CS 446 / ECE 449 — Homework 6

wl72

Instructions.

- Homework is due **Friday December 5th**, at 11:59 **PM** CST; you have **3** late days in total for **all Homeworks**.
- Everyone must submit individually at gradescope under `HW6` and `HW6-Programming Assignment`.
- The "written" submission at `HW6` **must be typed**, and submitted in any format gradescope accepts (to be safe, submit a PDF). You may use LaTeX, markdown, google docs, MS word, whatever you like; but it must be typed!
- When submitting at `HW6`, Gradescope will ask you to **mark out boxes around each of your answers**; please do this precisely!
- Please make sure your NetID is clear and large on the first page of the homework.
- Your solution **must** be written in your own words and you may not consult LLMs. Please see the course webpage for full **academic integrity** information. You should cite any external reference you use.
- We reserve the right to reduce the auto-graded score for `HW6-Programming Assignment` if we detect funny business (e.g., your solution lacks any algorithm and hard-codes answers you obtained from someone else, or simply via trial-and-error with the autograder).
- When submitting to `HW6-Programming Assignment`, only upload `hw6_q1.py`, `hw6_q3_autograd.py`, `hw6_q3_nn.py`, `hw6_q3_optim.py`, and `hw6_q3_utils.py`. Additional files will be ignored.

1. PCA (30 pt)

(a) (5 pt) Use SVD to compute the principal components ($k = 2$) of this matrix.

$$\begin{bmatrix} 5 & 1 & 1 \\ 1 & 5 & -1 \end{bmatrix}$$

(b) (2 pt) How would the resulting "principal components" differ if you didn't center the matrix?

In lecture, we learned that PCA is equivalent to maximizing the variances of datapoint projections and minimizing the error of the projections. In this problem, we will show the equivalences of various interpretations of PCA. To do so, assume you are given a centered data matrix $\boldsymbol{X} \in \mathbb{R}^{d \times N}$, where each of the N columns $\boldsymbol{x}^{(i)}$ in $\boldsymbol{X}$ are d-dimensional datapoints.

(c) You have a subspace $\mathcal{S} = \{\mathbf{b} + v\mathbf{u} : v \in \mathbb{R}\}$, where $\mathbf{b}$ is a vector of length d ($\mathbf{b} \in \mathbb{R}^d$) and $\mathbf{u}$ is a unit vector ($\mathbf{u}^\top \mathbf{u} = 1$).

i. (2 pt) Derive the formula for $\widetilde{x}$, the projection of x onto $\mathcal{S}$, in terms of $\mathbf{b}$ and $\mathbf{u}$.

ii. (3 pt) Derive the minimal squared distance $||\widetilde{x} - x||_2^2$ in terms of x, $\mathbf{b}$, and $\mathbf{u}$.

(d) (3 pt) What is the formula for the mean squared variance of $\boldsymbol{X}$ in a certain direction? How can you compute the direction of maximal variance of $\boldsymbol{X}$?

(e) (5 pt) Let's prove that the two definitions from lecture–maximizing the variances of the points projected onto a line and minimizing the errors between the points and their projections–are equivalent.

(f) (5 pt) Now, prove that finding a rank 1 approximation of the matrix $\boldsymbol{X}$ is equivalent to selecting the subspace $\mathcal{S}$ that maximizes the variances of the projected points.

Hint 1: Consider how to express a rank 1 matrix

Hint 2: Look into the properties of the Frobenius norm.

Since PCA is equivalent to minimizing reconstruction error, a linear autoencoder optimized using gradient descent can effectively recover the subspace spanned by PCA components (up to a rotation/reflection).

(g) (5 pt) Implement `hw6_q1.py` to empirically verify this. The output of your encoder may not be directly aligned with the subspace spanned by PCA components. However, assuming the autoencoder is trained correctly and its latent space spans the same subspace of the top k PCA components, you can recover the transformation from the encoder output to PCA components by employing simple least squares. After the transformation, the output of your encoder should align with the PCA components. A Jupyter notebook `hw6.ipynb` is provided for you to test your implementation.

Hint: You should not use two separate linear layers to implement the encoder and decoder.

Solution. Part (a): SVD for Principal Components

Given matrix: $\begin{bmatrix} 5 & 1 & 1 \\ 1 & 5 & -1 \end{bmatrix}$

To find the principal components using SVD, I need to compute $A = U\Sigma V^T$.

