Scalable Scientific Computing
CMDA 4634

Neural Networks from Scratch Using CUDA C++
Fall 2023

Patrick Dewey

March 13, 2024

Contents

Part I

Final projects

Lecture 1

Neural Networks from Scratch Using CUDA C++

Patrick Dewey

Contents

1.1 Introduction to Neural Networks

Neural networks, inspired by the human brain, are powerful computational models that are a staple in modern machine learning. Made up of interconnected nodes called neurons, these networks excel at learning complex patterns from data. Their importance lies in their ability to autonomously discover and represent intricate relationships within information, enabling applications such as image recognition, natural language processing, and predictive analytics.

1.1.1 Neural Network Components

Neurons

In neural networks, neurons serve as the fundamental computational units that process and transmit information. Inspired by the biological neurons in the human brain, their primary purpose is to transform input data through a series of mathematical operations, allowing the network to learn complex patterns and relationships within data. Artificial neurons receive input signals and multiply each by a corresponding weight, the results are then summed, a bias is added, and an activation function determines whether or not the neuron should activate.

They do this through the formula:

$$z = f(\vec{\mathbf{x}} \cdot \vec{\mathbf{w}} + b) \tag{1.1}$$

Where $\vec{\mathbf{x}}$ represents the input, $\vec{\mathbf{w}}$ the corresponding weights, b the bias, $f(z)$ is the activation function, and z the resulting output. This formula provides the foundation for the entire learning process of neural networks.

Activation Functions

Activation functions are pivotal components in neural networks, responsible for introducing non-linearity to models, allowing them to learn complex relationships in data. Serving as the decision-makers within each neuron, these functions determine whether or not the neuron should activate, based on the weighted input. Common activation functions include the sigmoid, hyperbolic tangent (tanh), rectified linear unit (ReLU), and softmax. Each activation function adds a layer of non-linearity to the network, enabling it to capture intricate patterns and make nuanced predictions. The choice of activation function is task-dependent, with ReLU or tanh often excelling in hidden layers while sigmoid and softmax are more suitable for output layers.

Layers

Layers are the fundamental part of neural networks that organize and structure the flow of information during both forward and backward propagation. Each layer consists of interconnected neurons, working collectively to transform input data into meaningful outputs. The input layer receives the initial data, and subsequent hidden layers progressively extract and learn hierarchical features from the input. The final output layer produces the network's prediction or classification. Layers are crucial for the network's ability to capture intricate patterns and representations within the data. The parameters within each layer, such as weights and biases, are fine-tuned during the training process, enabling the neural network to adapt and optimize its performance. The depth and architecture of layers contribute significantly to the network's capacity to model complex relationships, making the arrangement and configuration of layers a key factor in designing effective neural networks for various tasks.

Loss Functions

Loss (cost) functions play a central role in training machine learning models by quantifying the disparity between predicted and actual outcomes. Essential for optimization, these functions guide the learning process, helping models adjust their parameters to minimize errors. The choice of a suitable loss function hinges on the nature of the task, with some being suited to regression (mean squared error), while others are more suited to classification (cross-entropy). Loss functions guide model improvement, ensuring that predictions align closely with the desired outcomes.

1.1.2 Predictions and Learning

Forward and backward propagation are crucial components in the functioning of neural networks, encompassing the processes through which input data is transformed into predictions (forward propagation) and the processes through which the network learns from its errors to optimize its performance (backward propagation).

Forward Propagation

Forward propagation is the process through which input data is passed through a network to get an output. Starting at the input layer, each neuron processes the incoming data, and the information is successively passed through the network's layers until the final output is obtained. Taking the neuron activation formula 1.1, and expanding it to encompass the activations of every neuron in a given layer results in the following vectorized version of the formula:

$$Z = X \cdot W + \vec{\mathbf{b}}$$

Adding in the activation function, we get:

$$A = f(X \cdot W + \vec{\mathbf{b}})$$

To optimize the efficiency of forward propagation, this transition from computing the neuron activations sequentially to vectorizing the neuron update formula proves to be very advantageous as it allows for the addition of parallelization, significantly accelerating the speed at which a network can make a forward pass.

Forward propagation also plays an important role in the testing and deployment of neural networks, as making a forward pass over an input to get an output is all that is required to generate predictions for a given input.

Backward Propagation

Backward propagation (commonly called backpropagation or backprop), on the other hand, is the step in training in which gradients of the loss function are computed with respect to the weights and biases. This is done to allow the network to develop an understanding about how much each parameter contributed to the error. These gradients represent the sensitivity of the loss function to changes in the weights and biases, and the goal of training a neural network is to converge to a minimum along the gradients. This process is called gradient descent, and is a common numerical optimization method in machine learning that involves taking the negative gradient of the loss function in order to find the direction of steepest descent. The network then uses this descent direction to update the weights and biases, with the goal of reducing loss in future passes. Backpropagation starts at the output layer and works backwards through each layer of the network until it reaches the input layer, allowing the network to develop an understanding of how much each individual connection contributed to the overall error. The term "upstream gradients" refers to the flow of gradient information from the output layer back to the input layer, guiding the optimization algorithm in updating the weights and biases for enhanced learning and improved performance on specific tasks. (Note that there are alternative optimization methods that are sometimes used in place of gradient descent for training neural networks, but they are outside the scope of this project.)

Updated weights for a given neuron (i) and feature index (j) are given by:

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}$$

Where α represents a fixed learning rate and L the chosen loss function.

To update neuron biases (b), we use the formula:

$$b_i \leftarrow b_i - \alpha \frac{\partial L}{\partial b_i}$$

The processes of updating the weights and biases of neurons throughout a network during backpropagation is how neural networks actually learn. Making backward passes over inputs is the only way that a network can learn to make accurate predictions.

1.1.3 Network Architecture and Hyperparameters

Network architecture serves as the foundational blueprint for the organization and structure of a neural network, encompassing the arrangement and connectivity of layers, the number of neurons within each layer, and the configuration of parameters, crucially influencing the network's capacity to comprehend complex relationships and to excel in specific tasks. For a given network, the input layer will have a number of neurons equivalent to the dimension of the input data (i.e. 784 for MNIST digits), while the size of the output layer will be equivalent to the number of potential classes or response variables. In hidden layers, adjusting the number of neurons and choosing the correct activation functions are critical aspects of building a neural network, and as such, these variables are given a special name: hyperparameters

Hyperparameters are external settings, defined before training, that are crucial to the performance of a given neural network. They are values that control the learning process of a model, and as such, must be chosen carefully in order to get the desired result from the network. Hyperparameter choices influence accuracy, training speed, tendency to overfit, model complexity, and overall performance of a network. They are made up of model-architecture specific settings like layer sizes, the number of hidden layers, activation functions, and loss functions. They also include training-process related settings like learning rates, batch sizes and the number of epochs. There are a variety of different methods to find optimal hyperparameters including random search, grid search, and Bayesian optimization.

