

Handwritten Digit Recognition

A Neural Network

Term Paper



Submitted by:

Qaiser Abbas

P12-6055

Submitted to:

DR. Muhammad Nouman

Date: May 25, 2018

Contents

Abstract

Introduction

Data Collection

Methodology

- Perceptrons
- Sigmoid Neurons
- Stochastic Gradient Descent

Implementation

Conclusion

References

Abstract:

In this project, a Handwritten Digit Recognition system through a Neural Network is built based on the MNIST dataset. But along the way we'll develop many key ideas about neural networks, including two important types of artificial neuron (the perceptron and the sigmoid neuron), and the standard learning algorithm for neural networks, known as stochastic gradient descent.

Introduction

Handwritten digit recognition is already widely used in the automatic processing of bank cheques, postal addresses, etc. Some of the existing systems include computational intelligence techniques such as artificial neural networks or fuzzy logic, whereas others may just be large lookup tables that contain possible realizations of handwritten digits. Artificial neural nets have successfully been applied to handwritten digit recognition numerous times, with very small error margins

The work described in this paper does not have the intention to compete with existing systems, but merely served to illustrate to the general public how a neural network can be used to recognize handwritten digits.

Data Collection

The MNIST dataset is used in our project. MNIST is a database containing images of handwritten digits, with each image labeled by integer. The MNIST dataset contains a **training set** of 60,000 examples, and a **test set** of 10,000 examples. The training set is used to teach the algorithm to predict the correct label, the

integer, while the test set is used to check how accurately the trained network can make guesses.

Methodology

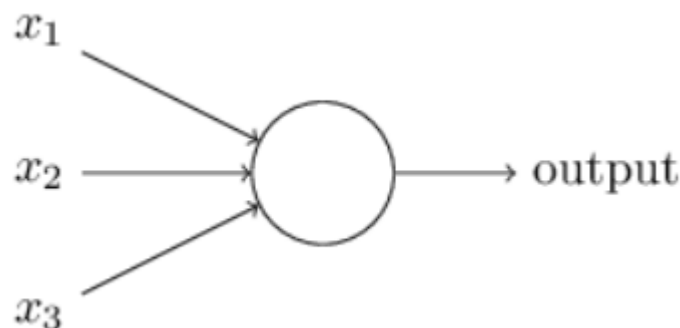
Having defined neural networks, let's return to handwriting recognition.

The followings are the techniques which leads Neural Network to recognize the handwritten images.

- Perceptrons
- Sigmoid Neurons
- stochastic gradient descent

Perceptrons

A perceptron takes several binary inputs, x_1, x_2, \dots, x_n , and produces a single binary output:



In general it could have more or fewer inputs.

Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, w_1, w_2, \dots, w_n , real numbers expressing the importance of the respective inputs to the output.

To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Sigmoid Neurons

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network?

To see how learning might work, suppose we make a small change in some weight (or bias) in the network. And we could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

The problem is that this isn't what happens when our network contains perceptrons. We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Just like a perceptron, the sigmoid neuron has inputs, x_1, x_2, \dots . But instead of being just 0 or 1, these inputs can also take on any values *between* 0 and 1. So, for instance, 0.638... is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w_1, w_2, \dots , and an overall bias, b . But the

output is not 0 or 1. Instead, it's $\sigma(w \cdot x + b)$, where σ is called the *sigmoid function* and is defined by:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}. \quad (3)$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs x_1, x_2, \dots , weights w_1, w_2, \dots , and bias b is

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}. \quad (4)$$

At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding if you're not already familiar with it. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

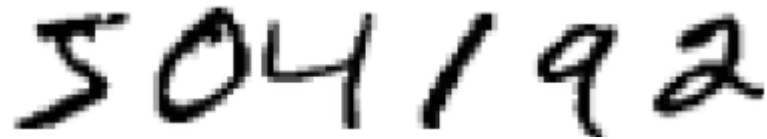
In fact, calculus tells us that Δ_{output} is well approximated by.

$$\Delta_{\text{output}} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b,$$

