Multi-layer perceptron for binary classification.

Pedro Rodriguez de Ledesma Jimenez

University of Groningen
Groningen, The Netherlands
p.rodriguez.de.ledesma.jimenez@student.rug.nl

May 11, 2022

Abstract

This paper aims to create two deep learning networks, one from scratch where we refrain from state-of-art neural network packages and instead we use basic python libraries, and other using machine learning framework, in particular Keras framework. We provide a thorough exploratory data analysis and base the decision in our neural networks outcomes. Both networks are created dynamically in order to evaluate different configurations and identify the optimal number of layers and perceptrons. Both model metrics are compared and next steps are provided to improve the performance.

1 Introduction

The perceptron algorithm is the simplest type of artificial neural network. It is a model of a single neuron that can be used for two-classification problems and provides the foundation for later developing much larger networks. It is inspired by the information processing of a single neural cell called a neuron. It receives input signals that are weighted and combined in a linear equation.

$$h(x) = bias + \sum_{i=1}^{+n} weight_i * x_i$$

The output is then transformed into an output value or prediction using a activation function.

$$activation(x) = act(bias + \sum_{i=1}^{+n} weight_i * x_i)$$

A multi-layer perceptron consists of multiple layers of perceptrons with feedforward feed-up, therefore the inputs of each layer are the outputs of the previous layer. This paper describes the components of these neural networks with different layers of perceptrons and the process for training the parameters that define these networks. We will discuss different configurations and architectures in order to find the best solution for generalizing the given dataset.

2 Methods

As previously mentioned, the aim of this paper is to create two neural networks for the classification of a binary dataset. In this section we will analyze the data sets used and provide a description of each component of both of the neural networks.

2.1 Dataset insights

The dataset consists of 500 samples created artificially by a given algorithm that defines the dataset by taking a multi-dimensional standard normal distribution with μ =0. The classes are defined separately by nested concentric multi-dimensional spheres such that roughly equal numbers of samples are in each class. The number of features and classes are two. As a result, the dataset is distributed in two concentric classes with center the origin (1) and the classification of the samples turns into a non-linear binary classification task. Despite the non-linearity of the dataset, the dataset is well distributed and low levels of overlapped points are present.

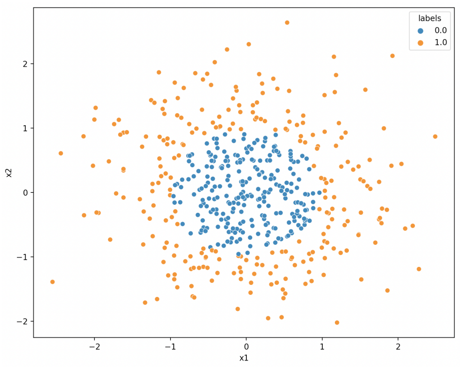

Figure (1) Data visualization

2.2 Creating a multi-layer perceptron from scratch

The created multi-layer perceptron networks have a dynamic configuration to give the possibility of easily changing the network configurations in order to find the optimal values of their parameters. Therefore, all the operations are done with vectors which not only helps to easier compute the operations but also requires less computational time.

The neural network is organized in layers and its initialization begins with the initialization of its parameters. Each neuron has a set of weights and biases that are stored in a dictionary and afterwards updated to better represent the dataset. The input layer receives samples of the training dataset and a single neuron is defined in the output layer. Different values of initialization have been taken for weights and bias, but all between the range of 0 and 1.

The output of each neuron is computed in the forward propagation process, where the input signal propagates through each layer until the output layer. Each neuron has a weighted input from each neuron of the previous layer plus the bias of the neuron. Neuron´s outputs are activated with the sigmoid function to add non-linearity to the model returning values between 0 and 1. In each epoch the layer's output values are stored for the backpropagation process for computation efficiency.

The cross entropy function is used for measuring the error between what our model predicts and the real labels of the sample due to the faster convergence (first derivative with higher slope for low probabilities). Additionally, the number of miss classifications are kept in each epoch to obtain the accuracy.

To improve the performance of the model and minimize the cost function of the model, the back-propagation process is carried out through the stochastic gradient descent optimization technique of loss functions with respect to the NN parameters (weights and bias), with the gradient of the final layer being calculated first and the gradient of the first layer calculated last. This backwards flow allows for efficient computation of the gradient at each layer versus the approach of calculating the gradient of each layer separately.

Once the network has computed the gradients of the cost functions, parameters are updated proportional to their gradient regulated by the learning rate, a hyperparameter that controls how much to change the model in response to the estimated error each time the model parameters are updated. Different step sizes are tested to reduce the number of iterations needed to converge in global minima.

This training process is repeated a number of iterations while the network updates the parameters and this way improves how it generalizes to similar data in relation to the training data. The best optimal structure was found through a process of trial and error experimentation.

Metrics of the training process like cost function values, decision boundaries or accuracy of the model are displayed for model performance evaluation.

2.3 Creating a multi-layer perceptron with Keras framework.

In this section the NN created in the Keras framework is going to be described. First of all, a Sequential model is defined that allows to create models layer-by-layer and Dense layers are added with the same number of neurons and activation function such as the previous described model to compare the performance of both models.