First, let I did calculate AA^T:

$$AA^T = \begin{bmatrix} 5 & 1 & 1 \\ 1 & 5 & -1 \end{bmatrix} \begin{bmatrix} 5 & 1 \\ 1 & 5 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 27 & 9 \\ 9 & 27 \end{bmatrix}$$

Now I found the eigenvalues by solving the characteristic equation:

$$\det(AA^T - \lambda I) = \det \begin{bmatrix} 27 - \lambda & 9 \\ 9 & 27 - \lambda \end{bmatrix} = (27 - \lambda)^2 - 81 = 0$$

Expanding this out: $(27-\lambda)^2 = 81$, so $27-\lambda = \pm 9$

This gives me $\lambda_1 = 36$ and $\lambda_2 = 18$. The singular values are the square roots: $\sigma_1 = 6$ and $\sigma_2 = 3\sqrt{2} \approx 4.24$.

For the eigenvectors of AA^T: - When $\lambda_1 = 36$: solving $(AA^T - 36I)\mathbf{v} = 0$ gives $\mathbf{u}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}$ - When $\lambda_2 = 18$: solving $(AA^T - 18I)\mathbf{v} = 0$ gives $\mathbf{u}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}$

The principal components are found using $\mathbf{v}_i = \frac{1}{\sigma_i}A^T\mathbf{u}_i$:

$$\mathbf{v}_1 = \frac{1}{6}\begin{bmatrix}5 & 1\\1 & 5\\1 & -1\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} = \frac{1}{3\sqrt{2}}\begin{bmatrix}3\\3\\0\end{bmatrix}$$

$$\mathbf{v}_2 = \frac{1}{3\sqrt{2}}\begin{bmatrix}5 & 1\\1 & 5\\1 & -1\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} = \frac{1}{3}\begin{bmatrix}2\\-2\\1\end{bmatrix}$$

So the first two principal components are $\mathbf{v}_1 = \frac{1}{\sqrt{18}}\begin{bmatrix}3\\3\\0\end{bmatrix}$ and $\mathbf{v}_2 = \frac{1}{3}\begin{bmatrix}2\\-2\\1\end{bmatrix}$.

Part (b): effect of not centering

If I don't center the data first, the principal components would be totally different. Instead of finding directions of maximum variance around the mean, they'd be influenced by where the data sits in space. The first component would probably just point toward the mean of all the data points rather than capturing the actual spread pattern. Basically, you'd be mixing up the location information with the variability information, which isn't what we want from PCA.

Part (c): projection onto subspace

(i) To project $\mathbf{x}$ onto the subspace $\mathcal{S} = \{\mathbf{b} + v\mathbf{u} : v \in \mathbb{R}\}$, I need to find the point in $\mathcal{S}$ that's closest to $\mathbf{x}$. The projection is:

$$\widetilde{\mathbf{x}} = \mathbf{b} + \langle \mathbf{x} - \mathbf{b}, \mathbf{u}\rangle \mathbf{u}$$

This works because I'm taking the component of $(\mathbf{x} - \mathbf{b})$ in the direction of $\mathbf{u}$ and adding it to the base point $\mathbf{b}$. The leftover part will be perpendicular to the subspace.

(ii) For the squared distance, I can expand it out:

$$\|\widetilde{\mathbf{x}} - \mathbf{x}\|_2^2 = \|\mathbf{b} + \langle \mathbf{x} - \mathbf{b}, \mathbf{u}\rangle \mathbf{u} - \mathbf{x}\|_2^2$$

Rearranging:

$$= \|(\mathbf{b} - \mathbf{x}) + \langle \mathbf{x} - \mathbf{b}, \mathbf{u}\rangle \mathbf{u}\|_2^2$$

Since $\mathbf{u}$ is unit length and the dot product gives us the projection coefficient, this simplifies to:

$$\|\mathbf{x} - \mathbf{b}\|_2^2 - \langle \mathbf{x} - \mathbf{b}, \mathbf{u}\rangle^2$$

Part (d): variance formula

The variance of the data $\mathbf{X}$ when projected onto direction $\mathbf{u}$ is:

$$\text{Var}(\mathbf{u}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{u}^T\mathbf{x}^{(i)})^2 = \mathbf{u}^T\left(\frac{1}{N}\mathbf{X}\mathbf{X}^T\right)\mathbf{u} = \mathbf{u}^T\mathbf{C}\mathbf{u}$$

where $\mathbf{C}$ is the covariance matrix. To find the direction with maximum variance, I solve:

$$\max_{\mathbf{u}} \mathbf{u}^T\mathbf{C}\mathbf{u} \text{ subject to } \|\mathbf{u}\|_2 = 1$$

This is just the standard eigenvalue problem - the solution is the eigenvector of $\mathbf{C}$ with the largest eigenvalue.

Part (e): showing the equivalence

Let me show that maximizing variance is the same as minimizing reconstruction error.

