CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 6

Total number of points: 100.

In this homework, you will accelerate training by performing forward and backward passes in parallel on multiple MPI processes, each with an associated GPU. Computation in the forward and backward passes on each MPI process (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.

Problem 1 Data Parallel Training

In data parallel training, each mini-batch is partitioned across processes and the neural network's parameters are replicated on each process. Each process computes the forward and backward passes for its partition of the mini-batch, and gradients computed in the backward pass are averaged across processes before a parameter update is performed on each process. In this problem, you will extend your implementation of `DataParallelNeuralNetwork` from Homework 5 to allow for training in parallel on multiple MPI processes.

Question 1.1

(70 points) Implement the member function `DataParallelNeuralNetwork::train` and complete the TODOs in the member functions `DataParallelNeuralNetwork::loss` and `DataParallelNeuralNetwork::backward`. As with the typedef `nn_real`, you should use the macro `MPI_FP` defined in `common.h` instead of using `MPI_FLOAT` or `MPI_DOUBLE`. You are welcome to use or modify your own implementation of `DataParallelNeuralNetwork` from Homework 5. We will check that the updated parameters of the neural network obtained using your implementation match the updated parameters obtained using the CPU implementation.

Problem 2 Performance Analysis

Measures of Performance

A floating-point operation (FLOP) is an operation that involves floating-point numbers, such as addition or multiplication. A common practice in the optimization of an implementation of an algorithm is to compare the theoretical maximum achievable FLOP/sec (FLOPs per second) with the achieved FLOP/sec. Note the difference between FLOPs and FLOP/sec; FLOPs is a count of floating-point operations and FLOP/sec (commonly shortened to FLOP/s) is a rate or throughput. The term 'FLOPS' in an NVIDIA datasheet corresponds to our usage of FLOP/sec (which is more common).

The theoretical maximum achievable FLOP/sec depends on the hardware. In your implementation of an algorithm to train a data parallel neural network, all floating-point operations are performed on an NVIDIA Quadro RTX 6000 GPU, so the theoretical maximum achievable FLOPs is 16.3 TFLOP/sec (see 'Single-Precision Performance' on the NVIDIA Quadro RTX 6000's datasheet).

The achieved FLOP/sec is calculated by counting the number of floating-point operations performed by an implementation and diving by the time taken to perform these operations. We refer to the ratio of the achieved FLOP/sec to the theoretical maximum achievable FLOP/sec as the effective compute utilization of the implementation.

Roofline Performance Model

Roofline performance charts are a visual tool used to model and understand the performance limits of computer systems, particularly in the context of high-performance computing (HPC).

The standard roofline performance model assumes that computation and memory accesses occur in parallel. Each point on the 'roofline' corresponds to the peak achievable performance in FLOP/sec for a given arithmetic intensity (FLOPs/byte). On the sloping part of the roofline, memory access time is greater than computation time so it corresponds to an application that is 'memory bound'; here performance is limited by the peak memory bandwidth of the system. On the flat part of the roofline, computation time is greater than memory access time so it corresponds to an application that is 'compute bound'; here performance is limited by the peak computational capability of the processor. The kink in the roofline corresponds to an application for which computation and memory access time are equal.

A standard roofline chart is a plot of $y =$ performance in FLOP/sec (this represents the computational performance) against $x =$ arithmetic intensity (AI) in FLOPs/byte (this represents the ratio of floating-point operations to memory accesses). The standard roofline is given by

$$
\begin{aligned}
y = \frac{\text{FLOPs}}{\text{Total time}} &= \frac{\text{FLOPS}}{\max\left(\text{Computation time}, \text{Memory access time}\right)} \\
&= \frac{\text{FLOPs}}{\max\left(\frac{\text{FLOPs}}{\text{Peak computation performance}}, \frac{\text{Bytes of memory accessed}}{\text{Peak memory bandwidth}}\right)} \\
&= \frac{\text{AI}}{\max\left(\frac{\text{AI}}{\text{Peak computation performance}}, \frac{1}{\text{Peak memory bandwidth}}\right)} \\
&= \min\left(\text{Peak computation performance}, \text{AI} \cdot \text{Peak memory bandwidth}\right) \\
&= \min\left(\text{Peak computation performance}, x \cdot \text{Peak memory bandwidth}\right).
\end{aligned}
$$

Thus a roofline plot graphically represents the theoretical peak performance of an application given the computational capability and memory bandwidth of a system. Plotting a point for an application on a roofline chart helps visualize where that application falls in terms of its performance and arithmetic intensity, indicating whether it is more limited by compute capabilities or memory bandwidth.

See `https://docs.nersc.gov/tools/performance/roofline/` for more information about roofline charts. You can also see the results section of the paper `https://arxiv.org/pdf/2009.05257` for a detailed example of roofline performance analysis for deep learning applications.

Question 2.1

(15 points)

- Approximate the number of FLOPs performed in a single forward pass and approximate the number of FLOPs performed in a single backward pass:
  - For each algorithm, the total number of FLOPs performed is dominated by a single matrix-matrix multiplication.
  - Assume that the operations of addition and multiplication each count as a single floating-point operation.
  - Computing the inner product of two n dimensional vectors requires n multiplications and $n-1$ additions, so the number of FLOPs needed to compute the inner product of two n-dimensional vectors can be approximated as $2n$.
- Approximate the number of floating-point operations performed per batch that is processed (forward pass, backward pass and parameter update).

