INFR10086 Machine Learning (MLG) Semester 2, 2024/5

Coursework

Instructions

- Due date: 10 March, Monday, at 12pm
- The submission is through Gradescope `https://www.gradescope.com/courses/946198`.
- It's best to typeset your answers, but it is fine to submit hand-written answers.
- For Q2, you do not need to submit the source code of the entire file. Copy-paste the snippets of your implementation, and submit 1 PDF document together with the answer in Q1.

Questions

1. In this question, we are going to work out the convergence rate of running gradient descent on the mean-squared error.

   Suppose we have a data set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ for $i = 1, \ldots, n$. Recall that in linear regression, the goal is to find the minimum of

   $$L(w) = \|Xw - y\|_2^2, \tag{1}$$

   where

   $$X = \begin{bmatrix} — & x_1 & — \\ — & x_2 & — \\ & \vdots & \\ — & x_n & — \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{2}$$

   We assume the last dimension of x_i is a constant 1, so we do not need to worry about the bias term.

   (a) Show that the gradient and the Hessian of L are

   $$\nabla L(w) = 2X^\top Xw - 2X^\top y \tag{3}$$

   $$H = 2X^\top X. \tag{4}$$

   [2 marks]

The loss function can be expanded into

$$L(w) = \|Xw - y\|_2^2 = (Xw - y)^\top (Xw - y) = w^\top X^\top Xw - 2y^\top Xw + y^\top y.$$

With the following

$$\frac{\partial w^\top Aw}{\partial w} = 2Aw \qquad \frac{\partial x^\top w}{\partial w} = x$$

we get $\nabla L(w) = 2X^\top Xw - 2X^\top y$ and $H = \nabla_w \nabla_w L(w) = 2X^\top X$.

(b) Recall that the optimal solution is the w^* where $\nabla L(w^*) = 2X^\top Xw^* - 2X^\top y = 0$. Show that

$$\frac{1}{2} Hw^* = X^\top y. \tag{5}$$

[3 marks]

By the result from (a), we have

$$2X^\top Xw^* - 2X^\top y = Hw^* - 2X^\top y = 0.$$

Rearranging the terms, we get $\frac{1}{2} Hw^* = X^\top y$.

(c) To optimize L with gradient descent, we iteratively perform

$$w_t = w_{t-1} - \eta \nabla L(w_{t-1}). \tag{6}$$

Use the results in (a) and (b) and show that

$$w_t = w_{t-1} - \eta H(w_{t-1} - w^*). \tag{7}$$

[5 marks]

Plugging in the result from (a) and (b), we have

$$\nabla L(w_{t-1}) = 2X^\top Xw_{t-1} - 2X^\top y = Hw_{t-1} - Hw^* = H(w_{t-1} - w^*).$$

(d) If we subtract both sides of (7) with w^*, show that

$$w_t - w^* = (I - \eta H)(w_{t-1} - w^*). \tag{8}$$

Now unroll the result in (d) and show that

$$w_t - w^* = (I - \eta H)^t (w_0 - w^*). \tag{9}$$

[5 marks]

$$
\begin{aligned}
w_t - w^* &= w_{t-1} - \eta H(w_{t-1} - w^*) - w^* = w_{t-1} - w^* - \eta H(w_{t-1} - w^*) \\
&= (I - \eta H)(w_{t-1} - w^*) \\
&= (I - \eta H)(I - \eta H)(w_{t-2} - w^*) = (I - \eta H)^2(w_{t-2} - w^*) \\
&\dots \\
&= (I - \eta H)^t(w_0 - w^*)
\end{aligned}
$$

(e) We will need to take a small detour here. Recall that in Optimization 4, we've talked about how solving

$$\max_w \frac{w^\top A w}{w^\top w} \tag{10}$$

is equivalent to solving

$$\max_w w^\top A w \qquad \text{s.t.} \quad \|w\|_2^2 = 1. \tag{11}$$

The solution is to form the Lagrangian

$$F(w) = w^\top A w - \lambda(w^\top w - 1) \tag{12}$$

and find the optimal solution by setting $\nabla F(w) = 0$. The optimal solution satisfies $Aw = \lambda w$, implying that w is an eigenvector of A and λ is its corresponding eigenvalue. Use this result and show that, for any matrix A whose largest eigenvalue is $\lambda_{\max}$,

$$w^\top A w \le \lambda_{\max} \|w\|_2^2, \tag{13}$$

for all w.

