Artifical Neural Networks

Sudarshan Kulkarni

May 2025

1 Data Preprocessing

A lot of different stuff was analyzed. This includes frequency distribution of all columns, month of appointment, correlation between different variables, and the following:

1. *No-shows* column was LabelEncoded and *Gender* was OneHot Encoded, with *NO* as 0, *YES* as 1.
2. A new column, *AppointmentTime*, for number of days between appointment and scheduled day was added.
3. *PatientID* and *AppointmentID* waas dropped as there is no relation between them and the patient not showing up
4. Z-score normalization was performed on *Age*.
5. *Neighbourhood* column was dropped because many neighbourhoods only have a few patients, so there was no point in categorizing them individually. The model will not be able to generalize to test set.
6. A new column *Appointment_on_weekend* was added, True if *AppointmentTime* is on weekend, False otherwise.

Observations on the data:

1. All appointments scheduled are from April, May and June only, with majority (80k+) in May.
2. There is one person with age of *-1* 🤔

2 Implementation from Scratch

I had previously implemented neural networks from scratch, but for MNIST dataset, and the layers were hardcoded. It is available here, on github. I had implemented vanilla, and also made use of various optimizations like Batch Gradient Descent, Momentum/Adam, L2/Dropout Regularization. Now I wanted to implement dynamic network.

2.1 Structure

In this section, we discuss the overall structure for implementing a small library supporting simple binary and multi class classification.

Note: Although functions for for mean_squared_loss *and such for regression are implemented, regression is not tested and supported in the library. Binary classification was tested using Medical No shows dataset, and multi-class with MNIST.*

1. **sigmoid**, **relu**, **softmax** activation functions (Ensuring cases for all outputs to be large/small in softmax)
2. **binary_cross_entropy_loss** and **sparse_categorical_cross_entropy** (though the input to latter assume the parameters to be one-hot encoded, not sparse 😅)
3. Functions for precision, recall and f1-score. These are implemented only for single label classification tasks
4. **accuracy** function. For single label, its straightforward. For multi-label, it checks by making sure the entire row (all labels for a particular example) is same.

To make usage of variable names more clear and consistent in the code, the following two notations were considered equivalent, and the former was used in code for its simplicity, which might have inadvertently slipped into the report as well. $\mathbf{dX} \Leftrightarrow \frac{\partial \mathbf{J}}{\partial \mathbf{X}}$, where $\mathbf{X}$ may be any parameters or activations in the network.

The class structure was inspired from tensorflow, which is what I've used in the past. Hence why we define two classes, **Dense** and **Sequential**, with the following structure. (The file defines an **Input** layer too, but there is no use of it.)

2.1.1 Dense

1. During initialization, only the number of units and type of activation to be given. Only *relu*, *sigmoid* and *softmax* activation supported. The weights and bias are not initialized. They can be done manually, or else its done automatically when training. An optional regularizer can be passed. Currently only **L2** regularizer is supported, which only adds regularization to weights, according to $\frac{1}{2}\sum_i \sum_j \mathbf{w}_{ij}^2$, where $\mathbf{W} = [\mathbf{w}_{ij}]$. This loss is added in **Sequential.fit**
2. **build** function, to initialize the weights and bias. They are done from Xavier/Glorot normal initialization, where $\mu = 0$, and

$$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

n_{in} and n_{out} are the number of input units to the layer, and output units from the layer, respectively. Further functionality allowing type of initializer can be added later.

3. **forward_prop** function, which takes input as the activation from the previous layer, calculates the activation of this layer using the below formula, and returns that. It also stores the value of previous layer, current layer, which can be further optimized (coming on this a bit later).

$$\mathbf{Z^{[l]}} = \mathbf{W^{[l]}} \cdot \mathbf{A^{[l-1]}} + \mathbf{b^{[l]}}$$
$$\mathbf{A^{[l]}} = \mathbf{g}(\mathbf{Z^{[l]}})$$

where, $\mathbf{A^{[l]}}$ is activation of layer $\mathbf{l}$. $\mathbf{W^{[l]}}, \mathbf{b^{[l]}}$ are parameters of layer $\mathbf{l}$ and $\mathbf{g}$ is the activation function.

4. **back_prop** function takes input, the derivative of activation of current layer, that is, $\frac{\partial \mathbf{J}}{\partial \mathbf{A^{[l]}}}$, and it calculates $\frac{\partial \mathbf{J}}{\partial \mathbf{Z^{[l]}}}$ (depending on the activation function).
It further calculates $\frac{\partial \mathbf{J}}{\partial \mathbf{W^{[l]}}}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{b^{[l]}}}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{A^{[l-1]}}}$ using the following formulae, and returns $\frac{\partial \mathbf{J}}{\partial \mathbf{A^{[l-1]}}}$ to be used for the previous layer.

$$\frac{\partial \mathbf{J}}{\partial \mathbf{W^{[l]}}} = \frac{1}{\mathbf{m}} \frac{\partial \mathbf{J}}{\partial \mathbf{Z^{[l]}}} \cdot \mathbf{A^{[l-1]^T}}$$

$$\frac{\partial \mathbf{J}}{\partial \mathbf{A^{[l-1]}}} = \mathbf{W^{[l-1]^T}} \cdot \frac{\partial \mathbf{J}}{\partial \mathbf{Z^{[l]}}}$$

$$\frac{\partial \mathbf{J}}{\partial \mathbf{b^{[l]}}} = \frac{1}{\mathbf{m}} \frac{\partial \mathbf{J}}{\partial \mathbf{Z^{[l]}}}$$

5. There is a way to add a **prev_layer** attribute to **Dense**, allowing so that $\mathbf{A^{[l-1]}}$ need not be stored, making it more memory efficient. This is only partially implemented yet.

