

HOMWORK SET 2

CSCI5527 Deep Learning (Spring 2025)

Due 11:59 pm, Mar 16 2025

Instruction Your writeup, either typeset or scanned, should be a single PDF file. For problems requiring coding, organize all codes for each top-level problem (i.e., Problem 1, Problem 2, etc) 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/>), Claude (<https://claude.ai/chats>), 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., \mathbf{a}) for vectors, and capital boldface letters (e.g., \mathbf{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.

Important notes Please provide detailed steps with justification for all problems; jumping to the final results leads to a zero score. Also, if we ask you to use certain facts/tools to obtain something, you have to use these facts/tools (perhaps plus others); otherwise, you get a zero score, e.g., if you prove everything from scratch when we ask you to use an existing theorem.

Preparation Please carefully work through the following two PyTorch official tutorials

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

before attempting any problems in this homework set. 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 $\mathbf{x}_1, \dots, \mathbf{x}_m$ be a collection of points in \mathbb{R}^n and suppose that they are zero-centered, i.e., $\sum_{i=1}^m \mathbf{x}_i = \mathbf{0}$. We write $\mathbf{X} = [\mathbf{x}_1 \dots \mathbf{x}_m]^\top \in \mathbb{R}^{m \times n}$, 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 $\mathbf{X}^\top \mathbf{X}$, collects them columnwise into a matrix $\mathbf{U} \in \mathbb{R}^{n \times r}$, and obtains a new representation of each data point \mathbf{x}_i as $\mathbf{U}^\top \mathbf{x}_i \in \mathbb{R}^r$. Geometrically, PCA amounts to deriving the best rank- r linear subspace approximation to the set of points $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$:

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

So a crucial step in PCA is to compute the subspace basis \mathbf{U} . Let's now generate a synthetic point set as shown in Fig. 1; see also `2025_HW2_Prob1.ipynb` (please do NOT change the random seed and the dimensions), and complete the following tasks—your $\mathbf{X}[0, 0]$ should be approximately

```

1 import torch
2 # fix the random seed; don't modify this
3 torch.manual_seed(55272025)
4
5 # torch default precision is float32. Uncomment below if you want to use float64
6 # torch.set_default_dtype(torch.float64)
7
8 n = 50
9 m = 200
10 r = 10
11
12 # generate a random dataset
13 A = torch.randn(n, r)
14 B = torch.randn(m, r)
15 X = B @ A.T + 0.01*torch.randn(m,n)
16
17 ## zero centering
18 X -= X.mean(0, keepdims=True)
19 print(f"X[0, 0]: {X[0, 0]}")
20 print(f"Mean of features: \n {X.mean(0, keepdims=True)}")

```

X[0, 0]: 4.165602207183838
Mean of features:
tensor([[5.7220e-08, -2.6226e-08, 0.0000e+00, 4.2915e-08, 6.1914e-08,
-4.7684e-09, -2.3842e-09, -1.1444e-07, 1.9073e-08, 1.7881e-08,

Figure 1: Code segment for generating the data for PCA

4.1656 (if you use the default `float32` precision).

- Continue the code in `2025_HW2_Prob1.ipynb` to compute the basis for the best rank-10 subspace approximation to \mathbf{X} , i.e., a matrix $\mathbf{A}_1 \in \mathbb{R}^{n \times 10}$ that contains the first 10 PCA basis vectors. (1/15)
- 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

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

- Implement the gradient descent method to solve the optimization problem, with **back-tracking 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. Since $f(\mathbf{A})$ is nonconvex, you may want to repeat your gradient descent algorithm multiple times using different random initializations to find a reasonably good solution. (2/15).

