Artificial Intelligence

Back Propagation

Claudio Hernández and David Ochoa

November 29, 2024

Index

1 Introduction

The implemented Algorithm tries to provide a solution for the classification of wine quality. We focus on these two data sets of wines for training our neural network using the BackPropagation algorithm.

2 Explanation

To start with, we have two classes in our program: Network and Neuron. The network is composed of 11 neurons because this is the amount of attributes for the problem. Each of those neurons receive inputs like the ph of the wine and calculates an output. In addition, we have one hidden layer and an output layer, so the predictions from one layer are used by the next neurons in the network.

3 Treatment of data

To explain in more detail the data set, we have the attributes as follows: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol and finally, the quality. The two csv contains the same parameters but the red wine has 1800 inputs and white wine has 4900. As you can see, some attributes for their own nature will have huge different values like density and total sulfur dioxide. The difference between the values of these two parameters is a problem for our neural network because it produces more significant weights for some attributes, so the first movement is to normalize every attribute, the method for this process is coded in the Data Class when we read the data and we use the common way to normalize: value-min/max-min. We have to say how we treat the quality parameter for comparing with our predictions, as the quality is some value between 0 and 10, and we are using 11 neurons, we can use each neuron to represent one quality: Neuron index 0 is for quality 0 Neuron index 1 is for quality 1 and so on. Using that we can represent the quality in a binary array of size 10, where the quality is marked by number 1, example: quality 1 is 0,1,0,0,0,0,0,0,0,0. This way is useful to compare the outputs from the neurons, because we'll mark the neuron with the better output with 1 and the others with 0, after that we will represent the state of the outputs as an array of 0's and 1's. Finally, in our implementation, the neurons contains the weights for the connections behind the neuron also start with a random bias, the weights in the other hand use the He normal initialization that consists in this rule:

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

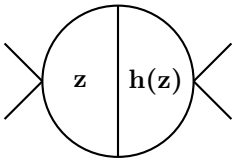
Where n_{in} is the number of inputs (in our case is 11), and we use this initialization because the performance is better if you are using ReLU functions. Therefore, we choose a random number between 0 or the root.

4 Algorithm: BackPropagation

Basically, BackPropagation consists of 3-4 steps: calculate the predictions for every neuron, calculate the error for the final layer, adjust the weights for the neurons and repeat. So we are doing this in the following functions: forward(x) for calculating the predictions, backward(x) for calculating the error and learning to adjust the weights.

4.1 Forward (x)

First of all, we have the forward function so we need to talk about how we calculate the predictions for a neuron and for this is useful thinking in the neurons composed of two values: z calculated with the inputs, and h(z), the evaluation of z in some function.



As the neuron has inputs from his left side we need to calculate the value z from all inputs in order to get our prediction represented by h(z) and we do this using this formula:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

We took the weights of the neuron and multiplied with their respective input and sum these values, after that we sum the bias. Once we have the z value from the inputs we need to evaluate our z in some function to get our predictions, in our project we are using two functions: ReLu and SoftMax.

4.2 ReLu

$$\text{ReLU}(x) = \max(0, x)$$

We use ReLU in hidden layers for two principal reasons: it's pretty easy to calculate and, more importantly, it helps us with the vanishing gradient problem. In the beginning, we used the sigmoid activation, but the results with only one hidden layer were too bad. We got between 30%-40% accuracy in the predictions, so we decided to implement ReLU, which consists only of calculating the $\max(0, z)$ value.

It is important to mention the potential problem with ReLU, the Dying ReLU problem: as the output for the neuron can be 0, the neuron could "die" and present a problem for deep neural networks. This problem can be handled using Leaky-ReLU that uses $\max(\lambda x, x)$, where λ is any small value like 0.01.

In our implementation, we use only ReLU activation because the neural network is too small for that problem.

4.3 SoftMax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

The softmax function is used in the output layer. This function calculates the exponential of each z calculated by the neurons and then divides each exponential by the sum of all the exponentials. The benefits for using softMax instead Sigmoid for example, consists in the normalization provided, as the values are some x between 0 and 1, we can see the outputs of each neuron like probabilities, then we select the neuron with the highest probability and is mark like a 1 in our array of predictions. Other reason for using SoftMax is related with the type of our problem, we are trying to classificate wines in multiples categories, for this softmax is better, also we using the combination of softmax with other loss function: cross-entropy, that combination simplifies the calculus in our network.

In summary, In the Forward function we explore each neuron from left side to the right side and we calculate their z values and his output value with ReLu if it's a hidden neuron, for the final layer we use softmax to get our predictions based on the probabilities provided by softmax.

4.4 Backward

When we are done with the first step we need to calculate the error for the final layer in order to know how good our prediction is.The Backward function is in charge of this process, we explore each neuron again but in the contrary direction in order to calculate their error. In our project we are "using" the Cross-entropy as a loss function, in fact we are using his derived, because the mix with softmax function allow us to express de errorweight (the gradient) in this simple way (thanks to the chain rule):

$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

That means we use the prediction minus the expect output 1/0 to get the gradient, but this is only in the final layer due to softmax function used is this layer, for the hidden layer we are using the error from the next layers so we use:

$$\delta_i^{(l)} = \left(\sum_j \delta_j^{(l+1)} \cdot w_{ji}^{(l+1)} \right) \cdot f' \left(z_i^{(l)} \right)$$

Basically we took the delta errors and the weights for the neuron from the next layer and we calculate the summatory after that we use the derived of our activation function, in the hidden layers is ReLu so we do:

$$f'(z) = \begin{cases} 1 & \text{si } z > 0, \\ 0 & \text{si } z \leq 0. \end{cases}$$

So we visit each neuron in an opposite direction and we use one of these two forms to get the

$$\frac{\partial \text{error}}{\partial \text{weight}}$$

for the neuron.

