Neural Networks and Optimizations

Siva G Perumal*
Sivaramakrishnan Sankarapandian*

May 12, 2017

*equal contributions

Abstract

Neural networks are growing in stature day by day.Neural networks have been hugely successful in approximating fairly complex models.This project aims to implement a three layer network in C to classify the digits in MNIST data. First the baseline serial code was implemented from scratch. Then different optimization techniques such as loop unrolling, Intel intrinsics, OpenMP and CUDA were implemented. The performance of the optimized versions are compared with the serial baseline code and the results are presented.

Contents

1 Neural Networks

1.1 Introduction

Neural networks are a bunch of processing elements (PE) connected to the PEs in the next layer.The most basic network is the multilayer perceptron which has the structure in figure 1:

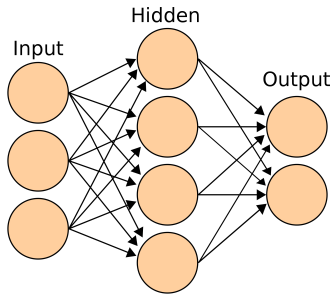

Figure 1: A simple neural network
[1]

As indicated by the arrows in figure 1, the direction of flow of data is from left to right. There is no feedback in these types of neural networks. Each circle is a Processing Element(PE) and each arrow has a weight associated with it.

All the PEs function in the same way. In neural networks, the PEs are called neurons.The details of each neuron is shown in figure 2. From the figure, the working of a neuron can be summarized

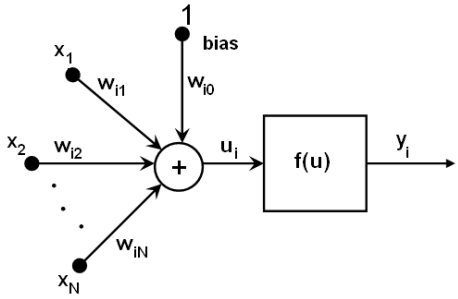

Figure 2: Neuron structure
[2]

as follows:

- The inputs x are multiplied by the weights w and summed.
- A bias is added to the sum.
- A function $f(u)$, called the activation function is applied to the total sum.

There are many activation functions available, but in this project sigmoid activation(shown in figure 3)is used. It is a non-linear activation function and gives output in between zero and one.

Mathematically, it is expressed as:

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

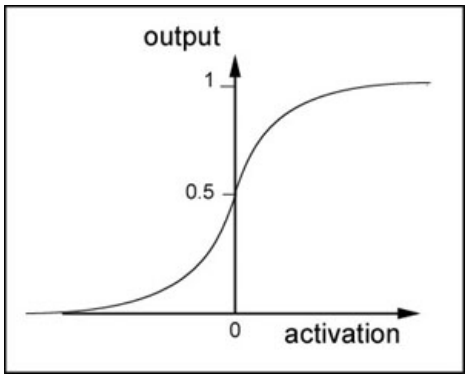

Figure 3: Sigmoid function

So far the structure of a neural network and the working of a neuron is explained. The next section explains the working of the neural network as a whole.

1.2 Neural network working

A neural network has to be **trained** before it can be useful for making predictions.Training the neural network involves two steps[3]:

- Forward pass
- Backpropagation

At the end of backpropagation we get the values by which the weights and the biases have to be changed so as to reduce the error between the predicted and the desired outputs. If we assume a two layer network with an input dimension of n and output dimension of m, for a mini-batch of size d, the complexity would be **O(mnd)**
Here we explain the process for the three layer neural network (NN) in figure 1. The idea can be easily extended to NNs with any number of layers.

Forward pass :

We start by defining a few terms:

- $x = [x_1, x_2, x_3]$ is the input vector.
- w_{21}, w_{32} are first and the second weight matrices.
- b_1, b_2 are the biases for the hidden layer and the output layer.
- sigmoid function is indicated by the symbol σ.

Forward pass explained

x is multiplied with w_{21} to get the input to the hidden layer.

$$input\_hidden = w_{21} * x + b_1$$

At the hidden layer we apply the activation funtion sigmoid(σ)

$$ouput\_hidden = sigmoid(input\_hidden)$$

The output from the hidden layer is then multiplied by w_{32} to get the input to the output layer.

$$input\_outlayer = w_{32} * ouput\_hidden + b_2$$

Finally, we apply the activation at the output layer to get the final output.

$$output = sigmoid(input\_outlayer)$$

Before looking at backpropagation , we define the cost function as Mean Square Error (MSE). Cost function is a measure of how accurate the network is. A large value of cost function means that the network is inaccurate. Hence, the aim of backpropagation is to manipulate the weights and the biases in such a way that the value of cost function reduces.

