Implement a model from scratch

Zhihan Hu

February 2025

1 Introduction

In this document, I will present my methodology on how to implement a model from scratch, that is, without using advanced tools like Pytorch, and only with the help of Numpy, to create a model from 0 to 1.

2 Split of a model

The following list shows the items composed of a ML model

- Data preparation.
- Parameter initialization
- Propagation
- Back-propagation (gradient computation)
- Gradient update
- Loss computation
- Model modularization (integration)
- Prediction

2.0.1 Data preparation

In this step, the aim is to create training dataset and test dataset. And we basically have 4 datasets to be prepared. The dimensions that I determined are flexible and can be changed as you wish.

- X_train $\in R^{d \times n1}$: training data set.
- Y_train $\in R^{m \times n1}$: training data set label. Exist when the ML is supervised.
- X_test $\in R^{d \times n2}$: test data set for model performance

- Y_test $\in R^{m \times n2}$: test data label for model performance.

Note that different dataset has different structure and the data load procedure is somewhat arbitrary and will vary for each different data set. So what I recommend is to have a util file for each data set.

2.1 Parameter initialization

In this step, we will initialize some starting values for learnable parameters for our model. And returns a dictionary called **parameters** that stores the state of each parameter set.

If your model is a simple linear model, you don't have to explicitly state the size of your parameter set. **If your model is some complex model such as deep learning model**, a good practice will be to explicitly write out your size(dimensions) of each of your parameter sets.

One more important thing to notice is that when initializing parameters, do not initialize the values to be equal, especially in the case of deep learning model, as there will be an issue called model symmetry occurred.

2.2 Propagation

In this step, we will send the training data set X into the model, and transform it based on the current learnable parameter states to generate an output. And returns a dictionary called **cache** to store current outputs.

This step depends solely on your model, and a good practice is to write out your propagation formulas first, and use Numpy to efficiently do the linear algebraic calculation. Additionally, remember to implement activation functions if required.

2.3 Back-propagation (gradient computation)

In this step, we will calculate the gradient of the Loss w.r.t each parameters. This step is the most difficult part mathematically, but usually the coding will be easy as long as you have gracefully simplify your gradients into matrix form. We will input the **parameters** and **cache** dictionaries, along with X_train and Y_train to get the gradients, as gradients are commonly computed by linear algebraic calculation among outputs, inputs weights, etc. We then put the gradient of each parameter set to a dictionary called **grads**

2.4 Gradient update

In this step, we predefine a learning rate, and use current **parameters** and **grads** to update the gradient

$$paramters = parameters - learning\ rate * grads \tag{1}$$

2.5 Loss computation

The loss computation is to track how the loss changes over the training process for parameter tuning, making the model run more efficiently. The loss function is commonly easy to be implemented using numpy.

2.6 Model modularization

This step will integrate all the above steps into one function. The input will be

- X_train
- Y_train
- runs: how many learning iterations to be run
- learning rate

And it will output a learned parameter sets

2.7 Prediction

This step checks how well the performance of the model on the X_test data set.