Rochester Institute of Technology

Project report

Handwritten digit recognition using Neural Networks

Srinath Obla, Viral Parman, Dhaval Chauhan

April 21, 2017

Introduction

Even with the advancement in technology, one of the primary means of communication and recording of data by humans is in written format. The handwriting of a person is unique characteristic that develops over time with its own variations; It differs from person to person.

To process these documents digitally, we first need to convert them into the digital domain. Considering the rate at which documents are generated, this would be a herculean task without the help of machines. Considering the large variation in handwriting styles and the amount of data to be processed, the solution appears to be automation, specifically Artificial Intelligence.

It would be an enormous task to develop a model to recognize various handwriting styles. Moreover, there is no guarantee that the developed model would necessarily work for styles it is yet to process with acceptable accuracy. In this situation, machine learning can be our solution. This project describes method to recognise handwritten digits. After introducing a solution with a basic cost function, further improvements are made to increase the accuracy of the algorithm.

Proposed method

Dataset

The MNIST database is a record of 60,000 handwritten digit samples, stored in the form of pixel data. This data is has been normalized into a 28 x 28 binary image and along with it, we have the digit which the image represents. The dataset it divided into 50,000 training samples and 10,000 validation samples.
Dataset link: http://deeplearning.net/tutorial/gettingstarted.html

Network structure

Our neural network operates on each pixel value to classify the image, and therefore there are 784(28 x 28) input layers, one for each pixel. Since we would classify the digits into a range of 0-9, there are 10 output layers. The number of neurons in the hidden layer(middle layer) depends on characteristics of the neural network, such as the cost function. In this experiment, we have constructed three neural networks with slight differences in its learning algorithm

- A neural network learning using the **quadratic** cost function
- A neural network learning using the **cross-entropy** cost function
- The above two variants with **regularization**

Every neuron accepts an set of inputs, weighing each of them based on importance, processes its output value.

$$\text{z} = \text{wa} + \text{b}, \tag{1}$$

where w is the weight of the incoming edge, a is the input from the previous node, and b is the bias at this node. The neuron uses a sigmoid function to process z and produce an output.

$$a' = \sigma(z), \tag{2}$$

where a' is the output of the current neuron, and σ is the sigmoid function.

Learning methods

- **Stochastic Gradient Descent**
  To gauge the efficiency and train a neural network, we use a cost function, which records how *wrong* the neural network is. A high cost function value corresponds to a high rate of error. The cost function, including the two mentioned above, make use of the output produced by the network and the actual desired output to produce a value. Gradient descent is used to minimize a function by tweaking the values its based on. In this experiment, we have trained the neural network using stochastic gradient descent, a variation of gradient descent, on its cost function.

  Unlike gradient descent, where we process the entire dataset, stochastic gradient descent executes on randomly selected batch values. Stochastic gradient descent helps us converge faster where we have large datasets.

- **Backpropagation**
  Backpropagation is a correctional method used along with gradient descent, to train a neural network. This algorithm calculates the amount by which the weights and the biases of the neural network must be changed to reduce the cost function, that is, the rate of change of the cost function with respect to the weights and biases. We first calculate the error at the output layer and then propagate the error through the remaining layers.

Quadratic cost function

The quadratic cost function is the square difference between the expected value and the obtained value:

$$C(w,b) = \frac{1}{2n}\sum_x (y(x) - a)^2, \tag{3}$$

where w and b refer to the weights and biases in the network respectively, x being the input, $y(x)$ being the expected output, and a the predicted output.

With respect to the quadratic function, the error calculated at the output layer, δ_j^L, corresponds to the change in cost with respect to the output. By differentiating the cost function with respect to a, we get:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L), \tag{4}$$

where $\sigma'(z_j^L)$ is the derivative of the sigmoid function with value calculated at the output layer. This equation is first used by the backpropagation function.