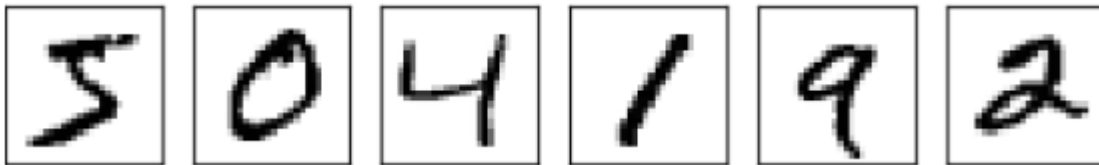
Neural Network

Having defined neural networks, let's return to handwriting recognition. We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images,

each containing a single digit. For example, we'd like to break the image

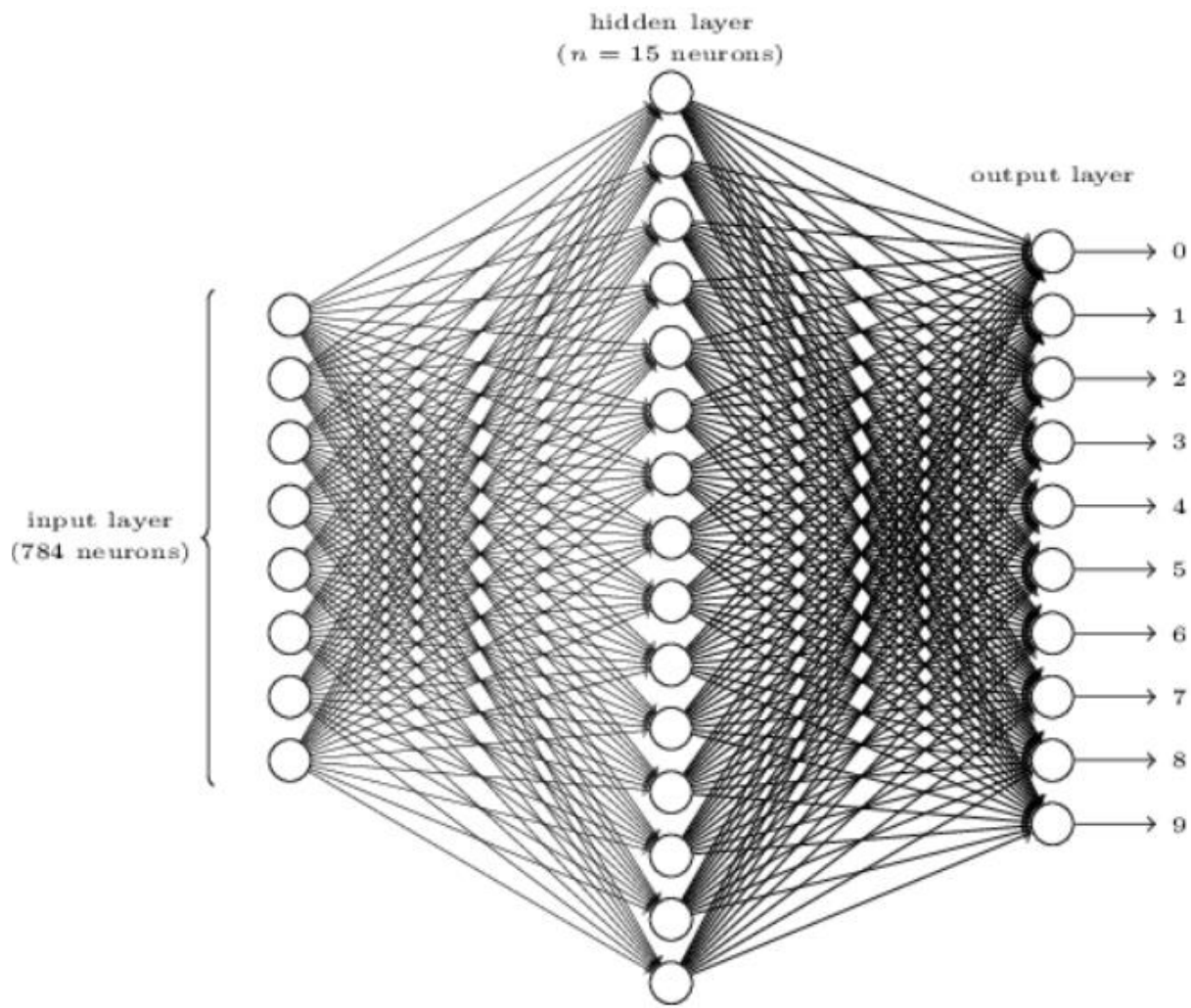
A handwritten number '504192' in a cursive, slightly slanted font.

into six separate images,



There are many approaches to solving the segmentation problem. One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation. A trial segmentation gets a high score if the individual digit classifier is confident of its classification in all segments, and a low score if the classifier is having a lot of trouble in one or more segments.

To recognize individual digits we will use a three-layer neural network:



The input layer of the network contains neurons encoding the values of the input pixels.

Our training data for the network will consist of many 2828 by 2828 pixel images of scanned handwritten digits, and so the input layer contains $784=28\times28$ neurons.

