Effects of Bit Precision on Training Accuracy of Neural Networks

Tejas Nikumbh tejasnikumbh@gmail.com +91 7506081238

Guide : Prof. Bipin Rajendran

Abstract

With the invention of powerful embedded devices like the Raspberry Pi 3.0 and BeagleBoard , it is has become very interesting to observe the possibilities of implementing ideas in Machine learning to these systems. If Machine Learning Algorithms can be implemented (and possibly trained) on these pocket sized devices, the possibilities of coming up with powerful hardware applications are endless. A robot which can learn to walk, a real time health monitoring system, an intelligent coffee machine, and what not. Implementing these algorithms on embedded devices would mean inducing intelligence into the machines we see all around us, and training them to learn via experience. But this is just one of the various applications of the area that the current project explores. The project aims at studying the effects of changing bit precision on the training accuracy of one of the most powerful ideas in Machine Learning - Neural Networks. The project specifically studies the effects of changing the bit precision of the trained weight matrix on the training accuracy of the neural network. The results would indicate how much loss of accuracy we face, in case we bring in hardware limitations on the trained neural network parameters. This would mean an estimate of the power of a Neural Network Implementation on a limited bit precision system as well as the hardware costs that we can bring down, wherever they are not necessary. Another interesting area to explore can be in the field of parallelizing a particular Machine Learning task, which is heavily hardware oriented. This study can provide interesting insights into it as well. However, the topic will be talked about in the future work section and is something which remains to be explored. Following is the Index of terms for this paper.

Index of Terms: Training task, Sample size, Network architecture and Neuron model, Training, Feed Forward Propagation, Back propagation ,Optimization algorithm , Effects of Bit Precision, Data Visualization.

Introduction

Artificial Neural Networks have been used in a variety of applications ranging from image processing to speech processing to speech synthesis and analysis and high energy physics and so on. The traditional Neural Networks are implemented using the Feed Forward and the Back propagation algorithms for training, and are usually trained and executed on a 32-64 bit precision system. In the current project, we use a 32 bit system to train a fully connected 3 layered Neural Network and obtain a trained weight matrix. We then observe the effect of reducing the bit precision of all the weights in the weight matrix by rounding them off to discrete levels, depending on the bit precision specified. We then use feed forward algorithm predict the training accuracy of the network given this set of weights and observe its effects. The task that we use here is classification of handwritten digits, which are all 1 Bit images.(Black if a pixel is ON, white if it is OFF). So the input already obeys our hardware requirements. In the following sections we would briefly specify the task and the sample size, the Network Architecture and Neuron model we have used, the algorithms we use for training our neural network , and finally the effects of bit precision on the training accuracy of our algorithm.

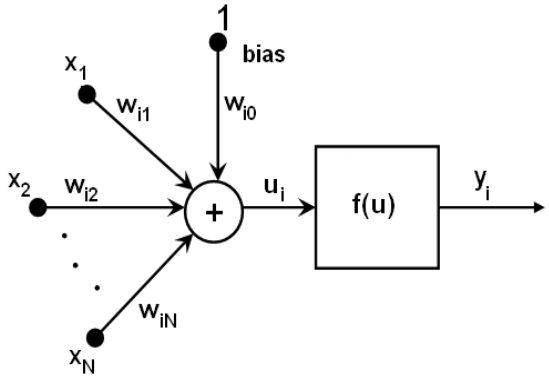
I. Evaluation Metric – The Task

The task that we use to evaluate the performance of the neural network is classification of handwritten digits. This is a classical task that is used to evaluate the performance of an artificial neural network.

The images that we have used here are 0/1 Bit images in nature. The grey color indicates that a pixel is 1 whereas 0 is indicative of the pixel being white. The dimensions of the input image are 20X20, and each pixel is mapped to a corresponding input neuron. The sample size that we have used here is 5000. The MNIST database has 60000 images, but since training such a large set would take a lot of time, we have used a subsample of 5000 images. Training on a higher training set can lead to higher accuracy, at least in this case.

