EECS545 WN23 Machine Learning
Homework #3

Due date: 11:55 PM, Tuesday Feb 21, 2023

Reminder: While you are encouraged to discuss problems in a small group (up to 5 people), you should write your solutions and code independently. Do not share your solution with anyone else in the class. If you worked in a group for the homework, please include the names of your collaborators in the homework submission. Your answers should be as concise and clear as possible. Please address all questions to `http://piazza.com/class#winter2023/eecs545` with a reference to the specific question in the subject line (e.g., Homework 3, Q2(c): foobar).

Submission Instruction: You should submit both **writeup** and **source code**. We may inspect your source code submission visually and rerun the code to check the results. Please be aware your points can be deducted if you don't follow the instructions listed below:

- Submit **writeup** to **Gradescope** (`https://www.gradescope.com/`)
  - Your writeup should contain your answers to **all** questions (typeset or hand-written), except for the implementation-only questions (marked with **(Autograder)**).
  - **For each subquestion, please select the associated pages from the pdf file in Gradescope. The graders are not required to look at pages other than the ones selected, and this may lead to some penalty when grading**.
  - Your writeup should be self-contained and self-explanatory. You should include the plots and figures in your writeup whenever asked, even when they are generated from the ipython notebook.
  - Please typeset (with LaTeX) or hand-write your solutions legibly. **Hand-writing that is difficult to read may lead to penalties**. If you prefer hand-writing, please use scanners or mobile scanning apps that provide shading corrections and projective transformations for better legibility; you should ***not*** just take photos and submit them without any image processing.
- Submit **source code** to **Autograder** (`https://autograder.io/`)
  - Each ipython notebook file (i.e., `*.ipynb`) will walk you through the implementation task, producing the outputs and plots for *all* subproblems. You are required to write code on its matched python file (`*.py`). For instance, if you are working on `soft_margin_svm.ipynb`, you are required to write code on `soft_margin_svm.py`. Note that the outputs of your source code must match with your **writeup**.
  - We will read the data file in the `data` directory (i.e., `data/q4_data/mnist.npz`) **from the same (current) working directory**:
    for example, `X_train = np.load('data/q4_data/mnist.npz')`.
  - When you want to evaluate and test your implementation, please submit the `*.py` and `*.ipynb` files to Autograder for grading your implementations in the middle or after finish everything. You can partially test some of the files anytime, but please note that this also reduces your daily submission quota. You can submit your code up to **five** times per day.

- Your program should run under an environment with following libraries:
  - Python 3.10 [1]
  - NumPy (for implementations of algorithms)
  - Matplotlib (for plots)

  Please do not use any other library unless otherwise instructed (no third-party libraries other than numpy and matplotlib are allowed). You should be able to load the data and execute your code on Autograder with the environment described above, but in your local working environment you might need to install them via `pip install numpy matplotlib`.
- For this assignment, you will need to submit the following files:
  - `soft_margin_svm.py`
  - `soft_margin_svm.ipynb`
  - `two_layer_net.py`
  - `two_layer_net.ipynb`

  Do not change the filename for the code and ipython notebook file because the Autograder may not find your submission. Please do not submit any other files except `*.py` and `*.ipynb` to Autograder.
- When you are done, please upload your work to Autograder (sign in with your UMich account). To receive the full credit, your code must run and terminate without error (i.e., with exit code 0) in the Autograder. Keep all the cell outputs in your notebook files (`*.ipynb`). We strongly recommend you run **Kernel → Restart Kernel and Run All Cells** (in the Jupyter Lab menu) before submitting your code to Autograder.

Credits

Some problems were adopted from Stanford CS229 and Bishop PRML.