Backpropagation :

Again we start by defining a few terms:

- C is the cost function.
- δ_{l2} and δ_{l1} are the errors at the output and the hidden layers respectively.
- y is the actual output vector for input x

Backpropagation explained
The cost can be found as follows:

$$C = \frac{1}{2}\sum(y - output)^2$$

We take the gradient of the cost function with respect to output:

$$\nabla_{output} C = output - y$$

Then the error at the last layer can be found as:

$$\delta_{l2} = \nabla_{output} C \ \odot \ \sigma'(input\_outlayer)$$

Once the error at the last layer is found, the values by which w_{32} and b_2 have to be changed can be found as:

$$db_2 = \delta_{l2}$$
$$dw_{32} = output\_hidden * \delta_{l2}$$

The error at the hidden layer can be found as follows:

$$\delta_{l1} = (w_3 2 * \delta_{l2}) \ \odot \sigma'(input\_hidden)$$

And as we have seen in the case of w_{32} and b_2, w_{21} and b_1 can be updated as follows:

$$db_1 = \delta_{l1}$$
$$dw_{21} = x * \delta_{l1}$$

Finally we update the weights and the biases:

$$
\begin{aligned}
b_1 &= b_1 - \eta db_1 \\
w_{21} &= w_{21} - \eta dw_{21} \\
b_2 &= b_2 - \eta db_2 \\
w_{32} &= w_{32} - \eta dw_{32}
\end{aligned}
$$

where η is the learning rate.

This completes both the forward pass and the backward pass for neural networks. However, the network isn't completely trained yet. This process has to be done multiple times till the error converges and we get a steady error. One way to do it is gradient descent which will be explained in the next section.

2 Gradient Descent Algorithm

2.1 Training Neural Network

Predictions from an untrained neural network are only as good as random guesses. So effectively training the neural network to make correct predictions is as important as building a neural network. There are different algorithms by which a neural network can be trained and one of the popular algorithms is the Gradient Descent Algorithm. Gradient Descent algorithm adjusts the weights in such a way that the gradients of the loss function with respect to weights are minimized. If we assume a convex objective function - an objective function where there is only a global minimum, gradient descent moves in the direction of the negative derivative of the objective function. In this way gradient descent always minimizes the function and it reaches a global optimum if it is a convex function. The algorithm is iterative where the gradients with respect to each examples are added and the final value is subtracted from the objective function which is considered as a step towards the optimum value. As the number of forward and backward pass increases, the weights are adjusted in such a way that the data in the input space are mapped to the correct data in the output space.

2.2 Learning Rate

The parameters while training a network that are tuned to specific data are referred to as hyper parameters. Learning rate determines the length of the step after each iteration (through all the data) towards the direction of minimum loss. Higher the learning rate, faster the learning would be but at the same time, higher learning rate causes oscillations near the optimal point. Lower learning rate causes slower learning while helping to reach closer to the optimal point. There is usually a trade-off between magnitude (time for convergence) and the how close the weights need to be in their optimal value. Cross-validation is the process by which ideal value for learning rate can be chosen.

2.3 Cross-validation

Cross-validation is the process of choosing a particular value of a hyper parameter while training a machine learning model such that benefits from it are maximized. For different values of learning rate, the loss after 10 epochs are determined (an epoch is computing the forward and backward pass through the entire training set once) and the learning rate with the minimum loss is chosen for the data in hand.

2.4 Weights Initialization

Weights initialization is an important process in training a neural network. If the weights are initialized to the same value or closer, there would be no or not much learning after every back propagation. Again, there are different ways by which the weights can be initialized, and we have chosen to implement normal weights initialization. In this method, the values are drawn from a particular normal distribution whose mean and variance are specified beforehand which also become hyper parameters for the network.

Algorithm 1 Gradient Descent Algorithm

```
while loss not converged do
    weights_prev ← initialize
    for for each data point do
        forward pass
        compute loss using the loss function Mean Square Error(MSE)
        compute gradient of the loss with respect to weights (backward pass)
        gradient_total ← gradient_total + gradient_for_each_data_point
    end for
    weights_next ← weights_prev − (learning_rate ∗ gradient_total)
end while
```

3 Serial Implementation of neural networks

3.1 Dataset

We implemented a neural network to classify the famous 'Modified National Institute of Standards and Technology' MNIST dataset. It is a database of handwritten digits which is commonly used to train image recognition and classification systems.Only a subset of 10,000 images was used in this project.Each image is 28x28. Training and testing were carried out on the same set of images.

