CS4787/CS5777 Problem Set 6

Anissa Dallmann, Kaitlyn Chen, Allison Hsu

TOTAL POINTS

28 / 31

QUESTION 1

1 1a **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts iii Incorrect

- 6 pts Incorrect

QUESTION 2

2 1b **3 / 4**

- 0 pts Correct

- 2 pts Not in-place.

- 4 pts Missing / Incorrect

✓ **- 1 pts** *Minor mistake / lack of explanation*

QUESTION 3

3 1c **1 / 3**

- 0 pts Correct

- 1 pts Arithmetic Error

✓ **- 2 pts** *Understanding error*

- 3 pts Incorrect

QUESTION 4

4 2a **3 / 3**

✓ **- 0 pts** *Correct*

QUESTION 5

5 2b **3 / 3**

✓ **- 0 pts** *Correct*

- 0.1 pts Either give bfloat ~= 32 *and* fred ~= 64 or a strict ranking, not one and the other

- 1 pts minor mistake in ranking

- 0.1 pts typo

QUESTION 6

6 2c **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Definitely not 64 bit floats

QUESTION 7

7 3a **4 / 4**

✓ **- 0 pts** *Correct*

QUESTION 8

8 3b **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect final answer

Correct: $$O(dMT)$$

- 0.25 pts Minor errors in reasoning or computation

QUESTION 9

9 3c **2 / 2**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect answer

Correct: It does not change

Kaitlyn Chen (kjc256), Anissa Dallmann (ad792), Allison Hsu (alh293)

1. Memory

   (a) i. Recall the matrix $W \in \mathbb{R}^{c \times d}$. On the MNIST dataset, W contains $c \times d = 10 \times 28 \times 28 = 7840$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$W = 7840 \times 32 \text{ bits} = 250880 \text{ bits} = 31360 \text{ B} = 30.625 \text{ KiB}$$

   Consequently, it will take 31,360 B of memory to store a single copy of the parameter vector W. The smallest cache in which this data fits is the L1 cache.

   ii. A single training example is of dimensions $x_i \in \mathbb{R}^d$, which contains $28 \times 28 = 784$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$x_i = 784 \times 32 \text{ bits} = 25088 \text{ bits} = 3136 \text{ B} = 3.0625 \text{ KiB}$$

   Consequently, it will take 3,136 B of memory to store a single training examples x_i. The smallest cache in which this data fits is the L1 cache.

   iii. The entire training set of $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ contains $n(c + d) = 60000(10 + 28 \times 28) = 47640000$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$47640000 \times 32 \text{ bits} = 1524480000 \text{ bits} = 190560000 \text{ B} = 181.732 \text{ MiB}$$

   Consequently, it will take 190,560,000 B of memory to store the entire training set. This data will not fit in any of the aforementioned caches.

   (b) $mult\_res \leftarrow W_t x_{i_t}$
   $softmax\_res \leftarrow softmax(mult\_res)$
   $diff \leftarrow softmax\_res - y_{i_t}$
   $tmp1 \leftarrow diff \mathbin{@} x_{i_t}^T$
   $tmp2 \leftarrow \alpha_t \cdot tmp1$
   $W_t \leftarrow W_t - tmp2$

   (c) The sizes for each of the variables are as follows:

   - W_t: c x d = 10 x 28 x 28 = 7840
   - $mult\_res$: c = 10
   - $softmax\_res$: c = 10
   - $diff$: c = 10
   - $tmp1$: c x d = 10 x 28 x 28 = 7840
   - $tmp2$: c x d = 10 x 28 x 28 = 7840

   which is a total of 23550 floating point numbers and 23550 x 32 bits = 753600 bits = 91.99 KiB = 0.0898 MiB. Thus, the smallest cache these fit into is the L2 cache.