Change Log

- rev0 (2023/2/7 1PM): Initial Release
- rev1 (2023/2/8 9:30AM): Update the problem description of Q1 from "if you think it is not, please give a counterexample." to "if you think it is not, please prove it or give a counterexample".
- rev2 (2023/2/11 9:30AM):
  - Q1: Explicitly allowed using the properties proved in the previous or later sub-questions. (Piazza @269)
  - Q3.(b): matched $\eta(j) = 1e^{-3}$ not to confuse with the python file. (Piazza @276)
- rev3 (2023/2/13 11:30PM):
  - Q2: updated Line 1 of Algorithm 1 and 2 can start with any real number, while assuming $\mathbf{w}_0 = \mathbf{0}$ in (a).(ii). (Piazza @289)
  - Q1.(i): Added a comment that we cannot directly apply Mercer's theorem.
- rev4 (2023/2/20 2:30PM): Added more hint for Q5.(a) $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$.

[1] We recommend using Miniconda (`https://docs.conda.io/en/latest/miniconda.html`). You can use other python distributions and versions as long as they are supported, but we will run your program on Python 3.10 on Autograder.

1 [21 + 3 points] Direct Construction of Valid Kernels

In class, we saw that by choosing a kernel $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$, we can implicitly map data to a high dimensional space, and have the SVM algorithm work in that space. One way to generate kernels is to explicitly define the mapping ϕ to a higher dimensional space, and then work out the corresponding k.

However, in this question we are interested in direct construction of kernels. Suppose we have a function $k(\mathbf{x}, \mathbf{z})$ that gives an appropriate similarity measure (similar to inner product) for our learning problem, and consider plugging k into a kernelized algorithm (like SVM) as the kernel function. In order for $k(\mathbf{x}, \mathbf{z})$ to be a valid kernel, it must correspond to an inner product in some higher dimensional space resulting from some feature mapping ϕ. Mercer's theorem tells us that $k(\mathbf{x}, \mathbf{z})$ is a (Mercer) kernel if and only if for any finite set $\left\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\right\}$, the matrix K is symmetric and positive semi-definite, where the square matrix $K \in \mathbb{R}^{N \times N}$ is given by $K_{ij} = k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$.

Now here comes the question: Let k_1, k_2 be kernels over $\mathbb{R}^D \times \mathbb{R}^D$, let $a \in \mathbb{R}^+$ be a positive real number, let $f : \mathbb{R}^D \to \mathbb{R}$ be a real-valued function, let $\phi : \mathbb{R}^D \to \mathbb{R}^M$ be a function mapping from $\mathbb{R}^D$ to $\mathbb{R}^M$, let k_3 be a kernel over $\mathbb{R}^M \times \mathbb{R}^M$, and let $p : \mathbb{R} \to \mathbb{R}$ be a polynomial with positive coefficients.

For each of the functions k below, state whether it is necessarily a kernel. If you think it is a kernel, please prove it; if you think it is not, please prove it or give a counterexample. You can use both definitions (inner prod of feature map or PSD) when proving the validity of the kernel. If you once proved a property in one sub-question (e.g., you proved Q1.(a)), then it is okay to use it in other questions, e.g., Q1.(f), by mentioning that you proved that property in Q1.(a); please make sure to provide the proof at least once. However, you are NOT allowed to use the 'results' from the 'Constructing kernels' slide without further proof.

(a) **[2 points]** $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) + k_2(\mathbf{x}, \mathbf{z})$

(b) **[2 points]** $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) - k_2(\mathbf{x}, \mathbf{z})$

(c) **[2 points]** $k(\mathbf{x}, \mathbf{z}) = -ak_1(\mathbf{x}, \mathbf{z})$

(d) **[2 points]** $k(\mathbf{x}, \mathbf{z}) = f(\mathbf{x})f(\mathbf{z})$

(e) **[3 points]** $k(\mathbf{x}, \mathbf{z}) = k_3(\phi(\mathbf{x}), \phi(\mathbf{z}))$

(f) **[3 points]** $k(\mathbf{x}, \mathbf{z}) = p\left(k_1(\mathbf{x}, \mathbf{z})\right)$

(g) **[4 points]** $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z})k_2(\mathbf{x}, \mathbf{z})$

(h) **[3 points]** For this subproblem you don't have to check if k is a kernel or not. Instead, given a kernel $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z} + 1)^2$, find a feature map ϕ associated with $k(\mathbf{x}, \mathbf{z})$ such that $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$. You may assume $D = 2$ for this subproblem.