For maximizing variance with unit vector $\mathbf{u}$:

$$\max_{\mathbf{u}} \frac{1}{N} \sum_{i=1}^{N} (\mathbf{u}^T \mathbf{x}^{(i)})^2$$

For minimizing reconstruction error (assuming centered data so $\mathbf{b} = \mathbf{0}$):

$$\min_{\mathbf{u}} \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}^{(i)} - (\mathbf{u}^T \mathbf{x}^{(i)})\mathbf{u}\|_2^2$$

If I expand the reconstruction error term:

$$\|\mathbf{x}^{(i)} - (\mathbf{u}^T \mathbf{x}^{(i)})\mathbf{u}\|_2^2 = \|\mathbf{x}^{(i)}\|_2^2 - (\mathbf{u}^T \mathbf{x}^{(i)})^2$$

So the total reconstruction error becomes:

$$\frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}^{(i)}\|_2^2 - \frac{1}{N} \sum_{i=1}^{N} (\mathbf{u}^T \mathbf{x}^{(i)})^2$$

The first term doesn't depend on $\mathbf{u}$, so minimizing the whole thing is equivalent to maximizing the second term, which is exactly the variance we want to maximize.

Part (f): rank-1 approximation equivalence

Any rank-1 matrix can be written as $\mathbf{A} = \mathbf{a}\mathbf{b}^T$ for some vectors $\mathbf{a}$ and $\mathbf{b}$.

The problem of finding the best rank-1 approximation is:

$$\min_{\mathbf{a},\mathbf{b}} \|\mathbf{X} - \mathbf{a}\mathbf{b}^T\|_F^2$$

Using the fact that $\|\mathbf{A}\|_F^2 = \text{tr}(\mathbf{A}\mathbf{A}^T)$, I can expand this:

$$\|\mathbf{X} - \mathbf{a}\mathbf{b}^T\|_F^2 = \|\mathbf{X}\|_F^2 - 2\text{tr}(\mathbf{X}^T \mathbf{a}\mathbf{b}^T) + \|\mathbf{a}\|_2^2 \|\mathbf{b}\|_2^2$$

The middle term is $2\mathbf{b}^T \mathbf{X}^T \mathbf{a}$. Since the first term is constant, minimizing the whole expression is equivalent to maximizing $\mathbf{b}^T \mathbf{X}^T \mathbf{a}$ subject to normalization constraints.

This is exactly the same as finding the direction $\mathbf{a}$ that maximizes the variance of the projections of the data, which is what PCA does. So the rank-1 approximation and PCA variance maximization are solving the same problem.

2. Variational Auto-Encoders (25 pt)

We are training a variational auto-encoder (VAE). It consists of the following parts: the input vector $\boldsymbol{x}$, the latent vector $\boldsymbol{z}$, the encoder that models the probability $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, and the decoder that models $p_\theta(\boldsymbol{x}|\boldsymbol{z})$. Based on these notations, let's look at several problems:

(a) We assume the latent vector $\boldsymbol{z} \in \mathbb{R}^2$ follows a multi-variate Gaussian distribution $\mathcal{N}$. Please compute the output dimension of the encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ under the following cases and briefly explain why. (If "output dimension" is not clear enough for you, think of it as "how many real numbers $r \in \mathbb{R}$ are needed to output for the sampling of latent vectors.")

- (2 pt) We assume $\mathcal{N}$ has an **identity matrix** as the covariance matrix.
- (2 pt) We assume $\mathcal{N}$ has an **diagonal matrix** as the covariance matrix.

(b) We then consider the problems related to the understanding of KL-Divergence.

i. (5 pt) Using the inequality of $\log(x) \leq x - 1$, prove that $D_{KL}(p(\boldsymbol{x}), q(\boldsymbol{x})) \geq 0$ holds for two arbitrary distributions $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$.

ii. (7 pt) Consider a binary classification problem with input vectors $\boldsymbol{x}$ and labels $y \in \{0, 1\}$. The distribution of the ground truth label is denoted as $P(y)$. The expression of $P(y)$ is as Eq 1, where y_{gt} is the ground truth label.

$$P(y = y_{gt}) = 1, P(y = 1 - y_{gt}) = 0 \tag{1}$$

Suppose we are trying to predict the label of $\boldsymbol{x}$ with a linear model $\boldsymbol{w}$ and sigmoid function, then the distribution of y is denoted as $Q(y)$ and computed as Eq. 2.

$$Q(y = 0|\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x})}, \quad Q(y = 1|\boldsymbol{x}) = \frac{\exp(-\boldsymbol{w}^\top \boldsymbol{x})}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x})} \tag{2}$$

With the above information, compute the KL Divergence between the distributions of $P(y)$ and $Q(y|\boldsymbol{x})$, specifically $D_{KL}(P(y) \| Q(y|\boldsymbol{x})) = \mathbb{E}_{y \sim P(y)}[\log \frac{P(y)}{Q(y|\boldsymbol{x})}]$.
Expand your solution to the clearest form. To get full credits, you may only use $y_{gt}, \boldsymbol{w}, \boldsymbol{x}$ and related constants in your expression.