2. Low-Precision Arithmetic

1 1a **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts iii Incorrect

- 6 pts Incorrect

Kaitlyn Chen (kjc256), Anissa Dallmann (ad792), Allison Hsu (alh293)

1. Memory

   (a) i. Recall the matrix $W \in \mathbb{R}^{c \times d}$. On the MNIST dataset, W contains $c \times d = 10 \times 28 \times 28 = 7840$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$W = 7840 \times 32 \text{ bits} = 250880 \text{ bits} = 31360 \text{ B} = 30.625 \text{ KiB}$$

   Consequently, it will take 31,360 B of memory to store a single copy of the parameter vector W. The smallest cache in which this data fits is the L1 cache.

   ii. A single training example is of dimensions $x_i \in \mathbb{R}^d$, which contains $28 \times 28 = 784$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$x_i = 784 \times 32 \text{ bits} = 25088 \text{ bits} = 3136 \text{ B} = 3.0625 \text{ KiB}$$

   Consequently, it will take 3,136 B of memory to store a single training examples x_i. The smallest cache in which this data fits is the L1 cache.

   iii. The entire training set of $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ contains $n(c + d) = 60000(10 + 28 \times 28) = 47640000$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$47640000 \times 32 \text{ bits} = 1524480000 \text{ bits} = 190560000 \text{ B} = 181.732 \text{ MiB}$$

   Consequently, it will take 190,560,000 B of memory to store the entire training set. This data will not fit in any of the aforementioned caches.

   (b) $mult\_res \leftarrow W_t x_{i_t}$
   $softmax\_res \leftarrow softmax(mult\_res)$
   $diff \leftarrow softmax\_res - y_{i_t}$
   $tmp1 \leftarrow diff \ @ \ x_{i_t}^T$
   $tmp2 \leftarrow \alpha_t \cdot tmp1$
   $W_t \leftarrow W_t - tmp2$

   (c) The sizes for each of the variables are as follows:

   - W_t: c x d = 10 x 28 x 28 = 7840
   - $mult\_res$: c = 10
   - $softmax\_res$: c = 10
   - $diff$: c = 10
   - $tmp1$: c x d = 10 x 28 x 28 = 7840
   - $tmp2$: c x d = 10 x 28 x 28 = 7840

   which is a total of 23550 floating point numbers and 23550 x 32 bits = 753600 bits = 91.99 KiB = 0.0898 MiB. Thus, the smallest cache these fit into is the L2 cache.

2. Low-Precision Arithmetic

2 1b **3 / 4**

- **- 0 pts** Correct
- **- 2 pts** Not in-place.
- **- 4 pts** Missing / Incorrect

✓ **- 1 pts** *Minor mistake / lack of explanation*

Principles of Large-Scale Machine Learning	Homework 6
Kaitlyn Chen (kjc256), Anissa Dallmann (ad792), Allison Hsu (alh293)	

1. Memory

   (a) i. Recall the matrix $W \in \mathbb{R}^{c\times d}$. On the MNIST dataset, W contains $c \times d = 10 \times 28 \times 28 = 7840$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$W = 7840 \times 32 \text{ bits} = 250880 \text{ bits} = 31360 \text{ B} = 30.625 \text{ KiB}$$

   Consequently, it will take 31,360 B of memory to store a single copy of the parameter vector W. The smallest cache in which this data fits is the L1 cache.

   ii. A single training example is of dimensions $x_i \in \mathbb{R}^d$, which contains $28 \times 28 = 784$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$x_i = 784 \times 32 \text{ bits} = 25088 \text{ bits} = 3136 \text{ B} = 3.0625 \text{ KiB}$$

   Consequently, it will take 3,136 B of memory to store a single training examples x_i. The smallest cache in which this data fits is the L1 cache.

   iii. The entire training set of $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ contains $n(c + d) = 60000(10 + 28 \times 28) = 47640000$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$47640000 \times 32 \text{ bits} = 1524480000 \text{ bits} = 190560000 \text{ B} = 181.732 \text{ MiB}$$

   Consequently, it will take 190,560,000 B of memory to store the entire training set. This data will not fit in any of the aforementioned caches.

