Developing a Neural Network From Scratch

Elliott Chang

Abstract

This document presents a comprehensive overview of a neural network project aimed at recognizing and classifying handwritten digits using the MNIST dataset. The project involves the implementation of a neural network from scratch, providing insights into the code, methodology, and the underlying mathematical principles. Through this work, I explore the intricacies of training a model to perform image classification tasks, shedding light on the architecture, training process, and results achieved.

1 Introduction

Handwritten digit recognition serves as a fundamental problem in the field of machine learning and artificial intelligence. The ability to develop a model capable of accurately identifying and classifying digits not only has practical applications in optical character recognition but also serves as a foundational exercise for understanding neural networks.

In this project, I delve into the MNIST dataset, a benchmark dataset widely used for image classification tasks. The dataset consists of a collection of 28x28 pixel grayscale images of handwritten digits (0 through 9), making it an ideal playground for training and evaluating machine learning models.

The focus of this endeavor is on implementing a neural network entirely from scratch. By doing so, I aim to provide a transparent view into the inner workings of the network, from the fundamental mathematical concepts to the code implementation. This document will guide the reader through the architecture of the neural network, the intricacies of training on the MNIST dataset, and the evaluation of the model's performance.

2 The Math Behind the Madness

In the realm of neural networks, the underlying mathematics revolves around the concept of a perceptron, the basic building block of these models. A perceptron takes multiple input values, each multiplied by a corresponding weight, and sums these weighted inputs. The result undergoes an activation function, introducing non-linearity to the model. This process can be succinctly expressed as

$$\text{output} = \text{activation}\Big(\sum_i \text{input}_i \times \text{weight}_i + \text{bias}_i\Big).$$

Stacking multiple perceptrons in layers creates a neural network. The first layer, known as the input layer, receives the initial features of the data. Subsequent layers, referred to as hidden layers, perform intermediate computations, adjusting weights and applying activation functions. The final layer, the output layer, produces the model's predictions. This architecture is visualized in Figure 1. The connections between neurons, governed by weights, are adjusted during training using algorithms like gradient descent to minimize the difference between predicted and actual outputs. The backpropagation algorithm computes the gradient (derivative) of a loss function with respect to the weights, facilitating this optimization process. The mathematical foundation of neural networks thus lies in linear algebra, calculus, and the interplay between weights and activation functions, enabling these models to learn complex patterns and relationships in data.

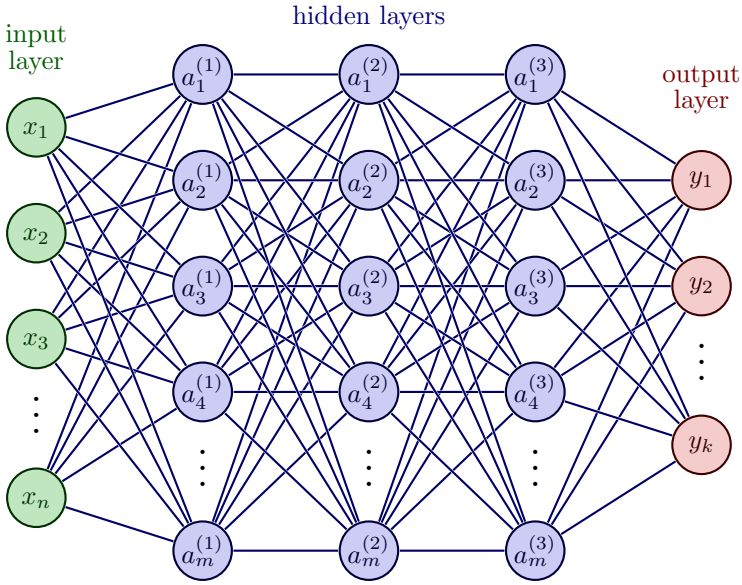

Figure 1: A neural network with 3 hidden layers

For this project, I designed a straightforward neural network comprising a single hidden layer. The input layer, denoted as X, encompasses 784 nodes corresponding to the 784 pixels in each 28×28 pixel image. Each node represents the luminance of its corresponding pixel, ranging between 0 and 1. To enable digit predictions, the output layer requires 10 nodes, each representing a possible digit (0-9). Consequently, the neural network processes 784-dimensional input vectors to produce 10-dimensional output vectors. To handle the entire dataset with m observations, the network takes in a matrix of dimensions $784 \times m$ (where each column represents an individual observation) and outputs a matrix of dimensions $10 \times m$. This matrix-based approach efficiently allows the network to perform parallel computations on the entire dataset during forward and backward passes, optimizing the training process. I chose to incorporate a 10-node hidden layer. For the activation function in the hidden layer, I opted for the simple ReLU function, defined as

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

The activation function guiding the transition from the inactivated output layer values to the activated output layer is the softmax function, defined as

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}},$$

which transforms values from the hidden layer into probabilities. The ensuing equations outline the network's computations:

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= \text{ReLU}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= \text{softmax}(Z^{[2]}).
\end{aligned}$$

However, in order to effectively classify our inputs, we need to tune our weights and biases. This is done through backward propagation. Backward propagation is an iterative process that fine-tunes the model's weights and biases based on how much a given prediction deviated from the true label. After the forward propagation phase, where the input data passes through the network to generate predictions, the backpropagation algorithm is employed to calculate the gradient (derivative) of the loss with respect to each weight in the network. This involves traversing the network in the reverse direction, starting from the output layer and moving backward through the hidden layers to the input layer. The gradient is then used to update the weights, nudging them in the direction that reduces the overall error. This iterative cycle of forward and backward propagation continues until the model's performance converges to a desirable state. Using the chain rule, this process is able to identify how much each weight and bias contributed to the overall error. For our particular network, backward propagation is performed using the following equations:

$$\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} (A^{[1]})^T \\
db^{[2]} &= \frac{1}{m} \sum dZ^{[2]} \\
dZ^{[1]} &= (W^{[2]})^T dZ^{[2]} * \text{ReLU}'(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} (A^{[0]})^T \\
db^{[1]} &= \frac{1}{m} \sum dZ^{[1]}.
\end{aligned}$$

To see how these backward propagation equations are computed, watch this video. Based on these

equations, we can update our parameters by

$$W^{[1]} = W^{[1]} - \alpha dW^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha db^{[1]}$$
$$W^{[2]} = W^{[2]} - \alpha dW^{[2]}$$
$$b^{[2]} = b^{[2]} - \alpha db^{[2]}$$

where α is the learning rate of the model, the updated parameters are on the left and the previous parameters are on the right. The more iterations we perform, the more accurate our data will be.

3 Implementation in Python

To see the implementation of this neural network in Python, visit the `app.py` file in this repository. The neural network ended up reaching a training accuracy of about 85%.