3.2 Preprocessing in python

A python code was written to pre-process the data into the format that is needed for our neural network implementation. The following things were implemented:

- 28x28 image matrix is converted into 784x1 vector.
- Since there are 10 digits each label is converted into a one-hot vector.
- As explained in section 2.4, the initial selection of weight is important. Weights have been initialized from a normal distribution in the code.

The processed inputs, labels and weights are stored in file.txt, onehot.txt, weights_1.txt, weights_2.txt respectively.

3.3 Helper functions in C

As seen in section 1.2, there are a lot of matrix operations in forward pass and backward pass.The following table shows the number of times, each matrix operation is needed.

Matrix Operation	Number of times used
Addition	2
Subtraction	6
Element-wise multiply	7
Multiplication	5
Transpose	2

Since, these operations are used many times they are written as functions. Apart from these operations, functions are also written for sigmoid activation and derivative operations.

3.4 Main function in C

Read files :
The text files obtained after preprocessing from python are read first and stored in arrays. There is an array for input,labels and both the weights respectively.

Iteration through images :
From the input array each input image is retrieved. Similarly the one hot vector associated with each image is retrieved. The forward pass and backpropagation is completed with one image using the **helper functions** and the weights and the biases are updated.
The number of epochs is a hyper-parameter that can be adjusted.
One can find that the neural network is learning is by observing the MSE. In our implementation the loss reduces with every **epoch**.

4 Parallelization of the Serial Code

In this section we try a few optimization techniques such as:

- Loop unrolling
- Intel intrinsics
- OpenMP
- Implementation in CUDA

4.1 Loop Unrolling

We start with the most basic optimization technique called Loop Unrolling.

Loop Unrolling helps in reducing the loop overhead by reducing the number of times the 'loop-end' condition has to be checked. It also reduces the number of times the loop variable is incremented. The disadvantage of loop unrolling is increased code size.

Our implementation requires **for** loops for matrix operations and hence this is a good method to increase the speed of code execution. Figure 4 shows the way we implement loop unrolling in our code:

Loop unrolling results :
Loop unrolling was implemented with unroll factor of **4** and **8**. Interestingly, **better** performance was observed with unroll factor of 4 rather than the unroll factor of 8. We suspect that in case of unroll factor 8, the benefit due to less number of 'loop-end' tests is not significantly

```
void matrix_multiply_unroll(int db1,int db2,double *m_b,int da1,int da2,double *m_a,double *r)
{
    int i,j,k;
    double sum_0 = 0;
    double sum_1 = 0;
    double sum_2 = 0;
    double sum_3 = 0;
    double sum_4 = 0;
    double sum_5 = 0;
    double sum_6 = 0;
    double sum_7 = 0;
    for (i = 0;i<db1;i++)
    {
        for(j = 0;j<da2;j++)
        {
            for(k = 0;k<da1-7;k = k+8)
            {
                sum_0 += m_b[i*da1+k]*m_a[k*da2+j];
                sum_1 += m_b[i*da1+(k+1)]*m_a[(k+1)*da2+j];
                sum_2 += m_b[i*da1+(k+2)]*m_a[(k+2)*da2+j];
                sum_3 += m_b[i*da1+(k+3)]*m_a[(k+3)*da2+j];
                sum_4 += m_b[i*da1+(k+4)]*m_a[(k+4)*da2+j];
                sum_5 += m_b[i*da1+(k+5)]*m_a[(k+5)*da2+j];
                sum_6 += m_b[i*da1+(k+6)]*m_a[(k+6)*da2+j];
                sum_7 += m_b[i*da1+(k+7)]*m_a[(k+7)*da2+j];
            }
            r[i*da2+j] = sum_0 + sum_1 + sum_2 + sum_3 + sum_4 + sum_5 + sum_6 + sum_7;
            sum_0 = 0;
            sum_1 = 0;
            sum_2 = 0;
            sum_3 = 0;
            sum_4 = 0;
            sum_5 = 0;
            sum_6 = 0;
            sum_7 = 0;
        }
    }
}
```

Figure 4: unroll snippet

greater than the time taken to execute more instructions. However, both the cases performed better than the baseline serial code. The results are shown below :

