Optical Character Recognition Using Neural Network And Backpropagation

Tian Lin Tan, Eric Yu

CMPT 414 project report, 2017 Spring

Introduction

Our project is optical character recognition (OCR) using a standard neural network (NN) algorithm. The program is given one set of images of characters and their labels for training, then the program will be asked to identify the labels of a second set of images. The standard backward propagation algorithm will be applied in an attempt to minimize the error. A successful trial should have most of the identifications correct.

Motivations

We chose to implement NN because it is exotic. NN is unlike other programming algorithms we have learned: they require training, answer questions with estimates rather than precise answers, and some amount of randomness is required for initialization. These chaotic aspects makes it feel quite organic. We want to find out why and how it is able to solve complex problems such as OCR. Furthermore, this allows us to explore a different area of the artificial intelligence. The traditional A.I will most likely be confined by the knowledge of its creator. However, this method allows the algorithm to possibly solve problems that we have yet have a firm understanding of. Through this understanding, we can find new ways to apply the algorithm for other problems.

Related works

There are other types of neural networks, including convolutional neural networks (CNN) which makes use of the convolution operation [1], and recurrent neural networks (RNN) which allow cycles in the network and introduces a temporal aspect to the algorithm [2]. They can solve image processing problems. Other types of algorithm such as generalized Hough transform (GHT) and curvature scale space (CSS) can also be used for template matching [3], which could be applied for OCR problems.

Description

High level view

Our neural network is a feedforward neural network. The network uses directed weighted edges, and vertices (called Neurons) are grouped in layers. The layers are ordered, and the neurons of each layer can only connect to the neurons of the following layer. Cycles are not allowed in the network. Each neuron on one layer forms one connection with every neuron in the next layer, so that every pair of adjacent layers form a complete bipartite graph.

The incoming connections for each neuron are called the inputs of that neuron, and the outgoing connections are called the outputs. Each neuron is a function of its input neurons, and produces a value that is in turn the input for another neuron. Beginning from the initial inputs, this function application will happen propagate to the next layers, until the neurons of the last layers produces values. This process is forward propagation.

The operation of each neuron is as follows:

$$O_j = f(\sum W_{ij} O_i + W_{bj})$$

Where O_j is the output of the neuron, O_i is an input value of the neuron, and W_{ij} is a weight of the edge connecting neuron O_i and neuron O_j, W_{bj} is a weight of the edge connecting a constant bias neuron and neuron O_j, and f is the sigmoid function.

The additional bias neuron is a single neuron with the constant value of 1. It has an outgoing connection to every other neuron, each with their own edge weights. This allows the output of the sigmoid function to be shifted, depending on the bias weight value.

The whole network can be thought of as a single function. By adjusting its graph shapes and edge weights to problem specific values, it will become able to estimate the output values with high probability.

Backpropagation

Backpropagation is a popular algorithm for finding the correct weights for a problem. It requires a large set of known input and outputs for the problem. The NN will first run forward propagation on the input to produce an output, then the computed output is used with the true output to compute the error. Then using partial derivatives of the squared error with respect to the weights, the weights between each pair of layers are adjusted in backwards order. Doing this repeatedly will change the weights and shift the squared error towards the negative infinity direction [5]. But because the error is squared, it can never become negative, only approach zero; this effectively minimizes the error. Our implementation follows the equations specified in chapter 4 lecture notes.

The experiment

The main goal of this project is to use backpropagation to find the edge weights for the OCR problem, for a particular set of images known as MNIST. These images are of handwritten digits, and there are a total of 70,000 of them, and they all shared the dimensions of 28x28. 60,000 will be used as training material for backpropagation, and the remaining 10,000 will be used to verify the correctness of the trained network. If it is able to estimate correctly the digits depicted in these images most of the time (over 75%), then we can consider it a success.

The main parameters to each run is the gain term, the momentum term, and the shape of the hidden layers. The gain term is a multiplier for the weight adjustment of every backpropagation. A higher value will make bigger changes to the weights, but may cause the weights to fluctuate and never to reach the minimum. The momentum term allows the weight adjustment to have a tendency towards the direction of the previous weight adjustment. Having some momentum for the weights may cause the errors to converge on a minimum faster [3]. The shape of the layer defines the number of hidden layers and the number of neuron in each hidden layer. Too few layers or too few neurons per layer may cause the network to be ineffective, while the opposite may bring no significant improvements at the cost of higher running time.

