# Exploring Multi-Layer Neural Networks:
## Training Techniques on the MNIST Dataset in CS 431

Ethan Hyde
The University of Oregon
Eugene, Oregon, USA
ehyde2@uoregon.edu

Eric Edwards
The University of Oregon
Eugene, Oregon, USA
eedwar10@uoregon.edu

December 6, 2023

# Abstract

This paper presents a detailed implementation and exploration of our neural network trained on the MNIST dataset. MNIST is a widely used and accepted benchmark in the field of image recognition machine learning. Our research focuses on the the creation and optimization of a neural network, introducing OpenMP parallelization to increase efficiency as well as the learning rate. We thoroughly describe the process of how our neural network understands these images, and links together the different layers in the end. Our methodology involves processing MNIST, normalizing the pixel values, manipulating the MNIST dataset to make it readable by our model. We explain the architecture of our network, which is made up of an input layer, $N$ hidden layers, and an output layer. We also describe how the use of activation functions (Sigmoid function) is able to introduce non-linearity into our network and describe the importance of learning rates and neuron sizes to lend a powerful model. The core of our research is based upon the implementation of the Stochastic Gradient Descent algorithm. Our study provides a comprehensive analysis of all things neural network, and despite facing challenges with creating a perfect model, this study lays the foundation for future improvements and research to be done.


# Introduction

In the everchanging field of machine learning, the MNIST dataset has continued to serve as a strong benchmark for evaluating the performance of different machine learning algorithms, especially those being trained on image recognition tasks. This dataset comprises of handwritten digits (numbers 0-9) and is widely used for testing a machine learning model. This paper aims to explain how a neural network is able to understand these images, and give a deep understanding how all the nuances of a neural network architecture.

In machine learning, the MNIST dataset's significance can't be overstated. Relative to other datasets, MNIST is much more simple. But this simplicity yields a solid foundation for one to build upon in order to improve or train more complex models. MNIST consists of 60,000 training images, and 10,000 test images. These images have a wide variation between them, which makes the model able learn and recognize patterns between the images. The goal here is to give the model exposure to as many examples as possible. The test images are then used to evaluate the model after it has completed the training stage.

Neural networks are inspired by the workings of the human brain. At a minimum, a neural network must have 3 layers, but it can have more. This consists of one input later, 1 or more hidden layers, and one output layer. The input layer first layer in the neural network. It receives the input data, then passes it to the hidden layers to be processed further. In the context of MNIST, this represents a pixel's value in the image. This information is in the form of a vector. The hidden layers perform most of the computations that take place in the neural network. The hidden layers transform the inputs from the previous layer, and update the previous layer's weights and biases and use an activation function. The one we chose to use is a sigmoid function:

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

This is used to introduce non-linearity into the neural network, which is important for training the neural network and having it learn over time. These layers are dense, meaning they are fully connected. Each hidden layer's neuron is connected to every other neuron in the next and previous layers. The output layer is the final layer of the neural network. It's job is to write an output or final value of the neural

network. In our case, this value is a size 10 vector, consisting of the probability that an input is a certain digit. For example, if the vector from the output layer is [0.1, 0.05, **0.6**, 0.1, 0.02, 0.04, 0.01, 0.04, 0.1, 0.2], then there is a 60% chance that the image is the digit 2. The term "hidden" means that they aren't directly in contact with the input/output of the neural network. In the case that there are multiple hidden layers, each layer can have a different purpose in the network. For example early layers may detect more simple color features, while deeper layers may look for more complex patterns or shapes. The number of layers can depend on how complex the task is. MNIST is able to be processed with just 3 layers, but it could improve if processed with some arbitrarily chosen amount of layers.