4.5 Learning(x)/Adjust(x)

The process of adjusting the weights is very easy, we follow this equation:

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial E}{\partial w_i}$$

We took the actual weight and we do the difference between w_i and the gradient of the neuron multiplied by our η which is the learning rate. For the bias, it is something similar, but easier as we don't have to use the gradient:

$$b \leftarrow b - \eta \cdot \frac{\partial E}{\partial b}$$

Now, this is the standard version. In order to get better results, we must implement the L2 regularization. This method sanctions big weights and helps with the stability of the neural network. To apply this, we only need to add a lambda term multiplied by the weight or the bias. So in fact, we are using these formulas:

$$w_i \leftarrow w_i - \eta \cdot \left(\frac{\partial E_{\text{original}}}{\partial w_i} + \lambda \cdot w_i \right)$$

$$b \leftarrow b - \eta \cdot \left(\frac{\partial E_{\text{original}}}{\partial b} + \lambda \cdot b \right)$$

So with these two formulas we explore every neuron and update their weights and bias.

5 Training and results

Explained the 3 steps before, we can tell how we train our model. Using a function, we repeat the process of Forward(x) → Backward(x) → Adjust(x) n times. Also, in every iteration, we adjust the learning rate with a factor: a decayRate that follows an exponential decay:

$$\eta_{\text{new}} = \eta_{\text{initial}} \cdot (\text{decayRate})^{\text{iteration}}$$

Adding this term was necessary to get better results. Probably, this term was the most important change in the code. Then we tried with values of 0.99 to 0.9999, and we observed an improvement in accuracy 10% for both datasets.

In summary, we do the forward (x) → backward (x) → adjust (x) process and decrease the learning rate in late iterations to avoid exploration at the end and focus more on exploitation, controlling the velocity of the learning. For values closer to 1, the learning is slower.

Another important fact is how we manage the inputs for training, as the sizes of datasets are different. We use 80% of the data for training and 20% left for evaluating our algorithm. This involves introducing the inputs, calculating the outputs with the Forward(x) function, and comparing the expected versus the predicted quality (with the array method mentioned before).

After that, we counted the hits and misses of the algorithm. The highest rate obtained was like 60%. We can change the parameters: number of layers, iterations, learning rate, λ, and decayRate to get different results, as we can see in the table below:

6 For red wine data set

Changing the decayRate

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.0005	0.000007	0.99	42.3%
2	10000	0.0005	0.000007	0.999	50.7%
2	10000	0.0005	0.000007	0.9999	51.7%
2	10000	0.0005	0.000007	0.99999	55.1%
2	10000	0.0005	0.000007	0.999999	51.7%

Changing the lambda

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.0005	0.0007	0.99999	52.0%
2	10000	0.0005	0.00007	0.99999	52.9%
2	10000	0.0005	0.000007	0.99999	54.2%
2	10000	0.0005	0.0000007	0.99999	55.7%
2	10000	0.0005	0.00000007	0.99999	54.8%

Changing the Learning Rate

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.05	0.0000007	0.99999	6.5%
2	10000	0.005	0.0000007	0.99999	42.6%
2	10000	0.0005	0.0000007	0.99999	52.6%
2	10000	0.00005	0.0000007	0.99999	50.1%
2	10000	0.000005	0.0000007	0.99999	44.8%

7 For white wine data set

Changing the decayRate

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.0005	0.0000007	0.99	52.8%
2	10000	0.0005	0.0000007	0.999	54.9%
2	10000	0.0005	0.0000007	0.9999	57.6%
2	10000	0.0005	0.0000007	0.99999	54.4%
2	10000	0.0005	0.0000007	0.999999	56.2%

Changing the lambda

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.0005	0.0007	0.9999	54%
2	10000	0.0005	0.00007	0.9999	57.6%
2	10000	0.0005	0.000007	0.9999	56.3%
2	10000	0.0005	0.0000007	0.9999	58.5%
2	10000	0.0005	0.00000007	0.9999	56.6%

Changing the Learning Rate

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	10000	0.05	0.0000007	0.9999	15.4%
2	10000	0.005	0.0000007	0.9999	57.8%
2	10000	0.0005	0.0000007	0.9999	56.4%
2	10000	0.00005	0.0000007	0.9999	53.2%
2	10000	0.000005	0.0000007	0.9999	52.7%

Using the best parameters for red wine

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	20000	0.0005	0.0000007	0.99999	54.2%
2	30000	0.0005	0.0000007	0.99999	60.5%
2	40000	0.0005	0.0000007	0.99999	60.5%
2	50000	0.0005	0.0000007	0.99999	57.9%
2	60000	0.0005	0.0000007	0.99999	57.9%

Using the best parameters for white wine

Number of layers	Iterations	Learning Rate	Lambda	decayRate	Accuracy
2	20000	0.005	0.0000007	0.9999	59.6%
2	30000	0.005	0.0000007	0.9999	61.4%
2	40000	0.005	0.0000007	0.9999	62.8%
2	50000	0.005	0.0000007	0.9999	61.7%
2	60000	0.005	0.0000007	0.9999	59.9%