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by NN , and we'll experiment with different values for NN . The example shown illustrates a small hidden layer, containing just $n=15$ neurons.

The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈ 1 , then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

Learning with Gradient Descent

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x . To quantify how well we're achieving this goal we define a *cost function*.

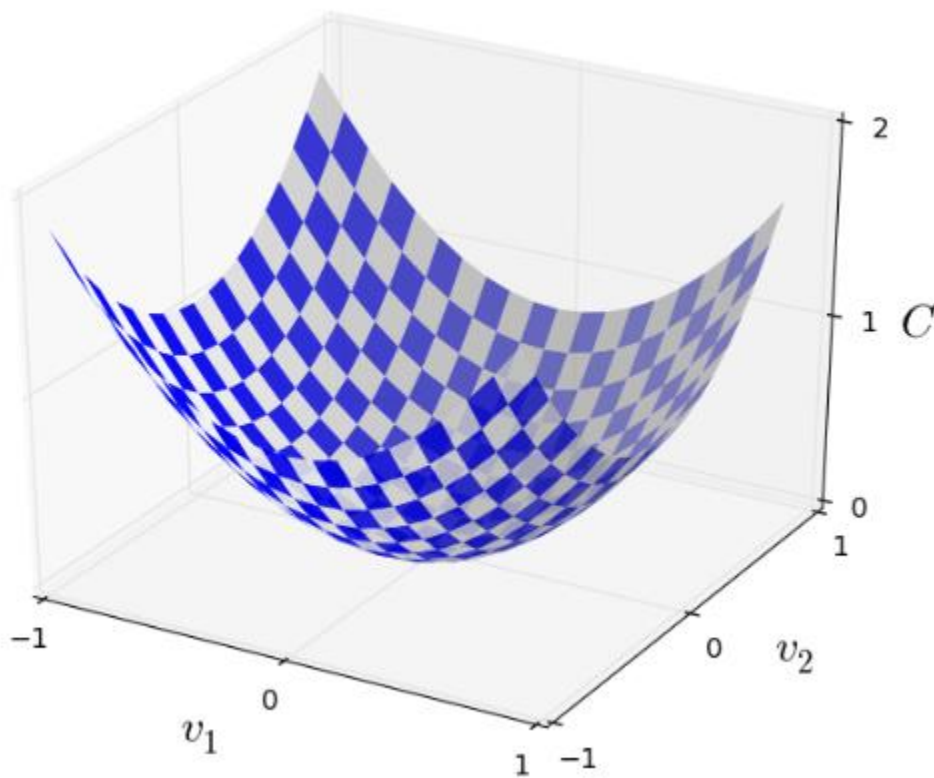
$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

We want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known

as *gradient descent*. Recapping, our goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w,b)$.

Okay, let's suppose we're trying to minimize some function, $C(v)$. This could be any real-valued function of many variables, $v = v_1, v_2, \dots$.

To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2 :



What we'd like is to find where C achieves its global minimum. Now, of course, for the function plotted above, we can eyeball the graph and find the minimum.

One way of attacking the problem is to use calculus to try to find the minimum analytically. We could compute derivatives and then try using them to find places where CC is an extremum. With some luck that might work when CC is a function of just one or a few variables. But it'll turn into a nightmare when we have many more variables.

let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that CC changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

We denote the gradient vector by ∇C , i.e.:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T.$$

I've explained gradient descent when CC is a function of just two variables. But, in fact, everything works just as well even when CC is a function of many more variables. Suppose in particular that CC is a function of m variables, v_1, \dots, v_m . Then the change ΔC in CC produced by a small change $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v,$$

where the gradient ∇C is the vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T$$

Just as for the two variable case, we can choose.

$$\Delta v = -\eta \nabla C,$$

This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C.$$

Implementation

Using stochastic gradient descent and the MNIST training data. I implemented one machine learning algorithm which gave high accuracy.

Algorithm:

- Neural Network

Conclusion:

The lesson to take away from this is that debugging a neural network is not trivial, and, just as for ordinary programming, there is an art to it. You need to learn that art of debugging in order to get good results from neural networks.

References:

1. DARPA: DARPA Neural Network Study. AFCEA International Press (1988) 60
2. Jain, L.C., Lazzerini, B. (eds.): Knowledge-based Intelligent Techniques in Character Recognition. CRC Press, Boca Raton FL (1999)
3. Kohonen, T.: Self-Organizing Maps. Springer-Verlag, Berlin (1995)
4. LeCun, Y., Jackel, L.D., Bottou, L., Brunot, A., Cortes, C., Denker, J.S., Drucker,