More Information

For more information on how neural networks learn and make predictions, as well as a more in depth explanation of how the math works, I highly recommend this video series by 3Blue1Brown (Grant Sanderson) explaining them in more detail.

1.2 Implementation in CUDA C++

Almost all neural network libraries are written to heavily utilize the power of GPUs, rather than merely executing on a CPU. Neural networks are considered embarrassingly parallel, meaning their operations are quite easy to separate and execute in parallel. This is due to the fact that every neuron in a layer can take in inputs and compute outputs independently of all other neurons in the same layer. Since layers are often made up of very large numbers of neurons, and individual calculations are relatively simple, this makes parallelization of in-layer calculations very well suited to GPUs with their many threads. GPUs are also immensely valuable in the forward and backward propagation processes as both perform multiplication operations on what are often quite large matrices. The process of gradient descent is also very well suited to parallelism since the gradients for different weights throughout the network are completely independent of each other. Add to this fact that neural networks are typically trained with very large amounts of data that are often independent (like images), and it's not hard to see how adding parallelization can result in significant speedup overall.

For this project, my focus was implementing a basic fully-connected feed-forward neural network (aka multilayer-perceptron) using CUDA.

1.2.1 Base Structure and Expansions

I started with a base implementation explained in this guide, and sourced from this repository [1] [2]. This base code was helpful in that it removed the need for me to implement a matrix class from scratch (which saved a large amount of time), and it provided a base structure for later expansion. However, much of the base implementation was inefficient in that it relied heavily upon the use of atomic operations, and was additionally only capable of performing binary classification on a single computationally simple (and quite boring) dataset. This dataset simply consisted of a grid of random points in two-dimensional space, and the network would classify if these points were in quadrants one and three or two and four. Fortunately, these limitations left me with plenty of room for improvement over the span of this project.

One of my main focuses throughout this project was optimizing the base code. This mainly took the form of replacing atomic operations with more advanced reductions, as well as replacing calls to unified (managed) memory with separate host and device allocations. While not originally part of my plan for this project, I actually spent the vast majority of my time expanding the capabilities of the neural network implementation. I first added a new dataset, the MNIST database of handwritten digits, which is a much higher dimensional and more realistic dataset than the 2D random points. In addition to this, I decided to implement new activation and loss functions to allow the network to work with multi-class classification problems. Hypothetically, with what is implemented now, the model could also perform regression tasks, but I don't have infrastructure to import other datasets at this point in time.

1.2.2 Activation Functions

The base code I started with came with the CUDA kernels for the ReLU and sigmoid activation functions, which in tandem, are a common choice for binary classification tasks. Since both functions are fairly simple mathematically, and their execution speeds were quite fast when profiled, I didn't see a need to make any modifications.

When I started working on expanding the capabilities of the neural network, I decided to add the softmax activation function since it is better suited to multi-class problems than a function like sigmoid. I also implemented the hyperbolic tangent activation function, because it is another common choice for usage inside hidden layers (like ReLU).

Rectified Linear Unit (ReLU)

The rectified linear unit activation function, or ReLU for short, is one of the most commonly used activations in deep learning. It is primarily used in the hidden layers of networks, and it adds non-linearity to neural networks, which allows them to more effectively identify and recognize complex relationships in data.

ReLU is represented by the piecewise function:

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

Compared to most other activation functions, ReLU is very computationally simple. ReLU also serves as a remedy to the vanishing gradient problem, an issue that can arise during backpropagation where the gradients of the loss function approach zero. Vanishing gradients are a problem since they reduce the efficiency of training and result in increased inaccuracies. [3]

Implementing ReLU in CUDA is quite simple:

```
__global__ void reluActivationForward(float* Z, float* A, int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        A[n] = fmaxf(Z[n], 0);
    }
}
```

Each thread evaluates one input and either outputs the input or zero, depending on which is bigger.

For backpropagation, the derivative of the ReLU function is given by:

$$\frac{\partial ReLU}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

Moving this to CUDA and adding the multiplication of the upstream gradient matrix (dA), this can be written:

```
__global__ void reluActivationBackprop(float* Z, float* dA, float* dZ,
                                        int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        dZ[n] = (Z[n] > 0) ? dA[n] : 0;
    }
}
```

In this kernel, Z represents the input data, dA the gradient matrix, and dZ the resulting gradient matrix after passing through the ReLU activation. Here, the step of multiplying the output of the derivative by the upstream gradient is skipped (since the ouput is just 1), and the upstream gradient value is returned directly in cases with a positive input. We also treat input values equal to zero as being less than zero to avoid the undefined case.

Sigmoid

The sigmoid function is another very common activation function utilized in neural networks, especially for binary classification tasks. It is a special case of the logistic function, where the input value is compressed to be between zero and one. Sigmoid is an incredibly versatile activation function, used for both hidden layers and output layers, and has a variety of different applications in all kinds of different neural network architectures. The sigmoid function is particularly useful in cases where the goal is to model probabilities since the output can be interpreted as the chance of a given outcome occurring. Another benefit of sigmoid is that it is differentiable everywhere (unlike ReLU), making it ideal for gradient descent.

Mathematically, the sigmoid function is represented:

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

CUDA implementation:

```
__inline__ __device__ float sigmoid(float x) {
    return 1.0f / (1 + expf(-x));
}

__global__ void sigmoidActivationForward(float* Z, float* A, int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        A[n] = sigmoid(Z[n]);
    }
}
```

The derivative of the function is given by:

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Where $\sigma(x)$ denotes the sigmoid function evaluated at x.

The CUDA implementation is as follows:

```
__global__ void sigmoidActivationBackprop(float* Z, float* dA, float* dZ,
                                          int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        dZ[n] = dA[n] * sigmoid(Z[n]) * (1 - sigmoid(Z[n]));
    }
}
```

Again, the gradient implementation does not require any reduction, with each thread computing a single value and writing it to the output matrix dZ.