- Using the default hyperparameters in `main_q2.cpp` below, i.e., with $D = 784$, $H = 512$, and $B = 3200$, what is the effective compute utilization of 1, 2, and 4 GPUs when training with 1, 2, and 4 MPI processes respectively?

```
  hyperparameters hparams_custom =
      hyperparameters{784, 512, 10, 1e-4, 1e-4, 38400, 1, 3200};
```

with

```
class hyperparameters {
public:
  int input_size; // D
  int hidden_size; // H
  int num_classes; // C
  nn_real reg;
  nn_real learning_rate;
  int N;
  int num_epochs;
  int batch_size; // B
  ...
};
```

- Comment on your observations. What happens when you increase the number of processes or change the batch size B? Experiment with different values.

Question 2.2

(15 points) Suppose that we train with 38,400 images for one epoch. Measure the total time taken to train when H increases from 512 to 1024 to 2048, and measure the total time taken to train when `num_procs` increases from 1 to 2 to 4 while B/`num_procs` remains constant. You can do this by modifying the test `gtestTrain.custom` in `main_q2.cpp`. For each combination of hyperparameters, plot a point on a roofline chart:

- Calculate the number of FLOPs performed per batch using your approximation from Question 2.1.
- Approximate the number of bytes moved per batch that is processed, considering only the number of bytes moved via `cudaMemcpy` between the CPU and GPU. We only consider these memory accesses because the bandwidth of the PCIe connection between the CPU and GPU is only about 13 GB/sec, which is much lower than the bandwidth at which other bytes in the program are moved (kernels can read from and write to GPU memory at a speed of up to 672 GB/sec on the Quadro RTX 6000). The PCIe connection is typically the bottleneck in a GPU-accelerated program, and we focus on it here to visually compare the speed of computing on the GPU with the time taken to move data from the CPU to the GPU.
- Two implicit `cudaMemcpy`'s are performed when using `MPI_Allreduce` with `gpu_mem_pool` to average gradients across MPI processes. Make sure to count the bytes transferred between the CPU and the GPU during this operation.
- The standard roofline performance model assumes that computation and memory accesses occur in parallel. Since we are only considering `cudaMemcpy` operations which do not occur in parallel with computation, we need to modify the standard roofline performance model. Show that

$$\frac{\text{FLOPs}}{\text{Total time}} = \frac{\text{AI}}{\frac{\text{AI}}{\text{Peak GPU computation performance}} + \frac{1}{\text{Peak PCIe bandwidth}}},$$

and plot the corresponding roofline. Comment on your observations. Where do the markers for each case fall on the roofline plot? What about their relative location? Do your measurements agree with the theoretical model? Are the trends correct? Explain.

References

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

A Running the Code

1. We have provided a script `hw6.sh` that compiles the code and runs all tests for a specific number of MPI processes. You can run this script on the cluster using `sbatch hw6.sh`. To run all tests with a different number of MPI processes, comment out a different `mpirun` command in `hw6.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 an implementation of `DataParallelNeuralNetwork` from Homework 5 in the starter code, you are welcome to use or modify your own implementation.

4. To submit, execute

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw6 <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.

3. A written answer to Problem 2 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.

Figure 1: Examples of MNIST digits.

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 $x^{(i)}$).

Model: Neural Networks

Neurons

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

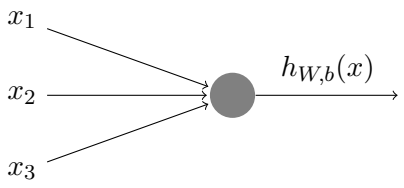

Figure 2: A single neuron.

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.

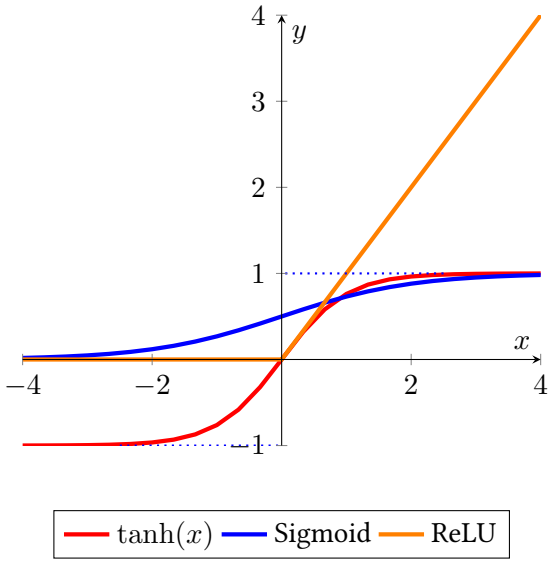

Figure 3: Examples of three activation functions: $\tanh(x)$, $1/(1 + \exp(-x))$ (sigmoid), and the rectified linear unit (ReLU).

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

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

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

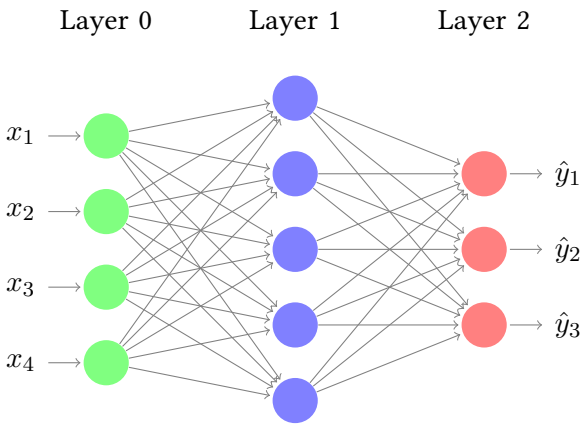

Figure 4: Fully connected feedforward neural network with two layers.

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.

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:

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

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.