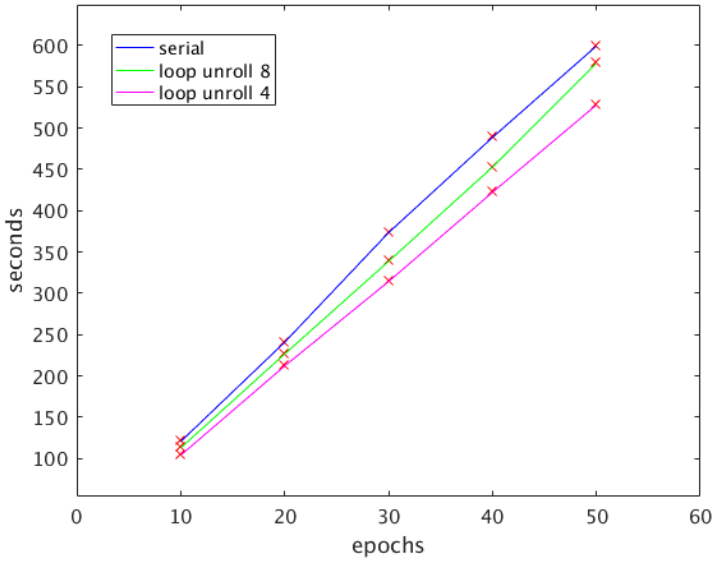

Figure 5: Time taken as the number of epochs increases

4.2 Intel Streaming SIMD Extensions(SSE)

In our baseline serial code, for matrix add, subtract and element wise multiplication, we loop through both the operand matrices element by element. Instead of that we can vectorize this operation by using Intel SSE [4].
SSE enabled processors have 8 **128-bit registers**. Each register can therefore store **2** double precision numbers. In our case, we use 3 such registers, one register each for operand matrices and one for the result matrix.
SSE allows us to add the numbers stored in the registers in parallel. Thus we add two pairs of doubles at once. This helps in reducing the loop iterations by **half** and improves the speed enormously.

SSE results :

As described above, we implemented SSE for matrix add, subtract and element wise multiply. The result of its comparison with the baseline serial code is shown in the figure below:

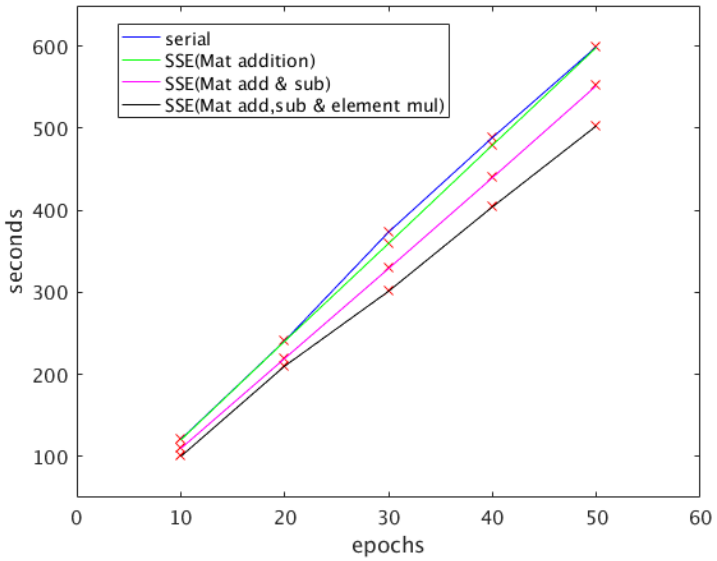

Figure 6: Time taken as the number of epochs increases

4.3 OpenMP

OpenMP (open Multi-threading) is a multi-threading technique where the master thread creates a number of slave threads and the work is distributed amongst the slave threads.

In our implementation, we used openMP to parallelize the for loops in the matrix operations.

OpenMP results :

The results obtained by implementing OpenMP weren't good because the threads were created and destroyed for every matrix multiplication operation in each **epoch**. This overhead of threads was greater than any benefits achieved due to multi-threading.

