CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 5

Total number of points: 100.

In this homework, you will accelerate the forward and backward passes of a neural network using a single GPU. The neural network will be trained on a single CPU (one MPI node) with an associated GPU. Computation in the forward and backward passes on each MPI node (such as matrix multiplication and softmax) should be accelerated using the GPU kernels you implemented in Homeworks 2 and 4. Please refer to the appendix for details of the forward and backward passes. You can follow the CPU implementation in `neural_network.cpp` to guide your GPU implementation.

Problem 1 Neural Network on the GPU

Similar to how you implemented a class `DeviceMatrix` to represent matrices on the GPU in Homework 2, you will implement a class to represent neural networks on the GPU. The purpose of these abstractions is to make it easier to write and debug code.

In Homework 6, you will accelerate training by performing forward and backward passes in parallel on multiple MPI nodes, each with an associated GPU. Two general techniques to parallelize training across multiple nodes are data parallelism and model parallelism:

- In data parallelism, each mini-batch is partitioned across nodes and the neural network's parameters are replicated on each node. Each node computes the forward and backward passes for its partition of the mini-batch, and gradients computed in the backward pass are averaged across nodes before updating the parameters.
- In (vertical) model parallelism, each mini-batch is replicated on each node and the neural network's parameters are partitioned across nodes. E.g. layer 1 of the neural network may be on node 0 and layer 2 of the neural network may be on node 1. Each node computes the part of the forward pass and the part of the backward pass corresponding to its partition of the parameters, and gradients computed in the backward pass are communicated across nodes before updating the parameters.

Different modes of parallelism are used to achieve the twin goals of accelerating training and fitting large models onto limited hardware. See e.g. `https://huggingface.co/docs/transformers/v4.15.0/en/parallelism` for more examples.

In this homework, you will implement a class `DataParallelNeuralNetwork`. For the final project, you will have the opportunity to implement a class `ModelParallelNeuralNetwork` as a bonus.

Question 1.1

(5 points) Implement the constructor for the class `DataParallelNeuralNetwork` in `neural_network.cpp`.

Question 1.2

(5 points) Implement the member function `DataParallelNeuralNetwork::to_cpu`.

Question 1.3

(5 points) Implement the constructor for the class `GPUGrads`.

Problem 2 Forward Pass

Question 2.1

(30 points) Implement the member function `DataParallelNeuralNetwork::forward` in `neural_network.cpp`. We will test your implementation by comparing the cached outputs of each layer with the cached outputs of the CPU implementation.

Problem 3 Loss

Question 3.1

(10 points) Implement the member function `DataParallelNeuralNetwork::loss` in `neural_network.cpp`. We will test your implementation by comparing its output with the output of the CPU implementation.

Problem 4 Backward Pass

Question 4.1

(40 points) Implement the member function `DataParallelNeuralNetwork::backward` in `neural_network.cpp`. You can use the wrapper function `tiledGEMM` in `gpu_stencil.cu` to perform the GEMM operation

$$C \leftarrow \alpha AB + \beta C$$

when either A or B is transposed. You are welcome to use or modify your own GEMM implementation from Homework 4. Note that only `neural_network.cpp` will be submitted for grading. We will test your implementation by comparing the cached gradients with respect to each parameter with the cached gradients of CPU implementation.

Problem 5 Optimizer Step

Question 5.1

(5 points) Implement the member function `DataParallelNeuralNetwork::step` in `neural_network.cpp`. We will test your implementation by comparing the updated parameters with the updated parameters of the CPU implementation.

References

[1] Yann LeCun et. al. MNIST. `http://yann.lecun.com/exdb/mnist/`. [Online].

A Running the Code

1. We have provided a script `hw5.sh` that compiles the code and runs all tests. You can run it on the cluster using `sbatch hw5.sh`.
2. You can compile the code using `make` or
   `DOUBLE_FLAG=-DUSE_DOUBLE make`
   to create executables that use double-precision floating point numbers.
3. You can add additional tests with different size configurations to make sure your code runs correctly, but we only require `neural_network.cpp` for submission.