(c) VAE is a special branch of generative method in sampling the latent vectors $\widetilde{\boldsymbol{z}}$ from $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ instead of directly regressing the values of $\boldsymbol{z}$. Read an example implementation of VAE at `https://github.com/AntixK/PyTorch-VAE/blob/master/models/vanilla_vae.py` and answer the following questions:

i. (1 pt) Find the functions and lines related to the sampling of $\widetilde{\boldsymbol{z}}$ from $q_\phi(\boldsymbol{z}|\boldsymbol{x})$. Specifying the names of the functions and the related lines can lead to full credits. Please note that if your range is too broad (in the extreme case, covering every line in the file) we cannot give your full credit.

ii. (5 pt) Suppose our latent variable is $\boldsymbol{z} \in \mathbb{R}^2$ sampled from a Gaussian distribution with mean $\boldsymbol{\mu} \in \mathbb{R}^2$ and a diagonal covariance matrix $\boldsymbol{\Sigma} = \texttt{Diag}\{\sigma_1^2, \sigma_2^2\}$. Then another random variable $\boldsymbol{\varepsilon} \in \mathbb{R}^2$ is sampled from a Gaussian distribution $\mathcal{N}(0, \boldsymbol{I})$. Show that $\boldsymbol{V} = [\sigma_1, \sigma_2]^\top \odot \boldsymbol{\varepsilon} + \boldsymbol{\mu}$ follows the same distribution as $\boldsymbol{z}$. ($\odot$ denotes element-wide product; $\mathcal{N}(0, \boldsymbol{I})$ denotes the multi-variate Gaussian with zero mean and identity matrix as covariance.)

iii. (2 pt) Under the same setting of the Question ii, we can sample the latent vector $\widetilde{\boldsymbol{z}}$ by the process $\widetilde{\boldsymbol{z}} = [\sigma_1, \sigma_2]^\top \odot \widetilde{\boldsymbol{\varepsilon}} + \boldsymbol{\mu}$, where $\widetilde{\boldsymbol{\varepsilon}}$ is a sampled random variable from $\mathcal{N}(0, \boldsymbol{I})$. Consider the process of training, where we apply back-propagation to train the neural networks. Given the gradient on $\widetilde{\boldsymbol{z}}$ as $\widetilde{\boldsymbol{g}} \in \mathbb{R}^2$, which can be written as $[\widetilde{g}_1, \widetilde{g}_2]^\top$. **What are the gradients of the output of the encoder: $\boldsymbol{\mu}, \sigma_1, \sigma_2$?** (Assume the KL-Divergence loss is not considered in this part.)
Note: To get full credit, you may use any constants and the variables $\widetilde{\boldsymbol{\varepsilon}} = [\widetilde{\varepsilon_1}, \widetilde{\varepsilon_2}]$, $\widetilde{\boldsymbol{g}} = [\widetilde{g}_1, \widetilde{g}_2]^\top$, and $\boldsymbol{\mu}, \sigma_1, \sigma_2$.

iv. (1 pt) After reading the code, explain why we sample $\widetilde{\boldsymbol{z}}$ with $\mathcal{N}(0, 1)$, instead of directly generating the values.

Solution. Part (a): encoder output dimensions

(i) identity covariance matrix: When the latent vector $\mathbf{z} \in \mathbb{R}^2$ follows $\mathcal{N}(\boldsymbol{\mu}, \mathbf{I})$, the covariance is just fixed to be the identity matrix. So I only need the encoder to output the mean vector $\boldsymbol{\mu}$, which has 2 components since we're in 2D.

Output dimension: **2** (just the mean vector components)

(ii) diagonal covariance matrix: Now if $\mathbf{z}$ follows $\mathcal{N}(\boldsymbol{\mu}, \text{diag}(\sigma_1^2, \sigma_2^2))$, I need both the mean and the variances. The mean is still 2 numbers, but now I also need to output σ_1^2 and σ_2^2, which is 2 more numbers.

Output dimension: **4** (2 for mean + 2 for the diagonal variances)

Part (b): understanding KL-divergence

(i) proving non-negativity: I'll use the given inequality $\log(x) \leq x - 1$ where equality happens when $x = 1$.

