csog-in-ann

Sebastian K Sabu

May 2025

1 Introduction

This week I have made a neural network that was implemented in Python as well as Pytorch

2 Code explanation

```
from sklearn.preprocessing import normalize
data["No-show"] = [[1,0] if x == "No" else [0,1] for x in
    data["No-show"]]
data["ScheduledDay_Year"] = pd.to_datetime(data["ScheduledDay"]).dt.year
data["ScheduledDay_Month"] =
    pd.to_datetime(data["ScheduledDay"]).dt.month
data["ScheduledDay_Day"] = pd.to_datetime(data["ScheduledDay"]).dt.day
data["ScheduledDay_Hour"] = pd.to_datetime(data["ScheduledDay"]).dt.hour

data["AppointmentDay_Year"] =
    pd.to_datetime(data["AppointmentDay"]).dt.year
data["AppointmentDay_Month"] =
    pd.to_datetime(data["AppointmentDay"]).dt.month
data["AppointmentDay_Day"] =
    pd.to_datetime(data["AppointmentDay"]).dt.day
data["AppointmentDay_Hour"] =
    pd.to_datetime(data["AppointmentDay"]).dt.hour

data.drop("ScheduledDay", axis=1, inplace=True)
data.drop("AppointmentDay", axis=1, inplace=True)



data.drop('PatientId',axis = 1,inplace=True)
data.drop('AppointmentID',axis = 1,inplace=True)
data.drop('Neighbourhood',axis = 1,inplace=True)
data["Gender"] = [0 if x == "F" else 1 for x in data["Gender"]]
data["No-show"] = data["No-show"].replace({'yes': 1, 'no':
    0}).apply(lambda x: x[0] if len(x) > 0 else None)
```

```
data
```

In the above code i changed the date data type and split it into day,month, hour since the original assumption was that it depended on the time whether a person showed up or not. Later on, by graphing i did notice some pattern, so I included. The date was split to make it easier to feed into the neural network. The gender feature was changed to 1 and 0 same was done for No-show but due to some problem I had to go through this long way to convert it into 1 or 0.

```
    c=[]
temp = data["No-show"]
for column in data.columns:
    if column not in ["No-show"]:
        c.append(column)
data =pd.DataFrame(normalize(data.drop("No-show",axis=1), norm='l2'),
     columns=c)
data["No-show"] = temp
data
```

Here I am scaling the data in order to prevent large values in weights due to the year features except for output ”No-show”

```
import numpy as np
import time

# Optimized sigmoid functions
def _positive_sigmoid(x):
    return 1 / (1 + np.exp(-x))

def _negative_sigmoid(x):
    exp = np.exp(x)
    return exp / (exp + 1)
#taken from stackoverflow
def sigmoid(x):
    positive = x >= 0
    negative = ~positive
    result = np.empty_like(x, dtype=np.float32)
    result[positive] = _positive_sigmoid(x[positive])
    result[negative] = _negative_sigmoid(x[negative])
    return result

def sigmoid_derivative(x):
    sig = sigmoid(x)
    return sig * (1 - sig)

class NeuralLayer:
    def __init__(self, input_size, output_size):
        self.weights = np.random.randn(input_size, output_size) * 0.01
        self.bias = np.zeros((1, output_size))
    #returned both activation and pre-activation in tuple to reduce
         computation
    def forward(self, X):
        self.output_1 = np.dot(X, self.weights) + self.bias
        self.output_2 = sigmoid(self.output_1)
        return self.output_2, self.output_1

class NeuralNetwork:
    def __init__(self, input_size, layers_size, output_size):
        self.input_size = input_size
        self.layers_size = layers_size
        self.output_size = output_size

        # Create layers
        self.layers = []
        self.layers.append(NeuralLayer(input_size, layers_size[0]))

        # Hidden layers
        for i in range(len(layers_size)-1):
            self.layers.append(NeuralLayer(layers_size[i],
                 layers_size[i+1]))

        # Output layer
        self.layers.append(NeuralLayer(layers_size[-1], output_size))

    def cross_entropy_loss(self, y_true, y_pred, eps=1e-15):
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return -np.mean(y_true*np.log(y_pred)+(1 - y_true)*np.log(1 -
             y_pred))
    #train function cleaned and refactored using AI
    def train(self, X, Y, epochs=100, learning_rate=0.1):
        X = np.array(X)
        Y = np.array(Y).reshape(-1, 1)

        for epoch in range(epochs):
            activations = [X]
            pre_activations = []

            # Forward pass
            A = X
            for layer in self.layers:
                A, Z = layer.forward(A)
                activations.append(A) # Store layer output
                pre_activations.append(Z) # Store pre-activation

            # Backward pass
            m = X.shape[0]
            dZ_list = []
            dW_list = []
            db_list = []


            dZ = (activations[-1] - Y) / m
            dZ_list.append(dZ)

            for l in range(len(self.layers)-1, -1, -1):
                A_prev = activations[l]

                dW = np.dot(A_prev.T, dZ_list[-1])
                db = np.sum(dZ_list[-1], axis=0, keepdims=True)

                dW_list.append(dW)
                db_list.append(db)
                if l > 0:
                    dA_prev = np.dot(dZ_list[-1], self.layers[l].weights.T)
                    dZ_prev = dA_prev *
                        sigmoid_derivative(pre_activations[l-1])
                    dZ_list.append(dZ_prev)
            dW_list.reverse()
            db_list.reverse()

            for l in range(len(self.layers)):
                self.layers[l].weights -= learning_rate * dW_list[l]
                self.layers[l].bias -= learning_rate * db_list[l]

            if epoch % 10 == 0:
                loss = self.cross_entropy_loss(Y, activations[-1])
                print(f"Epoch {epoch}, Loss: {loss:.4f}")
    def predict(self, X):
        for layer in self.layers:
            X = layer.forward(X)[0]
        self.y_pred = X
        return (self.y_pred)
```

Here we use backpropagation to minimize our loss function. The loss function used is Binary cross entropy loss function this is because our output is a single float representing our probability of output. The activation function is sigmoid function this is used to obtain a smoother classification and the final output stays between 1 and 0.The sigmoid function implementation has two parts one for positive values and one for negative values this is due to overflow of value when we use np.exp().I have used two hidden layer each of 30 neurons as adding more didn't seem to change loss my much.The same number of layers and neruros are used in the Pytorch implementations.

3 Evaluation

3.1 Time

1. Numpy : 7.33s
2. Pytorch :0.46s

This huge change in time due to the fact that Pytorch in my setup uses CUDA cores. It has a better optimization of its operations for GPU acceleration.It uses a JIT(just in time) compiler like java for faster performance.It has better memoery managment and uses multi-core cpu operation.

3.2 Scores

3.2.1 Acurracy

The accuracy of both the implemetiation is around 79.80% making this model very efficient

3.2.2 F 1 score

The F 1 score is around 0.887 this also is makes our model very good.Closer the F 1 score is to 1 better it is at predicting

3.2.3 Memory

The memory consumption of Pytorch is higher than Numpy this is since Numpy stores data in contiguous manner but Pytorch stores tensors is stored to optimize deep learning task but small tasks may take up larger chunk of memory