II. Neural Network Architecture and Neuron Model

The study of artificial neural networks has been inspired in part by the observation that biological learning system are built of very complex webs of interconnected neurons. Typically, the human brain consists of approximately 10^{11} neurons, each with an average of 10^{4} connections. It is believed that the immense computing power of the brain is the result of the parallel and distributed computing performed by these neurons . Based on the biological model of the neuron, this is the characteristic artificial neuron model proposed by McCulloch and Pitts [3] attempts to reproduce. This neuron model is widely used in artificial neuron networks with some variations (Figure 1).

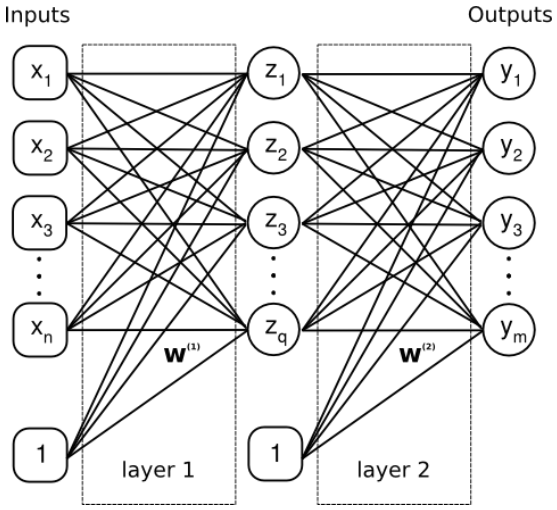


Neuron Model [Figure 1]

Think of the neuron model as a graph. The inputs from x_1 to x_N are the initial set of vertices(with values x_1 to x_N, respectively). Each of the weights to w_{iN} are the edges connected to these vertices. Each edge has values from w_{i0} to w_{iN} respectively. We multiply each vertex x_j with corresponding weight w_{ij} .The transfer function computes the sum of all the incoming vertex-edge products and produces a net input u_i. This input is then passed to the activation function which produces an output based on the input. The final output y_i is called the activation. The activation function which is usually used is either $tanh(x)$ or $sigmoid(x)$, but for the purposes of this study, we use sigmoid.

Next up is the network architecture. Based on the model proposed above, we have used a fully connected 3 layer neural network. A model for the neural network is shown below. (Figure 2)

The network is fully connected, meaning each neuron in a layer is connected to all other neurons in the previous layer. There are other models as well, which are not fully connected, but for simplicity we have assumed the fully connected model as shown below.



Neural Network Architecture [Figure 2]

In our case, we have 400 neurons in the first layer, (n = 400), 25 neurons in the second layer(q = 25) and 10 neurons in the final layer (m = 10) where n , q, m correspond to the variables in the figure above. The first layer neurons correspond to 400 pixels of the image that we have used, the second layer corresponds to the 25 neurons which are 'hidden' , and the 10 neurons in the final 3rd layer correspond to 10 digits which we output. The values stored in the output layer are the probabilities of a particular digit occurring.

The '1' present at each layer is called the bias unit, and the weights associated with it are called thresh holds. They play an important part in training. A bias unit is meant to allow units in the neural net to learn an appropriate threshold (i.e. after reaching a certain total input, start sending positive activation), since normally a positive total input means a positive activation. For example if the bias unit has a weight of -2 with some neuron x, then neuron x will provide a positive activation if all other input adds up to be greater then -2.

III. Training the Neural Network

Training the neural network is the process of evaluating the set of Wi's as described in the previous sections. We use a 32 bit system to train the neural network, and for training we make use of the following algorithms.

The training of a neural network is a two step process, and it involves re-iterating again ang again to arrive at an optimum set of weights to arrive at a fitting performance oriented set of weights. The first of the two steps used to train the algorithm is Feed forward propagation, which is followed by back propagation, both of which are described below.

A. Feed forward propagation

The output of our program implementing a neural network is the values stored at the 10 output neurons with value at each neuron corresponding to the probability of that particular digit being the input image's description. The ideal scenario is a 1X10 matrix with one value being 1 and all other values being 0. To set the algorithm in action, we initially randomly initialize the set of weights w_{ij} [edge values] as shown in the network architecture.

After randomly initializing the weights, the algorithm works as follows.