Softmax

Softmax is an activation function designed for use in multi-class classification problems. An expansion of the sigmoid activation function, softmax is a generalization of the logistic function into multiple dimensions. It outputs a vector where each entry corresponds to a possible class label, and where all entries sum to one. While this output is not technically a vector of true probabilities, it is treated as one for the purposes of training and generating predictions from neural networks. In neural network architectures, softmax is usually used in the output layer for multi-class classification networks.

The formula for softmax is given by:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{n=1}^{N} e^{x_n}}$$

Implementation of softmax in CUDA:

```
__inline__ __device__ float expsum(float* Z, int Z_y_dim, int row) {
    float esum = 0;
    for (int i = 0; i < Z_y_dim; i++) {
        esum += expf(Z[row * Z_y_dim + i]);
    }
    return esum;
}

__global__ void softmaxActivationForward(float* Z, float* A, int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        int r = n / Z_y_dim;
        A[n] = expf(Z[n]) / expsum(Z, Z_y_dim, r);
    }
}
```

Migrating softmax to CUDA code required computing the exponential sum of each row in the output matrix, which I initially implemented as a separate kernel to reduce computation. But after some profiling, it turned out to be faster to allow each thread to just compute the sum for the row it is a part of. This timing difference is likely dataset dependent, since datasets with very large numbers of possible classes would likely be quicker with a separate row sum kernel.

The gradient of softmax is defined as:

$$\frac{\partial Softmax}{\partial x} = Softmax(x_i) \cdot (1 - Softmax(x_i))$$

This formula is quite visibly similar to the sigmoid gradient, which makes sense since both are derived from the logistic function.

Writing this in CUDA, we get:

```
__global__ void softmaxActivationBackprop(float* Z, float* dA, float* dZ,
                                          int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        int r = n / Z_y_dim;
        float smax = expf(Z[n]) / expsum(Z, Z_y_dim, r);
        dZ[n] = dA[n] * smax * (1.f - smax);
    }
}
```

This kernel is very similar to the formula, except that it adds the multiplication of the output with the upstream gradient (dA) as part of the backpropagation step.

Hyperbolic Tangent (tanh)

Hyperbolic tangent or tanh, is another very commonly used activation function that is primarily used inside hidden layers. Like sigmoid, the hyperbolic tangent function compresses the input into a range, but instead of being from zero to one, tanh is zero-centered and squashes values into the -1 to 1 range. Also like the sigmoid, hyperbolic tangent is fully differentiable, making it easily calculable during gradient descent. Tanh being centered around zero also helps alleviate the vanishing gradient problem (but not fully), so in some cases it is used in place of sigmoid as an activation function in hidden layers. Recently however, ReLU (and variants of ReLU) have surpassed tanh as the most popular choice for hidden layers since ReLU is more simplistic and tends to accelerate convergence. Despite this, tanh is still a valid option and can be very valuable in many network architectures.

The formula for hyperbolic tangent is given by:

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

Translating this to code was quite easy since the CUDA development toolkit includes a hyperbolic tangent function (tanh):

```
__global__ void tanhActivationForward(float* Z, float* A,int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        A[n] = tanh(Z[n]);
    }
}
```

Hyperbolic tangent's gradient is defined as:

$$\frac{\partial tanh}{\partial x} = 1 - tanh^2(x)$$

For backpropagation this is written as follows:

```
__global__ void tanhActivationBackprop(float* Z, float* dA, float* dZ,
                                       int Z_x_dim, int Z_y_dim) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < Z_x_dim * Z_y_dim) {
        dZ[n] = dA[n] * (1 - tanh(Z[n]) * tanh(Z[n]));
    }
}
```

This kernel is again quite simple, with each thread computing the hyperbolic tangent for one value, then multiplying by the upstream gradient.

1.2.3 Loss Functions

Over the course of my project, I worked primarily with three different loss functions: binary cross-entropy, categorical cross-entropy, and mean square error. Binary cross entropy was included in the base implementation I started with, and I added code for both categorical cross-entropy and mean square error.

Binary Cross Entropy

Binary cross entropy is a popular type of loss function that is used for binary classification tasks to measure the difference between the predicted probabilities and the actual labels.

Binary cross entropy can be calculated with the formula:

$$BCE = -\frac{1}{N}\sum_{n=1}^{N}(y_n \cdot log(\hat{y}_n) + (1 - y_n) \cdot log(1 - \hat{y}_n))$$

Where y represents actual values and $\hat{y}$ represents predicted values.

In the base code [2], this was implemented with the following kernel:

```
// v0: original implementation
__global__ void binaryCrossEntropyCost(float* yhat, float* y, int N, float* cost) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < N) {
        float partial_cost = y[n] * logf(yhat[n])
            + (1.0f - y[n]) * logf(1.0f - yhat[n]);
        atomicAdd(cost, - partial_cost / N);
    }
}
```

As optimization of the base code was one of my original goals, and this proved to be the slowest kernel by far, I spent time refactoring and optimizing it.

The first optimization I made was breaking down the atomic into a shared memory thread block tree-based reduction:

```
// v1: shared memory reduction
__global__ void binaryCrossEntropyCost(float *yhat, float *y, int N, float *cost) {
    int t = threadIdx.x;
    int n = blockIdx.x * blockIdx.x + threadIdx.x;

    __shared__ float s_pc[256];

    if (n < N) {
        float partial_cost = y[n] * logf(yhat[n]) +
            (1.0f - y[n]) * logf(1.0f - yhat[n]);

        // shared memory tree reduction
        s_pc[t] = (-1 * partial_cost) / N;

        if (t < 128) { s_pc[t] += s_pc[t + 128]; } __syncthreads();
        if (t < 64)  { s_pc[t] += s_pc[t + 64];  } __syncthreads();
        if (t < 32)  { s_pc[t] += s_pc[t + 32];  } __syncthreads();
        if (t < 16)  { s_pc[t] += s_pc[t + 16];  } __syncthreads();
        if (t < 8)   { s_pc[t] += s_pc[t + 8];   } __syncthreads();
        if (t < 4)   { s_pc[t] += s_pc[t + 4];   } __syncthreads();
        if (t < 2)   { s_pc[t] += s_pc[t + 2];   } __syncthreads();

        if (t == 0) {
            s_pc[t] += s_pc[t + 1];
            atomicAdd(cost, s_pc[t]);
        }
    }
}
```

In this verison, every thread in a block computes part of the final cost, then a tree-based sum reduction is performed to reduce the thread block sum to the first thread in every block. The result is finally added to the global cost accumulator via an atomic operation.

