HOMEWORK SET 2

CSCI5527 Deep Learning (Fall 2023)

Due 11:59 pm, Nov 01 2023

Instruction Your writeup, either typeset or scanned, should be a single PDF file. For problems requiring coding, organize all codes for each problem into a separate Jupyter notebook file (i.e., `.ipynb` file). Your submission to Gradescope should include the single PDF and all notebook files—**please DO NOT zip them**! No late submission will be accepted. For each problem, you should acknowledge your collaborators—**including AI tools**, if any.

About the use of AI tools You are strongly encouraged to use AI tools—they are becoming our workspace friends, such as ChatGPT (https://chat.openai.com/, which does not yet accept PDFs, but we provide the LaTeX source codes for our problems that you can copy and enter), Claude (https://claude.ai/chats, which accepts numerous forms of inputs, including PDFs), and Github Copilot (https://github.com/features/copilot), to help you when trying to solve problems. It takes a bit of practice to ask the right and effective questions/prompts to these tools; we highly recommend that you go through this popular free short course **ChatGPT Prompt Engineering for Developers** offered by https://learn.deeplearning.ai/ to get started.

If you use any AI tools for any of the problems, you should include screenshots of your prompting questions and their answers in your writeup. The answers provided by such AI tools often contain factual errors and reasoning gaps. **So, if you only submit an AI answer with such bugs for any problem, you will obtain a zero score for that problem**. You obtain the scores only when you explain the bugs and also correct them in your own writing. You can also choose not to use any of these AI tools, in which case we will grade based on the efforts you have made.

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.

Preparation Please carefully work through the following two PyTorch official tutorials before attempting the following problems:

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

You can also use TensorFlow or Jax, but you should figure out how to start by yourself. Basic torch built-in functions are listed at

https://pytorch.org/docs/stable/torch.html.

The tensor class and its built-in functions are listed at

https://pytorch.org/docs/stable/tensors.html.

Problem 1 (Autoencoder, factorization, and PCA; gradient descent and back-tracking line search; 6/15) Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ be a collection of points in $\mathbb{R}^n$ and suppose that they are zero-centered, i.e., $\sum_{i=1}^m \boldsymbol{x}_i = \boldsymbol{0}$. We write $\boldsymbol{X} = [\boldsymbol{x}_1 \ \ldots \ \boldsymbol{x}_m]^\intercal \in \mathbb{R}^{m\times n}$, i.e., stacking the data points row-wise into a data matrix. Recall that PCA extracts the top r (with $r \leq n$) eigenvectors of $\boldsymbol{X}^\intercal\boldsymbol{X}$, collects them columnwise into a matrix $\boldsymbol{U} \in \mathbb{R}^{n\times r}$, and obtains a new representation of each data point $\boldsymbol{x}_i$ as

$\boldsymbol{U}^\top \boldsymbol{x}_i \in \mathbb{R}^r$. Geometrically, PCA amounts to deriving the best rank-r linear subspace approximation to the set of points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$:

$$\min_{\boldsymbol{U}\in\mathbb{R}^{n\times r}, \boldsymbol{Z}\in\mathbb{R}^{m\times r}} \frac{1}{m} \|\boldsymbol{X} - \boldsymbol{Z}\boldsymbol{U}^\top\|_F^2 \quad \text{s.t.} \quad \boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$$

So a crucial step in PCA is to compute the subspace basis $\boldsymbol{U}$. Let's now generate a synthetic point set as shown in Fig. 1; see also `2023_HW2_Prob1.ipynb` (please do NOT change the random seed and the dimensions), and complete the following tasks—your X $[0, 0]$ should be approximately

```python
import torch
# fix the random seed; don't modify this
torch.manual_seed(55272023)

# torch default precision is float32. Uncomment below to use float64
# torch.set_default_dtype(torch.float64)

n = 50
m = 200
r = 10

# generate a random dataset
A = torch.randn(n, r)
B = torch.randn(m, r)
X = B @ A.T + 0.01*torch.randn(m,n)

# # zero centering
X -= X.mean(0, keepdims=True)
print(f"X[0, 0]: {X[0, 0]}")
print(f"Mean of features: \n {X.mean(0, keepdims=True)}")
```

```
X[0, 0]: 0.3308468163013458
Mean of features:
 tensor([[-9.5367e-09,  7.3910e-08,  1.1325e-08,  5.2452e-08,  3.4571e-08,
          9.5367e-09,  3.0994e-08, -9.5367e-09,  1.9073e-08,  1.4305e-08,
```

Figure 1: Code segment for generating the data for PCA

0.3308. **Please submit your code for this problem as a separate .ipynb file, not to be combined with the codes for other problems.**

(a) Continue the code in `2023_HW2_Prob1.ipynb` to compute the basis for the best rank-10 subspace approximation to $\boldsymbol{X}$, i.e., a matrix $\boldsymbol{A}_1 \in \mathbb{R}^{n\times 10}$ that contains the first 10 PCA basis vectors. (1/15)

(b) A classic unsupervised learning technique in deep learning is the autoencoder (we'll cover it later in the course). The mathematical formulation specialized to our case is

$$\min_{\boldsymbol{A}\in\mathbb{R}^{n\times 10}} f(\boldsymbol{A}) \doteq \frac{1}{m} \|\boldsymbol{X} - \boldsymbol{X}\boldsymbol{A}\boldsymbol{A}^\top\|_F^2 .$$

(i) Implement the gradient descent method to solve the optimization problem, with **backtracking line search** for the step size and **appropriate stopping criterion** (say, by checking the gradient norm). You can use whatever methods to compute the numerical gradients, including auto-differentiation. (2/15).