   (b) $mult\_res \leftarrow W_t x_{i_t}$
   $softmax\_res \leftarrow softmax(mult\_res)$
   $diff \leftarrow softmax\_res - y_{i_t}$
   $tmp1 \leftarrow diff \mathbin{@} x_{i_t}^T$
   $tmp2 \leftarrow \alpha_t \cdot tmp1$
   $W_t \leftarrow W_t - tmp2$

   (c) The sizes for each of the variables are as follows:

   - W_t: c x d = 10 x 28 x 28 = 7840
   - $mult\_res$: c = 10
   - $softmax\_res$: c = 10
   - $diff$: c = 10
   - $tmp1$: c x d = 10 x 28 x 28 = 7840
   - $tmp2$: c x d = 10 x 28 x 28 = 7840

   which is a total of 23550 floating point numbers and 23550 x 32 bits = 753600 bits = 91.99 KiB = 0.0898 MiB. Thus, the smallest cache these fit into is the L2 cache.

2. Low-Precision Arithmetic

3 1c 1 / 3

- **- 0 pts** Correct
- **- 1 pts** Arithmetic Error
- ✓ **- 2 pts** *Understanding error*
- **- 3 pts** Incorrect

Principles of Large-Scale Machine Learning	Homework 6
Kaitlyn Chen (kjc256), Anissa Dallmann (ad792), Allison Hsu (alh293)	

1. Memory

   (a) i. Recall the matrix $W \in \mathbb{R}^{c\times d}$. On the MNIST dataset, W contains $c \times d = 10 \times 28 \times 28 = 7840$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$W = 7840 \times 32 \text{ bits} = 250880 \text{ bits} = 31360 \text{ B} = 30.625 \text{ KiB}$$

   Consequently, it will take 31,360 B of memory to store a single copy of the parameter vector W. The smallest cache in which this data fits is the L1 cache.

   ii. A single training example is of dimensions $x_i \in \mathbb{R}^d$, which contains $28 \times 28 = 784$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$x_i = 784 \times 32 \text{ bits} = 25088 \text{ bits} = 3136 \text{ B} = 3.0625 \text{ KiB}$$

   Consequently, it will take 3,136 B of memory to store a single training examples x_i. The smallest cache in which this data fits is the L1 cache.

   iii. The entire training set of $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ contains $n(c+d) = 60000(10 + 28 \times 28) = 47640000$ single-precision floating point numbers. Since each single-precision floating point number consists of 32 bits of memory, the storage cost is

   $$47640000 \times 32 \text{ bits} = 1524480000 \text{ bits} = 190560000 \text{ B} = 181.732 \text{ MiB}$$

   Consequently, it will take 190,560,000 B of memory to store the entire training set. This data will not fit in any of the aforementioned caches.

   (b) $mult\_res \leftarrow W_t x_{i_t}$
   $softmax\_res \leftarrow softmax(mult\_res)$
   $diff \leftarrow softmax\_res - y_{i_t}$
   $tmp1 \leftarrow diff \ @ \ x_{i_t}^T$
   $tmp2 \leftarrow \alpha_t \cdot tmp1$
   $W_t \leftarrow W_t - tmp2$

   (c) The sizes for each of the variables are as follows:

   - W_t: c x d = 10 x 28 x 28 = 7840
   - $mult\_res$: c = 10
   - $softmax\_res$: c = 10
   - $diff$: c = 10
   - $tmp1$: c x d = 10 x 28 x 28 = 7840
   - $tmp2$: c x d = 10 x 28 x 28 = 7840

   which is a total of 23550 floating point numbers and 23550 x 32 bits = 753600 bits = 91.99 KiB = 0.0898 MiB. Thus, the smallest cache these fit into is the L2 cache.

2. Low-Precision Arithmetic

(a) In ascending order of machine epsilon, we have IEEE standard 64-bit floats, IEEE standard 32-bit floats, IEEE standard 16-bit floats, bfloat, and FredFloat. IEEE standard 64-bit floats have the smallest machine epsilon, and FredFloats have the largest machine epsilon.