For each input image(called a training example), we initialize the x_1 to x_N as the value of the pixel in the image. 400 Neurons, 400 pixels. We then compute the value at each neuron in the 2nd layer using the neuron model described earlier. The activation function we use is the sigmoid function which is shown below. Y here is the activation output which is the value

$$f(x) = \frac{1}{1 + e^{-x}}$$

Of the neuron and x is the summation of products of all the vertices and edge weight values as described in the neuron model. This process is carried forward to compute the value of each neuron at the output layer, the inputs now being the 25 neurons in the middle layer.In this way, by observing the calculated value of the neurons at the output layer, we determine which digit the image is as described in the first paragraph. Following is some data relevant to our current simulation.

We have [400+1(biasunit)]X25 weights mapping the first layer to the second layer, which culminates to 1025 weights. Similarly in the mapping from 2nd -3rd layer we have [25+1]X10 = 260 weights. This adds up to a total of 1285 weights and 400 + 1 +25 + 1 +10 = 437 neurons in the entire network.

The actual training part happens when we determine the error on the output produced by the neural network, and improve the weights to reduce this error. The process is described below.

B. Back propagation algorithm

After having computed the output matrix of dimension 1X10 (corresponding to the output layer), we proceed on the determine the error (labeled δ). Once we are given a training set, we already know the expected output matrix. The expected output matrix A is a 1X10 matrix with one entry as 1[corresponding to the correct digit], and all

other entries are 0. Of course, our first guess according to random initialization produces some other matrix, say B. Now, we compute the error at each neuron as A[neuronNumber] – B[neuronNumber]. This gives us an error matrix delta of dimension 1X10 corresponding to the output layer.

We will now compute the errors at the hidden layer neurons, which are computed as follows. Error at each neuron is the weighted sum of errors of all of the neurons in the next layer that it is connected to, with the weights being the same as the feed forward propagation network. The details of this require considerable mathematical analysis, for which we won't discuss it here.

After determining the errors at each of the neurons in the network, our next job is to update them in order to improve the performance of our neural network. There are various algorithms to do this, one of which is gradient descent. It is the simplest to understand. We perform the weight update for the gradient descent optimization algorithm as follows.

$$W_{new} = W_{old} + h * x_{ij} * \frac{d(f(e))}{d(e)} * \delta$$

Where $\frac{d(f(e))}{d(e)}$ is the derivative of the activation function at the activation of at the output neuron of the weight and δ is the error at the output neuron of the weight as calculated above. X_{ij} is the value at the input neuron of the weight and h is the learning rate. 'h' is indicative of how fast we want the updates to happen, and is usually a practical/ implementational aspect of training the algorithm.

Note that the rate of change of the the weight is given by $x_{ij} * \frac{d(f(e))}{d(e)} * \delta$ as per the above equation. If we know the rate of change of weight for each weight and the weight itself, we can use many advanced optimization techniques for training, other than gradient descent to increase the speed of the neural network. One such algorithm is fminunc [function minimization unconstrained] and we have used it to update the weights and train the algorithm for simulation purposes, as our prime objective was to observe the effects of the bit precision on the trained weight matrix, not delving into the intricacies of how optimization works.

Once we have updated the weights according to the algorithm specified above for one sample image, we repeat it for others as well. An entire iteration through all the sample set is known as an epoch. We usually perform 50-100 epochs on a fully connected 3 layered network to train the algorithm well. Shared weight networks use lesser epochs. We have used 50 epochs in our current training and simulation. Another way to know when to stop is to keep a certain threshold for the training accuracy/error percentage on the training set. Once this threshold is crossed, we stop the training. Method 1 is followed in our simulations. We compute 50 iterations or epochs.

