HOMEWORK SET 2

CSCI 5980 Think Deep Learning (Spring 2020)

Due 11:59 pm, Apr 18 2020

Instruction Please typeset your homework in LaTeX and submit it as a single PDF file in Canvas. No late submission will be accepted. For each problem, your should acknowledge your collaborators if any. For problems containing multiple subproblems, there are often close logic connections between the subproblems. So always remember to build on previous ones, rather than work from scratch.

Notation We will use small letters (e.g., u) for scalars, small boldface letters (e.g., $\boldsymbol{a}$) for vectors, and capital boldface letters (e.g., $\boldsymbol{A}$) for matrices. For a matrix $\boldsymbol{A}$, $\boldsymbol{a}^i$ (supscripting) means its i-th row as a *row vector*, and $\boldsymbol{a}_j$ (subscripting) means the j-the column as a column vector, and a_{ij} means its (i,j)-th element. $\mathbb{R}$ is the set of real numbers. $\mathbb{R}^n$ is the space of n-dimensional real vectors, and similarly $\mathbb{R}^{m\times n}$ is the space of $m \times n$ real matrices. The dotted equal sign $\doteq$ means defining.

Problem 1 (Nonlinear least-squares and 2nd order methods) Solving linear equation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, or equivalently the linear least-squares $\min_{\boldsymbol{x}} \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2$, is classic. What about quadratic equations? Suppose we have $y_i = \left(\boldsymbol{a}_i^\intercal \boldsymbol{x}\right)^2$ for $i = 1, \dots, m$. Given y_i's and $\boldsymbol{a}_i$'s, we want to recover $\boldsymbol{x} \in \mathbb{R}^n$. It turns out all of sudden the problem becomes NP-hard in general.

Fortunately, the case $\boldsymbol{A}$ is random is qualitatively easier. Let's explore this a bit. *To make sure we can reproduce your results, please fix a random seed in Numpy by setting `numpy.random.seed` to a number you like*. Let's generate the data as follows: fix $n = 20$ and $m = 100$. Pick an $\boldsymbol{x} \neq \boldsymbol{0}$ you like (say random), and generate $\boldsymbol{a}_i$'s as iid standard normal. Compute $\boldsymbol{y} = [y_1, \dots, y_m]$.

(a) Consider a nonlinear least-squares formulation of the problem

$$\min_{\boldsymbol{x}} \ \frac{1}{4}\sum_{i=1}^{m}\left(y_i - \left(\boldsymbol{a}_i^\intercal \boldsymbol{x}\right)^2\right)^2. \tag{1}$$

Derive the Hessian of the objective (hint: applying Taylor expansion method with 2nd order expansion might be easier than other means) and implement the Newton's method. Does it solve your problem? The global minimum should be zero. (1/12)

(b) We did not cover it in the class, but Gauss-Newton method is a specialized method for solving nonlinear least-squares problems and can be considered as an approximate Newton's method. You can learn the method from https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm, or whatever sources you prefer. Implement the method and check if you find the global minimum. (1/12)

Problem 2 (Automatic differentiation—scalar version) Consider the the following three-variable function

$$f\left(x_1, x_2, x_3\right) = \frac{1}{x_3}\left(x_1 x_2 \sin x_3 + e^{x_1 x_2}\right), \tag{2}$$

and review slides 21–22 of 02/25 handout before you attempt the following questions. We are assuming the same convention as used in the slides.

(a) Draw the computational graph for this function. (1/12)

(b) List the detailed computational steps to compute the partial derivative $\frac{\partial f}{\partial x_2}$ at the point $(1, 1.5, 2)$ **using the forward mode**. Specifically, provide the numerical values of v_i and $\dot{v}_i$ for all i. For numerical values, you only need to **keep four digits** after the decimal point. To help you get started, let's assume that x_1, x_2 and x_3 are renamed into variables v_{-2}, v_{-1} and v_0. Then

$$v_{-2} = 1, \quad \dot{v}_{-2} = \frac{\partial v_{-2}}{\partial x_2} = 0, \tag{3}$$

$$v_{-1} = 1.5, \quad \dot{v}_{-1} = \frac{\partial v_{-1}}{\partial x_2} = 1, \tag{4}$$

$$v_0 = 2, \quad \dot{v}_0 = \frac{\partial v_{-1}}{\partial x_2} = 0. \tag{5}$$

Please continue and provide the values for all other nodes in your computational graph. (1.5/12)