Starting with the KL divergence:

$$D_{KL}(p(\mathbf{x}) \| q(\mathbf{x})) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x}$$

I then sub in $x = \frac{q(\mathbf{x})}{p(\mathbf{x})}$ into the inequality. Then:

$$\log \frac{p(\mathbf{x})}{q(\mathbf{x})} = -\log \frac{q(\mathbf{x})}{p(\mathbf{x})} \geq -\left(\frac{q(\mathbf{x})}{p(\mathbf{x})} - 1 \right) = 1 - \frac{q(\mathbf{x})}{p(\mathbf{x})}$$

Plugging this back into the KL divergence:

$$D_{KL}(p(\mathbf{x}) \| q(\mathbf{x})) \geq \int p(\mathbf{x}) \left(1 - \frac{q(\mathbf{x})}{p(\mathbf{x})} \right) d\mathbf{x}$$

$$= \int p(\mathbf{x}) d\mathbf{x} - \int q(\mathbf{x}) d\mathbf{x} = 1 - 1 = 0$$

So KL divergence is always non-negative, and it's zero only when the distributions are identical.

(ii) binary classification KL-divergence: The ground truth distribution $P(y)$ is super simple - it's 1 for the correct label and 0 for everything else.

So the KL divergence becomes:

$$D_{KL}(P(y) \| Q(y|\mathbf{x})) = \sum_y P(y) \log \frac{P(y)}{Q(y|\mathbf{x})}$$

Since $P(y) = 0$ for $y \neq y_{gt}$, only the $y = y_{gt}$ term survives:

$$D_{KL}(P(y) \| Q(y|\mathbf{x})) = 1 \cdot \log \frac{1}{Q(y_{gt}|\mathbf{x})} = -\log Q(y_{gt}|\mathbf{x})$$

I gotta now do $Q(y_{gt}|\mathbf{x})$ is from the sigmoid formula. From equation (2): - $Q(y = 0|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$ - $Q(y = 1|\mathbf{x}) = \frac{\exp(-\mathbf{w}^T\mathbf{x})}{1+\exp(-\mathbf{w}^T\mathbf{x})} = \frac{1}{1+\exp(\mathbf{w}^T\mathbf{x})}$

Wait, let me double-check this. Actually, I think there's an error in the given equation. The standard sigmoid should give us: - $Q(y = 1|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$ - $Q(y = 0|\mathbf{x}) = \frac{\exp(-\mathbf{w}^T\mathbf{x})}{1+\exp(-\mathbf{w}^T\mathbf{x})} = \frac{1}{1+\exp(\mathbf{w}^T\mathbf{x})}$

Using the given formulation though, the final result is:

$$D_{KL}(P(y) \| Q(y|\mathbf{x})) = -y_{gt} \log Q(y = 1|\mathbf{x}) - (1 - y_{gt}) \log Q(y = 0|\mathbf{x})$$

Substituting the expressions and simplifying:

$$= y_{gt} \log(1 + \exp(\mathbf{w}^T \mathbf{x})) + (1 - y_{gt}) \log(1 + \exp(-\mathbf{w}^T \mathbf{x}))$$

Part (c): VAE implementation analysis

(i) finding the sampling functions: Looking at the VAE code, the sampling happens in a few key places:

- The 'encode' function (around lines 35-36) outputs the mean and log-variance
- The 'reparameterize' function (lines 45-47) does the actual sampling using the reparameterization trick
- Specifically, line 46 does the sampling: 'std * torch.randn$_like(std) + mu$'

(ii) showing distribution equivalence: I need to show that $\mathbf{V} = [\sigma_1, \sigma_2]^T \odot \boldsymbol{\epsilon} + \boldsymbol{\mu}$ has the same distribution as $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\sigma_1^2, \sigma_2^2))$.

Since $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, each component ϵ_i is independent with $\epsilon_i \sim \mathcal{N}(0, 1)$.

For the mean:

$$E[\mathbf{V}] = E[[\sigma_1, \sigma_2]^T \odot \boldsymbol{\epsilon} + \boldsymbol{\mu}] = [\sigma_1, \sigma_2]^T \odot E[\boldsymbol{\epsilon}] + \boldsymbol{\mu} = \mathbf{0} + \boldsymbol{\mu} = \boldsymbol{\mu}$$

For the covariance, since the components are independent:

$$\text{Var}(V_1) = \text{Var}(\sigma_1 \epsilon_1) = \sigma_1^2 \text{Var}(\epsilon_1) = \sigma_1^2$$

$$\text{Var}(V_2) = \text{Var}(\sigma_2 \epsilon_2) = \sigma_2^2 \text{Var}(\epsilon_2) = \sigma_2^2$$

And $\text{Cov}(V_1, V_2) = 0$ since ϵ_1 and ϵ_2 are independent.