Backpropagation then uses δ_j^L to 'propagate' the error through previous layers. Using the error obtained at the output layer, the error at level l can be defined as:

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l),$$

where w^{l+1} is the matrix of weights and δ^{l+1} is the calculated error of the layer after l. We then take its *hadamard* product(depicted with $\odot$) with the derivative of $\sigma(z^l)$(z^l is the calculated output before applying the sigmoid function).

Using the two δ errors defined above, we can now derive how the cost function changes with respect to the weights w and bias b. We use these equations to determine how much to change the weights and biases to minimize the cost function:

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}, \tag{5}$$

where a_{in} is the output value from the previous node and δ_{out} is the output error calculated.

$$\frac{\partial C}{\partial b} = \delta, \tag{6}$$

where δ is the output error calculated. On every input, based on the output and the error calculated, the network weights are adjusted to minimize the cost function. But there is an intrinsic issue one might come across while using the quadratic cost function. δ_j^L has $\sigma'(z)$ as a component. When the output nears towards 0 or 1, the sigmoid function becomes flat, and hence $\sigma'(z) \approx 0$. In this case, the neurons would be corrected slowly, that is, the network would learn slowly. In order to correct this issue, we use the cross-entropy function.

Cross-entropy cost function

Keeping the issue encountered while using the quadratic cost function, the cross-entropy function was designed:

$$C = -\frac{1}{n}\sum_x \left[y \ln a + (1-y)\ln(1-a)\right], \tag{7}$$

where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

The change in the cost function with respect to the weights is obtained as follows:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y) \tag{8}$$

We can see that the equation is void of $\sigma'(z)$, which causes a slowdown of learning while using the quadratic cost function. The change in the C with respect to the bias b is as follows, which is also void of $\sigma'(z)$:

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y) \tag{9}$$

With these modifications, the neural network trains much faster when neurons outputs are near saturation(near 0 or 1).

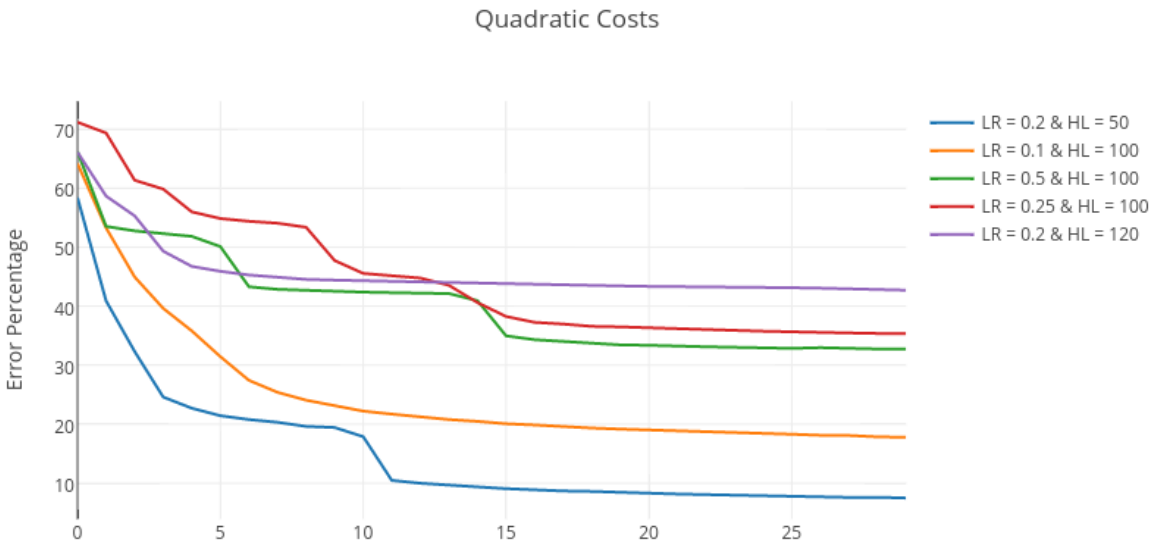
Regularization

Regularization is a method to avoid the overfitting of data. One way to perform this is using *weight decay*. We add an extra term to the cost function, called the *regularization term*. Upon differentiating it with the cost function, the partial derivative of the cost function with respect to the weights is:

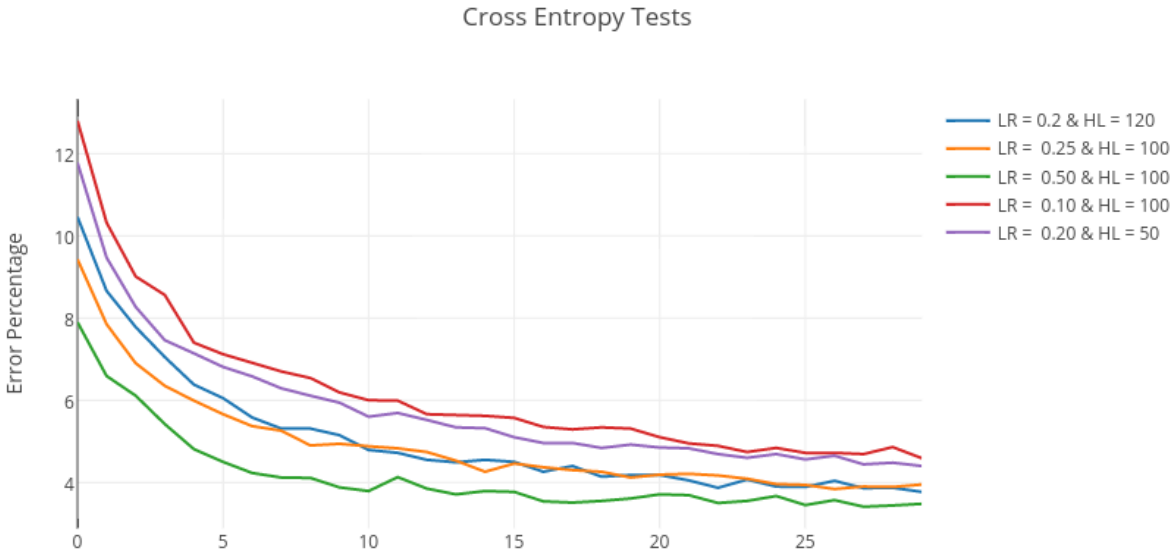
$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w, \tag{10}$$

where λ is the regularization constant.