(ii) Let's say the solution computed from the last step is $\boldsymbol{A}_2$. Now we want to compare the subspaces represented by $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$. We cannot directly do $\boldsymbol{A}_1 - \boldsymbol{A}_2$, as from linear algebra we know that $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ can span the same column/range space but take very different forms. Instead, a reasonable metric here is the difference between the subspace projectors induced by them, i.e., $\|\boldsymbol{A}_1\boldsymbol{A}_1^\dagger - \boldsymbol{A}_2\boldsymbol{A}_2^\dagger\|_F / \|\boldsymbol{A}_2\boldsymbol{A}_2^\dagger\|_F$[1], where $\boldsymbol{A}^\dagger$ denotes the

[1]We use this relative difference to remove the dependency of the metric on dimensionality. It is fine to use $\|\boldsymbol{A}_1\boldsymbol{A}_1^\dagger\|_F$ on the denominator also.

matrix pseudoinverse (https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse; in PyTorch, you can call this function `torch.linalg.pinv`https://pytorch.org/docs/stable/generated/torch.linalg.pinv.html). Report your result here. Is it close to 0 or not? (0.5/15)

(c) Consider another formulation, which is normally called **factorization**:

$$\min_{\boldsymbol{A}\in\mathbb{R}^{n\times 10},\boldsymbol{Z}\in\mathbb{R}^{m\times 10}} g(\boldsymbol{A},\boldsymbol{Z}) \doteq \|\boldsymbol{X}-\boldsymbol{Z}\boldsymbol{A}^{\intercal}\|_F^2 .$$

(i) Implement the gradient descent method to solve the optimization problem, with **backtracking line search** for the step size and **appropriate stopping criterion** (say, by checking the gradient norm). You can use whatever methods to compute the numerical gradients, including auto-differentiation. (2/15)

(ii) Let's say the solution computed from the last step is $\boldsymbol{A}_3$. Please compute the subspace differences between $\boldsymbol{A}_1$, $\boldsymbol{A}_2$ and $\boldsymbol{A}_3$ and report your results here. Are they close to 0 or not? (0.5/15)

Problem 2 (Automatic differentiation—scalar version; 3/15) Consider the the following three-variable function

$$f(x_1,x_2,x_3) = \frac{1}{x_3}\left(x_1x_2\sin x_3 + e^{x_1x_2}\right). \tag{1}$$

(a) Draw the computational graph for this function. (1/15)

(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. (1/15)

(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. (1/15)

Problem 3 (Automatic differentiation in DNNs; 4/15) In principle, we can perform reverse-mode autodifferentiation (aka backpropagation) for DNNs using scalar variables. But the scalar version is messy due to the many variables in typical DNNs. More importantly, modern computing hardware and software environments are optimized to perform direct matrix and tensor operations. So it makes perfect sense to perform autodifferentiation directly on 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

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

where the activation σ is ReLU. The computational graph is shown in Fig. 2. Let's fix a random

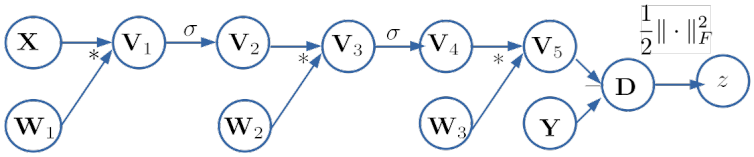

Figure 2: Computational graph of Eq. (5).

```
1  import torch
2  # fix the random seed; don't modify this
3  torch.manual_seed(55272023)
4
```

Figure 3: Data generation

seed 55272023 (as shown in Fig. 3), 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 that 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**/**backward-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 $\boldsymbol{V}_5.g$, where the numerical value $\boldsymbol{D}.g$ is known:

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

  where we recall the fact that $\nabla_{\boldsymbol{X}}\left\langle \boldsymbol{A}, \boldsymbol{X}\right\rangle = \boldsymbol{A}$ for any fixed $\boldsymbol{A}$ not dependent on $\boldsymbol{X}$.

- $\boldsymbol{Y}$ is given data and not optimization variable, so $\nabla_{\boldsymbol{Y}} f = \mathbf{0}$ (In Tensorflow or Pytorch, these variables do not require gradients so will be directly ignored for gradient calculation).
- For $\boldsymbol{V}_4.g$, where the numerical value $\boldsymbol{V}_5.g$ is known:

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

  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 4 (Classification of dry beans; 2/15) In this problem, we consider the classification of dry beans based on their visual features, based on this UCI Dry Bean Dataset: https://archive.ics.uci.edu/dataset/602/dry+bean+dataset. Please check out the details of the dataset from the webpage and also how to import the data into a Python environment by clicking the button "IMPORT IN PYTHON" and following their instruction. Split the data as $60\% : 20\% : 20\%$ as training, validation, and test sets, respectively. **Please submit your code for this problem as a separate .ipynb file, not to be combined with the codes for other problems.**

(a) Train a classifier based on one of the three models: support vector machines, random forest, or boosting. Note that for hyperparameter tuning, you can only use the training and validation sets, not the test data. You can use whatever machine learning libraries you want to use to build your training, hyper-parameter tuning, and test pipeline, to the level of calling any of their built-in functions; we highly recommend the scikit-learn library https://scikit-learn.org/stable/. You need to get 90% classification accuracy to get the score. (1/15)

(b) Train a classifier based on multi-layer perceptrons, i.e., multi-layer fully connected networks. Note that for hyperparameter tuning, you can only use the training and validation sets, not the test data. You are supposed to use PyTorch (or TensorFlow, Jax) to build your training, hyper-parameter tuning, and test pipeline. You may find this tutorial helpful when loading datasets not built into PyTorch: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html. You need to get 90% classification accuracy to get the score. (1/15)