HOMEWORK SET 3

CSCI5527 Deep Learning (Fall 2022)

Due 11:59 pm, Nov 13 2022

Instruction Your writeup, either typeset or scanned, should be a single PDF file. For problem requiring coding, please organize all codes for each problem into a separate Jupyter notebook file (i.e., `.ipynb` file). Your submission into Canvas/Gradescope should include the single PDF and all the notebook files—**Please do not zip them**! 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 whenever possible, try 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. $\mathbb{R}$ 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 in DNNs; 4.5/15) In principle, we can perform the reverse-mode auto-differentiation (aka back propagation) for DNNs using scalar variables (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 matrices and tensors. To illustrate the idea, let's consider a three-layer fully-connected neural network $\boldsymbol{x} \mapsto \boldsymbol{W}_3\sigma\left(\boldsymbol{W}_2\sigma\left(\boldsymbol{W}_1\boldsymbol{x}\right)\right)$, and the following training objective

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

where the activation σ is ReLU. The computational graph is shown in Fig. 1. Let's fix a random

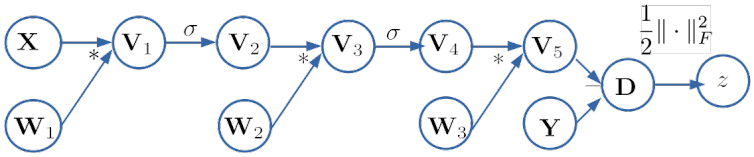

Figure 1: Computational graph of Eq. (1).

```
1  import numpy as np
2  from numpy.random import default_rng
3
4  # fix the random seed; don't modify this
5  rng = default_rng(55272022)
```

Figure 2: Data generation

seed 55272022 (as shown in Fig. 2; Note that Numpy has changed their interface and functions for random number generation in recent versions, and this is the recommended way of fixing the random seed. Check out https://numpy.org/doc/stable/reference/random/generator.html

for details), 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 generated**.

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.

(a) **Forward 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. (1/15)

(b) **Backward pass:** 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 $\boldsymbol{V}_5.g$,

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

  where we recall the fact that $\nabla_{\boldsymbol{X}} \langle \boldsymbol{A}, \boldsymbol{X} \rangle = \boldsymbol{A}$ for any fixed $\boldsymbol{A}$ not dependent on $\boldsymbol{X}$.
- $\boldsymbol{Y}$ is given data and not optimization variable, so $\nabla_{\boldsymbol{Y}} f = \boldsymbol{0}$ (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. (2.5/15)

(c) 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
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

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)? (1/15)

Problem 2 (Stochastic optimization methods for MNIST digital recognition; 4.5/15) 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.** You should go through this tutorial

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

to learn how to use Pytorch `torch.nn.sequential` to build simple sequential neural networks, and how to build more complex neural networks by subclassing the `torch.nn.Module`, before you attempt the following questions.

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, layers (either fully connected, or even convolutional layers if you're comfortable with it). Also, choose an appropriate loss for your training objective. (0.5/15)

(b) Implement the Adagrad algorithm. You're free to choose your hyperparameters (initialization, batch size, learning rate, etc). Please include a plot to how the objective evolves against the epoch. (1/15)

(c) Implement the RMSprob algorithm. Requirement is the same as (b). (0.5/15)

(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/1415.6980.pdf) that also handles the initial instability. Please implement the original version. (1.5/15)

(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. (1/15)

Problem 3 (Autoencoders, deep factorization, and deep sparse coding; 6/15)

(a) The geometric view of PCA says that PCA tries to fit a subspace to a collection of data points. From a modeling perspective, this means PCA makes sense only when the data points lie near a subspace. For example, if $\boldsymbol{x}_i = \boldsymbol{B}\boldsymbol{z}_i + \boldsymbol{\varepsilon}_i$ for all i, where $\boldsymbol{B}$ is a basis for a subspace and $\boldsymbol{\varepsilon}_i$'s represent small-magnitude noise, trying to find a basis for the subspace spanned by $\boldsymbol{B}$ may make a lot of sense. What happens when a small fraction of the data points deviate significantly from the subspace?

To investigate this, let's generate an orthonormal subspace basis $\boldsymbol{B} \in \mathbb{R}^{200\times 20}$ and 98 random data points on the subspace as $\boldsymbol{B}\boldsymbol{z}_i$'s where each $\boldsymbol{z}_i \in \mathbb{R}^{20}$ is iid Gaussian. So this portion of data is perfectly clean. Now let's generate 2 points that are iid Gaussian in $\mathbb{R}^{200}$—these two points will be far from the subspace $\boldsymbol{B}$ almost surely and they are "outliers". Now we have 100 data points and we collect them into a data matrix $\boldsymbol{X} \in \mathbb{R}^{100\times 200}$ (**Each row is a data**

point in our convention). Please normalize the 100 data points so that they all have unit ℓ_2 norm now. *Also, do NOT perform centering to the points for the following steps.*

- Perform PCA via SVD or eigen-decomposition on $\boldsymbol{X}$ (again, no centering step) and numerically compare the subspace spanned by the top 20 singular vectors with $\boldsymbol{B}$. Remember for two subspaces spanned by two bases $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$, their distance can be measured by $\|\boldsymbol{B}_1\boldsymbol{B}_1^\dagger - \boldsymbol{B}_2\boldsymbol{B}_2^\dagger\|_F$. What do you observe? (1/15)
- Specializing the factorization formulation for PCA for our setting is

$$\min_{\boldsymbol{A}\in\mathbb{R}^{200\times 20}, \boldsymbol{z}_i's\in\mathbb{R}^{20}} \sum_{i=1}^{100} \|\boldsymbol{x}_i - \boldsymbol{A}\boldsymbol{z}_i\|_2^2 . \tag{2}$$

  It's equivalent to the above PCA we have done of course. Now let's consider a slight modified version:

$$\min_{\boldsymbol{A}\in\mathbb{R}^{200\times 20}, \boldsymbol{z}_i\in\mathbb{R}^{20}} \sum_{i=1}^{100} \|\boldsymbol{x}_i - \boldsymbol{A}\boldsymbol{z}_i\|_2 , \tag{3}$$

  i.e., sum of the ℓ_2 norm, but norm squared. Numerically solve this optimization problem (choose whatever methods you're comfortable with, and auto-differentiation is also allowed: remember that PyTorch or TensorFlow can also be used to solve generic unconstrained optimization problems, not necessarily deep learning problems). Compare the subspace obtained here with $\boldsymbol{B}$. Do you get a better estimate than the plain PCA above? (2/15)

(b) Read the Science paper *Reducing the Dimensionality of Data with Neural Networks* (https://science.sciencemag.org/content/313/5786/504), which has revived deep learning since 2007.

- Reproduce the first two rows of Fig 2(B), i.e., PCA and autoencoder on MNIST. You should use exactly the same architecture as provided in Fig 1 (right). The original paper uses layer-wise pretraining for initialization and conjugate-gradient for training. You probably don't need these; instead, you can choose modern initialization and optimization methods in PyTorch or TensorFlow. If you are unsure how to use PyTorch to build a neural network and perform training, this tutorial will be helpful: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html (2/15)
- Reproduce the plot in Fig 3, i.e., PCA and autoencoder for visualization in the two-dimensional space. The autoencoder architecture is described in the caption of Fig 3. Again, you can use modern initialization and training methods instead of the original. (1/15)