(i) **[3 points, extra credit]** Prove that the Gaussian Kernel, $k(\mathbf{x}, \mathbf{z}) = \exp\left(-\|\mathbf{x} - \mathbf{z}\|^2 / 2\sigma^2\right)$ can be expressed as $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$, where $\phi(\cdot)$ is an infinite-dimensional vector. Specifically, find the explicit closed form of an *infinite* dimensional feature vector ϕ. (Note: you cannot directly apply Mercer's theorem here as ϕ is an *infinite* dimensional vector)

[Hint 1: $\|\mathbf{x}-\mathbf{z}\|^2 = \mathbf{x}^\top\mathbf{x}+\mathbf{z}^\top\mathbf{z}-2\mathbf{x}^\top\mathbf{z}$*. It might be useful to consider Power Series:* $\exp(x) = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$*.]*

[Hint 2: You might find the formula in the slide page 16 of Lecture 8 helpful.]

2 [18 points] Kernelizing the Perceptron

Consider a binary classification problem with labels $y \in \{-1, 1\}$. Assume that we have a (pre-processed) dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots (\mathbf{x}^{(N)}, y^{(N)})\}$ where $\mathbf{x}^{(n)} \in \mathbb{R}^{d+1}$ with $x_1^{(n)} = 1, \forall n$. The *perceptron* is a linear classifier, given by $\text{sign}(h(\mathbf{x}; \mathbf{w}))$ where $h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$ and $\mathbf{w} \in \mathbb{R}^{d+1}$ denotes the weights of the perceptron classifier. The perceptron is trained via Algorithm 1 described below.

Essentially, in each iteration, the algorithm picks a random sample from the dataset $(\mathbf{x}^{(n)}, y^{(n)}) \in \mathcal{D}$ and checks if it is misclassified *i.e.,* $\text{sign}(h(\mathbf{x}^{(n)}; \mathbf{w}_t)) \neq y^{(n)}$. This is implemented by checking if $y^{(n)} h(\mathbf{x}^{(n)}; \mathbf{w}_t)$ is negative (Line 5). If a positive example is misclassified, then we update the weights with $+\mathbf{x}^{(n)}$. If a negative example is misclassified, then we update the weights with $-\mathbf{x}^{(n)}$ (Line 6). If the example is correctly classified, nothing is done. This process is repeated for T iterations.

Algorithm 1: Perceptron training algorithm (using raw data $\mathbf{x}$, not feature vectors $\phi(\mathbf{x})$)

```
1  w_0 ∈ R^{d+1};
2  for t = 0 to T − 1 do
3  |   Pick a random training example (x^(n), y^(n)) from D (with replacement)
4  |   h ← w_t^⊤ x^(n)
5  |   if y^(n) h < 0 then
6  |   |   w_{t+1} ← w_t + y^(n) x^(n)
7  |   else
8  |   |   w_{t+1} ← w_t
9  |   end
10 end
11 return w_T
```

Assume that T is large and $\mathcal{D}$ is linearly separable. In 1943, Frank Rosenblatt showed that this simple algorithm converges and outputs a $\mathbf{w}_T$ that is able to perfectly separate the training data $\mathcal{D}$ [2].

What if the dataset $\mathcal{D}$ is **not linearly separable**? In this problem, you'll show that we can *kernelize* the perceptron algorithm. This allows us to operate in a high-dimensional (possibly infinite-dimensional) feature space $\phi(\mathbf{x})$ where we can assume that the data is linearly separable.