So $\mathbf{V}$ has mean $\boldsymbol{\mu}$ and covariance $\text{diag}(\sigma_1^2, \sigma_2^2)$, which is exactly the same as $\mathbf{z}$.

(iii) gradient computation: Given $\widetilde{\mathbf{z}} = [\sigma_1, \sigma_2]^T \odot \widetilde{\boldsymbol{\epsilon}} + \boldsymbol{\mu}$ and gradient $\widetilde{\mathbf{g}} = [\widetilde{g}_1, \widetilde{g}_2]^T$:

Using the chain rule:

$$\frac{\partial \widetilde{z}_1}{\partial \mu_1} = 1, \quad \frac{\partial \widetilde{z}_2}{\partial \mu_2} = 1$$

So:

$$\frac{\partial L}{\partial \boldsymbol{\mu}} = \widetilde{\mathbf{g}}$$

For the standard deviations:

$$\frac{\partial \widetilde{z}_1}{\partial \sigma_1} = \widetilde{\epsilon}_1, \quad \frac{\partial \widetilde{z}_2}{\partial \sigma_2} = \widetilde{\epsilon}_2$$

So:

$$\frac{\partial L}{\partial \sigma_1} = \widetilde{g}_1 \cdot \widetilde{\epsilon}_1, \quad \frac{\partial L}{\partial \sigma_2} = \widetilde{g}_2 \cdot \widetilde{\epsilon}_2$$

(iv) why we sample instead of direct generation: The whole point of sampling with the reparameterization trick is to make the stochastic operation differentiable. If we just directly generated values for $\widetilde{\mathbf{z}}$, we couldn't backpropagate through it because there'd be no connection between the encoder outputs ($\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$) and the sampled values.

By using $\widetilde{\mathbf{z}} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon}$ is the random part, we can compute gradients with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ and train the encoder properly. It's basically a clever way to "move the randomness outside" so the network parameters can still be optimized.

3. Neural Networks (29 pt)

In this problem, you will build a simple neural network framework from scratch, featuring a PyTorch-like API. At the core of popular deep learning frameworks such as PyTorch, TensorFlow, JAX, etc, are the automatic differentiation engines that compute gradients of common functions. During training, the losses are backpropagated through the network, and the resulting gradients are used to update the network parameters. You will implement several common functions, such as sigmoid activation and binary cross-entropy, as well as standard layers, such as fully connected (linear) layers, to build a simple multi-layer perceptron (MLP).

You must submit the following files to **Gradescope**:

- `hw6_q3_autograd.py`
- `hw6_q3_nn.py`
- `hw6_q3_optim.py`
- `hw6_q3_utils.py`

Do not submit the Jupyter notebook `hw6.ipynb` to Gradescope, as it will not be evaluated. The notebook is provided solely for you to test your implementation.

[**Written part**] (10 pt)

(a) **Manual backpropagation.**

To serve as a reference for your implementation, you will first manually perform backpropagation with a simple MLP. Consider a two-layer MLP with first layer weight $\mathbf{w} = [w_0, w_1, w_2]$, second layer weight $\mathbf{h} = [h_0, h_1]$. A ReLU activation is used for the first layer output, while no activation is used for the second layer output. Note that w_0 and h_0 are the bias terms. Each weight is initialized to 1. The dataset consists of 2 input-label pairs: $\{\boldsymbol{x}^{(0)} = [0, 1]^\intercal, y^{(0)} = 1\}$ and $\{\boldsymbol{x}^{(1)} = [1, 0]^\intercal, y^{(1)} = 0\}$. The MLP is trained on this dataset for 2 steps (1st step uses $\boldsymbol{x}^{(0)}$, 2nd step uses $\boldsymbol{x}^{(1)}$) and is evaluated using squared error $L(y, \widehat{y}) = (\widehat{y} - y)^2$. The neural network is optimized with gradient descent with a learning rate of 0.05.

Demonstrate your step-by-step backpropagation calculation for the first 2 training steps. Round your results up to 4 decimal digits.

i. (1.5 pt) First training step: What is the gradient (derivative) for w_0, w_1, w_2, h_0, h_1?

ii. (1.5 pt) First training step: What is the updated value for w_0, w_1, w_2, h_0, h_1?

iii. (1.5 pt) Second training step: What is the gradient (derivative) for w_0, w_1, w_2, h_0, h_1?

iv. (1.5 pt) Second training step: What is the updated value for w_0, w_1, w_2, h_0, h_1?