Implementation details

For the implementation, we used pure C++ with only the standard library. Intuitively, the neurons can be represented by a plain object, and the edges are pointers to other neuron objects. In practice however, it turned out to be not the ideal data structure to store the network. While one can use pointers to find a connected neuron, it becomes difficult to maintain the edges when we realized we need the edges to go in both directions (one for forward propagation, one for backpropagation). One solution was to create a duplicate edge in the opposite direction, but that meant for every two connected neurons, we would have to update two edge weights even though they are the same edge on the network.
In the end we decided to use matrices to represents the edge weights. Every layer is represented by a matrix, and each row of the matrix would represent a single neuron and contain all the input weights to that neuron; the network can simply be represented by an array of matrices. Not only is this compact, it is also easy to find a weight; the input weight k of neuron j of layer i is simply layers[i][j][k]. Performance-wise, the matrix representation also as an edge over the user of pointers; since the matrix is implemented as a single array, the weights are stored compactly, without the need to traverse pointers. Some operations such as the product sum stage of the neuron function can even be written as a matrix multiplication of the weights and the input vector. This possibility means neural networks can be implemented in graphics processors, where matrix operations perform the best.

Experiment results

Experiment results are recorded as average of 8 trials.

By default, the program will use these parameters:

Gain = 0.1

Inertia = 0.2

Shape: { input 30 25 output }

Average error rate: 0.122475

Average second best error rate: 0.480232

This will be the basis for the following experiments; one out of three parameters (gain, inertia, shape) will be changed to see if it positively affects the error rate.

gain	inertia	shape	Average error	Average second best error
0.1	0.2	30 25	0.122475	0.480232
0.1	**0.5**	30 25	0.124975	0.47535975
0.1	**1.0**	30 25	0.14635	0.553190375
0.5	0.2	30 25	0.1019375	0.472563
1.0	0.2	30 25	0.15595	0.509974125
0.1	0.2	**15**	0.146075	0.530237875
0.1	0.2	**15 15**	0.1392125	0.497443
0.1	0.2	**15 15 15**	0.1502375	0.492214625
0.1	0.2	**30**	0.15655	0.579757
0.1	0.2	**45 30**	0.1353625	0.53866075
0.1	0.2	**45 30 15**	0.122575	0.4764315
0.1	0.2	**45 45 45**	0.1277375	0.49009025

0.1	0.2	**15 30 45**	0.15455	0.55629

Conclusions

Adjustments in any of the three parameters resulted in differences in running time and error rate. For inertia, changing it from 0.2 to 0.5 showed a negligible change in error rate, while changing it to 1.0 increased the error rate. This could mean maximum inertia is not helpful as it could cause the error function to fluctuate as it approaches zero. Changing the gain term to 0.5 allowed error rate to reach 10%, the lowest we have achieved in this implementation; on the other hand, a gain of 1.0 resulted in a worse error rate, higher than that of 1.0 inertia. This means with the default learning rate of 0.1, the back propagation did not converge; it is only with 0.5 that it did. A gain of 1.0 most likely caused the same fluctuation as 1.0 inertia did. As for the shape, the results suggest that layers strictly decreasing in size from input to output gives the best results. This is consistent with our initial estimate. This structure allows the algorithm to first look at smaller parts of the picture, identify local characteristics, then look at the picture as a whole.

While the backpropagation algorithms can result in a highly accurate neural network, finding the correct parameters to reach that accuracy is another problem to solve, and it is one that requires experimentation.

Possible future works

While the results shows promise, these experiments suggest that this algorithm has trouble achieving an error rate that is less than 10%. This is due to the algorithm relies purely on the old weights. The reliance on previous graphs caused the algorithm to be somewhat influenced by the initial randomly generated graph. This is a fundamental flaw that leaves this algorithm vulnerable to being stuck in a local maxima. A possible augmentation to this algorithm would be to add random mutation on top of the backward propagation. The idea is that after we propagate the error and adjust the weights, we add random edges and make random changes to the network. For each graph we applied backward propagation to, we generate a family of random

graphs. Then we use the graph with the least error and move on to the next iteration. While this may cause the algorithm to converge on a low error rate much slower than pure backward propagation, this allows the algorithm to bypass local minima and achieve much lower error rate.

The learning rate, momentum and layer sizes are parameters that needs to be fine tuned for specific problems. One way of finding the optimal values is by hand, but another possible method is to use another neural network. The inputs for the second network will be the parameters, while the output will be the error rate of the first network. This would hypothetically be able to find the network parameters that results in a low error rate. The problem with this algorithm is deriving the function. Since part of the function is the first neural network, it may be very hard to come up with an analytical derivative for it; a numerical derivative will need to be used, which would have a long running time.

References

[1]. https://en.wikipedia.org/wiki/Convolutional_neural_network

[2]. https://en.wikipedia.org/wiki/Recurrent_neural_network

[3]. Lecture slides

[4]. https://en.wikipedia.org/wiki/Feedforward_neural_network

[5]. http://karpathy.github.io/neuralnets/