Let $\phi : \mathbb{R}^{d+1} \to \mathbb{R}^M$ be a high-dimensional feature transform that is intractable to compute. Assume that $\mathcal{D}$ is separable in the feature space *i.e.,* $\mathcal{D}_\phi = \{(\phi(\mathbf{x}^{(1)}), y^{(1)}), \dots (\phi(\mathbf{x}^{(N)}), y^{(N)})\}$ is linearly separable. Note that the perceptron classifier is now given by $\text{sign}(h(\phi(\mathbf{x}); \mathbf{w}))$ with $h(\phi(\mathbf{x}); \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x})$. Lastly, assume that we have a computationally-tractable kernel $\mathbf{k}$ such that $\mathbf{k}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \phi(\mathbf{x}')$.

(a) Define $\mathbf{\Phi} = \begin{bmatrix} \phi(\mathbf{x}^{(1)})^\top \\ \vdots \\ \phi(\mathbf{x}^{(N)})^\top \end{bmatrix} \in \mathbb{R}^{N \times M}$ as usual.

In this sub-problem, we assume we that we use Algorithm 2 in the feature space for analysis.

[2] https://search.lib.umich.edu/articles/record/cdi_proquest_miscellaneous_82268136

Algorithm 2: Perceptron training algorithm (using feature vectors)

$\mathbf{w}_0 \in \mathbb{R}^M$;
for $t = 0$ *to* $T - 1$ **do**
  Pick a random training example $(\mathbf{x}^{(n)}, y^{(n)})$ from $\mathcal{D}$ (with replacement)
  $h \leftarrow \mathbf{w}_t^\top \phi(\mathbf{x}^{(n)})$
  if $y^{(n)} h < 0$ **then**
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y^{(n)} \phi(\mathbf{x}^{(n)})$
  else
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t$
  end
end
return $\mathbf{w}_T$

(i) **[4 points]** Assuming $\mathbf{w}_t$ is of the form $\Phi^\top \boldsymbol{\alpha}_t$ where $\boldsymbol{\alpha}_t \in \mathbb{R}^N$, show that $\mathbf{w}_{t+1}$ is also of the form $\Phi^\top \boldsymbol{\alpha}_{t+1}$ for some $\boldsymbol{\alpha}_{t+1} \in \mathbb{R}^N$ (Please feel free to introduce the indicator function $\mathbb{I}[\cdot]$. You can also introduce one hot vector notation $\boldsymbol{e}^n$, the n^{th} standard basis vector in $\mathbb{R}^N$, where $e^n_n = 1$ and rest of the elements are zero in $\mathbb{R}^N$).

(ii) **[2 points]** Please show that all intermediate weights $\mathbf{w}_t$ where $0 \leq t \leq T$ (note that this includes the optimal weights $\mathbf{w}_T$), can be expressed as $\Phi^\top \boldsymbol{\alpha}_t$ where $\boldsymbol{\alpha}_t \in \mathbb{R}^N$, assuming $\mathbf{w}_0 = \mathbf{0}$ from this question. *[hint: induction!]*

(b) Note that by means of the previous subproblem, we have showed that the weights of the perceptron are a linear combination of the features.

(i) **[3 points]** Observing your solution for part (a), can you provide an update rule for $\boldsymbol{\alpha}_{t+1}$ from $\boldsymbol{\alpha}_t$ and other known quantities?

(ii) **[3 points]** Show that $h(\phi(\mathbf{x}^{(n)}), \mathbf{w}_t)$ can be expressed entirely in terms of $\boldsymbol{\alpha}_t$ and kernel $\mathbf{k}$ (doesn't involve ϕ or $\mathbf{\Phi}$).

(c) Kernelize the perceptron algorithm.

(i) **[4 points]** Write it exactly in the style of Alg. 2 *i.e.,* each line (wherever applicable) in Alg. 2 should be replaced with its "kernelized" version. The algorithm should be computationally tractable and thus must not contain any ϕ or $\mathbf{\Phi}$ terms, but with $\boldsymbol{\alpha}_0$, $\boldsymbol{\alpha}_t$, $\boldsymbol{\alpha}_{t+1}$, and k.

(ii) **[2 points]** You have trained the kernelized perceptron, following (i), and have a new test data point $\mathbf{x}$. How would you classify those test data point with the kernelized perceptron?