[5 marks]

Since

$$\max \frac{w^\top A w}{w^\top w} = \max \frac{\lambda \|w\|_2^2}{\|w\|_2^2} = \max \lambda,$$

we know that

$$\frac{w^\top A w}{w^\top w} \le \lambda_{\max}$$

for any arbitrary w. This gives the desired result $w^\top A w \le \lambda_{\max} \|w\|_2^2$ for all w. Note that this result does not require $\lambda_{\max} > 0$. If $\lambda_{\max} \le 0$, it simply implies that the matrix A is negative semi-definite.

(f) Apply the results in (d) to (e) and show

$$\|w_t - w^*\|_2^2 = (w_0 - w^*)^\top (I - \eta H)^{2t}(w_0 - w^*) \le \nu^{2t} \|w_0 - w^*\|_2^2, \tag{14}$$

where ν is the largest eigenvalue of $I - \eta H$.

[5 marks]

With (d), we have

$$\begin{aligned}\|w_t - w^*\|_2^2 &= \|(I - \eta H)^t (w_0 - w^*)\|_2^2 \\ &= (w_0 - w^*)^\top [(I - \eta H)^t]^\top (I - \eta H)^t (w_0 - w^*) \\ &= (w_0 - w^*)(I - \eta H)^{2t}(w_0 - w^*),\end{aligned}$$

where the last line uses the fact that $I - \eta H$ is symmetric. Now we use the result in (e),

$$\|w_t - w^*\|_2^2 = (w_0 - w^*)(I - \eta H)^{2t}(w_0 - w^*) \leq \nu^{2t}\|w_0 - w^*\|_2^2,$$

where the last line uses the fact that the largest eigenvalue of A^t is the $\lambda_{\max}(A)^t$. The last bit is easy to see, as $A^t w = A^{t-1}\lambda w = \lambda A^{t-1} w = \lambda A^{t-2}\lambda w = \lambda^2 A^{t-2} w = \cdots = \lambda^t w$.

(g) Use the results in (a) and show that

$$L(w_1) = L(w_2) + \nabla L(w_2)^\top (w_1 - w_2) + \frac{1}{2}(w_1 - w_2)^\top H (w_1 - w_2) \tag{15}$$

for all $w_1 \in \mathbb{R}^d$ and $w_2 \in \mathbb{R}^d$.[1]

[5 marks]

With (a), we have

$$\begin{aligned}L(w_1) - L(w_2) &= w_1^\top X^\top X w_1 - w_2^\top X^\top X w_2 - 2y^\top X(w_1 - w_2) \\ &= w_1^\top X^\top X w_1 - 2w_1^\top X^\top X w_2 + w_2^\top X^\top X w_2 + (2X^\top X w_2 - 2X^\top y)^\top (w_1 - w_2) \\ &= \frac{1}{2}(w_1 - w_2)^\top 2X^\top X (w_1 - w_2) + (2X^\top X w_2 - 2X^\top y)^\top (w_1 - w_2) \\ &= \nabla L(w_2)^\top (w_1 - w_2) + \frac{1}{2}(w_1 - w_2)^\top H (w_1 - w_2).\end{aligned}$$

(h) If we choose w_1 to be an arbitrary $w \in \mathbb{R}^d$ and w_2 to be an optimal solution w^* where $\nabla L(w^*) = 0$, show that

$$L(w) - L(w^*) = \frac{1}{2}(w - w^*)^\top H (w - w^*). \tag{16}$$

[5 marks]

[1] Since L is quadratic in w, the right hand side of (15) is exactly the second-order Taylor expansion of L. We did not talk about Taylor expansion in class, so please do not use it to solve this question.

By letting $w_1 = w$ and $w_2 = w^*$, we have

$$\begin{aligned} L(w) &= L(w^*) + \nabla L(w^*)^\top (w - w^*) + \frac{1}{2}(w - w^*)^\top H(w - w^*) \\ &= L(w^*) + \frac{1}{2}(w - w^*)^\top H(w - w^*), \end{aligned}$$

for any w.

(i) Use (h) to get

$$L(w_t) - L(w^*) = \frac{1}{2}(w_t - w^*)^\top H(w_t - w^*). \tag{17}$$

Apply (e) again, plug in (f), and show that

$$L(w_t) - L(w^*) \leq \frac{\lambda_{\max}}{2}\nu^{2t}\|w_0 - w^*\|_2^2, \tag{18}$$

where $\lambda_{\max}$ is the largest eigenvalue of H.