In the next version, I implemented a two-stage warp-based shuffle reduction:

```
// v2+: shuffle based warp reduction
__global__ void binaryCrossEntropyCost(float *yhat, float *y, int N, float *cost) {
    // switched to 2D thread blocks
    int t = threadIdx.x;
    int w = threadIdx.y;
    int nx = blockDim.x * blockIdx.x + threadIdx.x;
    int ny = blockDim.y * blockIdx.y + threadIdx.y;
    int n = ny * N + nx;

    __shared__ float w_pc[32];

    float pc = (n < N) ? -1 * (y[n] * logf(yhat[n] + 1e-5f) +
        (1.f - y[n]) * logf(1.f - yhat[n] + 1e-5f)) / N : 0;

    // shuffle reduction
    pc += __shfl_down_sync(MASK, pc, 16);
    pc += __shfl_down_sync(MASK, pc, 8);
    pc += __shfl_down_sync(MASK, pc, 4);
    pc += __shfl_down_sync(MASK, pc, 2);
    pc += __shfl_down_sync(MASK, pc, 1);

    // apppend results from first thread in each warp
    if (t == 0) {
        w_pc[w] = pc;
    }

    __syncthreads();

    // reduce full results in warp 0
    if (w == 0) {
        pc = w_pc[t];

        pc += __shfl_down_sync(MASK, pc, 16);
        pc += __shfl_down_sync(MASK, pc, 8);
        pc += __shfl_down_sync(MASK, pc, 4);
        pc += __shfl_down_sync(MASK, pc, 2);
        pc += __shfl_down_sync(MASK, pc, 1);

        // thread 0 in warp 0 adds to cost accumulator
        if (t == 0) {
            atomicAdd(cost, pc);
        }
    }
}
```

This reduction is similar to the previous since it reduces the partial sum to the first thread in every thread block. In this version however, shuffles are used in place of most shared memory storage in order to reduce pressure on the shared memory bus. This version was the fastest of the three, so later cost functions were implemented exclusively using shuffle reductions.

In my testing, performing a two-stage shuffle reduction did not have a noticeable performance improvement over a single-stage reduction, so later cost functions were also implemented with single stage reductions.

In order to perform backpropagation, we also need to compute the gradient of the binary cross entropy function.

$$\frac{\partial BCE}{\partial \hat{y}} = -\frac{1}{N}\sum_{n=1}^{N}\left(\frac{y_n}{\hat{y}_n} - \frac{1-y_n}{1-\hat{y}_n}\right)$$

In CUDA code, this takes the following form:

```
__global__ void dBinaryCrossEntropyCost(float *yhat, float *y, float *dY, int N) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < N) {
        dY[n] = -1 * (y[n] / (yhat[n] + 1e-5f) -
            (1 - y[n]) / (1 - yhat[n] + 1e-5f));
    }
}
```

In this gradient kernel, each thread computes an individual value and writes it to the dY vector. No reductions are required here since each gradient value can be calculated and written completely independently of the others.

Categorical Cross Entropy

Categorical cross-entropy (CCE or CE) is a type of loss function similar to binary cross-entropy, except that it is designed for classification problems with multiple classes rather than just binary classification. It is very commonly used with the softmax activation function since CE takes in a probability vector and encourages one element equal to one, and all others to be zero.

Mathematically, categorical cross-entropy is calculated through:

$$CCE = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C}(y_{nc} \cdot log(\hat{y}_{nc}))$$

Where C represents the total number of classes present in the data.

Converting this into CUDA code we get:

```
__global__ void crossEntropyCost(float* yhat, float* y, int N, int C, float* cost) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;

    int row = n / C;
    int c = n % C;
    float pc = (n < N * C) ? -1.f * (c == static_cast<int>(y[row])) *
        logf(yhat[n] + 1e-5f) : 0;

    pc += __shfl_down_sync(MASK, pc, 16);
    pc += __shfl_down_sync(MASK, pc, 8);
    pc += __shfl_down_sync(MASK, pc, 4);
    pc += __shfl_down_sync(MASK, pc, 2);
    pc += __shfl_down_sync(MASK, pc, 1);

    if (threadIdx.x % 32 == 0) {
        atomicAdd(cost, pc);
    }
}
```

To minimize execution time, this cost function kernel utilizes a warp based shuffle reduction.

Note that the predictions array ($\hat{y}$) is a matrix, but the actual values array (y) is a vector. To deal with this difference without changing the format of y, I made each thread compare its column index c to the true label $y[row]$, which will result in a one when they match and zero otherwise. An alternative approach to this would have been using one-hot encoding to convert the vector into a sparse matrix where each row is full of zeroes except for the index matching the true label.

The gradient formula for categorical cross entropy is defined as:

$$\frac{\partial CCE}{\partial \hat{y}} = \hat{y}_n - y_n$$

In this formula, it is important to note that $\hat{y}$ represents the predicted probability the input is a certain class, and y_n represents the true probability for that respective class, which will either be zero or one since there are no partial probabilities for true labels.

In CUDA this can be written:

```
__global__ void dCrossEntropyCost(float* yhat, float* y, float* dY, int N, int C) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < N * C) {
        int row = n / C;
        int c = n % C;
        dY[n] = yhat[n] - (c == static_cast<int>(y[row]));
    }
}
```

Again, the column index c is compared to $y[row]$ to determine whether the prediction should match with the true value.

Mean Square Error

Mean square error (MSE) is a common loss function that measures the average squared difference between actual and predicted values. It is most commonly used in regression problems, and is optimal when the underlying data is normally distributed. MSE is not typically used in classification problems since it is sensitive to class imbalances and outliers.

Mean square error is defined as:

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2$$

Where y represents actual values and $\hat{y}$ represents predicted values.