3 [14 points] Implementing Soft Margin SVM by Optimizing Primal Objective

Support Vector Machines (SVM) is a discriminative model for classification. Although it is possible to develop SVMs that do K class classifications, we are going to restrict ourselves to binary classification in this question, where the class label is either $+1$ (positive) or -1 (negative). SVM is not a probabilistic algorithm. In other words, in its usual construction, it does not optimize a probability measure as a likelihood. SVM tries to find the "best" hyperplane that maximally separates the positive class from the negative class.

Recall that the objective function for maximizing the soft margin is equivalent to the following minimization problem:

$$\begin{aligned} &\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi^{(i)} \\ &\text{subject to } y^{(i)}\left(\mathbf{w}^\top\phi(\mathbf{x}^{(i)}) + b\right) \geq 1 - \xi^{(i)},\ \forall i = 1,\ldots,N \\ &\qquad \xi^{(i)} \geq 0,\ \forall i = 1,\ldots,N \end{aligned} \tag{1}$$

The above is known as the primal objective of SVM. Notice the two constraints on the slack variables $\xi^{(i)}$. It means that $\xi^{(i)}$ must satisfy both of those conditions, while minimizing the sum of $\xi^{(i)}$'s times C. The constrained minimization is equivalent to following minimization involving the *hinge loss* term:

$$\min_{\mathbf{w},b} E(\mathbf{w},b), \quad E(\mathbf{w},b) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\max\left(0, 1 - y^{(i)}\left(\mathbf{w}^\top\phi(\mathbf{x}^{(i)}) + b\right)\right) \tag{2}$$

You will be working with minimization of the above objective in this problem.

(a) **[5 points]** Find the derivatives of the loss function $E(\mathbf{w},b)$ with respect to the parameters $\mathbf{w}, b$. Show that:

$$\nabla_{\mathbf{w}} E(\mathbf{w},b) = \mathbf{w} - C\sum_{i=1}^{N}\mathbb{I}\left[y^{(i)}\left(\mathbf{w}^\top\phi(\mathbf{x}^{(i)}) + b\right) < 1\right] y^{(i)}\phi(\mathbf{x}^{(i)}) \tag{3}$$

$$\frac{\partial}{\partial b} E(\mathbf{w},b) = -C\sum_{i=1}^{N}\mathbb{I}\left[y^{(i)}\left(\mathbf{w}^\top\phi(\mathbf{x}^{(i)}) + b\right) < 1\right] y^{(i)}, \tag{4}$$

where $\mathbb{I}[\cdot]$ is the indicator function. If $f(x) = \max(0,x)$, then assume that $\frac{\partial f}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$

(b) **[7 points]** (Autograder) Implement the SVM algorithm using batch gradient descent, following part 1 of `soft_margin_svm.ipynb`. In previous assignments, you have implemented gradient descent while minimizing over one variable. Minimizing over two variables ($\mathbf{w}$, b) is not different. Both gradients are computed from current parameter values, and then parameters are simultaneously updated.[3]

Example pseudocode for optimizing $\mathbf{w}$ and b is given by Algorithm 3 below.

[3]Note: it is a common mistake to implement gradient descent over multiple parameters by updating the first parameter, then computing the derivative w.r.t second parameter using the updated value of the first parameter. In fact, updating one parameter then computing the gradient of the second parameter using the updated value of the first parameter, is a different optimization algorithm, known as Coordinate Descent.

Algorithm 3: SVM Batch Gradient Descent

$\mathbf{w}^* \leftarrow 0$
$b^* \leftarrow 0$
for $j = 1$ *to* *NumEpochs* **do**
  $\mathbf{w}_{grad} \leftarrow \nabla_{\mathbf{w}} E(\mathbf{w}^*, b^*)$
  $b_{grad} \leftarrow \frac{\partial}{\partial b} E(\mathbf{w}^*, b^*)$
  $\mathbf{w}^* \leftarrow \mathbf{w}^* - \eta(j)\mathbf{w}_{grad}$
  $b^* \leftarrow b^* - \eta(j) b_{grad}$
end
return $\mathbf{w}^*$, b^*

Throughout this question, we will set C in the previous question to $C = 5$, NumEpochs $= 100$ and the learning rate η as a constant. i.e., $\eta(j) = 1e^{-3}$).

