HOMEWORK SET 3

CSCI 5980/8980 Think Deep Learning (Fall 2020)

Due 11:59 pm, Nov 19 2020

Instruction Typesetting your submission in LaTeX is now **optional**, and but you need to 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 (Automatic differentiation—scalar version; 4/12) Consider the the following three-variable function

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

Review slides 20–22 of the Oct 19th lecture on automatic differentiation. We are using the same notation 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

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

$$v_{-1} = 1.5, \quad \dot{v}_{-1} = \frac{\partial v_{-1}}{\partial x_2} = 1, \tag{3}$$

$$v_0 = 2, \quad \dot{v}_0 = \frac{\partial v_0}{\partial x_2} = 0. \tag{4}$$

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 2 (Automatic differentiation in DNNs; 4/12) 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

$$f\left(\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3\right) \doteq \frac{1}{2}\left\|\boldsymbol{Y} - \boldsymbol{W}_3 \sigma\left(\boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{X}\right)\right)\right\|_F^2, \tag{5}$$

where the activation σ is ReLU.

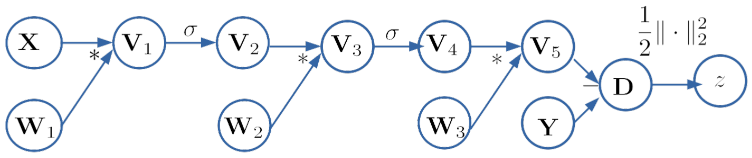

Figure 1: Computational graph of Eq. (5).

The computational graph is shown in Fig. 1. We briefly discussed this on Slides 24–28 in our lecture on auto-differentiation. Let's fix a random seed you like, and generate $\boldsymbol{Y} \in \mathbb{R}^{2\times 50}$, $\boldsymbol{X} \in \mathbb{R}^{5\times 50}$, $\boldsymbol{W}_1 \in \mathbb{R}^{4\times 5}$, $\boldsymbol{W}_2 \in \mathbb{R}^{3\times 4}$, and $\boldsymbol{W}_3 \in \mathbb{R}^{2\times 3}$ all as iid Gaussian. You should fix these matrices once done.

(a) Suppose each node in the computational graph has two fields: $.v$ holds the numerical value of the variable itself, and $.g$ holds the numerical value of the gradient of f with respect to the current variable.

Recall there are two-stages in **reverse-mode** auto-differentiation: forward pass and backward pass.

- Now that $\boldsymbol{X}.v, \boldsymbol{Y}.v, \boldsymbol{W}_1.v, \boldsymbol{W}_2.v, \boldsymbol{W}_3.v$ are known, compute the numerical values of all other variables (i.e., $\boldsymbol{V}_1.v$, $\boldsymbol{V}_2.v$, etc) in the computational graph. You only need to keep 4 digits after the decimal point. (0.5/12)

Now we start to work out the backward process. Obvious $z.g = 1$ as $\nabla_z f = 1$ (remember $f = z$). Moreover, $\nabla_{\boldsymbol{D}} z = \boldsymbol{D}$. So $\boldsymbol{D}.g = \boldsymbol{D}.v$. From this point onward, we start to see the trouble of Jacobians as tensors. For example, $\boldsymbol{V}_5 = \boldsymbol{W}_3 \boldsymbol{V}_4$ and so the Jacobian $\frac{\partial \boldsymbol{V}_5}{\partial \boldsymbol{V}_4}$ is a tensor as both $\boldsymbol{V}_4$ and $\boldsymbol{V}_5$ are matrices—direct implementation involves tensor-matrix product. Fortunately, we can get around the mess by the crucial observation: by implementing chain-rule for gradient, we only care about the result of the Jacobian-matrix product here, not the Jacobian itself. Now if $\boldsymbol{V}_i \mapsto \boldsymbol{V}_j$ and we want to compute $\nabla_{\boldsymbol{V}_i} f$ given $\nabla_{\boldsymbol{V}_j} f$, it turns out

$$\nabla_{\boldsymbol{V}_i} f = \mathcal{J}^{\intercal}_{\boldsymbol{V}_i \mapsto \boldsymbol{V}_j}\left(\nabla_{\boldsymbol{V}_j} f\right) = \nabla_{\boldsymbol{V}_i}\left\langle \boldsymbol{V}_j, \nabla_{\boldsymbol{V}_j} f\right\rangle.$$

The last inner product form avoids the Jacobian tensor $\mathcal{J}_{\boldsymbol{V}_i \mapsto \boldsymbol{V}_j}$ entirely, and now we only need to derive the gradient of matrix to scalar functions.