Translating this to CUDA code we get the following:

```
__global__ void meanSquareErrorCost(float* yhat, float* y, int N, int C, float* cost) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    int row = n / C;
    int c = n % C;

    float diff = (n < N * C) ? yhat[n] - (c == y[row]) : 0;
    float sum = (diff * diff) / C;

    // warp shuffle reduction
    sum += __shfl_down_sync(MASK, sum, 16);
    sum += __shfl_down_sync(MASK, sum, 8);
    sum += __shfl_down_sync(MASK, sum, 4);
    sum += __shfl_down_sync(MASK, sum, 2);
    sum += __shfl_down_sync(MASK, sum, 1);

    if (threadIdx.x % 32 == 0) {
        atomicAdd(cost, sum);
    }
}
```

In this kernel, each thread computes the partial mean square error between the actual and predicted value at one index, and then a single stage warp shuffle reduction is performed to aggregate the results from every warp. The cumulative warp sum is then added to a global accumulator (cost) by the first thread in every warp. In my testing, the loss functions were generally run with relatively small numbers of active threads so a multi-stage reduction wasn't necessary.

Again, I used the column comparison method to convert the vector of true values into a matrix, but it is possible that this would not work for regression tasks, since column indices wouldn't be one-to-one matches with response values.

For the backpropagation step, we need the derivative of the loss function:

$$\frac{\partial MSE}{\partial \hat{y}} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)$$

Translating this to CUDA code we get:

```
__global__ void dMeanSquareErrorCost(float* yhat, float* y, float* dY, int N, int C) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (n < N * C) {
        int row = n / C;
        int c = n % C;
        dY[n] = 2.f * (yhat[n] - (c == y[row])) / C;
    }
}
```

This kernel does not require any kind of reduction since every thread is able to write to a unique slot in the dY matrix.

1.2.4 Forward and Backward Propagation

Forward Propagation

As stated previously, the equation for forward propagation is given by:

$$Z = X \cdot W + \vec{\mathbf{b}}$$

In this formula, the input matrix X is multiplied with the weights matrix W, then a bias vector $\vec{\mathbf{b}}$ is added. We apply then this update formula for every layer in the network, starting at the input layer and ending with the output layer. Features from the input are extracted, and a prediction is produced.

Translating the formula into CUDA code, this can be written:

```
__global__ void linearLayerForward(float* W, float* A, float* Z, float* b, int W_x_dim,
                                   int W_y_dim, int A_x_dim, int A_y_dim) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int Z_x_dim = A_x_dim;
    int Z_y_dim = W_y_dim;

    float Z_value = 0;

    if (row < Z_y_dim && col < Z_x_dim) {
        // 8x8 thread blocks
        for (int i = 0; i < W_x_dim; i++) {
            Z_value += W[row * W_x_dim + i] * A[i * A_x_dim + col];
        }
        Z[row * Z_x_dim + col] = Z_value + b[row];
    }
}
```

This kernel performs the matrix multiplication of the weight (W) and input (A in this case) matrices, then adds the bias vector to the result. After this, a launcher function calls the desired activation function's kernel, and the resulting output is passed on to the next layer of the network.

Backward Propagation

Backpropagation performs a similar process, except this time multiplying the model gradient of the loss function for each output (dZ) with the weights matrix (W).

In CUDA this can be written:

```
__global__ void linearLayerBackprop(float* W, float* dZ, float *dA, int W_x_dim,
                                    int W_y_dim, int dZ_x_dim, int dZ_y_dim) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // W is treated as transposed
    int dA_x_dim = dZ_x_dim;
    int dA_y_dim = W_x_dim;

    float dA_value = 0.0f;

    if (row < dA_y_dim && col < dA_x_dim) {
        for (int i = 0; i < W_y_dim; i++) {
            dA_value += W[i * W_x_dim + row] * dZ[i * dZ_x_dim + col];
        }
        dA[row * dA_x_dim + col] = dA_value;
    }
}
```

Again, this is a standard (but not particularly advanced) matrix multiplication kernel that multiplies the loss gradient by the neuron weights. The output from one layer is then used to populate the upstream gradient matrix dA for the next layer.

To update the weights for a layer, we perform another matrix multiplication between the loss gradient dZ and the input matrix A. Afterwards, each weight in each neuron is adjusted according to formula:

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W}$$

Translating to CUDA:

```
__global__ void linearLayerUpdateWeights(float* dZ, float* A, float* W, int dZ_x_dim, int dZ_y_dim
    ,
                                          int A_x_dim, int A_y_dim, float learning_rate) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // A is treated as transposed
    int W_x_dim = A_y_dim;
    int W_y_dim = dZ_y_dim;

    float dW_value = 0.0f;

    if (row < W_y_dim && col < W_x_dim) {
        for (int i = 0; i < dZ_x_dim; i++) {
            dW_value += dZ[row * dZ_x_dim + i] * A[col * A_x_dim + i];
        }
        W[row * W_x_dim + col] = W[row * W_x_dim + col] - learning_rate * (dW_value / A_x_dim);
    }
}
```

Again, this is a typical matrix multiplication kernel where the gradient is multiplied with the input matrix, and an adjustment is applied before writing the result to the weights matrix.

Moving on to updating biases, we use a similar process:

$$\vec{\mathbf{b}} \leftarrow \vec{\mathbf{b}} - \alpha \frac{\partial L}{\partial \vec{\mathbf{b}}}$$

With the corresponding CUDA code being:

```
__global__ void linearLayerUpdateBias(float* dZ, float* b, int dZ_x_dim, int dZ_y_dim,
                                      int b_x_dim, float learning_rate) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < dZ_x_dim * dZ_y_dim) {
        int dZ_x = index % dZ_x_dim;
        int dZ_y = index / dZ_x_dim;
        float res = -learning_rate * (dZ[dZ_y * dZ_x_dim + dZ_x] / dZ_x_dim);
        atomicAdd(&b[dZ_y], res);
    }
}
```

Since bias is a vector rather than a matrix, each thread determines which entry of the vector it needs to update, then calculates the bias update for a unique element of the loss gradient matrix (dZ), normalizes by the number of features, and finally adds this result to the bias vector with an atomic add. Despite the usage of an atomic operation, this kernel was extremely fast, so a more advanced reduction didn't seem necessary.

1.2.5 Putting Everything Together

For the sake of modularity, the file "nn_layer.hh" from the base implementation defines an abstract class (similar to a Java interface) that allows for the easy creation and modification of neural network layers:

```
class NNLayer {
public:
        virtual ~NNLayer() = 0;
        virtual Matrix& forward(Matrix& A) = 0;
        virtual Matrix& backprop(Matrix& dZ, float learning_rate) = 0;
};
inline NNLayer::~NNLayer() {}
```

By making all the activation functions extend this abstract layer class, we can define an "addLayer" function (found in "neural_network.cu") that is able to take in an arbitrary layer type and add it to the neural network architecture. This class definition also makes a guarantee to the neural network class that the layer added to the network has forward and backward propagation functions defined that the network can use. It provides this guarantee in the form of virtual functions—a way of expressing that any derivative class is required to have a matching function of the same name, return type, and input arguments.

