Neural Network Implementation on Medical Appointment No-Show Dataset

Anshul Choudhary

1 June, 2025

Abstract

In this task, we implement a neural network model (both from scratch and using PyTorch) on a Medical Appointment No-show dataset and compare both implementations on various metrics.

Data Pre-Processing

First, let's check what we have been provided in the data.

```
[25]: dataset.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 110527 entries, 0 to 110526
Data columns (total 14 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   PatientId       110527 non-null  float64
 1   AppointmentID   110527 non-null  int64
 2   Gender          110527 non-null  object
 3   ScheduledDay    110527 non-null  object
 4   AppointmentDay  110527 non-null  object
 5   Age             110527 non-null  int64
 6   Neighbourhood   110527 non-null  object
 7   Scholarship     110527 non-null  int64
 8   Hipertension    110527 non-null  int64
 9   Diabetes        110527 non-null  int64
 10  Alcoholism      110527 non-null  int64
 11  Handcap         110527 non-null  int64
 12  SMS_received    110527 non-null  int64
 13  No-show         110527 non-null  object
dtypes: float64(1), int64(8), object(5)
memory usage: 11.8+ MB
```

We can see that there are no missing values. The number of classes in **Neighbourhoods** is 81. Since this is not too large, we can one-hot encode this feature along with the **Gender** feature.

- **Feature engineering**: Raw dates are not very useful. First, we convert them to `pandas.Timestamp` objects and then use them to create features like **waiting time** (the difference between scheduled date and appointment date), **weekday on which scheduling was done**, and **weekday of the appointment**. After creating these features, we drop the original ones.

  Patient IDs and **Appointment IDs** could potentially be useful features because some patients may follow certain patterns in whether they appear for an appointment.

However, we must be careful not to leak data. For example, if we create a feature like the no-show rate for a patient, it would include information from all their appointments. So, when splitting into training and test sets, using such features would leak test data into training. To avoid this, I dropped these two features, though they could potentially be useful.

- **Splitting the data**: I split the data into training (80%) and validation (20%) sets.

Results

2.1 Scratch Implementation

```
Final Test Metrics:
Precision: 0.3202
Recall:    0.6823
F1 Score:  0.4358
PR-AUC:    0.3555

Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.63      0.74     17642
           1       0.32      0.68      0.44      4464

    accuracy                           0.64     22106
   macro avg       0.60      0.66      0.59     22106
weighted avg       0.77      0.64      0.68     22106
```

Figure 1: Performance metrics of scratch implementation

I trained a neural network with a single hidden layer containing 256 neurons for 200 epochs using a learning rate of $\alpha = 0.003$. The training took **192.592 seconds**. I used cross-entropy loss as the objective function, **ReLU** activation for the hidden layer, and **softmax** activation for the output layer.

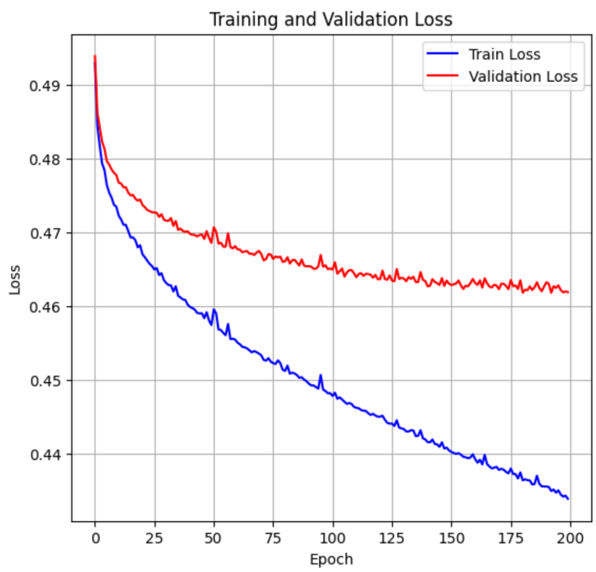


Figure 2: Loss vs Epochs graph

We can notice from the graph that while the model is still improving its loss on the training data, it has plateaued on the test set. This indicates that the model is starting to overfit, so training for more epochs would not help.

2.2 PyTorch Implementation

I trained the model for 10 epochs with a learning rate of $\alpha = 0.02$. It took **22.07 seconds** to train.

```
==================================================
FINAL EVALUATION ON TEST SET
==================================================
Final Test Metrics:
Accuracy:  0.6444
Precision: 0.3259
Recall:    0.7124
F1 Score:  0.4472
PR-AUC:    0.3685

Detailed Classification Report:
              precision    recall  f1-score   support

         0.0       0.90      0.63      0.74     17642
         1.0       0.33      0.71      0.45      4464

    accuracy                           0.64     22106
   macro avg       0.61      0.67      0.59     22106
weighted avg       0.78      0.64      0.68     22106
```