(b) **Gradient of common functions.**

The sigmoid activation $\sigma(z)$ and the binary cross-entropy loss $L(y, \widehat{y})$ is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \qquad \text{and} \qquad L(y, \widehat{y}) = -y \log \widehat{y} - (1 - y) \log(1 - \widehat{y})$$

where $\widehat{y} = \sigma(f(x))$ is the probability-valued output of neural network f. In practice, the sigmoid and binary cross-entropy loss are fused into one operation for numerical stability and efficiency (i.e, `BCEWithLogitsLoss` in PyTorch). Derive:

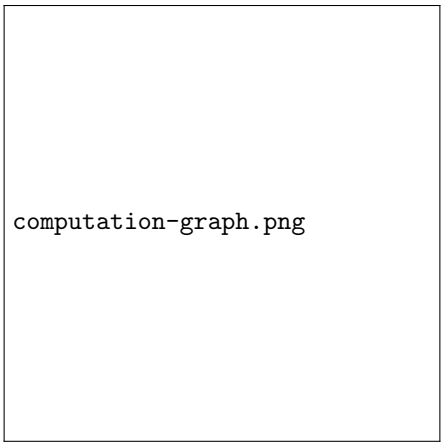
i. (1.5 pt) The gradient of $\sigma(z)$ with respect to its input z.

ii. (2.5 pt) The simplified binary cross-entropy $L(y, \sigma(z))$ with respect to the logit $z = f(x)$ (i.e, raw, unnormalized neural network output) and the gradient of $L(y, \sigma(z))$ with respect to the z.

[**Programming part**] (19 pt)

(c) **Reverse-mode automatic differentiation. Implement** `hw6_q4_autograd.py`.

You are **not allowed** to use any external libraries (Numpy, PyTorch, etc.) for this question. You are **allowed** to use the provided numerically stable `sigmoid` and `log_sigmoid` functions.



For a simple 2-layer MLP, we already have to perform lots of manual computation. In popular neural network frameworks, this gradient computation is usually handled by reverse-mode automatic differentiation. These engines perform gradient calculation by maintaining a "computation graph" (represented by a directed acyclic graph (DAG)) that records all operations that lead to the output. Consider the root node as f, we can do a depth-first traversal and obtain the following order $x_1, x_2, u_1, u_2, u_3, f$. Reversing this order to obtain the backpropagation order $f, u_3, u_2, u_1, x_2, x_1$. We will implement our own automatic differentiation module by chopping the neural network into its smallest unit, which is an individual scalar. The graph traversal mechanism has been provided. You would only need to implement the `forward` and `backward` methods for each operation.

(d) **Neural network. Implement `hw6_q3_nn.py` and `hw6_q3_optim.py`.**
You are **not allowed** to use any external libraries (Numpy, PyTorch, etc.) for this question.

(e) **Fit everything together. Implement `hw6_q4_utils.py`.**
A Jupyter notebook `hw6.ipynb` is provided for you to test your implementation against the PyTorch implementation. Your results should be very close to the PyTorch implementation.

Solution. Part (a): manual backpropagation

I have a two-layer network with weights $\mathbf{w} = [w_0, w_1, w_2]$ and $\mathbf{h} = [h_0, h_1]$, all starting at 1.0. The first layer uses ReLU, second layer has no activation, and I'm using squared error loss with learning rate 0.05.

(i) first training step gradients ($\mathbf{x}^{(0)} = [0, 1]^T, y^{(0)} = 1$):

Forward pass: - $z_1 = w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 1 = 1 + 0 + 1 = 2$ - $a_1 = \text{ReLU}(2) = 2$ (since it's positive) - $\hat{y} = h_0 \cdot 1 + h_1 \cdot a_1 = 1 + 1 \cdot 2 = 3$ - $L = (\hat{y} - y)^2 = (3 - 1)^2 = 4$

Now for backprop: - $\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) = 2(3 - 1) = 4$ - $\frac{\partial L}{\partial h_0} = 4 \cdot 1 = 4$ - $\frac{\partial L}{\partial h_1} = 4 \cdot a_1 = 4 \cdot 2 = 8$ - $\frac{\partial L}{\partial a_1} = 4 \cdot h_1 = 4 \cdot 1 = 4$ - $\frac{\partial L}{\partial z_1} = 4 \cdot 1 = 4$ (ReLU derivative is 1 since $z_1 > 0$) - $\frac{\partial L}{\partial w_0} = 4 \cdot 1 = 4$ - $\frac{\partial L}{\partial w_1} = 4 \cdot x_1^{(0)} = 4 \cdot 0 = 0$ - $\frac{\partial L}{\partial w_2} = 4 \cdot x_2^{(0)} = 4 \cdot 1 = 4$