For classifying MNIST digits, I created a network with an input layer of size 784 (number of pixels in each image), this input layer then feeds into a hidden layer containing 512 neurons, which in turn uses a ReLU activation function before moving on to the second hidden layer containing 128 neurons and using another ReLU activation. The network finishes by feeding the result from the second hidden layer into the output layer containing 10 neurons (one for each possible class label) and a softmax activation function to produce a normalized probability vector output.

Written in C++ code, this takes the form:

```
// instantiate new neural network
NeuralNetwork nn;

// layer definitions
nn.addLayer(new LinearLayer("linear_1", Shape(784, 512)));
nn.addLayer(new ReLUActivation("relu_1"));
nn.addLayer(new LinearLayer("linear_2", Shape(512, 128)));
nn.addLayer(new ReLUActivation("relu_2"));
nn.addLayer(new LinearLayer("linear_3", Shape(128, 10)));
nn.addLayer(new SoftmaxActivation("softmax_output"));
```

Before passing data into the network, we also need to choose a loss function. After adding both the categorical cross-entropy and mean square error loss functions, I created an abstract class definition for loss functions to improve modularity.

This class can be seen as follows:

```
class Cost {
public:
    virtual ~Cost() = 0;
    virtual float cost(Matrix predictions, Matrix target) = 0;
    virtual Matrix dCost(Matrix predictions, Matrix target, Matrix dy) = 0;
};
inline Cost::~Cost() {}
```

Again, virtual functions are utilized to tell the neural network class that any cost function extending the "Cost" class will have a "cost" and "dCost" function.

As an example of abstract class usage, here is my cross-entropy class definition:

```cpp
#include "cost.hh"

class CECost : public Cost {
public:
    CECost();
    ~CECost();
        float cost(Matrix predictions, Matrix target);
        Matrix dCost(Matrix predictions, Matrix target, Matrix dY);
};
```

Similar to the layer class, this cost class definition allows for more generic use of cost functions. This proved to be especially valuable in the call to "nn.backprop()", which is defined as follows:

```cpp
void NeuralNetwork::backprop(Matrix predictions, Matrix target, Cost* cost) {
    ...
    Matrix error = cost->dCost(predictions, target, dY);
    ...
```

By using the abstract cost class, the "cost" parameter can take in any type of loss function allowing the backprop function implementation to work generically by outsourcing the computation to the correct derivative class function.

To define a loss function for use in the network, we simply choose one of the following:

```cpp
// for BCE
BCECost cf;
// for CE
CECost cf;
// for MSE
MSECost cf;
```

To train the network, we need to make one forward and backward pass over each batch of the dataset for every epoch (training iteration). In code, this looks like:

```cpp
Matrix Y;
for (int epoch = 0; epoch < epochs + 1; epoch++) {
    float cost = 0.0;
    for (int batch = 0; batch < dataset.getNumOfBatches(); batch++) {
        Y = nn.forward(dataset.getBatches().at(batch));
        nn.backprop(Y, dataset.getTargets().at(batch), &cf);
        cost += cf.cost(Y, dataset.getTargets().at(batch));
    }
    if (print_epoch > -1 && epoch % print_epoch == 0) {
        std::cout       << "Epoch: " << epoch << ", Cost: " << cost / dataset.getNumOfBatches() <<
             std::endl;
    }
}
```

This code loops over every defined batch, once for each epoch, calling the forward and backprop functions to generate output and to update neurons. It also computes and prints the cost for each epoch as the network trains.

Note that the batch loop could certainly be parallelized to significant effect, which would be an excellent next step in improving the network implementation further.

Testing

The ability to pass in a testing dataset is critical in order to evaluate the prediction accuracy of a neural network, but was surprisingly not implemented in the base code. In another puzzling design decision, the base code evaluated the accuracy of only the last training batch and used that as the overall measure of accuracy. This is not a good representation of overall accuracy, especially when batch sizes are small.

In order to address these issues, I set up testing infrastructure and improved the accuracy computation function when I was working on importing the MNIST dataset. This infrastructure consisted of reading in the testing dataset separately, and then making forward passes over it after training the network.

The code can be seen as follows:

```
MNISTDataset test_set(batch_size, ts / batch_size, test_image_file, test_labels_file);
Matrix T;
float test_acc = 0.0;
for (int i = 1; i <= ts / batch_size; i++) {
    T = nn.forward(test_set.getBatches().at(test_set.getNumOfBatches() - i));
    T.copyDeviceToHost();
    test_acc += computeAccuracy(T, test_set.getTargets().at(test_set.getNumOfBatches() - i));
}
test_acc /= (ts / batch_size);
std::cout << "Test Accuracy: " << test_acc << std::endl;
```

Additionally, when moving to MNIST classification, I modified the "computeAccuracy" function to use the correct labels in the binary classifier case (zeroes and ones) as well as to accommodate for multi-class classification.

1.3 Results

My best results for each version, along with some of the important hyperparameter choices, are contained in the table below:

Dataset	Accuracy	N	Batch Size	Batches	Epochs
2D Points	91%	2100	100	21	1000
MNIST 0/1	98.7%	12000	2	6000	5
MNIST	91.3%	60000	2	30000	4

Table 1.1: Network Accuracies and Architectures

For the 2D points network, I used an input layer with 2 neurons, two hidden layers containing 30 neurons each, ReLU activation functions for both, and a sigmoid output layer with one neuron. This dataset utilizes the binary cross-entropy loss function.

For the MNIST zero and one classifier, the input layer had 784 neurons, there were two hidden layers with 128 and 64 neurons respectively, both using ReLU, and a sigmoid output layer with one neuron. This classifier also used binary cross-entropy loss.

For the full MNIST classifier, my input layer again had 784 neurons, there were two hidden layers with 512 neurons each, ReLU activation functions, and a softmax output layer with 10 neurons. Here I flipped between using categorical cross-entropy loss and mean square error, both yielding similar results (the higher accuracy was obtained using MSE, but it was close).

1.3.1 Profiling

Timing

One of my original goals for this project was to optimize and profile the kernels in each code version. I was able to determine which kernels were in need of improvement by profiling execution time on a kernel-by-kernel basis, allowing me to easily evaluate the optimizations I made.

