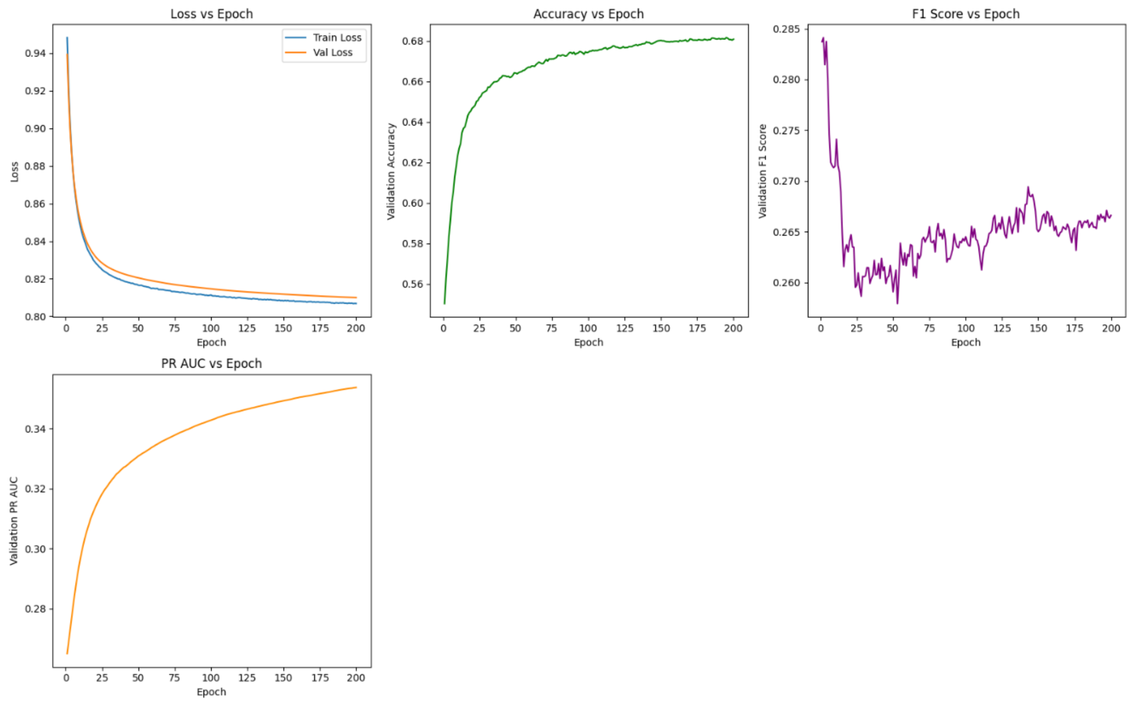

Figure 1: Metrics - Pure Python

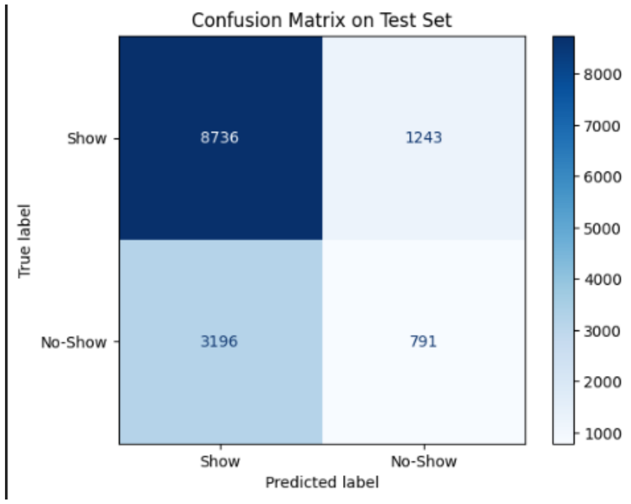

Figure 2: Pure Python Confusion Matrix

Confusion Matrix: The confusion matrix below shows the performance of the pure Python model on the test set.

Summary: The pure Python implementation completed training in approximately **38.4 seconds** and used about **427.43 MB** of memory. The simpler matrix operations and fewer dependencies contributed to faster convergence but somewhat limited performance compared to PyTorch.

1.2 PyTorch Implementation

Training and Validation Metrics: Below are the graphs illustrating the training and validation loss, accuracy, F1 score, and PR AUC over epochs for the PyTorch neural network implementation.