It is clear that the number of bits used to store an infinite-precision real number as a floating-point number is directly proportional with the numerical precision of that floating-point format. Consequently, the IEEE standard 64-bit floats have a smaller machine epsilon than IEEE standard 32-bit floats, which then subsequently has a smaller machine epsilon than the 16-bit floats. Among the formats of 16-bit floats (IEEE standard, bfloat, and FredFloat), the relative numerical error is then determined by the number of bits dedicated to the exponent and mantissa. IEEE standard 16-bit floats have a 5/10 exponent-mantissa split, bfloats have an 8/7 exponent-mantissa split, and FredFloats have an 11/4 exponent-mantissa split. The larger the number of exponent bits, the larger the range of numbers that can be represented by the floating-point format. This comes at the expense of numerical precision, so FredFloats must have the largest machine epsilon.

(b) In ascending order of numerical range, we have IEEE standard 16-bit floats, bfloats, IEEE standard 32-bit floats, FredFloats, and IEEE standard 64-bit floats.

The numerical range that can be expressed by a floating-point format is directly proportional to the number of exponent bits. IEEE standard 16-bit floats have 5 exponent bits, bfloats have 8 exponent bits, IEEE standard 32-bit floats have 8 exponent bits, FredFloats have 11 exponent bits, and IEEE standard 64-bit floats have 11 exponent bits. Among floating-point formats that have the same number of exponent bits, the extent of the numerical range is then decided by the number of mantissa bits. Since a general real number is represented as

$$\text{represented number } = (-1)^{\text{sign}} \cdot 2^{\text{exponent } -127} \cdot 1.b_{\text{mantissa bits}} \cdots b_0$$

having more mantissa bits increases the expressible numerical range slightly.

(c) FredFloats would result in the greatest quantization error for an ML application. Since quantization error is the difference between the infinite-precision real number and the nearest representable number in that floating-point format, this error is essentially determined by the number of mantissa bits in a low-precision computing setting. FredFloats have only 4 mantissa bits, so there is a lot of error due to rounding to this less precise format.

3. Distributed Machine Learning

(a) Distributed Adam optimizer

for $t = 1$ to T **do**
  for $m = 1$ to M run in parallel on machine m **do**
    Load w_0 from algorithm inputs
    Select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    Compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    All-reduce across all workers to compute $g_t \leftarrow \frac{1}{M} \sum_{m=1}^{M} g_{m,t}$
  end for
  for $j = 1$ to d **do**
    Accumulate first moment estimate $s_{t_j} \leftarrow \rho_1 s_{t_j} + (1 - \rho_1) g_{t_j}$

4 2a 3 / 3

✓ **- 0 pts** *Correct*

(a) In ascending order of machine epsilon, we have IEEE standard 64-bit floats, IEEE standard 32-bit floats, IEEE standard 16-bit floats, bfloat, and FredFloat. IEEE standard 64-bit floats have the smallest machine epsilon, and FredFloats have the largest machine epsilon.

It is clear that the number of bits used to store an infinite-precision real number as a floating-point number is directly proportional with the numerical precision of that floating-point format. Consequently, the IEEE standard 64-bit floats have a smaller machine epsilon than IEEE standard 32-bit floats, which then subsequently has a smaller machine epsilon than the 16-bit floats. Among the formats of 16-bit floats (IEEE standard, bfloat, and FredFloat), the relative numerical error is then determined by the number of bits dedicated to the exponent and mantissa. IEEE standard 16-bit floats have a 5/10 exponent-mantissa split, bfloats have an 8/7 exponent-mantissa split, and FredFloats have an 11/4 exponent-mantissa split. The larger the number of exponent bits, the larger the range of numbers that can be represented by the floating-point format. This comes at the expense of numerical precision, so FredFloats must have the largest machine epsilon.

(b) In ascending order of numerical range, we have IEEE standard 16-bit floats, bfloats, IEEE standard 32-bit floats, FredFloats, and IEEE standard 64-bit floats.