Figure 3: Performance metrics of PyTorch implementation

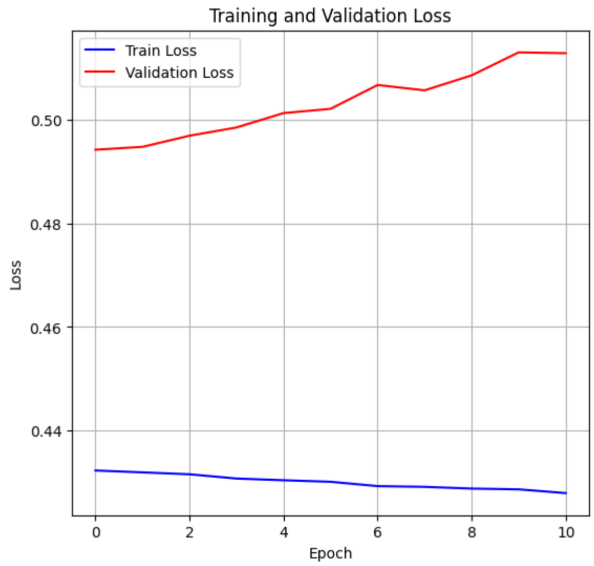
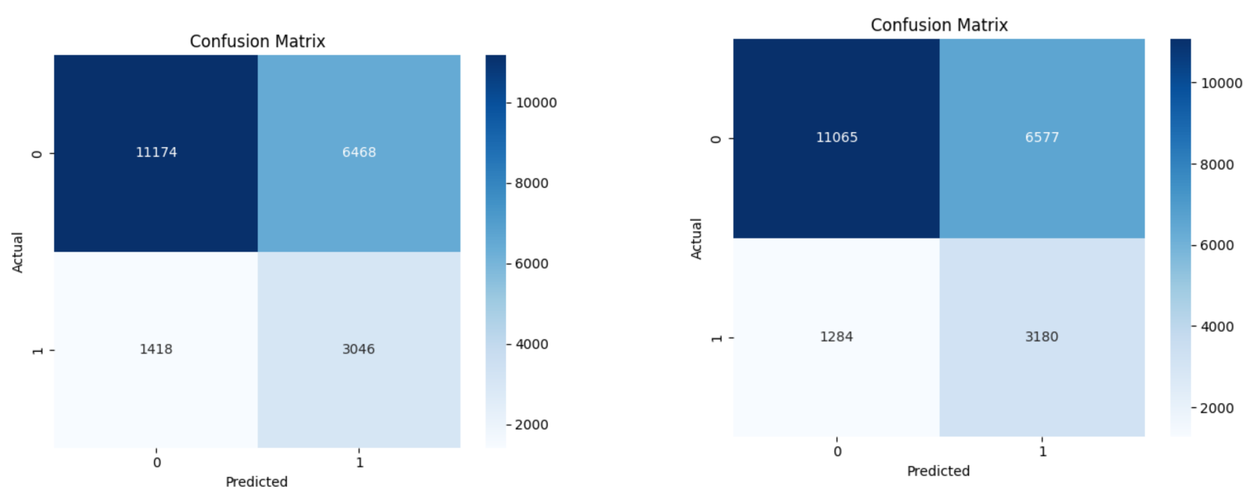


Figure 4: Loss vs Epochs

The architecture of the neural network is the same as in the scratch implementation. The only difference is the optimizer: in the scratch implementation, I used **SGD**, while in PyTorch, I used **Adam**.
We observe that in just 10 epochs, the PyTorch model reaches a similar performance level as the scratch model. However, training for more epochs doesn't help—only the training loss decreases, while the test loss keeps oscillating.

Comparison and Analysis

3.1 Confusion Matrix



(a) Confusion matrix for scratch implementation

(b) Confusion matrix for PyTorch implementation

We can note that the difference between the final metrics of both models is minimal, since both use the same architecture and were trained sufficiently to reach a local minimum.

The dataset has about 80% of data points from people who showed up for appointments and 20% from those who did not. Therefore, if the threshold is set to 0.5 or greater, the model tends to predict only the negative class, achieving a high accuracy of 80% but performing poorly on metrics like **F1 score**.
Using a simple program, I tested various threshold values and found that a value of **0.2** gives the best F1 score. However, lowering the threshold reduces precision in exchange for high recall, which is why our models misclassify around **6500 negative examples** as positive.

3.2 Convergence Time

The PyTorch model converged in fewer epochs than the scratch model, primarily due to the use of the **Adam optimizer** and possibly because of PyTorch's highly optimized backend.

3.3 Memory Usage

In the scratch implementation, we explicitly store all weight matrices, biases, and intermediate values (like activations and pre-activations) for each batch and perform matrix operations using NumPy. So memory usage is relatively stable, but we need to manually manage forward and backward values. Additionally, since we don't use any computational graph or autograd, there's no extra memory overhead for gradient tracking. Overall memory usage is lower, but there's no optimization like shared buffers or GPU support.

In contrast, the PyTorch model internally builds a computational graph for automatic differentiation, which increases memory consumption due to storage of gradients and intermediate tensors during backpropagation. Moreover, PyTorch by default uses `float32` precision and allocates buffers for Adam optimizer states (like momentum and variance for each parameter), which further adds to memory usage. However, since PyTorch is optimized and can leverage GPUs, the overall memory management is more efficient for larger models and datasets.

Summary: Scratch uses less memory per parameter due to simplicity but lacks scalability. PyTorch uses more memory due to autograd and optimizer states, but scales better and handles memory more efficiently.