Similar loss function was used and metrics of the Keras model framework were extracted (accuracy and loss) for measuring the performance of the model, metrics that are recorded at the end of each epoch on the training process.

This training process is repeated a number of iterations while the network updates the parameters and this way improves how it generalizes to similar data to the training dataset.

Metrics of the training process like cost function values or accuracy of the model are displayed for model performance evaluation.

3 Results

In this section results of both models are presented and their performance is compared. Multiple configurations have been tested to find the best network configuration structure for the given task. As starter configuration of the models, we selected a hidden layer of 4 neurons trained with the stochastic gradient descent technique in 10 iterations with a learning rate of 0.1. The weights were initialized with the standard normal distribution divided by 10.

None of the models converged in a solution, although they reach an accuracy of 94% the created from scratch model (model 1 from now on) and 82% the Keras model. Comparing the evolution of the cost function of both models we see that the cost function stayed steady in the firsts iterations. This could be due to the random initialization of the parameters that takes a significant amount of repetitions to converge to the least loss and reach the ideal weights matrix due to the far starter values. Furthermore, it is prone to vanishing or exploding gradient problems.

In the next experiment, the model 1 was trained with a higher learning rate to try to converge the model. Both models improved the accuracy and converged in a better solution. Although the accuracy of the model is high, a better tuning of the parameters and architecture can be reached.

The Xabier weight initialization technique was applied and the learning rate was increased to improve the performance of the model 1 and speed up the convergence of gradient descent. Furthermore, more neurons were added to give more deepness to the NN. With a hidden layer with 8 neurons and a training step of 0.4 the accuracy of the models increased to 97% and the convergence was faster.

Additionally, different architectures were tested by adding new layers with different numbers of neurons. Despite this, none of the architectures with more than one hidden layer reach similar levels of performance. On the contrary, the performance drops, requiring more iteration to converge and with higher fluctuations. By adding more depth to the single hidden layer the models performed dissimilarly. While the model one raised the model accuracy to 97.4% with 12 neurons, the Keras model accuracy dipped to 91% (0.2 learning rate). On the other hand, by increasing the learning rate with the same layout the performance of the Keras model increased and the model 1 decreased ending the model 1 in 92.8% accuracy and the Keras model in 96.2%. Other experiments were run initializing the weights with values equal to zero. The model required more iteration to converge in an optimal solution.

After testing with different parameters values and architecture, we assume that more training iterations are needed to increase the accuracy. The final model selected as optimal has 8 neurons in a single hidden layer, with a training rate of 0.2 and trained in 15 epochs. The model 1 obtained best metrics (99% of accuracy) despite both models had the same configuration. However, the standard deviation of the model 1 after running 10 iterations is of 0.41 and the train-test performance computed(Fig. 2)

In addition, other experiments were run by training the models with the batch gradient descent algorithm ending in similar results but in a more stable error gradient.

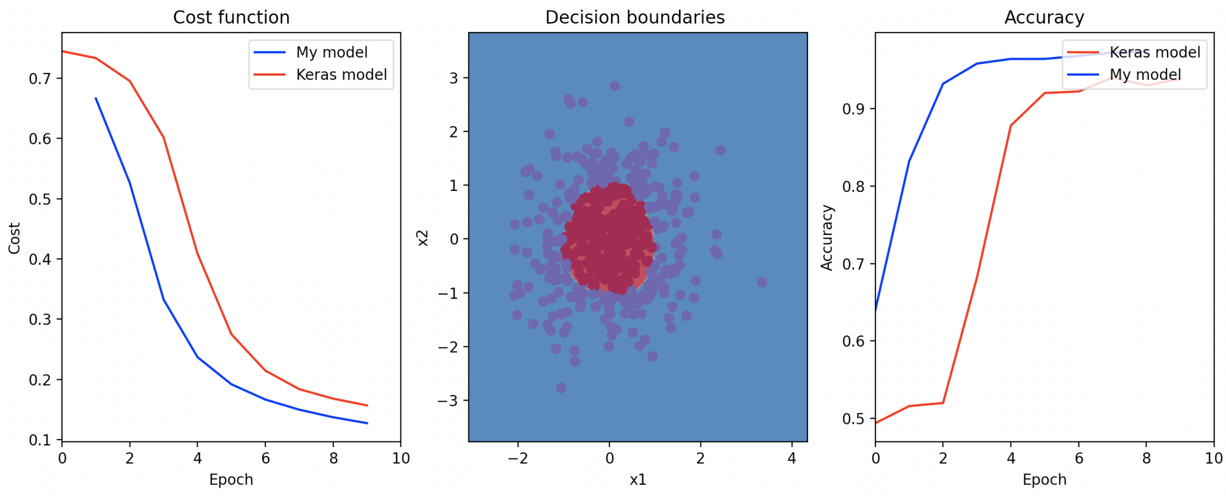

Figure (2) Performance of the models with one hidden layer with 8 neurons after 10 iterations. Training step = 0.2.