- For $\boldsymbol{V}_5.g$,

$$\nabla_{\boldsymbol{V}_5} f = \nabla_{\boldsymbol{V}_5}\left\langle \boldsymbol{D}, \boldsymbol{D}.g\right\rangle = \nabla_{\boldsymbol{V}_5}\left\langle \boldsymbol{Y} - \boldsymbol{V}_5, \boldsymbol{D}.g\right\rangle = \nabla_{\boldsymbol{V}_5}\left\langle -\boldsymbol{D}_g, \boldsymbol{V}_5\right\rangle = -\boldsymbol{D}.g.$$

- $\boldsymbol{Y}$ is given data and not optimization variable, so $\nabla_{\boldsymbol{Y}} f = \mathbf{0}$ (In Tensorflow or Pytorch, these variables do not require gradients so will be directly ignored for gradient calculation).
- For $\boldsymbol{V}_4.g$,

$$\nabla_{\boldsymbol{V}_4} f = \nabla_{\boldsymbol{V}_4} \langle \boldsymbol{V}_5, \boldsymbol{V}_5.g \rangle = \nabla_{\boldsymbol{V}_4} \langle \boldsymbol{W}_3 \boldsymbol{V}_4, \boldsymbol{V}_5.g \rangle = \nabla_{\boldsymbol{V}_4} \langle \boldsymbol{W}_3^\intercal \boldsymbol{V}_5.g, \boldsymbol{V}_4 \rangle = \boldsymbol{W}_3^\intercal \boldsymbol{V}_5.g.$$

  So $\boldsymbol{V}_4.g = \boldsymbol{W}_3.v^\intercal \boldsymbol{V}_5.g$.

Question: You should substitute and obtain the numerical values for the above quantities. Now carry on the backward pass and obtain all the numerical values of gradients for all variables. (2.5/12)

(b) Go through the tutorials below and learn how to call Pytorch autograd to compute numerical gradients and read off the gradient values

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.

Now use it to compute the values of $\boldsymbol{W}_1.g$, $\boldsymbol{W}_2.g$, and $\boldsymbol{W}_3.g$. Do they agree with your results in part (a)? (1/12)

Problem 3 (Stochastic optimization methods for MNIST digital recognition; 4/12) In this problem, we're going to train a shallow neural network based on different stochastic gradient descent (SGD) methods that we learned in the lecture. *Neural networks modules and autograd are allowed in this problem, but no built-in optimizers in Pytorch are allowed.* I strongly recommended you go through this tutorial

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#learning-pytorch-with-examples

to learn how to use Pytorch `torch.nn.sequential` to build simple sequential neural networks (this example https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#learning-pytorch-with-examples) and to build more complex neural networks by subclassing `torch.nn.Module` (this example https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#learning-pytorch-with-examples).

We will first load the MNIST dataset into your workspace as in previous homework sets.

(a) Design a 3-layer neural network model. You're free to choose the architecture, i.e., number of nodes, activation functions, convolutional layers if you're comfortable with. Also, choose an appropriate loss for your training objective. (0.5/12)

(b) Implement the Adagrad algorithm. You're free to choose your hyperparameters (initialization, batch size, learning rate, etc). Please include a plot of how the objective vs. the epoch. (0.5/12)

(c) Implement the RMSprob algorithm. Requirement is the same as (b). (0.5/12)

(d) Implement the Adam algorithm. Requirement is the same as (b). The version covered in our lecture is a reduced version of Algorithm 1 of the original paper (`https://arxiv.org/pdf/1412.6980.pdf`) that also handles the initial instability. Please implement the original version. (1.5/12)

(e) A “98%" test: MNIST is a relative easy classification task and the state-of-the-art learning models can achieve near perfect recognition performance. If you get a $\geq 98\%$ test accuracy for any two of (b), (c), and (d), you get 1 point here. For this, so long as your network remains 3-layer, you are free to adjust your network architecture in (a) and/or adopt any strategy to avoid overfitting. You may also compare the performance of your implementation with the built-in (`https://pytorch.org/docs/stable/optim.html`), but the results you report must be produced from your own implementation. (1/12)