The numerical range that can be expressed by a floating-point format is directly proportional to the number of exponent bits. IEEE standard 16-bit floats have 5 exponent bits, bfloats have 8 exponent bits, IEEE standard 32-bit floats have 8 exponent bits, FredFloats have 11 exponent bits, and IEEE standard 64-bit floats have 11 exponent bits. Among floating-point formats that have the same number of exponent bits, the extent of the numerical range is then decided by the number of mantissa bits. Since a general real number is represented as

$$\text{represented number } = (-1)^{\text{sign}} \cdot 2^{\text{exponent } -127} \cdot 1.b_{\text{mantissa bits}} \cdots b_0$$

having more mantissa bits increases the expressible numerical range slightly.

(c) FredFloats would result in the greatest quantization error for an ML application. Since quantization error is the difference between the infinite-precision real number and the nearest representable number in that floating-point format, this error is essentially determined by the number of mantissa bits in a low-precision computing setting. FredFloats have only 4 mantissa bits, so there is a lot of error due to rounding to this less precise format.

3. Distributed Machine Learning

(a) Distributed Adam optimizer

for $t = 1$ to T **do**
  for $m = 1$ to M run in parallel on machine m **do**
    Load w_0 from algorithm inputs
    Select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    Compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    All-reduce across all workers to compute $g_t \leftarrow \frac{1}{M} \sum_{m=1}^{M} g_{m,t}$
  end for
  for $j = 1$ to d **do**
    Accumulate first moment estimate $s_{t_j} \leftarrow \rho_1 s_{t_j} + (1 - \rho_1) g_{t_j}$

5 2b **3 / 3**

✓ **- 0 pts** *Correct*

- 0.1 pts Either give bfloat ~= 32 *and* fred ~= 64 or a strict ranking, not one and the other

- 1 pts minor mistake in ranking

- 0.1 pts typo

(a) In ascending order of machine epsilon, we have IEEE standard 64-bit floats, IEEE standard 32-bit floats, IEEE standard 16-bit floats, bfloat, and FredFloat. IEEE standard 64-bit floats have the smallest machine epsilon, and FredFloats have the largest machine epsilon.

It is clear that the number of bits used to store an infinite-precision real number as a floating-point number is directly proportional with the numerical precision of that floating-point format. Consequently, the IEEE standard 64-bit floats have a smaller machine epsilon than IEEE standard 32-bit floats, which then subsequently has a smaller machine epsilon than the 16-bit floats. Among the formats of 16-bit floats (IEEE standard, bfloat, and FredFloat), the relative numerical error is then determined by the number of bits dedicated to the exponent and mantissa. IEEE standard 16-bit floats have a 5/10 exponent-mantissa split, bfloats have an 8/7 exponent-mantissa split, and FredFloats have an 11/4 exponent-mantissa split. The larger the number of exponent bits, the larger the range of numbers that can be represented by the floating-point format. This comes at the expense of numerical precision, so FredFloats must have the largest machine epsilon.

(b) In ascending order of numerical range, we have IEEE standard 16-bit floats, bfloats, IEEE standard 32-bit floats, FredFloats, and IEEE standard 64-bit floats.

The numerical range that can be expressed by a floating-point format is directly proportional to the number of exponent bits. IEEE standard 16-bit floats have 5 exponent bits, bfloats have 8 exponent bits, IEEE standard 32-bit floats have 8 exponent bits, FredFloats have 11 exponent bits, and IEEE standard 64-bit floats have 11 exponent bits. Among floating-point formats that have the same number of exponent bits, the extent of the numerical range is then decided by the number of mantissa bits. Since a general real number is represented as

$$\text{represented number } = (-1)^{\text{sign}} \cdot 2^{\text{exponent } -127} \cdot 1.b_{\text{mantissa bits}} \cdots b_0$$

having more mantissa bits increases the expressible numerical range slightly.

(c) FredFloats would result in the greatest quantization error for an ML application. Since quantization error is the difference between the infinite-precision real number and the nearest representable number in that floating-point format, this error is essentially determined by the number of mantissa bits in a low-precision computing setting. FredFloats have only 4 mantissa bits, so there is a lot of error due to rounding to this less precise format.