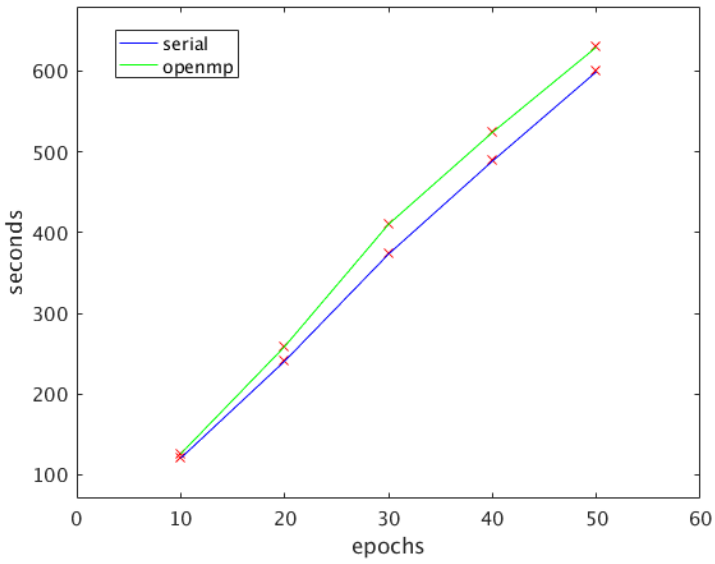

Figure 7: Time as number of epochs increases

4.4 CUDA implementation

GPUs can be used when same operations have to be performed on huge chunks of data. However a single core of GPU is slower than a CPU core. In our version of the GPU code, we implement matrix multiplications on GPU and compare the performance with the serial code.
We used the cuBLAS library for matrix multiplication [5]. However it is important to note that cuBLAS uses column-first indexing while in C its row-first indexing. Here we explain the cuBLAS multiplication function that we used:

cublasStatus_t cublasSgemm(cublasHandle_t handle,cublasOperation_t transa, cublasOperation_t transb,int m, int n, int k,const float *alpha,const float *A, int lda,const float *B, int ldb, const float *beta, float *C, int ldc)

The following image shows how we call the gpu function in our code:

```
float *w1,*in,*act;
cudaMalloc(&w1,HIDDEN_LAYER*INPUT_LAYER* sizeof(float));
cudaMalloc(&in,1*INPUT_LAYER * sizeof(float));
cudaMalloc(&act,HIDDEN_LAYER*1 * sizeof(float));

    cudaMemcpy(w1,final_weights_i_h,HIDDEN_LAYER*INPUT_LAYER* sizeof(float),cudaMemcpyHostToDevice)
    cudaMemcpy(in,instance_input,1*INPUT_LAYER * sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(act,input2layer1,HIDDEN_LAYER*1 * sizeof(float),cudaMemcpyHostToDevice);
    gpu_blas_mmul(w1,in,act,HIDDEN_LAYER,INPUT_LAYER,1);
    cudaMemcpy(input2layer1,act,HIDDEN_LAYER*1 * sizeof(float),cudaMemcpyDeviceToHost);
```

Figure 8: GPU implementation snippet

And this is snippet of the our gpu function:

```
void gpu_blas_mmul(const float *A, const float *B, float *C, const int m, const int k, const int n)

{
    const float alf = 1;
    const float bet = 0;
    const float *alpha = &alf;
    const float *beta = &bet;


     cublasHandle_t handle;
     cublasCreate(&handle);
    // Do the actual multiplication
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, alpha, B, n, A, k, beta, C, n);
    cublasDestroy(handle);
}
```

As seen in the snippet, it is important to swap the input matrices and set the leading dimensions of the matrices lda,ldb,ldc properly because cuBLAS follows column major indexing.

CUDA results :

GPU is advantageous only when we deal with huge amount of data. Here we did not have a lot of data and so the timing overhead of copying the data to the GPU from the host and copying the data back from the GPU to the host proved to be more than the speedup achieved and the GPU version performed **10x** worse than the CPU version.

5 Summary

- Loop unrolling helped in improving the speed of execution.
- Multithreading methods did not work since the implementation suffered from thread overheads.
- GPU implementation suffered from data copy overheads and did not give good results.
- Intel Intrinsics gave the best improvement over serial code since vectorization helped in parallelizing the matrix operation functions.

Therefore we conclude that multi-threading options have to be carefully chosen and that the best option to optimize neural networks according to us is to vectorize the code wherever possible.

References

[1] Wikimedia. `https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg`.

[2] Blog explaining neural net. `http://blanco.io/blog/machine-learning/neural-networks-and-backpropagation/`.

[3] Michael A. Neilson. Neural networks and deep learning. 2015.

[4] Intel intrinsics documentation. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`.

[5] Cublas documentation. `http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemm`.

6 Appendix

Description of the files and instructions to run them :

- The serial implementation can be found in the **serial** folder. Execute the **make** command and then execute the **./serial_neural** command.
- Loop unrolled version of the code can be found in the **loop_unrolling** folder. Execute the **make** command and then execute the **./loop_unroll_neural** command.
- SSE version of the code can be found in the **intrinsics_serial** folder. Execute the **make** command and then execute the **./intrinsics_neural** command.
- The OpenMP version of the code can be found in the **openmp** folder. Execute the following commands to run the code:

  gcc -fopenmp openmp_neural_network.c -o neural
  export OMP_NUM_THREADS=2
  ./neural

- The CUDA version of the code can be found in the **GPU** folder. To run the file use the following commands on the shared computing cluster(scc):

  qrsh -l gpus=1 -l gpu_c=3.5
  nvcc gpu.cu -o gpu -lcublas
  ./gpu