B Submission Instructions

1. For all questions that require answers besides source code, put those answers in a separate PDF file and upload this file on Gradescope.

2. Make sure your code compiles on `cme213-login.stanford.edu` and runs. To check your code we will run `make` on `cme213-login.stanford.edu`, and this should produce an executable `main`.

3. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files while debugging, but make sure you test with the default files before submitting. Do not use external libraries, additional header files, etc. that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   `neural_network.cpp`

   Although we will provide implementations of the GPU kernels from Homeworks 2 and 4 in the starter code, you are welcome to use or modify your own implementations.

4. To submit, execute

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw5 <directory with your submission files>
```

5. You can submit up to 10 times before the deadline and each submission will replace the previous one.

The final list of items that we expect is:

1. `neural_network.cpp` in the code submission.

2. A screenshot of the output of the provided tests for each question in your PDF submission on Gradescope.

C Neural Networks on CUDA

In the final project, you will use both CUDA and MPI to implement a parallel training process for a two-layer neural network which can identify digits from handwritten images (a specific case of image classification problem). Neural networks are widely used in machine learning problems, specifically in the domains of image processing, computer vision, and natural language processing. There is a flurry of research projects on deep learning, which uses more advanced variants of the simpler neural network we cover here. Therefore, being able to train neural networks efficiently is important and is the goal of this project.

Data: MNIST

We will be using the MNIST [1] dataset, which consists of 28×28 greyscale images of handwritten digits from 0 to 9. Some examples from this dataset are shown in Figure 1.

The dataset is divided into a *training* set of 60,000 images and a *test* set of 10,000 images. We will use the training set to optimize the parameters of our neural network and we will use the *unseen* test set to measure the performance of the trained network. We denote the i^{th} example in the training set by $(x^{(i)}, y^{(i)})$, where $x^{(i)}$ denotes the image and $y^{(i)}$ denotes the corresponding class label (i.e. the digit shown in the image $x^{(i)}$).

Figure 1: Examples of MNIST digits.

Model: Neural Networks

Neurons

To describe neural networks we begin by describing the simplest neural network, which comprises a single *neuron.*

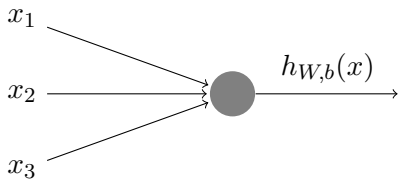

Figure 2: A single neuron.

The neuron illustrated in Figure 2 is a computational unit that takes as input $x = (x_1, x_2, x_3)$ and outputs

$$h_{W,b}(x) = f(Wx + b) = f\Big(\sum_{i=1}^{3} W_i x_i + b\Big),$$

where $f : \mathbb{R} \to \mathbb{R}$ is some non-linear *activation function*, W is the *weight* of the neuron, and b is the *bias* of the neuron. The row vector W and the scalar b are referred to as the *parameters* of the neuron, and the output of the neuron is referred to as its *activation.*

In this project, we let f be the sigmoid function given by

$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}.$$

The derivative of the sigmoid function with respect to its input is