(c) List of detailed computational steps to compute the partial derivative $\frac{\partial f}{\partial x_2}$ at the point $(1, 1.5, 2)$ **using the reverse mode**. Specifically, provide the numerical values of v_i and $\overline{v}_i$ for all i. For numerical values, you only need to **keep four digits** after the decimal point. (1.5/12)

Problem 3 (Automatic differentiation in DNNs) In principle, we can perform the reverse-mode auto-differentiation (aka back propagation) for DNNs using scalar variables as above. If you're interested in this form, please refer to http://neuralnetworksanddeeplearning.com/chap2.html.

But the scalar version is messy due to the many variables in typical DNNs. More importantly, modern computing hardware and software environments are optimized for performing direct matrix/tensor operations. So it makes perfect sense to perform auto-differentiation directly in matrix/tensor notation.

To illustrate the idea, let's consider a three-layer neural network and the following training objective

$$f\left(\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3\right) \doteq \frac{1}{2}\left\|\boldsymbol{y} - \boldsymbol{W}_3 \sigma\left(\boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right\|_F^2, \tag{6}$$

where the activation σ is ReLU, and we only have a single data point $(\boldsymbol{x}, \boldsymbol{y})$—both $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors (I hope you can see how we can extend to a general objective with many data points!). The

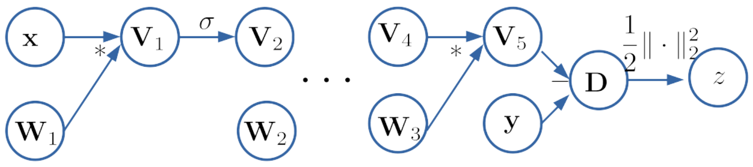

Figure 1: Computational graph of a DNN.

computational graph is shown in Fig. 1. We briefly discussed this on Slide 25 of the 02/25 lecture.

(a) Derive the analytic gradient of f, i.e., $\nabla_{\boldsymbol{W}_1} f$, $\nabla_{\boldsymbol{W}_2} f$, and $\nabla_{\boldsymbol{W}_3} f$. (1/12)

(b) Suppose each node in the computational graph has two fields: $.v$ holds the numerical value for the associated node variable (assuming $\boldsymbol{V}$), and $.d$ holds the derivative value—the derivative of f wrt the current variable, i.e., $\frac{\partial f}{\partial \boldsymbol{V}}$, evaluated at the current value $.v$.

Recall there are two stages in the reverse mode: forward pass and backward pass. Suppose that we have performed the forward pass, so that the value fields $.v$'s are all filled. Let's now perform the backward pass.

- $z = f$ and hence $z.d = \frac{\partial f}{\partial z} = 1$
- $z = \frac{1}{2}\left\|\boldsymbol{D}\right\|_2^2$, so $\frac{\partial z}{\partial \boldsymbol{D}} = \boldsymbol{D}^\intercal$[1]. By the chain rule, $\frac{\partial f}{\partial \boldsymbol{D}} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial \boldsymbol{D}}$, and thus $\boldsymbol{D}.d = (\boldsymbol{D}.v)^\intercal$.
- $\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{y}} = \boldsymbol{0}$ as $\boldsymbol{y}$ is part of the given data and not a variable.
- $\boldsymbol{D} = \boldsymbol{y} - \boldsymbol{V}_5$, so $\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{V}_5} = -\boldsymbol{I}$. To see this, one can use the Taylor expansion method again: recall that for any general vector-to-vector function $f(\boldsymbol{x})$, $f(\boldsymbol{x}+\boldsymbol{\delta}) \approx f(\boldsymbol{x}) + \boldsymbol{J}_f(\boldsymbol{x})\,\boldsymbol{\delta}$ for small $\boldsymbol{\delta}$ when we ignore higher-order terms in $\boldsymbol{\delta}$. Now $\boldsymbol{y} - (\boldsymbol{V}_5 + \boldsymbol{\delta}) = (\boldsymbol{y} - \boldsymbol{V}_5) - \boldsymbol{\delta}$ and so the Jacobian is $-\boldsymbol{I}$. By the chain rule, $\frac{\partial f}{\partial \boldsymbol{V}_5} = \frac{\partial f}{\partial \boldsymbol{D}}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{V}_5}$, and so $\boldsymbol{V}_5.d = -(\boldsymbol{D}.v)^\intercal$.

