Neural Network Implementation on Medical Appointment No-Show Dataset

Prerna Tripathi

June 1, 2025

1. Initial Procedures

1.1. Approach

The procedure common across both the parts were Data Preprocessing, Feature Selection and Feature Engineering. All the missing values were removed from the dataset and features like gender and neighbourhood were one hot encoded. Feature Engineering included combination of features like Scheduled Day and Appointment Day to give a more useful feature for predicting No-Show which was the number of Waiting Days. The next step was feature selection which dropped lesser important features like PatientId and AppointmentId.The final step was to normalise the features using Z-Score normalisation technique. The dataset was then divided into training and testing sets for both the implementations.

2. Neural Network from Scratch

The implementation of Neural Network from scratch involved first defining all the activation functions, their derivatives and the output function. The activation function used is ReLU and the output function is Softmax. The next step was the initialisation of weights and biases. The weights were initialised using Xavier initialisation and the biases were set to zero. The neural network built has 2 hidden layer with 64 and 32 neurons respectively and since the output function used is softmax there are 2 neurons in the output layer. The neural network was then trained using a mini-batch gradient descent algorithm with a learning rate of 0.05 in 2700 epochs which trained the model in batches of size 128 using the conventional feed forward and backpropogation. The other thing the problem statement focussed on is class imbalance. The dataset was imbalanced as it had a very substantial difference in the number of samples of No-Shows and non No-Shows. To deal with this problem the first thing I did was to change conventional cross entropy loss function to a balanced cross entropy that

can deal with class imbalances. The loss function was implemented using the concept of class weights. The majority class was given a lesser weightage compared to the Minority class such that the loss function does not only get affected by the majority class but also the minority class. The second thing I did was threshold tuning. Usually the predicted class of the model is the class with probability greater than or equal to 0.5. I changed that to 0.35 so as to improve my F1-score. This was purely done by hit and trial. The final step was to then evaluate the metrics such as accuracy F1-score PR-AUC and the Confusion Matrix.

2.1. Analysis

Convergence time: 1041.6400 sec
Accuracy= 0.54
F1-Score(corresponding to the postive class)= 0.42
PR-AUC= 0.3332
Confusion Matrix=[[8253 9472] [731 3650]]
Inference From Confusion Matrix:Since the dataset was highly imbalanced, certain class imbalance strategies were used which increased the number of true positives getting detected by the model and the number of false negatives came down to a low number of 731 thus increasing our recall of the minority class to a very large value. But this increase in recall came at the cost of precision. As we can see from the confusion matrix the number of false positives is very high thus lowering our precision to a very low percentage.Because of the threshold tuning and the class weights the model is over predicting the positive class thus reducing our precision but improving the recall.

3. PyTorch Implementation

The concept applied behind the neural network implementation using PyTorch was pretty much the same as the from scratch one but PyTorch helped us to shorten our code as we did not need to define functions like feed forward and backpropogation and many more. They were all done by the pytorch operations like the autograd for backpropogation and gradient calculation. Thus Pytorch simplified the process of building the neural network to a large extent.

3.1. Analysis

Convergence time: 6145.7082 sec
Accuracy= 0.57

F1-Score(corresponding to the postive class)= 0.42
PR-AUC= 0.3295
Confusion Matrix=[[9177 8492] [1003 3434]]
Inference From Confusion Matrix:The confusion matrix obtained in part2 was quite similar to part1. The number of false negatives was slightly higher than part1 thus reducing the recall and the number of false positives on the other hand reduced thus slightly improving the precision.

4. Convergence Time Comparison

Unlike normal circumstances the code implemented using pytorch came out to be way slower than the scratch one.The possible reason for this could be due to the limited GPU utilisation. The main reason for pytorch to work faster than a simple from scratch code is because of its integration with CUDA to perform computations on GPUs which speed up the tensor operations but maybe in our case the model is too small to benefit from this GPU acceleration. Pytorch benefits can be seen when we have more complex deep learning models and larger datasets.

5. Memory Usage Comparison

On measuring the memory usage of the from scratch implementation we found that a total of 1285.28 MB amount of RAM is being used in this implementation. On measuring the memory usage of the Pytorch implementation the max CUDA memory allocated came out to be 17.63 MB. This memory highlights the memory allocated or used up by pytorch for all the tensors and their operations on the GPUs. The main reason for this difference to exist is again the hardware accelerations and framework optimisations.Pytorch uses an advanced caching memory allocator which reuses memory chunks and allows more efficient use of availaible memory unlike the from scratch implementation which uses basic memory allocation strategies which can lead to higher memory usage. Also another big reason is Pytorch's GPU utilisation which offers memory optimisations and parallelism which are ideal for the functioning of neural network operations with large datsets. The from scratch implementation runs on CPU and hence cannot make use of these GPU optimisations. Other factors like Pytorch's Dataloading and parallel processing capabilites also allow efficient memory usage when dealing with large datasets and batches.

6. Performance Metrics Comparison

Although the performance metrics are almost the same for both the implementations there is a slight difference in accuracy,PR-AUC and the counts of confusion matrix. This is because manual gradient calculations in the from scratch implementation may introduce rounding errors and approximations wheraas in Pytorch the gradients are calculated using high precision intermediate values and optimised backward passes leading to more numerical stability.

7. Possible Improvements

The first way in which the model can be made to perform better is to increase the number of features by doing more feature engineering.I left out several features like PatientId ,thinking they won't affect the outcome much, but it could have been included by certain transformations. The second thing which i can think to improve the model is to use different optimisers. The optimiser used right now is SGD(mini-batch gradient descent). It can be replaced with other optimisers like Adam with an adaptive learning rate which might yield better results. The learning rate is 0.05 right now. Other lower learning rates were tried but they were too slow to converge within a large number of epochs. Although the learning rate of 0.05 was not too high that the gradient descent overshooted or diverged, it enabled the model to converge within the given number of epochs but one thing which can be improved here is to use lower learning rates with NAG(nesterov accelerated gradient descent) to speed up convergence.

8. Use of AI

The use of AI was restricted to just write certain blocks of code in both the parts. All the concepts anf strategies behind it were strictly thought of and implemented by me.