$$\frac{\partial \sigma(x)}{\partial x} = -\frac{1}{(1 + \exp(-x))^2} \frac{\partial \exp(-x)}{\partial x} = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \sigma(x)(1 - \sigma(x));$$

we will use this fact repeatedly in the following sections. Other common activation functions include $f(z) = \tanh(z)$ and the rectified linear unit (ReLU) $f(z) = \max(0, z)$. These are illustrated in Figure 3.

A single neuron can be trained to perform the task of binary classification. Consider the example of cancer detection, where the task is to classify a tumor as benign or malignant. We can provide as input $x =$ (size of tumor, location of tumor, length of time for which the tumor has existed), and if the label is

$$y = \begin{cases} 1 & \text{the tumor is malignant} \\ 0 & \text{the tumor is benign} \end{cases},$$

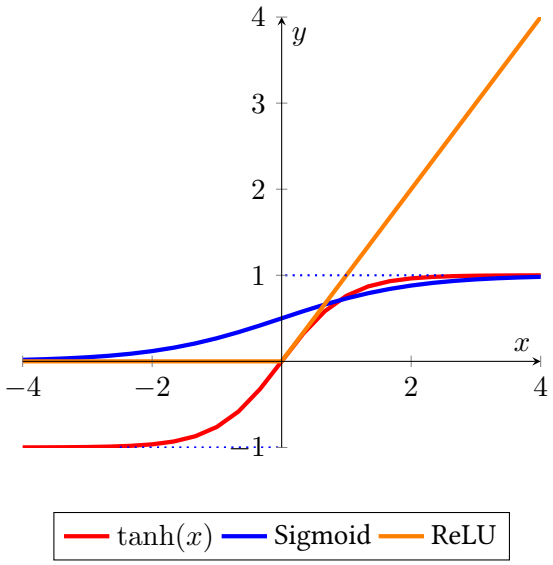

Figure 3: Examples of three activation functions: $\tanh(x)$, $1/(1+\exp(-x))$ (sigmoid), and the rectified linear unit (ReLU).

we can say that the neuron predicts that the tumor is malignant if and only if $f(Wx+b) > 0.5$.

Since the value of $f(Wx+b)$ depends on the sign of $Wx+b$, the neuron effectively partitions the input space $\mathbb{R}^3$ using a 2-dimensional hyperplane. On one side of the hyperplane we have $f(Wx+b) > 0.5$, and on the other side of the hyperplane we have $f(Wx+b) < 0.5$. Through an optimization process referred to as *training*, we want to find values of the parameters W and b such that the hyperplane represented by the neuron is as close as possible to the 'true' hyperplane.

More generally, we want to find values of the parameters W and b such that the network's predictions are 'good' on an unseen test set, since this would imply that our choice of model (here, a neuron with certain values of W and b) is close to the 'true' model corresponding to reality.

It is insufficient to observe good predictions on the training set. Sufficiently complex networks can be trained to make perfect predictions on the training set but they perform much worse on unseen data that they were not trained on, implying that the trained model is not close to the 'true' model.

In this project, we would like to train a neural network to perform multi-class classification rather than binary classification. Instead of simply predicting true or false, we would like the network we train to be able to accurately predict which of 10 different digits is shown in the input image.

Fully connected feedforward neural network

Figure 4 shows a fully connected feedforward neural network with an input layer, one hidden layer, and an output layer. Such a network is referred to as a 'two-layer fully connected feedforward neural network' or a 'two-layer multilayer perceptron (MLP)'.

The input layer is not counted since a neuron in the input layer performs no computation. For example, the first neuron in the input layer takes as input x_1 and outputs x_1. As this output travels along the edge connecting the first neuron in the input layer to the first neuron in the hidden layer, it is multiplied by the weight W_1 of the first neuron in the hidden layer. Once it reaches the first neuron in the hidden layer, it is added to the bias b of the first neuron in the hidden layer, and the result is passed through the sigmoid function to obtain the activation of the first neuron in the hidden layer.

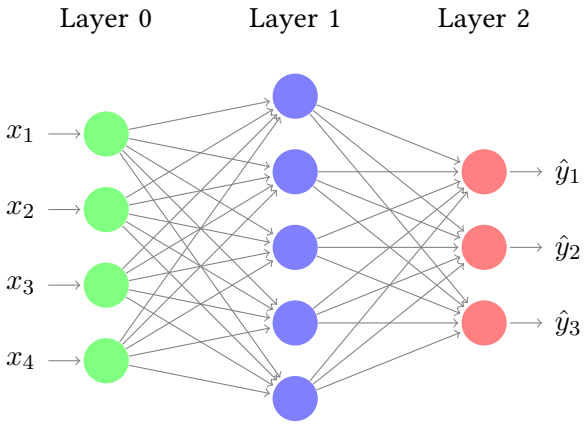

Figure 4: Fully connected feedforward neural network with two layers.

