Neural Network models for predicting patient no-shows using the same dataset: From scratch and using PyTorch.

Kushagra Yadav

May 2025

1 Introduction

The assignment dealt with developing a neural network to predict no-shows by patients using two techniques: From scratch (without deeplearning frameworks) libraries such as NumPy or Pandas for numerical operations and one using the Pytorch framework. One restriction was provided that, even though the class was imbalanced, no class balancing was to be used.

2 Data Preprocessing

The dataset initially had 14 columns:- 13 features and the target column. The first decision I had to make was how to deal with the Neighbourhood and gender columns. The Gender column was easy to deal with but the Neighbourhood had much more decisions to make. I used pandas "get dummies" method (one-hot encoding) for the Gender feature. The problem with using the same methodology on the "Neighbourhood" feature was that it would result in much more columns (14 vs 92).
The other alternative was to use the ordinal encoding, but it is not generally not advised for data that where inputs are discrete rather than in an order as the model could interpret that the particular neighbourhoods have some inherent order.
Thus, I ran the models in both the cases. There wasn't any noticeable difference in the metrics, thus it could be concluded that ordinal encoding wasn't having any detrimental effect on the model predictions. But there was a prominent increase in model's training time when using one-hot encoding. Thus, considering all these things, I decided to use Ordinal encoder for my model.
I used the Scheduled Day and AppointmentDay columns to get relevant info. This included "ScheduledDayDayOfWeek", "ScheduledDayHour", "AppointmentDayDayOfWeek", "AppointmentDayMonth" and "DaysBeforeAppointment". Following that i dropped the 'PatientID' and 'AppointmentID' columns as they

are unique to the patients and have no effect on the predictions.
Another important thing about the dataset was that it was highly biased towards the 0 class(majority class). To overcome this assigned weights to both the classes.

```
from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y_tra
```

This assigns weight to the classes inversely proportionally to their frequency in input data.

3 Scratch Model

3.1 Neural Network Components

The core components are designed to facilitate the forward and backward passes essential for training a neural network.

Layer_Dense

This class represents a fully connected (dense) layer.

- **Initialization:**
  - `n_inputs`: Number of input features.
  - `n_neurons`: Number of neurons in the layer.
  - **Weights:** I have initialized using the Glorot uniform initialization method to help prevent vanishing/exploding gradients as it was leading to big variations in the loss function graph rather than a smooth decreasing curve:

$$\text{weights} \sim U\left(-\sqrt{\frac{6}{\text{n\_inputs} + \text{n\_neurons}}}, \sqrt{\frac{6}{\text{n\_inputs} + \text{n\_neurons}}}\right)$$

  - **Biases:** Initialized to zeros.
- **'forward(inputs)':** This calculates the output of the layer:

$$\text{output} = \text{inputs} \cdot \text{weights} + \text{biases}$$

- **'backward(dvalues)':** The backward function computes gradients with respect to weights, biases, and inputs:
  - 'dweights': Gradient with respect to the weights.

$$\text{dweights} = \text{inputs}^T \cdot \text{dvalues}$$

– ‘dbiases‘: Gradient with respect to the biases.

$$\text{dbiases} = \sum_{\text{samples}} \text{dvalues}$$

– ‘dinputs‘: Gradient with respect to the inputs to this layer.

$$\text{dinputs} = \text{dvalues} \cdot \text{weights}^T$$

Activation_RELU

I used the Rectified Linear Unit (ReLU) activation function as my activation function.

- **‘forward(inputs)‘:** This applies the ReLU function:

$$\text{output} = \max(0, \text{inputs})$$

- **‘backward(dvalues)‘:** It computes gradients, setting gradients to zero where the input was less than or equal to zero during the forward pass as the function is constant in that domain and for inputs greater than zero, it returns the dvalues as Relu has derivative 1 in this domain.

$$\text{dinputs}_i = \begin{cases} \text{dvalues}_i & \text{if inputs}_i > 0 \\ 0 & \text{if inputs}_i \leq 0 \end{cases}$$

Activation_Softmax

I have used the Softmax activation function, as the output layer because this is a classification problem,

- **‘forward(inputs)‘:** Calculates the Softmax probabilities:

$$\text{output}_i = \frac{e^{\text{inputs}_i - \max(\text{inputs})}}{\sum_j e^{\text{inputs}_j - \max(\text{inputs})}}$$

(The subtraction of max(inputs) along ‘axis=1‘ as exponential functions increase at a very fast rate.)
(There is no backward function here as i am implementing it with the loss function directly)

Loss Function

Loss_CategoricalCrossentropy

It calculates the categorical cross-entropy loss, which is commonly used for classification tasks. It supports both one-hot encoded and sparse labels for ‘y_true‘.