Binary Classification

Versions 0 through 2 make up the binary classification component of my implementation, with version 0 being the base implementation with modifications allowing for MNIST classification and better testing, version 1 optimizing the binary cross-entropy loss function by utilizing a shared memory tree reduction, and version 2 optimizing the loss function further using shuffle instructions.

Overall timing for 2-dimensional coordinate binary classification:

Version	Execution Time
0	7.629 seconds
1	0.852 seconds
2	0.830 seconds
2.5	0.849 seconds

Table 1.2: 2D Points Full Execution Time Per Version

In terms of execution time, version 0 was by far the slowest, with all later versions being 9 times faster. We can see also see that version 2 is the fastest overall.

Timing results for 2-dimensional points binary classification network:

	Name	v0	v1	v2	v2.5
1	binaryCrossEntropyCost	200.94	3.50	2.49	2.39
2	dBinaryCrossEntropyCost	2.58	2.43	2.42	3.26
3	linearLayerBackprop	2.61	2.65	2.64	2.66
4	linearLayerForward	3.03	2.80	2.78	2.80
5	linearLayerUpdateBias	3.28	3.36	3.35	3.37
6	linearLayerUpdateWeights	4.24	4.35	4.35	4.36
7	reluActivationBackprop	2.11	2.14	2.11	2.12
8	reluActivationForward	1.95	1.95	1.96	1.95
9	sigmoidActivationBackprop	2.47	2.18	2.18	2.20
10	sigmoidActivationForward	2.04	2.05	2.05	2.06

Table 1.3: Average Kernel Timing Results for 2D Points Dataset (μs)

By looking at these timing results, we can see that all of these kernels run quickly, with the single exception of the v0 "binaryCrossEntropyCost" function. This significant time difference arises from the usage of unified (managed) memory. Unified memory is memory that is kind of shared between the host and device, and the slowdown is caused by the need to spend over $100\mu s$ moving the shared value from host memory to device memory. By switching from unified memory to separate allocations for the host and device, the "binaryCrossEntropyCost" kernel was sped up by 57x, with 87x speedup reached comparing version 0 to the best version (v2). This usage of unified memory was the accounted for the slowness of version 0 compared to all the others.

Overall timing for 0/1 MNIST classifier coordinates binary classification:

Version	Execution Time
0	2.352 seconds
1	0.992 seconds
2	0.987 seconds

Table 1.4: MNIST 0/1 Full Execution Time Per Version

It is obvious that versions 1 and 2 are substantially faster than 0 as a result of unified memory usage. Version 2 (the shuffle reduction version) is the fastest overall, but with only a slight improvement over version 1 with this dataset.

Interestingly, version 0 actually got faster when working with the new dataset, but this is simply a byproduct of running fewer training epochs (meaning fewer calls to unified memory).

Timing results for MNIST binary classifier:

	Kernel	v0	v1	v2
1	binaryCrossEntropyCost	142.77	3.25	2.41
2	dBinaryCrossEntropyCost	1.62	2.37	2.40
3	linearLayerBackprop	15.00	15.52	15.53
4	linearLayerForward	20.80	18.34	18.34
5	linearLayerUpdateBias	1.39	2.16	2.16
6	linearLayerUpdateWeights	7.02	7.81	7.80
7	reluActivationBackprop	1.16	2.07	2.09
8	reluActivationForward	1.40	2.05	2.05
9	sigmoidActivationBackprop	1.30	2.25	2.22
10	sigmoidActivationForward	1.16	2.01	2.02

Table 1.5: Average Kernel Timing Results for MNIST 0s and 1s Dataset (μs)

From this timing table, we can see that switching away from using unified memory again results in massive speedup (44x for v1, 59x for v2). We can also see that switching to a higher dimensional dataset did indeed increase the execution time for the kernels performing matrix multiplication.

Multi-Class Classification

Moving on to the multi-class classifier, there are 4 new versions: version 3 which contains the initial implementations for softmax and mean square error, version 4 which refactors the MSE kernel from version 3 into a shared reduction, version 5 which experiments with cuBLAS for faster matrix multiplication, and version 6 which changes the MSE function of version 3 to a shared memory tree reduction.

Overall timing for full MNIST classifier coordinates binary classification:

Version	Execution Time
3	4.400 seconds
4	4.330 seconds
5	$>$ 10 seconds
6	4.321 seconds

Table 1.6: MNIST Full Execution Time Per Version

Versions 3, 4, and 6 all run in similar amounts of time with versions 4 and 6 being the best overall. In version 5, I used cuBLAS in an attempt to speed up the matrix multiplications in the forward and backward propagation kernels. Unfortunately, this actually resulted in a noticeable slowdown, most likely due to small matrix dimensions (cuBLAS is optimized for large matrices).

(Note: Version 5 is not included in the kernel timings table due to issues with the cuBLAS library files on pascal.)

Average kernel timings:

	Name	v3	v4	v6
1	dMeanSquareErrorCost	2.69	1.25	1.22
2	exp_sum	1.57	n/a	n/a
3	linearLayerBackprop	14.09	14.46	14.38
4	linearLayerForward	17.54	17.59	17.51
5	linearLayerUpdateBias	1.24	1.25	1.25
6	linearLayerUpdateWeights	6.93	7.00	6.97
7	meanSquareErrorCost	1.68	1.37	1.63
8	reluActivationBackprop	1.11	1.11	1.11
9	reluActivationForward	1.01	0.98	0.99
10	softmaxActivationBackprop	1.28	1.89	1.91
11	softmaxActivationForward	1.15	1.80	1.83

Table 1.7: Average Kernel Timing Results for Full MNIST Dataset (μs)

According to this timing table, most of the kernels take very little time to execute, with the exception being the kernels performing matrix multiplication operations (forward/backprop/weights). We can see that the most noticeable difference is between the different versions of "dMeanSquareErrorCost". My initial MSE implementation (v3) was not particularly well parallelized, so when those issues were remedied (v4+), the kernel ran more than twice as fast. The kernel "meanSquareErrorCost" also showed significant speedup from version 3 to 4 (the version using shuffles), but no speedup from version 3 to 6 (the shared memory reduction). Again, the shuffle reduction implementation was the fastest.