This process must be repeated for each element of the input vector $x \in \mathbb{R}^{d\times 1}$ and for each of H_1 neurons in layer 1. An efficient way to do this is to use matrix-multiplication and compute $a^{(1)} = f(W^{(1)}x + b^{(1)})$, where $W^{(1)} \in \mathbb{R}^{H_1\times d}$, $b^{(1)} \in \mathbb{R}^{H_1\times 1}$, and f is the sigmoid function. The element $W^{(1)}_{ij}$ is the jth weight of the ith neuron in layer 1, the element $b^{(1)}_i$ is the bias of the ith neuron in the layer 1, and the $(W^{(1)}, b^{(1)})$ are referred to as the parameters of layer 1. The vector $a^{(1)} \in \mathbb{R}^{H_1\times 1}$ is referred to as the activation of layer 1 and it consists of the activations of the neurons in layer 1.

The activation function of the output layer is special. Instead of each neuron independently applying the sigmoid function to its input, all neurons in the output layer collectively compute $\text{softmax}(W^{(2)}a^{(1)} + b^{(2)})$. If C denotes the number of possible class labels, we have $C = 10$ since there are 10 possible digits $0, \ldots, 9$. For $1 \leq i \leq 10$, using the softmax activation function allows us to interpret the ith element of the output vector $\hat{y} \in \mathbb{R}^{C\times 1}$ as the neural network's prediction of the probability that the digit in the input image is digit $i - 1$.

In general, if H_i is the number of neurons in layer i, then the parameters of layer i are $W^{(i)} \in \mathbb{R}^{H_i\times H_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{H_i\times 1}$. In Figure 4 we have $d = H_0 = 4$, $H_1 = 5$, $H_2 = 3 \implies W^{(1)} \in \mathbb{R}^{5\times 4}$, $b^{(1)} \in \mathbb{R}^{5\times 1}$, $W^{(2)} \in \mathbb{R}^{3\times 5}$, and $b^{(2)} \in \mathbb{R}^{3\times 1}$. To efficiently process a batch of inputs $x_1, \ldots, x_N \in \mathbb{R}^{d\times 1}$, we can stack them horizontally to obtain a matrix $X = \begin{pmatrix} x_1 & \cdots & x_N \end{pmatrix} \in \mathbb{R}^{d\times N}$, and compute a batch of activations $A^{(1)} = f(W^{(1)}X + B^{(1)}) \in \mathbb{R}^{H_1\times N}$ where $B^{(1)} = \begin{pmatrix} b^{(1)} & \cdots & b^{(1)} \end{pmatrix} \in \mathbb{R}^{H_1\times N}$.

Forward pass

The forward pass is the process of computing the activations of all neurons in the network for an (batch of) input x. For a two-layer MLP, we compute

$$\begin{aligned} z^{(1)} &= W^{(1)}x + b^{(1)} \\ a^{(1)} &= \sigma(z^{(1)}) \\ z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\ \hat{y} = a^{(2)} &= \text{softmax}(z^{(2)}) \end{aligned}$$

The softmax function is defined by:

$$\text{softmax}(z^{(2)})_j \overset{\text{def}}{=} P(\text{label} = j|x) \overset{\text{def}}{=} \frac{\exp(z^{(2)}_j)}{\sum_{i=1}^{C} \exp(z^{(2)}_i)}$$

This equation is saying that the probability that the input has label j (i.e., in our case, the digit j is handwritten in the input image) is given by $\text{softmax}(z^{(2)})_j$. Therefore, our predicted label for the input x is given by:

$$\text{label} = \text{argmax}(\hat{y})$$

This is basically the digit the network believes is written in the input image.

Loss

Recall that our objective is to learn the parameters of the neural network such that it gets the best accuracy on the test set. Let y be the *one-hot* vector denoting the class of the input; $y_c = 1$ if c is the correct label and $y_i = 0$ for all $i \neq c$. We want $P(\text{label} = c|x)$ to be the highest (e.g., close to 1).

Without going into the mathematical details, we will use the following general expression to determine the error of our neural network. This expression turns out to be the most convenient for our purpose:

$$\text{CE}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

CE stands for cross-entropy. Since y is a one-hot vector, this simplifies to

$$\text{CE}(y, \hat{y}) = -\log(\hat{y}_c)$$

We can observe that CE is 0 when we have the optimal answer $\hat{y}_c = 1$. Similarly, CE is maximal ($+\infty$) when $\hat{y}_c$ is 0. This corresponds to a neural network that is "sure" that the digit is *not* c (maximally wrong).

The total cost for N input data points (such that the cross-entropy of the i^{th} training vector is denoted as $\text{CE}^{(i)}$) is:

$$\text{cost} = J(W, b; x, y) = \frac{1}{N} \sum_{i=1}^{N} \text{CE}^{(i)}(y, \hat{y})$$

The above cost measures the error, i.e. our "dissatisfaction", with the output of the network. The more certain the network is about the correct label (high $P(y = c|x)$), the lower our cost will be.

Clearly, we should choose the parameters that minimize this cost. This is an optimization problem, and may be solved using the method of Stochastic Gradient Descent (described below).

Our neural network applies a non-linear function to the input because of the sigmoid and softmax functions. When optimizing the neural network, we often add a penalization term for the magnitude of W in order to control the non-linearity of the network. If we make W smaller, the network becomes 'more linear' since $Wx \approx \sigma(Wx)$ when $Wx \approx 0$. Despite the possibility of making W too small and the fact that there is no rigorous justification for this penalization, it is found to work well in practice. With the penalization term, the cost function becomes

$$\boxed{J(W, b; x, y) = \frac{1}{N} \sum_{i=1}^{N} CE^{(i)}(y, \hat{y}) + \frac{\lambda}{2} \|W\|_2^2} \tag{1}$$

where $\|W\|_2^2$ is the sum of the l^2-norm of all the weights W of the network, and λ is a hyperparameter that needs to be tuned for best performance. In our implementation, only the weights W are penalized, not the biases b.

Backward Pass

The backward pass is the process of using the chain rule to compute $\nabla_p J$, the gradient of the loss function with respect to each parameter of the neural network. This process is also referred to as backpropagation, since gradients are 'propagated backward' through the network using the chain rule.

Let's compute the gradient for the parameters in the last layer (2) of our network:

$$\frac{\partial \text{CE}(y,\hat{y})}{\partial z_k^{(2)}} = -\frac{\partial}{\partial z_k^{(2)}} \log\left[\frac{\exp(z_c^{(2)})}{\sum_{i=0}^{C}\exp(z_i^{(2)})}\right] = -\frac{\partial\left[z_c^{(2)} - \log\left(\sum_{i=0}^{C}\exp(z_i^{(2)})\right)\right]}{\partial z_k^{(2)}}$$

There are two cases here:

1. Case I: $k = c = y_i$, i.e., k is the correct label

$$\frac{\partial \text{CE}(y,\hat{y})}{\partial z_k^{(2)}} = -1 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C}\exp(z_i^{(2)})} = -1 + \hat{y}_k = \hat{y}_k - y_k$$

2. Case II: $k \neq y_i$

$$\frac{\partial \text{CE}(y,\hat{y})}{\partial z_k^{(2)}} = 0 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C}\exp(z_i^{(2)})} = \hat{y}_k - y_k$$

Therefore, the gradient in vector notation simplifies to

$$\frac{\partial \text{CE}(y,\hat{y})}{\partial z^{(2)}} = \hat{y} - y \tag{2}$$

Recall that $z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$, such that $z^{(2)} \in \mathbb{R}^{H_2 \times 1}$, $a^{(1)} \in \mathbb{R}^{H_1 \times 1}$ and $W^{(2)} \in \mathbb{R}^{H_2 \times H_1}$. Therefore,