- **Initialization:**
  - `class_weights`: This is used to apply weights previously calculated, to the loss for each class.
- `forward(y_pred, y_true):` It computes the weighted negative log likelihood:
  - 'y_pred_clipped': Predicted probabilities are clipped to a small range (10^{-7} to $1 - 10^{-7}$) to avoid log(0) errors.

$$\text{y\_pred\_clipped} = \text{clip}(\text{y\_pred}, 10^{-7}, 1 - 10^{-7})$$

  - 'correct_confidences': The probability score of the true class.
  - 'log_likelihoods': Negative logarithm of the correct confidences.

$$\text{log\_likelihoods} = -\log(\text{correct\_confidences})$$

  - 'weighted_log_likelihoods':The log likelihoods are multiplied by the corresponding class weight.

$$\text{Loss} = \frac{1}{\text{samples}} \sum_{s=1}^{\text{samples}} \left(-\log(\text{correct\_confidence}_s) \cdot \sum_{c=1}^{\text{classes}} (\text{class\_weights}_c \cdot \text{y\_true\_one\_hot}_{s,c}) \right)$$

- `'backward(dvalues, y_true)':` It computes the gradient of the loss with respect to the predictions and the output activation function simultaneously:
  - Converts sparse 'y_true' to one-hot encoding if it is not in that form.
  - Calculates the gradient:

$$\text{dinputs} = \frac{\text{dvalues} - \text{y\_true\_one\_hot}}{\text{samples}}$$

NeuralNetwork Class

The 'NeuralNetwork' class encapsulates the architecture and training logic of a neural network.

```
class NeuralNetwork:
    def __init__(self,layers,activation_functions,
                 loss_class=Loss_CategoricalCrossentropy,learningrate=0.0001,class_we
        self.layers=layers
        self.activation_functions=activation_functions
        self.loss_function=loss_class(class_weights=class_weights)
        self.learning_rate=learningrate
        self.current_batch=None
```

layers: This is the list of Layer_Dense objects in my neural network.

Activation_functions: A list of activation function objects, corresponding to each layer. I have matched the order of activation functions with the order of layers.

loss_class: The class of the loss function to be used (defaulting to Loss_CategoricalCrossentropy).

Learning_rate: The step size for updating weights and biases during optimization. I chose the learning rate to 0.0001 as the cost function for any higher learning rate started to diverge.

class_weights: Weights for the loss function, passed to the loss_class.

```
def forward_propagation(self,X):
    current = X
    self.layer_outputs = []
    self.activation_outputs = []


    for k in range(len(self.layers)):
        self.layers[k].forward(current)
        self.layer_outputs.append(self.layers[k].output)

        self.activation_functions[k].forward(self.layers[k].output)
        self.activation_outputs.append(self.activation_functions[k].output)

        current = self.activation_functions[k].output

    return current
```

Forward Propagation: This method performs the forward pass through the network, calculating the output for a given input 'X'.

It iterates through each Layer_Dense and its corresponding activation function, for each layer, it first performs the linear transformation (layer.forward(current)). The output of that is then passed through the activation function (activation_function.forward(layer.output)).

The output of the activation function becomes the input for the next layer.

The outputs of both the linear layers and the activation functions are stored for use in backpropagation.

The final output of the last activation function is returned as the network's prediction.

```
def backward_propagation(self,output,y_true):
    self.loss_function.backward(output,y_true)
    gradients=self.loss_function.dinputs
    for i in reversed(range(len(self.layers))):
        if not isinstance(self.activation_functions[i], Activation_Softmax):
            self.activation_functions[i].inputs = self.layer_outputs[i]
            self.activation_functions[i].backward(gradients)
            gradients=self.activation_functions[i].dinputs
        self.layers[i].inputs = self.activation_outputs[i-1] if i > 0 else self.c
        self.layers[i].backward(gradients)
        gradients=self.layers[i].dinputs
```

Backward Propagation: This method performs the backpropagation algorithm to compute gradients of the loss with respect to the weights and biases of each layer.
It starts by calculating the initial gradients of the loss function w.r.t to the outputs and the activation function: ‘self.loss_function.backward(output, y_true)‘.
It then iterates backward through the layers of the network.