# Header

Listing 1: Layer Class Implementation

```
class Layer {
public:
    Layer(unsigned int inputs,
       unsigned int outputs);
    unsigned int inputs, outputs;
    float bias;

    std::vector<std::vector<double>>
       weight_matrix;
    std::vector<double>
       forward_propagation_serial(
       const std::vector<double>&
       input_vector);
    std::vector<double>
       forward_propagation_
       parallelized(const std::vector<
       double>& input_vector);
    std::vector<double>
       forward_propagation_
       parallelized_reduction(const
       std::vector<double>&
       input_vector);
    std::vector<double>
       forward_propagation_mkl(const
       std::vector<double>&
       input_vector);

    std::vector<double>
       back_propagation_serial(const
       std::vector<double>& input,
       const std::vector<double>&
       error, double learningRate);

private:
    double sigmoid(double x);
    double sigmoidDeriv(double x);
};
```

# Methodology

The preprocessing of our data consisted of normalizing the pixel values of the images. The pixel values were originally between 0 and 255, which we then normalized to a range of 0 and 1. This is intended to help with speeding up the process of the learning, as it takes less time to achieve convergence. The 2D images are then reshaped into a 1D vector of 784

pixels (28*28 instead of 28x28) which was then fed into our neural network. The `reverseInt` function in our code is also important, as it allows for us to reverse the bytes of an integer of the way MNIST is stored and the endianness of the system it's running on. MNIST is stored in a big-endian format. By using bitwise operations, we are able to isolate the least significant bit of `i`. For instance, `c1 = i & 255` will do this. After this, the bytes are recombined in reverse order, essentially converting the integer between endian formats.

Our input layer consists of 784 neurons, which each represent one pixel in the image. The neural network comprises of N hidden layers, which is defined in a variable that is able to be changed upon request. The output layer has 10 neurons which correspond to the 10 digit options. Our learning rate is also able to be adjustable, which is a crucial variable in any neural network. Too high of a learning rate may lend an unstable training process. The wirght updates can be too large and overshoot the optimal values that we are trying to find. This could actually lead to divergent behavior where the error gets worse over time instead of better. If the learning rate is too low, the training process will be extremely slow. This is due to the network only being able to make tiny adjustments to the weights at a time. This could also increase the risk that the algorithm gets stuck at a local minimum instead of finding the global minimum of the loss function. The ideal learning rate is dependent upon the task at hand. A quicker learning rate may lend towards simpler tasks for example.

Our `NeuralNetwork` class consists of private data members for te input size, output size, learning rate, and network layers. Here is also where the declarations for our training functions are. These functions are `train, computeLoss, back_propagate, updateWeights, computeOutputLayerError` and `predict`. We also have a `Layer` class consisting of `forward_propagation_serial, forward_propagation_parallelized, forward_propagation_parallelized_reduction, forward_propagation_mkl,` and `back_propagation_serial`. These functions all work together to create the neural network. The main for loop calls the `NeuralNetwork` class's `train` function. With each call, the `train` function is performing a forward pass through the network, computing the loss, then back propagating through the layers by using the Stochastic Gradient Descent algorithm. This is a popular optimization algorithm that is used in a wide variety of applications. It's different than the regulat Gradient Descent algorithm, as the work is processed in batches, rather than as a whole. SGD updates parameters using only one data point (small batch) at a time and updates the parameters immediately. There's several advantages with this approach. The main one would be that it's much more efficient than Gradient Descent, especially for large datasets like MNIST. Because of the frequent updates, SGD can lead to faster convergence than other algorithms. Lastly, the stochastic nature of this algorithm means that it more likely to escape local minima in the loss function, which may lead to better results. One downsize however may be that the learning rate needs careful tuning in order to find the best possible solution.

At this moment, the `foward()` function calls our `forward_propagation_serial` function in the `Layer` class. This function's goal is to compute the output of the neural netowrk for a given input of vectors. The function only takes a single parameter, `input_vector`, which is generated when we read in the MNIST dataset. The first thing the function does is initialize the output vector to the number of neurons in the layer. Each element in the output vector is originally set to 0, but it will store the output of the layer from this function. Next, the layer's output is computed. This

