MNIST Classification Neural Network

From scratch using C++

Vasilis Kyriafinis, 9797

Aristotle University Of Thessaloniki
Neural Networks
November 2022

Contents

List of plots

Introduction

Assignment description

The main point of focus of this project is to create a Neural Network and train it on a dataset. For the purposes of this assignment the MNIST data set was used. The code for this assignment can be found in this GitHub repository.

The MNIST dataset

"The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting." [3]

Each image is a 28x28 pixel gray-scale image. All the images are labeled with the correct digit. The purpose of this assignment is to create two + 1 classification algorithms that are able to identify the images in the test dataset.

State of the Art

Due to the fact that the MNIST dataset is a very well known dataset, there are many successful attempts in the classification. The majority of them use the python language and a relative library.

Objective and Methodology

Objective

This assignment will implement the algorithms found in those libraries in C++ to the best ability of the author. The goal is to create a flexible framework that can be used to create and train neural networks. The neural networks will be fully connected. The framework will be able to create and train neural networks with any number of layers and any number of neurons per layer. Also there will be the ability to choose the activation function for each layer and the initialization method for the weights.

Methodology

Fully connected neural networks

Fully connected neural networks are networks where each neuron in one layer is connected to every neuron in the next layer. Every layer has a predetermined number of neurons. The first layer is the input layer and the last layer is the output layer. The input layer in this project will be 748 neurons, one for each pixel in the MNIST dataset. The output layer will be 10 neurons, one for each digit.

Network evaluation

Once the network is created, the weights are initialized and the training begins. In order to determine the optimal parameters for the network, such as the learning rate, the number of epochs, the number of neurons per layer, the activation function and the initialization method, the network will be trained with different combinations of these parameters. The performance of the network will be measured by the accuracy of the network on the test set.

NN Algorithms

Neural Networks notation

Neural networks are total of neurons that are connected to each other. Each neuron has a set of inputs with the corresponding weights, a bias and an output. The way the neuron works is shown in the following equation:

$$a_i^l = f\left(\sum_{j=1}^{n_{l-1}} w_{ij}^l x_j^{l-1} + b_i^l\right) \tag{1}$$

Where:

- a_i^l is the output of the neuron i in the layer l
- f is the activation function
- w_{ij}^l is the weight of the connection between the neuron j in the layer $l-1$ and the neuron i in the layer l
- b_i^l is the bias of the neuron i in the layer l

This is the notation that will be used throughout the rest of the document.

Network initialization

In order to start the training of the network, the weights and the biases must be initialized. This is a very discussed topic in the field of neural networks. For this assignment, the weights will be initialized with one of the following methods.

Xavier (Glorot) initialization

In 2010, Xavier Glorot and Yoshua Bengio authored a paper [1] that proposed a new method for initializing the weights of a neural network. The method is based on a random normal distribution and on the number of neurons of the layers. The weights are initialized with the following equation:

$$w_{ij}^l \sim \mathcal{N}\left(0, \frac{2}{n_{l-1} + n_l}\right) \tag{2}$$

Where:

- n_{l-1} is the number of inputs of the neuron i in the layer l

- n_l is the number of neurons in the layer l

The Xavier initialization method is best used with the Sigmoid, Softmax and Tanh activation functions.

Kaiming He initialization

Glorot initialization is not a good choice for the ReLU activation function. In 2015, Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun proposed a new initialization method [2] that is better suited for the ReLU activation function. The weights are initialized with the following equation:

$$w_{ij}^l \sim \mathcal{N}\left(0, \frac{2}{n_{l-1}}\right) \tag{3}$$

Where:

- n_{l-1} is the number of inputs of the neuron i in the layer l

The Kaiming He initialization method is best used with the ReLU activation function.

Zero initialization

For testing purposes, instead of initializing the weights with a random distribution, they can be initialized with zeros.

Bias initialization

The biases are not as important as the weights. Many methods suggest that the biases can be initialized with a small constant value. For this assignment though, the biases will be initialized to 0.

Activation functions

The activation function is a function that is applied to the weighted sum of the inputs to obtain the output of a neuron. There are many activation functions that can be used in a neural network. For this assignment, the following activation functions will be implemented.

Sigmoid

The sigmoid activation function is defined as:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{4}$$

The derivative of the sigmoid function is will also be needed for the backpropagation algorithm. The derivative of the function is defined as:

$$f'(z) = f(z) \cdot (1 - f(z)) \tag{5}$$

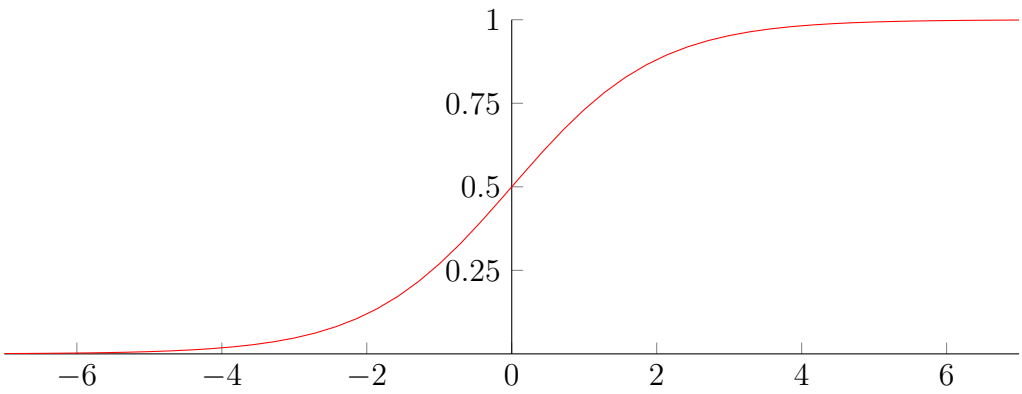

Figure 1: Sigmoid function

ReLU

The ReLU activation function is defined as:

$$f(z) = \max(0, z) \tag{6}$$

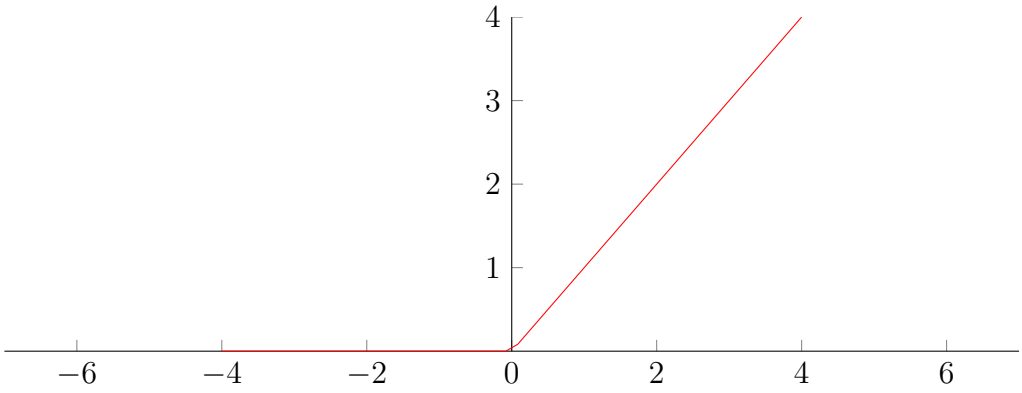

Figure 2: ReLU function

The derivative of the ReLU function is will also be needed for the backpropagation algorithm. The derivative of the function is defined as:

$$f'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \tag{7}$$

Softmax

The softmax activation function is defined as:

$$f(z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{8}$$

Where:

- z_i is the weighted sum of the inputs of the neuron i
- n is the number of neurons in the layer

Softmax is a special activation function that is used in the output layer of a neural network. It is used for classification problems. The output of the softmax activation function is a probability distribution. The probability of a class is the value of the output of the neuron that represents that class.

The derivative of the softmax function is will also be needed for the backpropagation algorithm. The derivative of the function is defined as:

$$\frac{df(z_i)}{dz_j} = \begin{cases} f(z_i) \cdot (1 - f(z_i)) & \text{if } i = j \\ -f(z_i) \cdot f(z_j) & \text{if } i \neq j \end{cases} \tag{9}$$

Training

After the network is initialized, the training can begin. The training is done by using the backpropagation algorithm along the MSE (Mean Square Error) function. But before that the forward pass must be done.

Forward pass

The forward pass is the process of calculating the output of the network. The output of the network is the prediction of the network. The forward pass is done by calculating the weighted sum of the inputs of each neuron and applying the activation function to the weighted sum. The output of the activation function is the input of the next layer. The forward pass is done for each layer of the network. The output of the last layer is the output of the network.

Passing a single image through the network and then calling the backpropagation algorithm is not very efficient. For this reason the training images are divided into batches. The forward pass is done for each batch. After every image the errors are calculated but the weights are updated at the end of the batch.

Backpropagation

The backpropagation algorithm is used to update the weights of the network. After an image is passed through the network, an output is obtained. The output is compared to the expected output and the error is calculated. The error function is the MSE function. The MSE function is defined as:

$$E = \frac{1}{2} \sum_{i=1}^{n} (y_i - t_i)^2 \tag{10}$$

Where:

- y_i is the output of the neuron i
- t_i is the expected output of the neuron i
- n is the number of neurons in the output layer

The goal of the backpropagation algorithm is to change the weights of the network in such a way that the error is minimized. To do this, the gradient of the error function with respect to all the weights and biases must be calculated.