$$\frac{\partial \text{CE}(y,\hat{y})}{\partial W^{(2)}} = \frac{\partial \text{CE}(y,\hat{y})}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\boxed{\frac{\partial \text{CE}(y,\hat{y})}{\partial W^{(2)}} = (\hat{y} - y)[a^{(1)}]^T} \tag{3}$$

Similarly,

$$\boxed{\frac{\partial \text{CE}(y,\hat{y})}{\partial b^{(2)}} = \hat{y} - y} \tag{4}$$

Going across L_2:

$$\frac{\partial z^{(2)}}{\partial a^{(1)}} = [W^{(2)}]^T$$
$$\frac{\partial \text{CE}(y,\hat{y})}{\partial a^{(1)}} = \frac{\partial \text{CE}(y,\hat{y})}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial a^{(1)}} = [W^{(2)}]^T(\hat{y} - y)$$

Going across the non-linearity of L_1:

$$\begin{aligned}\frac{\partial \text{CE}(y,\hat{y})}{\partial z^{(1)}} &= \frac{\partial \text{CE}(y,\hat{y})}{\partial a^{(1)}}\frac{\partial \sigma(z^{(1)})}{\partial z^{(1)}} \\ &= \frac{\partial \text{CE}(y,\hat{y})}{\partial a^{(1)}} \circ \sigma(z^{(1)}) \circ (1 - \sigma(z^{(1)}))\end{aligned}$$

Note that we have assumed that $\sigma(\cdot)$ works on vectors (matrices) by applying an element-wise sigmoid, and $\circ$ is the element-wise (Hadamard) product.

That brings us to our final gradients:

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(1)}} = \Big(\frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}}\Big) x^T} \tag{5}$$

Similarly,

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial b^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}}} \tag{6}$$

The above equations have been derived for a single training vector, but they extend seamlessly to a matrix of N column vectors. In that case, you need to sum up over all the input images x.

Gradient Descent

Gradient Descent is an iterative algorithm for finding local minima of a function parameterized by some parameters p. The gradient descent update rule is

$$\boxed{p \leftarrow p - \alpha \, \nabla_p J} \tag{7}$$

where α is the learning rate that controls how large the descent step is. $\nabla_p J$ is the gradient of J with respect to the network parameters p.

In practice, we often do not compute $J = \sum_{i=1}^{N} \text{CE}^{(i)}$ since this requires computing $\text{CE}^{(i)}$ for all $i = 1, \ldots, N$. Instead, we divide the input into 'mini-batches' containing M images and process one mini-batch at a time until all images are processed. For each mini-batch we calculate $J_{\text{mb}} = \sum_{i=k}^{k+M} \text{CE}^{(i)}$ (where $x^{(k)}$ is the first image in the mini-batch), and update the network parameters p according to the update rule

$$p \leftarrow p - \alpha \, \nabla_p J_{\text{mb}}. \tag{8}$$

This algorithm is also referred to as Mini-batch Gradient Descent. See below for the pseudo-code, where an 'epoch' refers to a single iteration over all N images and corresponds to $\lceil M/N \rceil$ updates to the parameters p. This approach usually leads to faster convergence than Batch Gradient Descent (or simply Gradient Descent) since we update the network coefficients more than once per epoch.

Algorithm 1 Mini-batch Gradient Descent

```
epoch ← 0
while epoch < MAX_EPOCHS do
    batches ← split(training_samples, M)
    for batch in batches do
        p ← p − step × gradient(batch)
    end for
    epoch ← epoch + 1
end while
```

In the final project, in each iteration of Mini-batch Gradient Descent, the images in a mini-batch should be distributed evenly among MPI nodes, and computation in the forward and backward passes on each MPI node (such as matrix multiplication and softmax) should be accelerated using GPU kernels.