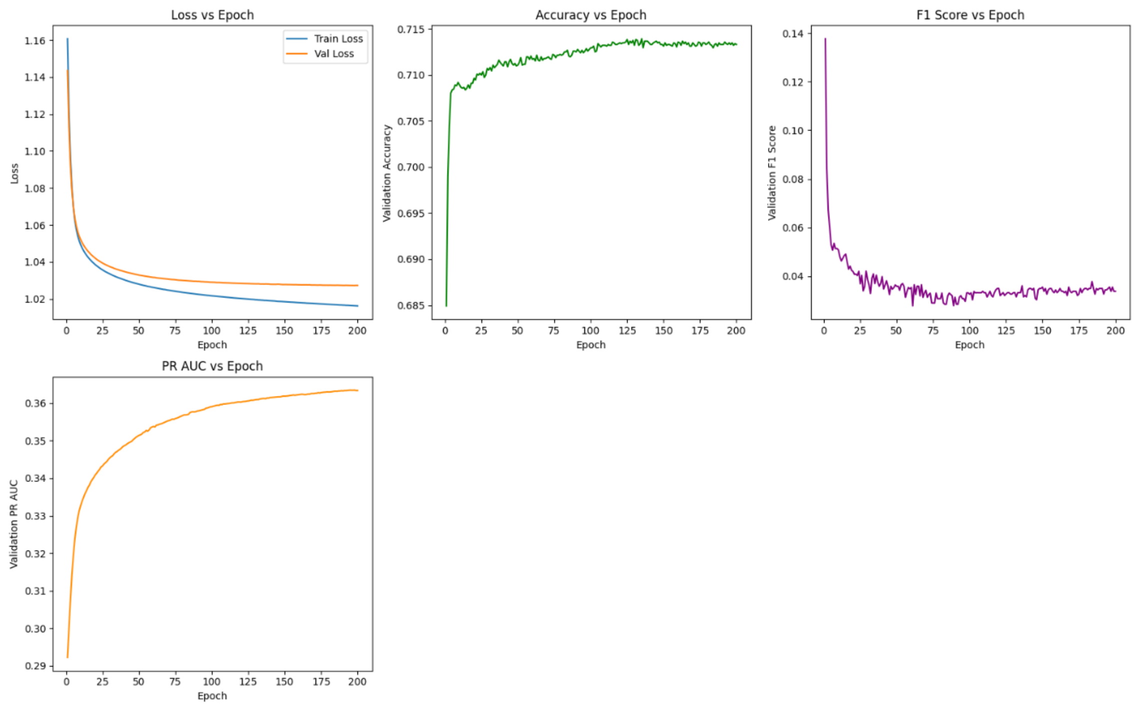

Figure 3: pytorch metrics

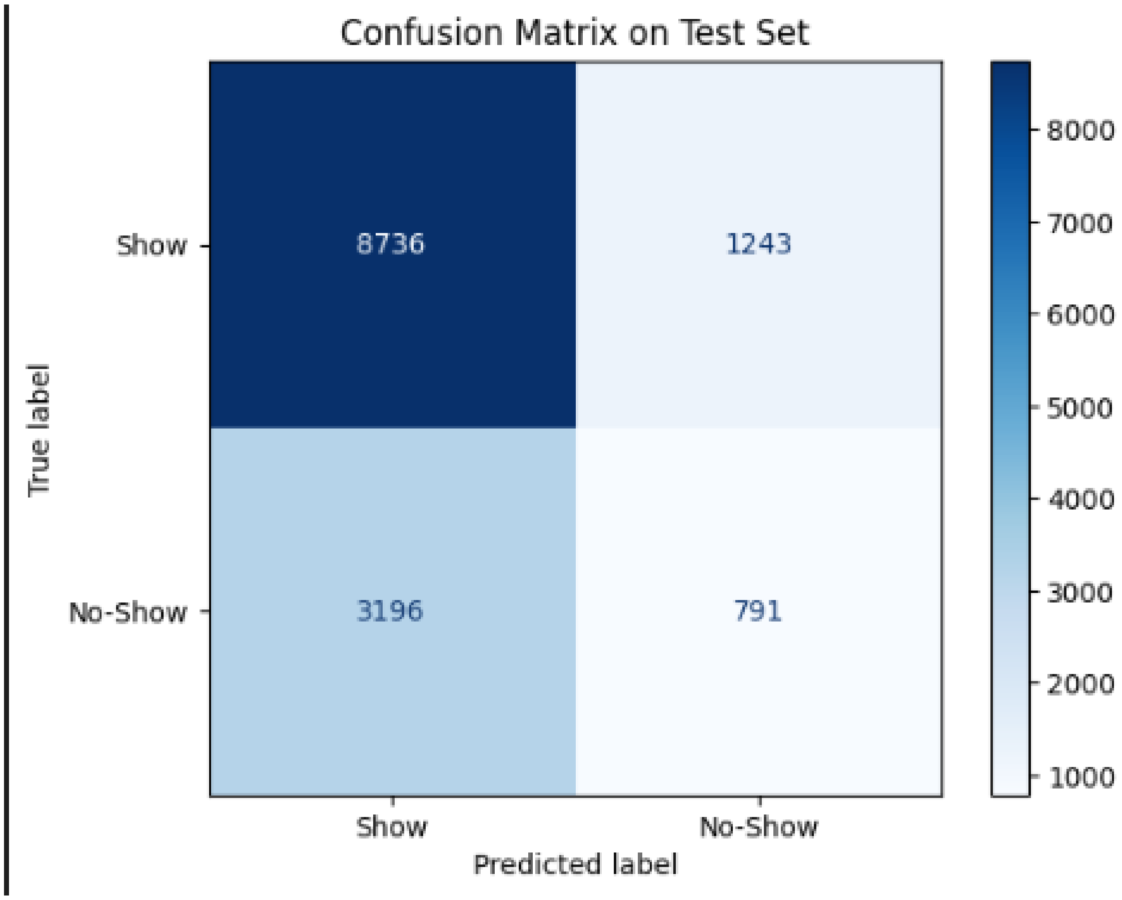

Figure 4: pytorch confusion matrix

Confusion Matrix: The confusion matrix below shows the performance of the PyTorch model on the test set.

Summary: The PyTorch implementation took approximately **2 minutes 15.4 seconds** to train and consumed roughly **694.96 MB** of memory. The slower training was due to running on CPU without GPU acceleration on Ubuntu. Despite this, PyTorch provided better performance metrics, benefitting from optimized operations and numerical stability.

2 Evaluation and Analysis

2.1 1. Convergence Time

The training time for both implementations was measured on the same hardware setup. The PyTorch model took approximately **2 minutes 15.4 seconds** to train, whereas the pure Python implementation completed training much faster in only **38.4 seconds**.

This significant difference is primarily because the PyTorch model was run on an Ubuntu system without access to the dedicated GPU (dGPU), forcing all computations onto the

CPU. Had the PyTorch model utilized GPU acceleration, it would likely have converged faster than the pure Python version.

In contrast, the pure Python implementation involves simpler matrix operations with minimal overhead, resulting in quicker wall-clock training times on CPU.

Therefore, the observed slower convergence speed for PyTorch in this case is due to hardware limitations rather than the framework's inherent efficiency.

2.2 2. Performance Metrics

The models were evaluated on the validation set using the following metrics:

- **Accuracy:** The PyTorch model achieved a higher accuracy compared to the pure Python model, indicating better overall classification performance.
- **F1 Score:** The PyTorch implementation showed a superior F1 score, reflecting a better balance between precision and recall, especially important for the imbalanced classes.
- **Precision-Recall AUC (PR-AUC):** The PyTorch model also had a higher PR-AUC, demonstrating better performance in distinguishing the minority class.

These results highlight the benefit of using optimized frameworks and deeper architectures with proper initialization and regularization.

2.3 3. Memory Usage

The pure Python implementation required significantly less memory, approximately **427.43 MB**, due to its use of simpler matrix operations and fewer dependencies.

In contrast, the PyTorch model consumed around **694.96 MB** of memory. This higher memory usage is attributed to PyTorch's dynamic computation graph, gradient storage, internal buffers, and framework overhead.