Activation Layer Backpropagation: For each activation function (except for ‘Activation_Softmax‘, which is being handled with the loss function’s backward pass in a combined step), it sets the ‘inputs‘ property and computes its gradients: ‘self.activation_functions[i].backward(gradients)‘. The ‘gradients‘ are then updated with the ‘dinputs‘ from the activation function.
Layer Backpropagation:For each layer, it sets its ‘inputs‘ property (which is the output of the previous activation) and computes its gradients: ‘self.layers[i].backward(gradients)‘. The ‘gradients‘ are then updated with the ‘dinputs‘ from the layer.

```
def train(self,X,y,epochs=2500,batch_size=64):
    self.epoch_losses=[]
    for epoch in range(epochs):
        permutation = np.random.permutation(len(X))
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]
        epoch_loss=0
        num_batches=0
        for i in range(0,len(X),batch_size):
            X_batch = X_shuffled[i : i + batch_size]
            y_batch = y_shuffled[i : i + batch_size]

            self.current_batch = X_batch
            output=self.forward_propagation(X_batch)
            batch_loss = self.loss_function.calculate(output, y_batch)
            epoch_loss += batch_loss
            num_batches += 1
            self.backward_propagation(output,y_batch)

            for layer in self.layers:
                layer.weights-=self.learning_rate*layer.dweights
                layer.biases-=self.learning_rate*layer.dbiases

        avg_epoch_loss = epoch_loss / num_batches
        self.epoch_losses.append(avg_epoch_loss)
```

Training (‘train(X, y, epochs=2500, batch_size=64)‘): This method trains the neural network using mini-batch gradient descent.
Mini-Batch Iteration and Data Shuffling: At the beginning of each epoch, the training data (‘X‘ and ‘y‘) are shuffled to randomize the order of samples presented to the network. The shuffled data is then divided into mini-batches of ‘batch_size‘.
Forward Pass: For each mini-batch:
‘forward_propagation‘ is called to get the network’s predictions for the batch.

The 'batch_loss' is calculated using the 'loss_function' for each batch. The 'epoch_loss' is accumulated from each batch loss.

Backward Pass: 'backward_propagation' is called to compute gradients for the current batch.
Weight Update: For each layer, the weights and biases are updated using the calculated gradients' based on the gradient descent rule:

$$\text{weights} = \text{weights} - \text{learning\_rate} \cdot \text{dweights}$$
$$\text{biases} = \text{biases} - \text{learning\_rate} \cdot \text{dbiases}$$

Neural Network Architecture Setup

This code snippet details the programmatic construction of a sequential neural network. It dynamically creates a series of **dense layers** and their corresponding **activation functions** based on a predefined list of neuron counts for each layer.

The network's structure is defined by the `neurons` list, where each element represents the number of neurons in a layer. For example, `[len(X_train_scaled[0]), 32, 16, 8, 2]` indicates an input layer with `len(X_train_scaled[0])` features, followed by three hidden layers with 32, 16, and 8 neurons respectively, and an output layer with 2 neurons(0,1). I tried running both various combinations of, 64,32,16 and 8 neurons for hidden layers, eventually opting for 32,16 and 8 combination as gave the most optimal results.

The code iterates through the `neurons` list to build the layers:

- For each pair of consecutive neuron counts (N_i, N_{i+1}), a `Layer_Dense` is initialized with N_i inputs and N_{i+1} neurons. These layers are appended to the `layers` list.
- For the **hidden layers** (all except the last one), an `Activation_RELU` function is associated with each `Layer_Dense`. The ReLU function is defined as:
$$f(x) = \max(0, x)$$
- For the **output layer** (the last layer created), an `Activation_Softmax` function is used:
$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

A specific threshold of 0.25 is applied to the probability of the positive class (typically at column index 1 for binary classification) to convert these continuous probabilities into binary class predictions. This implies that if the probability of the positive class meets or exceeds 0.25, the prediction is categorized as positive (1); otherwise, it's deemed negative (0). This is done to counter the heavily bias-ness towards the majority class.

*********Incomplete**********