Another important observation is the use of the "exp_sum" kernel which only exists in version 3. I implemented this kernel originally in an effort to reduce the total amount of computation each thread performed in softmax, by pre-computing the required exponential row sums. While the softmax forward and backward kernels are faster when using "exp_sum", the additional kernel launch overhead added by using multiple kernels is not worth the tradeoff, making version 3 slower.

Metrics

Unfortunately, I determined early on that there are a number of difficulties that arise when attempting to perform metrics profiling on neural networks. First, it took an unreasonably long time (more than 24 hours) to profile a single version of my code with only a few metrics, notwithstanding a realistic choice for batch size and a singular epoch. This made profiling incremental changes (an important part of my project) nearly impossible, and proved to be an insurmountable obstacle. With help from Dr. Warburton, I later determined profiling could be performed quickly by running one batch of a size equivalent to the length of an entire dataset. But this is not a realistic use case as it would result in a poor predictive network prone to overfitting. Additionally, I determined that the hyperparameter choices have an immense impact on metrics performance, and since they are drastically different in every neural network, there is no one-size-fits-all approach that would be able to provide results generalizable outside the specific network architecture I constructed. In light of these two major obstacles, I chose not to pursue metrics profiling or roofline modeling further, since any results would likely be subpar.

However, in the context of neural networks, some promising metrics to look at would be DRAM read and write throughput as well as floating point operation count (single precision). In tandem, these metrics are used to calculate arithmetic intensity, which is a valuable metric to consider when evaluating the performance of GPU kernels. For example, the propagation functions, as well as certain activation functions, should have high arithmetic intensities since they perform comparatively large numbers of floating point operations for every value moved in memory. Of course, hyperparameters have a sizeable impact on this, since batch and layer sizes both influence matrix sizes inside the network, so choosing small values for either of these can result in lowered arithmetic intensities. Checking memcpy throughput could also be useful, to make sure that there is no unnecessary data movement between the device and the host. Some other useful metrics would include achieved occupancy, warp execution efficiency, and SM efficiency, all of which are good for validating that GPU resources are being used effectively by our kernels.

1.3.2 Limitations and Future Improvements

I was able to make many improvements and expansions to the base implementation, yet there is considerable room for further improvement.

First, the matrix multiplication kernels could be drastically improved further since they utilize the most basic method of performing parallel matrix multiplications. More advanced techniques like block multiplications with strip mining, could also bring about improvement.

Another notable limitation is that training batches are processed sequentially, which I finally noticed late into the writing of this report. This limitation could be reasonably addressed by parallelizing the inner loop in the training code with the CPU (likely a band-aid solution), or by modifying the kernel launcher functions to allow for the processing of multiple batches simultaneously.

This implementation is also limited by the ability to only generate predictions for the 2D points dataset and MNIST digits. A realistic remedy for this would be generalizing the MNIST dataset code defined in “mnist_dataset.cu” to work for comma separated value (csv) files. This would vastly expand the capabilities of the network, allowing for a much larger range of use cases.

There is also room for expansion in terms of the available activation and loss functions. For activation functions, this could include common variants of ReLU, like leaky ReLU, or others such as the exponential linear unit (ELU). Other potential loss functions could include exponential cost or Hellinger distance.

Another somewhat reasonable expansion of this network would be the addition of convolutional layers. Convolutional layers can help networks better learn patterns in time series and image data.

Another major limitation I encountered while experimenting with different hyperparameters was numerical instability. The network would train normally at first, but at some point (and usually before the cost function converged) the calculated cost would suddenly become NaN. After a somewhat extensive investigation, I determined this issue happens when the gradient of the loss function abruptly grows very large before becoming NaN. This problem is called the exploding gradient problem, and prevents networks from converging to minima. While I was not able to solve this problem outside of merely changing hyperparameters, there are methods of dealing with exploding gradients, the most common choice being a process called “gradient clipping”. [4]

1.3.3 Conclusion

After all of this experimentation, I have a neural network capable of performing binary classification on the 2D points dataset in addition to the MNIST zeroes and ones. My network implementation is also capable of performing multi-class classification on the entire MNIST handwritten digits dataset. With modifications to allow the importing of csv files, the network could conceivably be generalized to solve any single-label classification problem, as long as the dataset is numeric. For each of the three datasets I tested, I was able to make networks capable of achieving upwards of 90% predictive accuracy, all in a reasonable amount of time. My improvements to the base code yielded 9.2x speedup, and I was able to expand the capabilities of the network to allow it to accomplish new types of problems. Through work on this project, I developed a deep understanding of how neural networks function and how their underlying math works. I also learned a great deal about developing large scale applications in CUDA, and about best practices and avoiding potential pitfalls when writing GPU code.

1.4 References

Project Links

Project GitHub repository: link

Overleaf write-up: link

Bibliography

[1] Paweł Lunial. “CUDA Neural Network Implementation (Part 1)”. In: *Online link* (2018).

[2] Paweł Lunial. 2018.

[3] Chi-Feng Wang. “The Vanishing Gradient Problem”. In: *Online link* (2019).

[4] Jason Brownlee. “A Gentle Introduction to Exploding Gradients in Neural Networks”. In: *Online link* (2019).

[5] Grant Sanderson. “Neural Networks from the Ground Up”. In: *Online link* (2017).

[6] Nvidia Corp. “cuBLAS API Reference Guide”. In: *Online link* (2023).

Codebase Overview

In this project, I started with a baseline GPU neural network implementation, but over time it evolved through the addition of optimizations, new features, and other modifications. For the sake of full transparency, I have created a comprehensive list of the files that were created, modified, or left unchanged over the span of the project.

Note: "*U*" denotes a file unchanged from the base implementation, "*M*" denotes a file with modifications, and "*N*" denotes a new file that I created. (Header files for non-abstract classes not included for brevity)

Filename	***Tag***
coordinates_dataset.cu	*U*
layers/linear_layer.cu	*U*
layers/nn_layer.hh	*U*
layers/relu_activation.cu	*U*
layers/sigmoid_activation.cu	*U*
layers/softmax_activation.cu	*N*
main.cu	*M*
makefile	*N*
mnist_dataset.cu	*N*
neural_network.cu	*M*
nn_utils/bce_cost.cu	*M*
nn_utils/ce_cost	*N*
nn_utils/cost.hh	*N*
nn_utils/matrix.cu	*U*
nn_utils/mse_cost.cu	*N*
nn_utils/nn_exception.hh	*U*
nn_utils/shape.cu	*U*