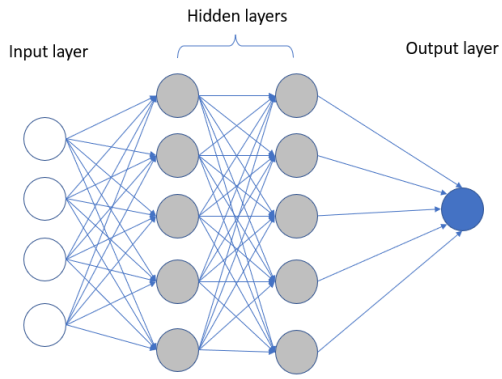
is done by entering a loop over each output neuron in the layer.
This looks like:
`for(int i = 0; i < outputs; i++)`. Inside the loop, the variable `val` is initialized to 0, but accumulates the weighted sum of the inputs for all the neurons. To compute the weighted sum, a nested loop iterates over each element in the `input_vector`. This looks like:
`val += input_vector.at(j) * weight_matrix.at(i).at(j);`.
The function is multiplying each value by it's weight from the `weight_matrix` and adds it to the `val` variable. The matrix is a 2D vector that has each row representing the weights of a particular neuron in a given layer. After this, the activation function is applied. This looks like:
`output_vector[i] = sigmoid(val + bias);`.
It maps the input which consists of the weighted sum plus the bias to a value between 0 and 1 with the idea of introducing non-linearity to the network. After processing all neurons in the layer, `output_vector` is returned, which holds the output for the layer.

Back in the `train` function, we then enter the loss function. The purpose of this function is to quantify the difference between the predictions made by the neural network and the actual target values of the MNIST dataset. This is done by calculating the Mean Squared Error loss. It takes in two parameters, which are the `output` vector which represents the neural network's output, and the `target` vector representing the correct values. First, we initialize the `loss` variable to 0.0. This variable will accumulate the the total squared error over all the output units. We then enter a loop that goes over every wlement in the output vector. This looks like:
`for (int i = 0; i < output.size(); ++i)`
Inside this loop, we find the squared difference of each element in the output and target vectors. The expression:
`pow(target.at(i) - output.at(i), 2)`
computes that squared difference of each element. This is added to the loss variable, which accumulates the total squared error. After the loop completes, the function then divides the total loss that was calculated over the number of output units, which is held in the `outputSize` variable. Finally, we return `loss / outputSize` which is the mean squared error.
Now, we enter the back_propagate function. This is a key component of the neural network, as backpropagation is used to compute the gradient of the loss function with respect to the network's weights, then adjusts the weights to minimize the loss. First, we calculate the error at the output layer by using the `computeOutputLayerError` function. Then store this into a `vector<double> error` variable. The function iterates backwards through the layers in the network with a loop. This looks like:
`for (int i = network_layers.size() - 1; i >= 0; --i)`
At each iteration, the function calls the `back_propagation_serial()` method. This works by taking an input, error, and learning rate as parameters. The gradients are initialized as vectors with zeros of a size that is equal to the number of outputs in the layers. The gradients are then computed by iterating over the output and input of each neuron using two nested for loops. For each pair of input and output neurons, the gradient of the weight that connects them is the product of the error of the output neuron and the activation of the input neuron. This looks like:
`gradientWeight[i] += error[i] * input[j]`.
Then, the gradient of the bias for each output neuron just the error of that neuron.
`gradientBias[i] += error[i]`.

Next, the function updates the weights and biases. It loops over the output neuron again to perform this update. for each weight,

we perform the following calculation: `weight_matrix[i][j] -= learningRate * gradientWeight[i]`. The errors of the previous later are then calculated, and we find the error for each input neuron as the sum of the product of the error for each output neuron and its corresponding weight. Finally, the function returns the error in the previous layer.

# Visualization




This is a simple visualization of what a neural network might look like. Of course, the number of neurons in the layers can vary, as well as the number of hidden layers. But the principle of how a neural network looks remains the same.

# SGD algorithm

The algorithm we tried to implement was the classic Stochastic Gradient descent algorithm.