IV. Effects of Bit Precision

Once this training is done we obtain a set of weight matrices which we then utilize to predict the outputs, using the feed forward propagation as described earlier. On a 32 bit system, the feed forward propagation algorithm gives us an accuracy of 96.00%

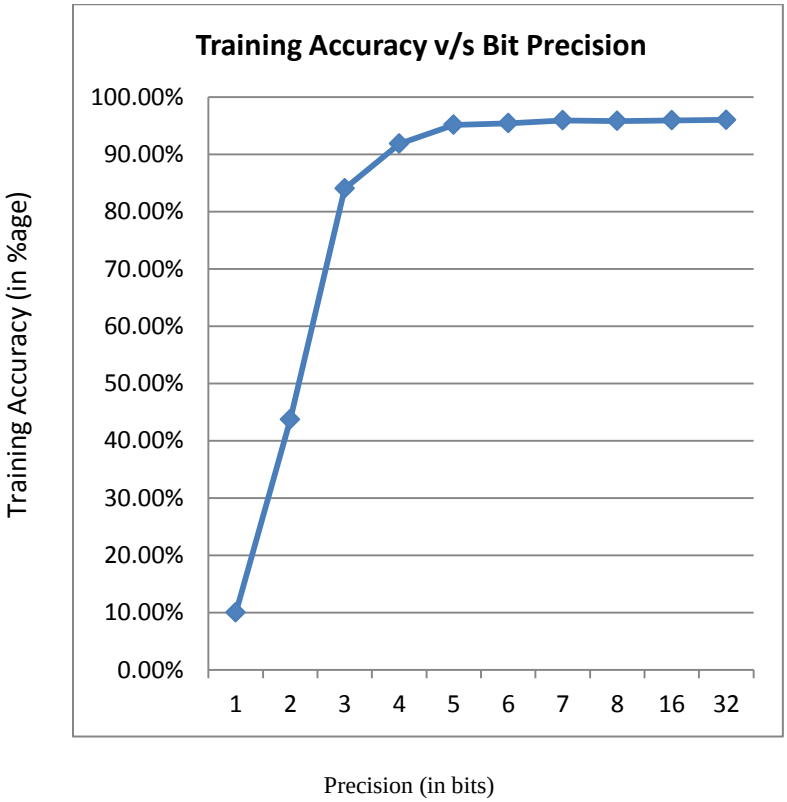
Now, for the purposes of our simulation, we take the trained weight matrix and determine the maximum and minimum value in the entire weight matrix. Let these values be W_{min} and W_{max} respectively. Once this is done, we split the range between W_{min} and W_{max} into n discrete levels. 'n' here is given by 2^b where b is the bit precision. This is logical since if we have a 4 bit precision(say), we can represent 16 different levels of data, and hence we split the range between W_{min} and W_{max} into 16 different levels.

We now take each of the weights in the weight matrix and round it off to the nearest of the discrete levels that we have split the range into. In this way, we achieve the effect of representing the weight matrix in different bit precisions. Each weight in the network architecture will now be one of the discrete 16 levels.

Following the above procedure, we were able to simulate results for 1,2,3,4,5,6,7,8,16 and 32 bit precision. At each precision level, the following training accuracy was observed. [Data described in a table]

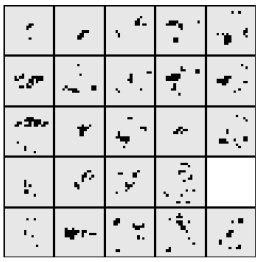
Bit Precision	Training Accuracy(%age)
1	10.00
2	43.68
3	84.00
4	91.84
5	95.14
6	95.42
7	95.90
8	95.80
16	95.92
32	96.00

As we just changed the bit precision of the weight matrix being trained at each level, keeping all other parameters the same, this data is indicative of the effect of changing the bit precision on the training accuracy. Following is a graph of the training accuracy (in %age) versus the weight matrix precision (in bits).

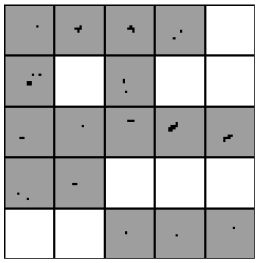


We observe from the graph that the training accuracy begins to saturate at 4 bits and saturates beyond 5 bits. This result has some really good implications described in the conclusions section. We also tried to visualize the weights as a gray scale images. We have shown below the gray scale images obtained for the 1000 weights mapping layer 1 to layer 2. Each of the 25 sections shown below is representative of the weights coming into one of the 25 neurons in the second layer. Within each of these 25 sections, you can see the 20X20 pixels, which are indicative of the values of the 400 weights which connect to each neuron in the second layer to the 400 input neurons. The visualizations clearly depict the change of weights w.r.t the changing bit precision, consolidating our simulation process.