[5 marks]

If we chain the result in (e) and (f), we have

$$\begin{aligned} L(w_t) - L(w^*) &= \frac{1}{2}(w - w^*)^\top H(w - w^*) \\ &\leq \frac{\lambda_{\max}}{2}\|w_t - w^*\|_2^2 \leq \frac{\lambda_{\max}}{2}\nu^{2t}\|w_0 - w^*\|_2^2, \end{aligned}$$

where $\lambda_{\max}$ is the largest eigenvalue of H and ν is the largest eigenvalue of $I - \eta H$.

(j) If we choose the step size to be $\eta = \frac{1}{K}$ for a constant K, then ν becomes the largest eigenvalue of $I - \frac{1}{K}H$, which is $1 - \lambda_{\min}/K$, where $\lambda_{\min}$ is the smallest non-zero eigenvalue of H. We arrive at

$$L(w_t) - L(w^*) \leq \frac{\lambda_{\max}}{2}\left(1 - \frac{\lambda_{\min}}{K}\right)^{2t}\|w_0 - w^*\|_2^2. \tag{19}$$

When we choose $K > \lambda_{\min}$, show that

$$L(w_t) - L(w^*) \leq O(r^t), \tag{20}$$

for some $0 < r < 1$.

[5 marks]

When $K > \lambda_{\min}$, then $0 < 1 - \lambda_{\min}/K < 1$. We have

$$L(w_t) - L(w^*) \leq O(r^t), \tag{21}$$

where $r = (1 - \lambda_{\min}/K)^2$.

(k) Given the result in (j), what type of convergence is running gradient descent on mean-squared error? Sublinear, linear, or quadratic?

[5 marks]

By definition, the convergence rate is linear.

2. In this question, we are going to write a small neural network library.

Download `https://homepages.inf.ed.ac.uk/htang2/mlg2024/coursework/nn.py`. The file `nn.py` provides an initial scaffolding.

A neural network is essentially a sequence of function compositions. To train a neural network with stochastic gradient descent, we need to be able to compute the gradient of arbitrary function compositions. The algorithm is called backpropagation, and the data structure is called computation graphs.

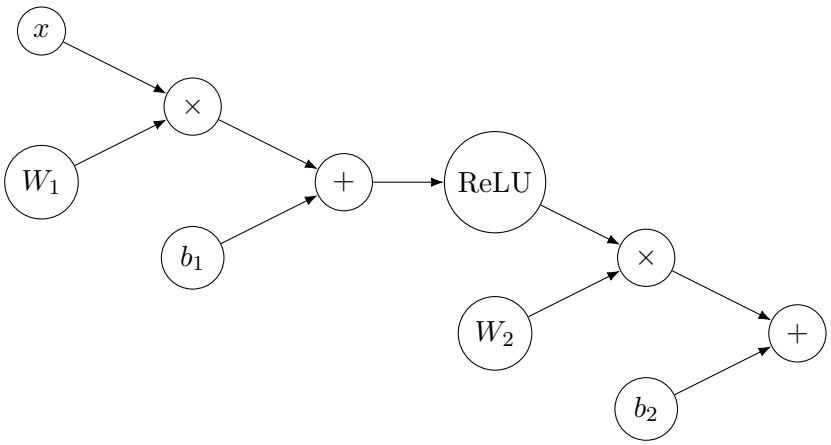
Take a two-layer ReLU network for example, where the network has the form

$$f(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2, \tag{22}$$

and $\text{ReLU}(x) = \max(0, x)$. To compute $f(x)$, we hope to be able to write the following snippet.

```
g = Graph()
output = g.add(g.matmul(g.relu(g.add(g.matmul(x, w1), b1)), w2), b2)
```

The result of running the snippet is the following graph.



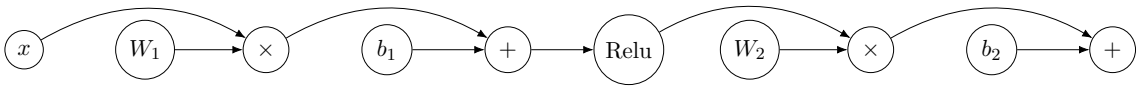
The graph is called a computation graph, where every vertex is either a variable or an operation. When values are given to the variables, we can run the computation graph to get the result of the computation. We hope to write the following snippet that prints the value of the output.

```
g.run_eval(output)
print(g.value[output])
```

To represent a computation graph, the `Graph` class has a set of vertices with integer IDs. Suppose we have a `Graph` instance `g`. A vertex with ID `i` has a single parent vertex `g.parent[i]`, a list of children vertices `g.children[i]`, its associated value `g.value[i]`, its associated gradient `g.grad[i]`, and finally the name of the vertex `g.name[i]`.