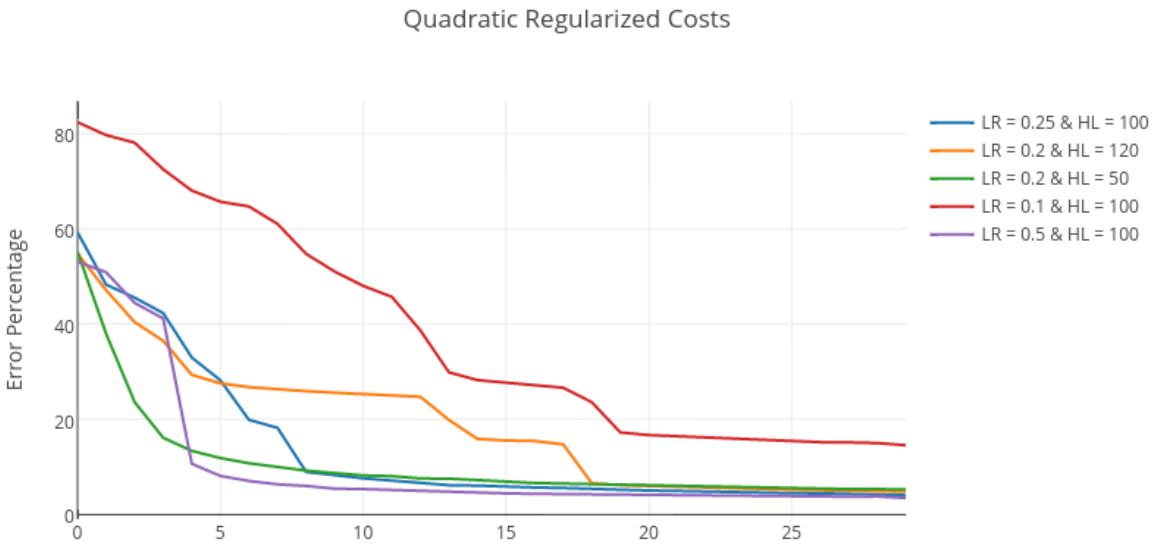
Results and analysis



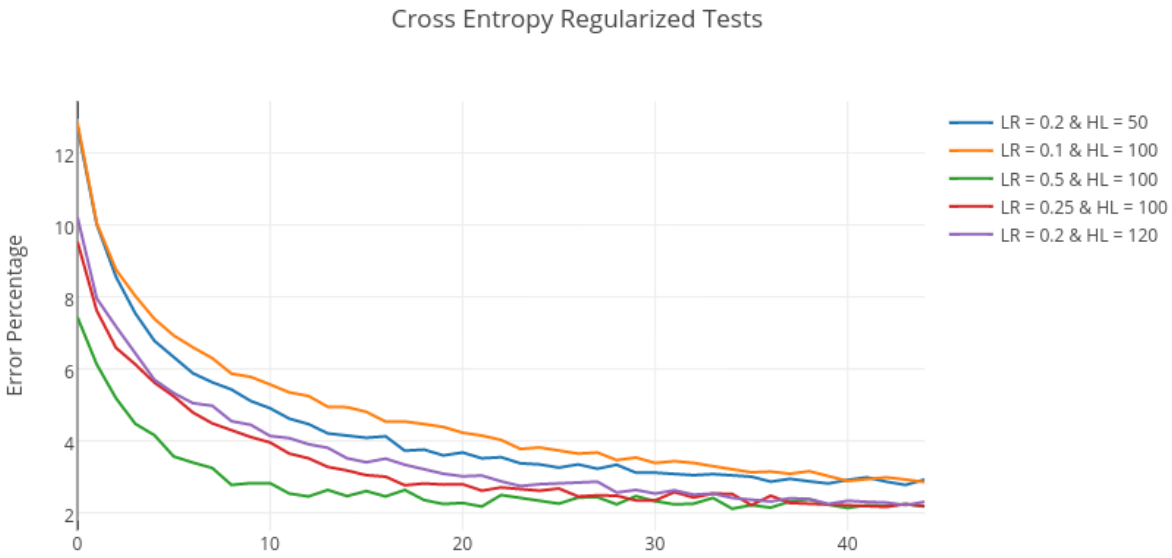
This model uses a quadratic cost function also called as mean squared error. This method doesn't give us good result even with 30 epochs. The best we get from this is a 7.46 percent error rate.



This model uses a cross-entropy cost function. This model gives us good result with best being only 4 percent error.



This model uses the same Mean squared error function used above but also improvizes on it by using a regularization function. which reduces overfitting, but does not contribute to the improvement of accuracy.



This model gives us the best results with minimum error rate of 2.1 percent. This is because the weight changes don't get reduced extremely so the training isn't likely to stall out.

Conclusions

The cross-entropy cost function used along with regularization gives us a good accuracy amongst the ones we have implemented.

References

Y. LeCun, O. Matan, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel and H. S. Baird: Handwritten Zip Code Recognition with Multilayer Networks, in IAPR (Eds), Proc. of the International Conference on Pattern Recognition, II:35-40, IEEE, Atlantic City, invited paper, 1990

O. Matan, J. Bromley, C. Burges, J. Denker, L. Jackel, Y. LeCun, E Pednault, W. Satterfield, C Stenard and T. Thompson: Reading Handwritten Digits: A Zip Code Recognition System., IEEE Computer, 25(7):59-63, July 1992

Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998

http://neuralnetworksanddeeplearning.com, Micheal Nielsen