Output layer

The target is to calculate the partial derivatives of the error function with respect to the weights and biases.

$$\frac{\partial E_i}{\partial w_{ij}}, \frac{\partial E_i}{\partial b_i} \tag{11}$$

Using the chain rule, the partial derivatives of the error function with respect to the weights and biases equals

$$\begin{aligned} \frac{\partial E_i}{\partial w_{ij}} &= \frac{\partial E_i}{\partial a_i}\frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ij}} \\ \frac{\partial E_i}{\partial b_i} &= \frac{\partial E_i}{\partial a_i}\frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial b_i} \end{aligned} \tag{12}$$

Where:

- a_i is the output of the neuron i
- z_i is the weighted sum of the inputs of the neuron i
- w_{ij} is the weight between the neuron i and the neuron j of the previous layer
- b_i is the bias of the neuron i

As a reminder:

$$a_i^l = f(z_i^l) \tag{13}$$

and

$$z_i^l = \sum (a_j^{(l-1)} \cdot wij) + b_i^l \tag{14}$$

The partial derivative of the error function with respect to the output of the neuron is calculated as:

$$\frac{\partial E_i}{\partial a_i} = a_i - t_i \tag{15}$$

The partial derivative of the output of the neuron with respect to the weighted sum z_i is simply the derivatives of the activation function:

$$\frac{\partial a_i}{\partial z_i} = f'(z_i) \tag{16}$$

Finally the partial derivative of the weighted sum with respect to the weight w_{ij} is simply the output of the previous neuron a_j^{l-1}:

$$\frac{\partial z_i}{\partial w_{ij}} = a_j^{(l-1)} \tag{17}$$

and the partial derivative of the weighted sum with respect to the bias b_i is simply 1:

$$\frac{\partial z_i}{\partial b_i} = 1 \tag{18}$$

From the equations 12 and equations 15, 16 we can identify a common "error" term:

$$\delta_i^l = \frac{\partial E_i}{\partial a_i}\frac{\partial a_i}{\partial z_i} = (a_i - t_i) \cdot f'(z_i) \tag{19}$$

The δ_i term can be calculated for each neuron in the output layer. After that using this term the gradient of the error function with respect to the weights and biases can be calculated as follows:

$$\begin{aligned} \frac{\partial E_i}{\partial w_{ij}} &= \delta_i^l \cdot a_j^{(l-1)} \\ \frac{\partial E_i}{\partial b_i} &= \delta_i^l \end{aligned} \tag{20}$$

Hidden layers

The equation 20 stands for all the layers of the network. The only thing missing is the calculation of the δ_i term for the hidden layers. The δ_i term for the hidden layers is backpropagated as follows:

$$\delta_i^{(l)} = \sum_{j=1}^{n} \delta_j^{(l+1)} \cdot w_{ij}^{(l+1)} \cdot f'(z_i^l) \tag{21}$$

Where:

- $\delta_j^{(l+1)}$ is the error term of the neuron j of the next layer
- $w_{ij}^{(l+1)}$ is the weight between the neuron i of the layer l and the neuron j of the next layer

It is easily seen that the δ^l is dependent on the δ and the weights of the next layer. This means that It can be calculated as long as the δ and the weights of the next layer are known. Since the δ and the weights of the output layer are known, the δ term for all the layers of the network can be calculated.

Weight and bias update

Once the gradient of the error function with respect to the weights and biases is calculated, the weights and biases can be updated as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \cdot \frac{\partial E_i}{\partial w_{ij}} \tag{22}$$

$$b_i^{(l)} = b_i^{(l)} - \eta \cdot \frac{\partial E_i}{\partial b_i} \tag{23}$$

Where:

- η is the learning rate

In practice the gradient of the weights and biases is averaged for each training example in a single batch. So the update equations are:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \cdot \frac{1}{m} \sum_{k=1}^{m} \frac{\partial E_i^{(k)}}{\partial w_{ij}} \tag{24}$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \eta \cdot \frac{1}{m} \sum_{k=1}^{m} \frac{\partial E_i^{(k)}}{\partial b_{i}} \tag{25}$$

Where:

- m is the number of training examples in the batch

Training in practice

The steps to train the network in practice are the following:

1. Create the batch of training samples by randomly selecting images

   For each image in the batch:

   - Calculate the output of the network by propagating the input through the network
   - Calculate the δ error term of the output layer using equation 19
   - Backpropagate the error term to calculate the error term of the hidden layers using equation 21
   - Calculate the gradients all the weights and biases using equation 20 and add them to the respecting average gradient sum

2. Update the weights and biases using equation 24 and 25
3. Repeat the steps 1-2 for the next batch

Results

At this point the plan was to start testing different configurations for the network and see which one performs the best. Unfortunately, despite the fact that cade implements the algorithms exactly as they are described in the literature, the network does not perform as expected. The network, regardless of the configuration, achieves an accuracy of 10% on the test set. This is the same as if the network randomly guesses the digit.

The behavior of the network is that despite the fact that the weights are updated, the the accuracy stays the same. After many hours of debugging and trying to find the problem, I was unable to find the cause of the problem.

In a nutshell, the learning rate was set from 0.00001 to 100 and it did not make any difference. Also, the batch size and number of epochs were changed and the accuracy stayed the same. Finally the number of layers also did not make any difference.

Bibliography

[1] Xavier Glorot Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010. URL: https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[2] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. URL: https://arxiv.org/abs/1502.01852.

[3] *MNIST*. http://yann.lecun.com/exdb/mnist/.