3. Distributed Machine Learning

(a) Distributed Adam optimizer

for $t = 1$ to T **do**
  for $m = 1$ to M run in parallel on machine m **do**
    Load w_0 from algorithm inputs
    Select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    Compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    All-reduce across all workers to compute $g_t \leftarrow \frac{1}{M} \sum_{m=1}^{M} g_{m,t}$
  end for
  for $j = 1$ to d **do**
    Accumulate first moment estimate $s_{t_j} \leftarrow \rho_1 s_{t_j} + (1 - \rho_1) g_{t_j}$

6 2c **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Definitely not 64 bit floats

(a) In ascending order of machine epsilon, we have IEEE standard 64-bit floats, IEEE standard 32-bit floats, IEEE standard 16-bit floats, bfloat, and FredFloat. IEEE standard 64-bit floats have the smallest machine epsilon, and FredFloats have the largest machine epsilon.

It is clear that the number of bits used to store an infinite-precision real number as a floating-point number is directly proportional with the numerical precision of that floating-point format. Consequently, the IEEE standard 64-bit floats have a smaller machine epsilon than IEEE standard 32-bit floats, which then subsequently has a smaller machine epsilon than the 16-bit floats. Among the formats of 16-bit floats (IEEE standard, bfloat, and FredFloat), the relative numerical error is then determined by the number of bits dedicated to the exponent and mantissa. IEEE standard 16-bit floats have a 5/10 exponent-mantissa split, bfloats have an 8/7 exponent-mantissa split, and FredFloats have an 11/4 exponent-mantissa split. The larger the number of exponent bits, the larger the range of numbers that can be represented by the floating-point format. This comes at the expense of numerical precision, so FredFloats must have the largest machine epsilon.

(b) In ascending order of numerical range, we have IEEE standard 16-bit floats, bfloats, IEEE standard 32-bit floats, FredFloats, and IEEE standard 64-bit floats.

The numerical range that can be expressed by a floating-point format is directly proportional to the number of exponent bits. IEEE standard 16-bit floats have 5 exponent bits, bfloats have 8 exponent bits, IEEE standard 32-bit floats have 8 exponent bits, FredFloats have 11 exponent bits, and IEEE standard 64-bit floats have 11 exponent bits. Among floating-point formats that have the same number of exponent bits, the extent of the numerical range is then decided by the number of mantissa bits. Since a general real number is represented as

$$\text{represented number } = (-1)^{\text{sign}} \cdot 2^{\text{exponent } -127} \cdot 1.b_{\text{mantissa bits}} \cdots b_0$$

having more mantissa bits increases the expressible numerical range slightly.

(c) FredFloats would result in the greatest quantization error for an ML application. Since quantization error is the difference between the infinite-precision real number and the nearest representable number in that floating-point format, this error is essentially determined by the number of mantissa bits in a low-precision computing setting. FredFloats have only 4 mantissa bits, so there is a lot of error due to rounding to this less precise format.

3. Distributed Machine Learning

(a) Distributed Adam optimizer

for $t = 1$ to T **do**
  for $m = 1$ to M run in parallel on machine m **do**
    Load w_0 from algorithm inputs
    Select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    Compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    All-reduce across all workers to compute $g_t \leftarrow \frac{1}{M} \sum_{m=1}^{M} g_{m,t}$
  end for
  for $j = 1$ to d **do**
    Accumulate first moment estimate $s_{t_j} \leftarrow \rho_1 s_{t_j} + (1 - \rho_1) g_{t_j}$

Accumulate second moment estimate $r_{t_j} \leftarrow \rho_2 r_{t_j} + (1-\rho_2) g_{t_j}^2$

end for

Correct first moment bias $\hat{s}_t \leftarrow \frac{s_t}{1-\rho_1^t}$

Correct second moment bias $\hat{r}_t \leftarrow \frac{r_t}{1-\rho_2^t}$

Update model $w_t \leftarrow w_{t-1} - \frac{\alpha}{\sqrt{r_t}} \cdot s_t$

end for

Return w_T from any machine

(b) Suppose that we are storing the parameters as single-precision floating point numbers, i.e. each parameter requires 4 bytes to store. According to the pseudocode in part (a), each machine m will send $g_{m,t}$ to the reducer on each iteration t (except the machine that is the one computing the full gradient). Since $g_{m,t} \in \mathbb{R}^d$, each machine then sends $d \times 4$ B $= 4d$ B to the reducer. Consequently, $4d(M-1)T$ B are sent to the reducer over T total iterations of Adam. Then, the reducer calculates the sum of g_t sends that vector back to each machine (except the one that calculated the gradient). This is also $d \times 4B = 4d$ B or $4d(M-1)T$ B over T iterations. Thus, the total bytes of data sent over the network are $8d(M-1)T$ B.