**Algorithm: Stochastic Gradient Descent**
**Input:** Learning rate $\eta$, initial parameters $\theta$
**Input:** Loss function $L(\theta)$
Initialize parameters $\theta$
**loop**
  Randomly shuffle examples in the training set
  **for** each training set **do**
    Compute gradient $\nabla_\theta L(\theta)$ w.r.t. to the parameters
    Update parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$
  **end for**
**until** convergence or maximum epochs reached

# Parallelization methods

At first, we planned to parallelize both the forward and backpropagation methods, however we found that only the forward propagation method was able to be parallelized, as the backpropagation method is dependent upon the previous layer in our implementation. It works similarly to our serial approach, but uses OpenMP support to distribute the loop across multiple threads. In the function `forward_propagation_parallelized()`, we are able to use a `#pragma omp parallel for` above the outer loop `for(int i = 0; i < outputs; i++)`. Each iteration over the outer loop is independent, which makes it a good candidate for parallelization. With this parallelization, OpenMp automatically distributes the loop iteration across multiple threads, which is able to be defined in the main. The idea is that each thread will compute a subset of the output neurons, then combine them at the end to produce the final result.

This process is repeated by each layer in the network. The output of each neuron is a function of the weighted sum of its inputs and biases. If $W$ is the weight matrix, and $b$ is the

bias vector, then the output $y$ for just one neuron could be expressed as:

$$y = f(W \cdot x + b)$$

where $F$ is the Sigmoid function that we described earlier.

# Challenges and Results:

# Neural Network Model

In an effort to get a deeper understanding of how our network functions, we chose to test it on the Talapas supercomputer. However, the resulting accuracy numbers of the model indicated that the model in its current form is not functional. After twenty runs, the model averaged only 9.73% accuracy, indicating a random chance guess out of 10 integers. In future development, this would be a starting point for where to debug, likely lying somewhere in the optimization approach.

# Isolated testing

Within an isolated testing environment on Talapas, we tested the average time for the forward operation on a vector of size 128, with a weight matrix of dimension 128 - the size of an average hidden layer in the neural network (not the first or last hidden layer). Interestingly, on an average of 500 forward operations, the serial forward took an average of 0.53 seconds, while the parallelized forward operation took 1.56 seconds - utilizing 28 threads. We hypothesize this is likely due to the overhead time taken to set up the threads was too significant for a layer size of only 128 values.

# Serial Implementation

On every run, the serial implementation, running on 60,000 training images and 100 epochs, took an average of 1 hour, 4 minutes, and 19 seconds to train.

# OpenMP Implementation

Using 28 threads on Talapas, the execution time of the training over 60,000 images and 100 epochs took 1 hour, 32 minutes, and 12 seconds. Experimenting with various omp directives showed that the fastest approach was the simplest, simply leaving it statically scheduled with shared a shared output vector seemed to be the fastest, with a dynamically scheduled

# Remarks

The peculiar nature of the results indicates that either the multiplication was implemented incorrectly, simply too naive of an approach for vector multiplication, or the parallel code is being bottlenecked by the thread creation due to the fairly small size of each layer. This question can be answered through the utilization of the intel math library's vector multiplication function, cblas_dgemv(). On Talapas, this was accessed using `module load intel` to access icpc (intel's compiler) and `module load mkl` for the function definition. However, upon attempt to test with this function, we were unable to link the cblas or mkl header files on Talapas, leaving the method implemented yet untested.

# Conclusion

Overall, our implementation of the MNIST-trained neural network was an interesting and engaging problem to tackle and attempt to parallelize. A lot of time was spent wrapping our heads around the operations and ideas surrounding the implementation of a serial neural network. Despite a disappointing outcome, the project sparked new questions and was a great learning experience as a whole. What we venture to explore after this is the reason that the learning process does not happen, maybe stepping through the back propagation and determining where the error lies in the learning process. Additionally, learning if the parallelization was really slowing the process down or if the issue stemmed from other areas in the project. It would be an interesting idea to take on a different optimization method to see if it could further take advantage of parallelization, or if our parallel multiply was a bottleneck for any optimization approach.

# References and Sources


https://www.jeremyong.com/cpp/machine-learning/2020/10/23/cpp-neural-network-in-a-weekend/

https://optimization.cbe.cornell.edu/index.php?title=AdaGrad

https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

https://www.youtube.com/watch?v=Ilg3gGewQ5U
*3Blue1Brown - What is backpropagation really doing?*