From this point onward, there are mappings from matrices to vectors, whose derivatives are tensors. We don't need to explicitly form the tensors, as we will use the *vector-Jacobian trick*, which is standard in auto-differentiation packages (e.g., Jax, PyTorch and TensorFlow). A crucial fact used here is that for any $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ and any vector $\boldsymbol{v} \in \mathbb{R}^m$,

$$\boldsymbol{v}^\intercal \boldsymbol{J}_f(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{x}}\left\langle \boldsymbol{v}, f(\boldsymbol{x})\right\rangle, \tag{7}$$

which is again based on the chain rule.

- To compute $\frac{\partial f}{\partial \boldsymbol{W}_3}$, note that $\frac{\partial f}{\partial \boldsymbol{W}_3} = \frac{\partial f}{\partial \boldsymbol{V}_5}\frac{\partial \boldsymbol{V}_5}{\partial \boldsymbol{W}_3}$ and $\frac{\partial f}{\partial \boldsymbol{V}_5}$ is a known row vector. So we can write an equivalent form

  $$\frac{\partial f}{\partial \boldsymbol{W}_3} = \frac{\partial}{\partial \boldsymbol{W}_3}\left\langle \left(\frac{\partial f}{\partial \boldsymbol{V}_5}\right)^\intercal, \boldsymbol{V}_5 \right\rangle. \tag{8}$$

  Now for any column vector $\boldsymbol{v}$ that has the same dimension as $\boldsymbol{V}_5$, $\langle \boldsymbol{v}, \boldsymbol{V}_5\rangle = \langle \boldsymbol{v}, \boldsymbol{W}_3\boldsymbol{V}_4\rangle = \langle \boldsymbol{v}\boldsymbol{V}_4^\intercal, \boldsymbol{W}_3\rangle$. Thus $\frac{\partial}{\partial \boldsymbol{W}_3}\langle \boldsymbol{v}, \boldsymbol{V}_5\rangle = \boldsymbol{v}\boldsymbol{V}_4^\intercal$. So

  $$\frac{\partial f}{\partial \boldsymbol{W}_3} = \left(\frac{\partial f}{\partial \boldsymbol{V}_5}\right)^\intercal \boldsymbol{V}_4^\intercal \quad \text{and} \quad \boldsymbol{W}_3.d = -\boldsymbol{D}.v\,(\boldsymbol{V}_4.v)^\intercal. \tag{9}$$

Question: carry on the backward pass and list each step clearly as above. Particularly, we need the formulas for $\boldsymbol{W}_1.d$ and $\boldsymbol{W}_2.d$. (3/12)

(c) Generate random vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$ and random matrices $\boldsymbol{W}_1 \in \mathbb{R}^{2\times 3}$, $\boldsymbol{W}_2 \in \mathbb{R}^{3\times 3}$ and $\boldsymbol{W}_3 \in \mathbb{R}^{3\times 2}$. You're free to choose your random seed and random distribution. Compute the numerical derivatives based on

  (i) your analytic formulas from (a);

  (ii) the reverse mode differentiation from (b);

  (iii) autograd from PyTorch. If you're not sure how to use this, please review the relevant tutorial sections

  `https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py`
  `https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#learning-pytorch-with-examples`

[1]NB: here D is a vector, not a matrix; also, we write $\frac{\partial(\cdot)}{\partial(\cdot)}$ to mean the derivative, not the gradient; if you get confused, review our lecture on multivariable calculus.

Do they agree with each other up to small numerical errors? Please submit your code and results. *To make sure we can reproduce your results, please fix a random seed in Numpy by setting* `numpy.random.seed` *to a number you like. Also, to make sure the data for comparison stay the same—you may want to generate the data in Numpy, and copy them to PyTorch tensor format when using the autograd in PyTorch.* (2/12)