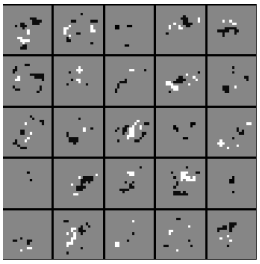
Gradual transition of weight matrices from 1 bit to 32 bits. (images shown on next page)



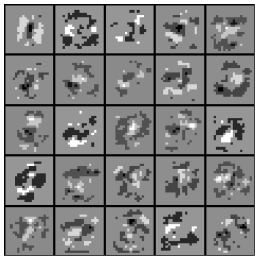
1 Bit



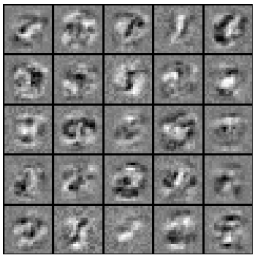
2 Bit



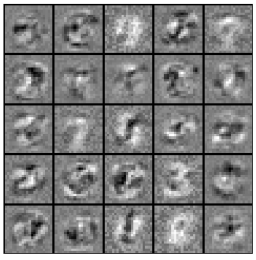
3 Bit



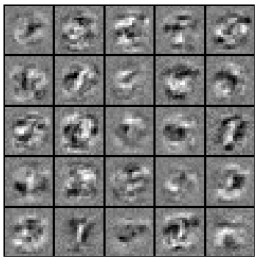
4 Bit



8 Bit



16 Bit



32 Bit

Visualizations of Weight Matrices [Figure 3]

V. Implications and Conclusions

Our observations have some really interestin g implications and conclusions. To begin this, the graph suggests a saturation of training accuracy beyond the 4 bit value. This suggests that we can have a relatively good performance while evaluating at 4 bits with a relatively less loss of accuracy. Given embedded devices like Raspberry Pi and Beagle Board at our disposal, we can develop interesting applications using a neural network implementation on one of these boards. We can also greatly reduce hardware costs at places where higher bit precision machines or devices are being used to do the evaluations or predictions. From making intelligent coffee machines, intelligent homes, to developing quad copters, building on interesting applications on the microcontroller, to using the the implementation for parallel processing tasks, we can carefully consider the pros and cons of the implementation and use a 4 bit system to implement the neural network evaluation procedure in various scenarios.

Two , this also means that small perturbations in weights don't affect the training accuracy beyond a certain point which can help us determine the number of epochs we should train a particular neural network for. This can greatly reduce the number of training hours and redundant training cycles, thereby greatly increasing efficiency.

VI. Future Work

As future work , I'd like to perform the following tasks.

- Implementation of the neural network in C++/C to see how much faster the training can happen. This would also provide me a good and unique opportunity to play with the algorithm and determine how the prototyped version in Octave could be implemented in C++/C, both of which are pretty close to real world applications of the same. It would also give me a chance to learn the language libraries and intricacies, which I think would be beneficial in developing any large scale system involving these kind of systems.

- I would like to study about parallelizing the training system so that the neural network is trained faster. One of the problems I faced was that even with a relatively fast optimization routine, the training was relatively slow. Since the neural network is something that is inherently parallelizable, I would like to see how the parallel procedures of map and reduce are applied to something like a 3 layered neural network in reality and what its implications are.

As a third task, I would like to implement Convolutional Neural Networks, which are currently the state of the art neural networks with shared weights to see how the 4 bit precision performs on that Network Architecture. Again, if I could do this, it would serve to be a very good achievement for me.

VII. References

[1] http://en.wikipedia.org/wiki/Backpropagation

[2] Advances in and problems of the implementation of neural algorithms By Yihua Liao Department of Computer Science, University of California.

[3] McCulloch, W. S. and Pitts, W., 1943, A Logical Calculus of the Ideas Immanet in Nervous Activity.Bulletin of Mathematical Biophysics, vol. 5, 115-133, 1943.

[4] Machine Learning Class, Professor Andrew Ng. www.coursera.org