(c) If SGD or AdaGrad were used, the bytes of data sent over the network would not change. For each of those algorithms, we would compute the gradients in batches and all-reduce the results, just as in Adam, so the same amount of data will be sent over the network.

7 3a **4 / 4**

✓ **- 0 pts** *Correct*

Accumulate second moment estimate $r_{t_j} \leftarrow \rho_2 r_{t_j} + (1-\rho_2) g_{t_j}^2$

end for

Correct first moment bias $\hat{s}_t \leftarrow \frac{s_t}{1-\rho_1^t}$

Correct second moment bias $\hat{r}_t \leftarrow \frac{r_t}{1-\rho_2^t}$

Update model $w_t \leftarrow w_{t-1} - \frac{\alpha}{\sqrt{r_t}} \cdot s_t$

end for

Return w_T from any machine

(b) Suppose that we are storing the parameters as single-precision floating point numbers, i.e. each parameter requires 4 bytes to store. According to the pseudocode in part (a), each machine m will send $g_{m,t}$ to the reducer on each iteration t (except the machine that is the one computing the full gradient). Since $g_{m,t} \in \mathbb{R}^d$, each machine then sends $d \times 4$ B $= 4d$ B to the reducer. Consequently, $4d(M-1)T$ B are sent to the reducer over T total iterations of Adam. Then, the reducer calculates the sum of g_t sends that vector back to each machine (except the one that calculated the gradient). This is also $d \times 4B = 4d$ B or $4d(M-1)T$ B over T iterations. Thus, the total bytes of data sent over the network are $8d(M-1)T$ B.

(c) If SGD or AdaGrad were used, the bytes of data sent over the network would not change. For each of those algorithms, we would compute the gradients in batches and all-reduce the results, just as in Adam, so the same amount of data will be sent over the network.

8 3b 3 / 3

✓ **- 0 pts** *Correct*

- 1 pts Incorrect final answer

Correct: $$O(dMT)$$

- 0.25 pts Minor errors in reasoning or computation

Accumulate second moment estimate $r_{t_j} \leftarrow \rho_2 r_{t_j} + (1-\rho_2) g_{t_j}^2$

end for

Correct first moment bias $\hat{s}_t \leftarrow \frac{s_t}{1-\rho_1^t}$

Correct second moment bias $\hat{r}_t \leftarrow \frac{r_t}{1-\rho_2^t}$

Update model $w_t \leftarrow w_{t-1} - \frac{\alpha}{\sqrt{r_t}} \cdot s_t$

end for

Return w_T from any machine

(b) Suppose that we are storing the parameters as single-precision floating point numbers, i.e. each parameter requires 4 bytes to store. According to the pseudocode in part (a), each machine m will send $g_{m,t}$ to the reducer on each iteration t (except the machine that is the one computing the full gradient). Since $g_{m,t} \in \mathbb{R}^d$, each machine then sends $d \times 4$ B $= 4d$ B to the reducer. Consequently, $4d(M-1)T$ B are sent to the reducer over T total iterations of Adam. Then, the reducer calculates the sum of g_t sends that vector back to each machine (except the one that calculated the gradient). This is also $d \times 4B = 4d$ B or $4d(M-1)T$ B over T iterations. Thus, the total bytes of data sent over the network are $8d(M-1)T$ B.

(c) If SGD or AdaGrad were used, the bytes of data sent over the network would not change. For each of those algorithms, we would compute the gradients in batches and all-reduce the results, just as in Adam, so the same amount of data will be sent over the network.

9 3c **2 / 2**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect answer

Correct: It does not change