Gradients: $w_0 : 4.0000, w_1 : 0.0000, w_2 : 4.0000, h_0 : 4.0000, h_1 : 8.0000$

(ii) first training step updates: Using gradient descent with lr = 0.05: - $w_0 = 1 - 0.05 \cdot 4 = 0.8000$ - $w_1 = 1 - 0.05 \cdot 0 = 1.0000$ - $w_2 = 1 - 0.05 \cdot 4 = 0.8000$ - $h_0 = 1 - 0.05 \cdot 4 = 0.8000$ - $h_1 = 1 - 0.05 \cdot 8 = 0.6000$

(iii) second training step gradients ($\mathbf{x}^{(1)} = [1, 0]^T, y^{(1)} = 0$):

Forward pass with the updated weights: - $z_1 = 0.8 \cdot 1 + 1.0 \cdot 1 + 0.8 \cdot 0 = 0.8 + 1.0 + 0 = 1.8$ - $a_1 = \text{ReLU}(1.8) = 1.8$ - $\hat{y} = 0.8 \cdot 1 + 0.6 \cdot 1.8 = 0.8 + 1.08 = 1.88$ - $L = (1.88 - 0)^2 = 3.5344$

Backprop: - $\frac{\partial L}{\partial \hat{y}} = 2(1.88 - 0) = 3.76$ - $\frac{\partial L}{\partial h_0} = 3.76$ - $\frac{\partial L}{\partial h_1} = 3.76 \cdot 1.8 = 6.768$ - $\frac{\partial L}{\partial a_1} = 3.76 \cdot 0.6 = 2.256$ - $\frac{\partial L}{\partial z_1} = 2.256 \cdot 1 = 2.256$ - $\frac{\partial L}{\partial w_0} = 2.256$ - $\frac{\partial L}{\partial w_1} = 2.256 \cdot 1 = 2.256$ - $\frac{\partial L}{\partial w_2} = 2.256 \cdot 0 = 0$

Gradients: $w_0 : 2.2560, w_1 : 2.2560, w_2 : 0.0000, h_0 : 3.7600, h_1 : 6.7680$

(iv) second training step updates: - $w_0 = 0.8 - 0.05 \cdot 2.256 = 0.6872$ - $w_1 = 1.0 - 0.05 \cdot 2.256 = 0.8872$ - $w_2 = 0.8 - 0.05 \cdot 0 = 0.8000$ - $h_0 = 0.8 - 0.05 \cdot 3.76 = 0.6120$ - $h_1 = 0.6 - 0.05 \cdot 6.768 = 0.2616$

Part (b): gradient derivations

(i) sigmoid gradient: For $\sigma(z) = \frac{1}{1+e^{-z}}$, I need to find $\frac{\partial \sigma(z)}{\partial z}$.
Using the quotient rule (or chain rule):

$$\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) = \frac{0 \cdot (1+e^{-z}) - 1 \cdot (-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2}$$

rewritten:

$$= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \sigma(z) \cdot \frac{e^{-z}}{1+e^{-z}}$$

Since $\frac{e^{-z}}{1+e^{-z}} = 1 - \frac{1}{1+e^{-z}} = 1 - \sigma(z)$:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))$$

(ii) binary cross-entropy with sigmoid: The combined loss is:

$$L(y, \sigma(z)) = -y \log(\sigma(z)) - (1-y) \log(1-\sigma(z))$$

Taking the derivative with respect to z:

$$\frac{\partial L}{\partial z} = -y \frac{1}{\sigma(z)} \frac{\partial \sigma(z)}{\partial z} - (1-y) \frac{1}{1-\sigma(z)} \frac{\partial (1-\sigma(z))}{\partial z}$$

The second term: $\frac{\partial (1-\sigma(z))}{\partial z} = -\frac{\partial \sigma(z)}{\partial z} = -\sigma(z)(1-\sigma(z))$
Substituting everything:

$$\frac{\partial L}{\partial z} = -y \frac{\sigma(z)(1-\sigma(z))}{\sigma(z)} - (1-y) \frac{-\sigma(z)(1-\sigma(z))}{1-\sigma(z)}$$

$$= -y(1-\sigma(z)) + (1-y)\sigma(z)$$

$$= -y + y\sigma(z) + \sigma(z) - y\sigma(z) = \sigma(z) - y$$

The result was good. The gradient is just the difference between the prediction and the true label. That's why this combination is so popular in practice.