2.1.2 Dropout

This is layer for implementing Dropout via the Inverted dropout method. It doesn't have any trainable parameters, just a hyperparameter of *keep_prob*, the fraction of units to keep in a single forward pass. It is not necessarily stable, that is, the number of units kept is not the same every time. There might even be cases where the entire layer gets removed. This was implemented just for the sake of completeness and because I had implemented it previously with MNIST the same way.

2.1.3 Sequential

1. Layers of the network (only Dense) are to be given during initialization, or added later using **add_layer** function.

2. The type of loss function and optimizer is to be defined when calling **compile** function. it must be called before fit. Optimizers supported are **SGD**, **Momentum** and **Adam**.

3. **build** function initializes weights for all layers of the network. It takes input the number of features for the network. Not necessary to call as it is called in **fit** function if parameters are not initialized.

4. **fit** function performs gradient descent for the number of epochs given, using the given batch size. It shuffles the training set for every epoch, so an optional *random_state* argument is provided. It measures and stores accuracy and loss for the training and test sets, and returns it at the end of training, so that it can be used for plotting.
   It calculates the gradient of last layer on the basis of given loss function and performs backpropagation for every layer.
   Note: for softmax activation and categorical_cross_entropy, the value calculated is $\frac{\partial \mathbf{J}}{\partial \mathbf{Z}^{[l]}}$*, instead of* $\frac{\partial \mathbf{J}}{\partial \mathbf{A}^{[l]}}$*, which is handled appropriately in* **Dense** *class.*

2.1.4 Momentum and Adam

$\mathbf{V_{dW}}$ and $\mathbf{V_{db}}$ are exponentially weighted averages of $\frac{\partial \mathbf{J}}{\partial \mathbf{W}}$ and $\frac{\partial \mathbf{J}}{\partial \mathbf{b}}$. Similarly $\mathbf{S_{dW}}$ and $\mathbf{S_{db}}$ are exponentially weighted averages of $(\frac{\partial \mathbf{J}}{\partial \mathbf{W}})^2$ and $(\frac{\partial \mathbf{J}}{\partial \mathbf{b}})^2$. All are initialized to zero at start.

$$\mathbf{V_{dW}} = \beta_1 \cdot \mathbf{V_{dW}} + (\mathbf{1} - \beta_1) \cdot \frac{\partial \mathbf{J}}{\partial \mathbf{W}} \tag{1}$$

$$\mathbf{S_{dW}} = \beta_2 \cdot \mathbf{S_{dW}} + (\mathbf{1} - \beta_2) \cdot (\frac{\partial \mathbf{J}}{\partial \mathbf{W}})^{\mathbf{2}} \tag{2}$$

(1) and (2) are defined equivalently for **db** as well. Generally, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

1. Now in momentum, we can perform the gradient descent as follows (bias has an equivalent equation, replace $\mathbf{W}$ with $\mathbf{b}$ everywhere in the equation):

   $$\mathbf{W} = \mathbf{W} - \alpha \cdot \frac{\mathbf{V_{dW}}}{\mathbf{1} - \beta_1^{\mathbf{t}}}$$

   where $\mathbf{t}$ is the iteration count, and the term $(\mathbf{1} - \beta^{\mathbf{t}})$ is called bias correction. This can be ignored for momentum, and it will still work just fine, albeit a bit slower.

2. In Adam, we do as follows:

$$\mathbf{W} = \mathbf{W} - \alpha \cdot \frac{\frac{\mathbf{V_{dW}}}{\mathbf{1} - \beta_1^t}}{\sqrt{\frac{\mathbf{S_{dW}}}{\mathbf{1} - \beta_2^t}} + \epsilon}$$

The bias correction is important in Adam and cannot be ignored, unlike the case for Momentum. ϵ is a small number, added for numerical stability, to avoid division by zero.
Note: ϵ- should be added outside of the square root. It normally wouldn't make an issue, but when L2-Regularization was applied, the training slowed down in later Epochs, taking ≈10s for 1 epoch, while it was taking only 1s initially. This is shown in the image in this directory: training_slowdown_after_epoch35.jpeg

3 Results

In implementation from scratch, a deep neural network, with sizes: 64, 32, 16, 16, 8, 8, 1 was used, without any regularization, learning rate of 10^{-3} and Adam optimizer. The max (unweighted) f1 was **.447**, PR-AUC **.35** (with threshold around 0.216). *batch_size* was kept to be 256. It was found that the f1score was not crossing 0.442 with *batch_size* of 32, even if we tried things like giving class weights in loss (which was experimented with and failed in pytorch implementation.)

In pytorch, the same architecture but with a weighted loss function, with weights from 2.0 to 7.0 for *YES*, and 1.0 for *NO*, was tried. for weight 3.0, the f1 score was almost **.45**, but it also had a factor of luck with weight initialization, because the score couldn't be replicated. Focal loss was also tried with $\alpha \in \{0.5, 0.6, 0.7, 0.8\}$ and $\gamma \in \{1, 2, 3, 4\}$, but it didn't not result in improve score.