(a) The computation needs to be done in a particular order, we need a concept called topological ordering of vertices. Suppose there are n vertices in the graph. A topological ordering of vertices is a sequence $v_1, \ldots, v_n$ in which for every pair v_i and v_j with $i < j$, if there is an edge between the two, then v_i needs to be a child of v_j.

In simple terms, if we line up the vertices in topological order, the edges can only point forward, not backward. For example, below is a topological order of the graph above.



Have a look at `add` and `_new_vertex`. Suppose we have n vertices. If every time an operation is called (e.g., `c = g.add(a, b)`), we create a new vertex (`c` in this case), connect the children (`a` and `b`), and assign a new ID to it. Use induction and show that the IDs of the vertices follow a topological order.

[10 marks]

When there is only 1 vertex, it is trivially in topological order. Suppose the graph has more than 1 vertex and we can list them in topological order. A new vertex having existing vertices as children can be placed at the end, i.e., having the largest ID, while maintaining the toplogical ordering.

(b) Have a look at `run_eval`. Where is topological order used in `run_eval`? What happens if the IDs of the vertices do not follow a topological order?

[5 marks]

The for loop is iterating through the vertices in topological order. It's just that topological order coincides with the IDs, which we prove in (a). If the loop does not iterate vertices in topological order, then `g.value[v]` might not be defined for some `v` when it needs to be used.

(c) Have a look at `eval_add`. The forward computation is straightforward, so let's focus on implementing the backward computation `grad_add`.

Given that `c = g.add(a, b)` which represents $c = a + b$, we can write down the chain rule

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} = \frac{\partial f}{\partial c} \cdot 1 = \frac{\partial f}{\partial c}. \tag{23}$$

Note that `a` is the first child of `c`. We can the write the following line in `grad_add` for an arbitrary vertex `n` (in this case, `c`).

```
self.grad[self.children[n][0]] = self.grad[n]
```

Derive $\frac{\partial f}{\partial b}$ and show the your final implementation of `g.grad_add` by completing the gradient from `n` to the second child.

[10 marks]

> The gradient to the second child is the same of the first, because
>
> $$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = \frac{\partial f}{\partial c} \cdot 1 = \frac{\partial f}{\partial c}.$$
>
> The implementation is as follows.
>
> ```
> self.grad[self.children[n][1]] = self.grad[n]
> ```

(d) Have a look at `eval_matmul`. The forward computation is again straightforward, so let's focus on implementing the backward computation `grad_matmul`.

Given that `y = g.matmul(w, A)` which represents $y = wA$, we can write down the chain rule

$$\frac{\partial f}{\partial w_i} = \sum_j \frac{\partial f}{\partial y_j}\frac{\partial y_j}{\partial w_i} = \sum_j \frac{\partial f}{\partial y_j} a_{ij}. \tag{24}$$

In the matrix form, we have

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial y} A^\top. \tag{25}$$

Note that w here is the first child of y, so we can write the following line to implement the chain rule for an arbitrary vertex `n` (in this case, `y`).

```
self.grad[self.children[n][0]] = self.grad[n] @ self.value[self.children[n][1]].T
```

Derive $\frac{\partial f}{\partial a_{ij}}$ and its matrix form. Show the your final implementation of `grad_matmul` by completing the gradient from `n` to the second child.

[10 marks]

Following the same argument,

$$\frac{\partial f}{\partial a_{ij}} = \sum_k \frac{\partial f}{\partial y_k}\frac{\partial y_k}{\partial a_{ij}} = \sum_k \frac{\partial f}{\partial y_k}\mathbb{1}_{k=j} w_{i\cdot} = \frac{\partial f}{\partial y_j} w_{i\cdot}.$$

In terms of code, we have

```
self.grad[self.children[n][1]]
    = np.outer(self.value[self.children[n][0]], self.grad[n])
```

(e) Have a look at `run_grad`. Where is topological order used in `run_grad`? What happens if the IDs of the vertices do not follow a topological order?

[5 marks]

It's the for loop again, but this time it needs to iterate in reversed topological order. If the loop does not iterate vertices in (reversed) topological order, then `g.grad[v]` might not be defined for some `v` when it needs to be used.

(f) Now that we have implemented backpropagation, we need a way to verify whether the implementation is correct. A simple approach to estimating gradients is called the finite difference method. The idea is that we can simulate derivative by making a tiny change in the function input. For example,

$$\frac{\partial f}{\partial w_i} \approx \frac{f(w + e_i\epsilon) - f(w)}{\epsilon}, \tag{26}$$

where e_i is a vector that has 1 on the i-th coordinate and 0 everywhere else, and ϵ is a small number, say 10^{-3}.