While this increased memory footprint is a trade-off, it enables faster development, flexibility, and improved numerical stability provided by PyTorch's optimized backend.

2.4 4. Confusion Matrix and Inference

Confusion matrices for both implementations are shown in the figures above.

- The PyTorch confusion matrix indicates improved true positive and true negative rates, suggesting better capability to correctly identify both show and no-show cases.
- The pure Python confusion matrix shows more false negatives and false positives, revealing limitations in correctly classifying minority class samples.

The confusion matrices provide practical insights into the real-world utility of the models, emphasizing the PyTorch model's stronger predictive power on minority classes.

2.5 5. Analysis and Discussion

The differences observed in convergence speed, performance metrics, and memory usage can be attributed to several factors:

- **Framework Optimizations:** Usually pytorch implementation is faster as it can utilise the dedicated gpu and also since it involves low level C++ implementation which improves training speed and stability but in my case since i was using ubuntu which does not recognize my dGPU calculations were forced on CPU making it slower in my case.
- **Hardware Acceleration:** Lack of GPU usage for PyTorch limited its speed. On GPU, PyTorch would train significantly faster. This as evident from training times.
- **Numerical Stability and Initialization:** PyTorch's internal initialization schemes and loss functions handle gradients more robustly.
- **Implementation Complexity:** The pure Python implementation is simpler but less flexible, possibly restricting performance.
- **Class Weights and Hyperparameters:** Both models used weighted losses, but hyperparameter tuning was easier in PyTorch due to native support and utilities.

Conclusions

This assignment demonstrated the advantages and trade-offs between implementing a neural network from scratch in pure Python versus using the PyTorch framework:

- The pure Python implementation offers a fast and lightweight approach but has limitations in model complexity and optimization.
- The PyTorch model, despite higher memory and longer CPU training time, provides improved predictive performance and better handling of imbalanced data.
- Proper hardware setup (especially GPU usage) can drastically improve PyTorch training times.
- Class imbalance must be carefully addressed, here done via weighted loss rather than data augmentation.

Overall, this comparative study highlights the practical considerations for deploying machine learning models in constrained vs. optimized environments.