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

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

- (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. Since $f(\mathbf{A}, \mathbf{Z})$ is nonconvex, you may want to repeat your gradient descent algorithm multiple times using different random initializations to find a reasonably good solution. (2/15)
- (ii) Let's say the solution computed from the last step is \mathbf{A}_3 . Please compute the subspace differences between \mathbf{A}_1 , \mathbf{A}_2 and \mathbf{A}_3 and report your results here. Are they close to 0 or not? (0.5/15)

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

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

- (a) Draw the computational graph for this function. Note that depending on how you parse the function, computational graph for a function may not be unique. But the AD results should not be affected by this. (1/15)
- (b) List 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

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

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

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

Please continue and provide the values for all other nodes in your computational graph. (1/15)

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

- (c) List 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 \bar{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 (AD) in DNNs; 4/15) In principle, we can perform reverse-mode AD 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 AD directly on matrices and tensors. To illustrate the idea, let's consider a three-layer fully-connected neural network $x \mapsto \mathbf{W}_3 \sigma(\mathbf{W}_2 \sigma(\mathbf{W}_1 x))$, and the following training objective

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

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

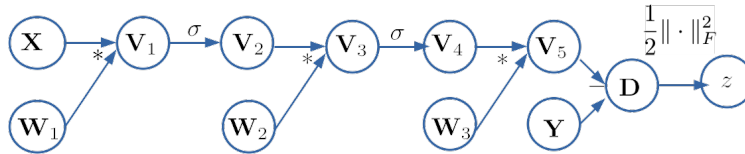


Figure 2: Computational graph of Eq. (5).

55272025, and generate $\mathbf{Y} \in \mathbb{R}^{2 \times 50}$, $\mathbf{X} \in \mathbb{R}^{5 \times 50}$, $\mathbf{W}_1 \in \mathbb{R}^{4 \times 5}$, $\mathbf{W}_2 \in \mathbb{R}^{3 \times 4}$, and $\mathbf{W}_3 \in \mathbb{R}^{2 \times 3}$ 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** AD: forward pass and backward pass.

- (a) **Forward pass:** Now that $\mathbf{X}.v, \mathbf{Y}.v, \mathbf{W}_1.v, \mathbf{W}_2.v, \mathbf{W}_3.v$ are known, compute the numerical values of all other variables (i.e., $\mathbf{V}_1.v, \mathbf{V}_2.v$, 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 pass. Obvious $z.g = 1$ as $\nabla_z f = 1$ (remember $f = z$). Moreover, $\nabla_D z = \mathbf{D}$. So $\mathbf{D}.g = \mathbf{D}.v$. From this point onward, we start to see the trouble of working with Jacobians as tensors. For example, $\mathbf{V}_5 = \mathbf{W}_3 \mathbf{V}_4$ and so the Jacobian $\frac{\partial \mathbf{V}_5}{\partial \mathbf{V}_4}$ is a tensor as both \mathbf{V}_4 and \mathbf{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, suppose that \mathbf{V}_j is a function of \mathbf{V}_i , and we want to compute $\nabla_{\mathbf{V}_i} f$ given $\nabla_{\mathbf{V}_j} f$. It turns out

$$\nabla_{\mathbf{V}_i} f = \mathcal{J}_{\mathbf{V}_i \mapsto \mathbf{V}_j}^T (\nabla_{\mathbf{V}_j} f) = \nabla_{\mathbf{V}_i} \langle \mathbf{V}_j, \nabla_{\mathbf{V}_j} f \rangle.$$

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

- For $\mathbf{V}_5.g$, where the numerical value $\mathbf{D}.g$ is known:

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

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

- Y is given data and not an optimization variable, so $\nabla_Y f = \mathbf{0}$ (In PyTorch, these variables do not require gradients so will be directly ignored for gradient calculation).
- For $V_4.g$, where the numerical value $V_5.g$ is known:

$$\nabla_{V_4} f = \nabla_{V_4} \langle V_5, V_5.g \rangle = \nabla_{V_4} \langle W_3 V_4, V_5.g \rangle = \nabla_{V_4} \langle W_3^T V_5.g, V_4 \rangle = W_3^T V_5.g.$$

$$\text{So } V_4.g = (W_3.v)^T V_5.g.$$

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 $W_1.g$, $W_2.g$, and $W_3.g$. 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 using 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 instructions. Split the data as 60% : 20% : 20% into training, validation, and test sets, respectively.

- (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 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)