Below is an implementation of the finite difference method to check the gradient to the first child in `grad_matmul`.

```
eps = 1e-3

g1 = Graph()
w1 = g1.tensor(np.array([1, 2, 3]))
a1 = g1.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y1 = g1.matmul(w1, a1)
g1.run_eval(y1)

g2 = Graph()
w2 = g2.tensor(np.array([1 + eps, 2, 3]))
a2 = g2.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y2 = g2.matmul(w2, a2)
g2.run_eval(y2)
```

```
g3 = Graph()
w3 = g3.tensor(np.array([1, 2 + eps, 3]))
a3 = g3.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y3 = g3.matmul(w3, a3)
g3.run_eval(y3)

g4 = Graph()
w4 = g4.tensor(np.array([1, 2, 3 + eps]))
a4 = g4.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y4 = g4.matmul(w4, a4)
g4.run_eval(y4)

g1.run_grad(y1, np.array([1, 0]))

g5 = Graph()
w5 = g5.tensor(np.array([1, 2, 3]))
a5 = g5.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y5 = g5.matmul(w5, a5)

g5.run_grad(y5, np.array([0, 1]))

print((g2.value[y2] - g1.value[y1]) / eps)
print((g3.value[y3] - g1.value[y1]) / eps)
print((g4.value[y4] - g1.value[y1]) / eps)
print(g1.grad[w1])
print(g5.grad[w5])
```

Use the finite difference method to check your implementation of the gradient to the second child in `grad_matmul`. Show your implementation of the finite difference method and the respective output to confirm that the gradient is implemented correctly.

[10 marks]

The following code (albeit verbose) implements the finite difference method to check the gradient.

```
eps = 1e-3

g1 = Graph()
w1 = g1.tensor(np.array([1, 2, 3]))
a1 = g1.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y1 = g1.matmul(w1, a1)

g1.run_eval(y1)
g1.run_grad(y1, np.array([1, 0]))
```

```
g2 = Graph()
w2 = g2.tensor(np.array([1, 2, 3]))
a2 = g2.tensor(np.array([[1, 2], [3, 4], [5, 6]]))
y2 = g2.matmul(w2, a2)

g2.run_eval(y2)
g2.run_grad(y2, np.array([0, 1]))

g3 = Graph()
w3 = g3.tensor(np.array([1, 2, 3]))
a3 = g3.tensor(np.array([[1 + eps, 2], [3, 4], [5, 6]]))
y3 = g3.matmul(w3, a3)
g3.run_eval(y3)

g4 = Graph()
w4 = g4.tensor(np.array([1, 2, 3]))
a4 = g4.tensor(np.array([[1, 2 + eps], [3, 4], [5, 6]]))
y4 = g4.matmul(w4, a4)
g4.run_eval(y4)

g5 = Graph()
w5 = g5.tensor(np.array([1, 2, 3]))
a5 = g5.tensor(np.array([[1, 2], [3 + eps, 4], [5, 6]]))
y5 = g5.matmul(w5, a5)
g5.run_eval(y5)

g6 = Graph()
w6 = g6.tensor(np.array([1, 2, 3]))
a6 = g6.tensor(np.array([[1, 2], [3, 4 + eps], [5, 6]]))
y6 = g6.matmul(w6, a6)
g6.run_eval(y6)

g7 = Graph()
w7 = g7.tensor(np.array([1, 2, 3]))
a7 = g7.tensor(np.array([[1, 2], [3, 4], [5 + eps, 6]]))
y7 = g7.matmul(w7, a7)
g7.run_eval(y7)

g8 = Graph()
w8 = g8.tensor(np.array([1, 2, 3]))
a8 = g8.tensor(np.array([[1, 2], [3, 4], [5, 6 + eps]]))
y8 = g8.matmul(w8, a8)
g8.run_eval(y8)
```

```
print((g3.value[y3] - g1.value[y1]) / eps)
print((g4.value[y4] - g1.value[y1]) / eps)
print((g5.value[y5] - g1.value[y1]) / eps)
print((g6.value[y6] - g1.value[y1]) / eps)
print((g7.value[y7] - g1.value[y1]) / eps)
print((g8.value[y8] - g1.value[y1]) / eps)

print(g1.grad[a1])
print(g2.grad[a2])
```

The output of the finite difference method is

```
[1. 0.]
[0. 1.]
[2. 0.]
[0. 2.]
[3. 0.]
[0. 3.]
```

which is exactly the same as the gradient from backpropagation

```
[[1 0]
 [2 0]
 [3 0]]
[[0 1]
 [0 2]
 [0 3]]
```