(Remarks: In this problem, we only asked you to implement batch gradient descent for primal SVM, but it's quite straightforward to implement stochastic gradient descent in a very similar way (although you may need to decay the learning rate and set the initial learning rate properly). Unlike stochastic gradient descent, we don't need to decay the learning rate for convergence.)

(c) [**2 points**] Run gradient descent over the training data 5 times, once for each of the NumEpochs=[1, 3, 10, 30, 100]. For each run, please report your trained parameters $(\mathbf{w}, b)$ and the test classification accuracy in your **writeup**.

4 [20 points] Asymmetric Cost SVM

Consider applying an SVM to a supervised learning problem where the cost of a false positive (mistakenly predicting +1 when the label is -1) is different from the cost of a false negative (predicting -1 when the label is +1). The asymmetric cost SVM models these asymmetric costs by posing the following optimization problem:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C_0 \sum_{i:y^{(i)}=-1} \xi^{(i)} + C_1 \sum_{i:y^{(i)}=1} \xi^{(i)}$$

s.t.

$$y^{(i)}(\mathbf{w}^\top\phi(\mathbf{x}^{(i)}) + b) \geq 1 - \xi^{(i)}, \forall i = 1, \ldots, N$$
$$\xi^{(i)} \geq 0, \forall i = 1, \ldots, N$$

Here, C_0 is the cost of a false positive; C_1 is the cost of a false negative. (Both C_0 and C_1 are fixed, known, constants.)

(a) **[4 points]** We will find the dual optimization problem. First, write down the Lagrangian. Use $\alpha^{(i)}$ and $\mu^{(i)}$ to denote the Lagrange multipliers corresponding to the two constraints ($\alpha^{(i)}$ for the first constraint, and $\mu^{(i)}$ for the second constraint (slack variables)) in the primal optimization problem above.

$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) =$

(b) **[6 points]** Calculate the following derivatives with respect to the primal variables:

$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) =$

$\frac{\partial\mathcal{L}(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\mu})}{\partial b} =$

$\nabla_{\boldsymbol{\xi}}\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) =$
[Hint] You can define $C^{(i)}$ as $C^{(i)} = C_0\mathbb{I}[y^{(i)} = -1] + C_1\mathbb{I}[y^{(i)} = 1]$.

(c) **[10 points]** Find the dual optimization problem. You should write down the dual optimization problem in the following form. Try to simplify your answer as much as possible. In particular, to obtain full credit, the Lagrange multipliers $\mu^{(i)}$ should not appear in your simplified form of the dual. [Hint: Refer to the KKT conditions.]

$\max_{\boldsymbol{\alpha}} W(\boldsymbol{\alpha}) =$

s.t.

5 [27 points] Neural Network Layer Implementation

In this problem, you will get the opportunity to implement various neural network layers. $\mathbf{X}, \mathbf{Y}$ will represent the input and the output of the layers, respectively. For this problem, we use the row-major notation here (i.e. each row of $\mathbf{X}$ and $\mathbf{Y}$ corresponds to one data sample) to be compatible with the multi-dimensional array structure of Numpy. Furthermore, $\mathcal{L}$ is the scalar valued loss function of $\mathbf{Y}$.

For the Fully-Connected Layer and ReLU, you should include your derivation of the gradient in your written solution.

Important Note: In this question, any indices involved (i, j, etc.) in the mathematical notation start from 1, and not 0. In your code, however, you should use 0-based indexing (standard Numpy notation).

(a) **[8 points] Fully-Connected Layer**
In this question, you will be deriving the **gradient** of a fully-connected layer. For this problem, consider $\mathbf{X} \in \mathbb{R}^{N \times D_{\text{in}}}$, where N is the number of samples in a batch. Additionally, consider a dense layer with weight parameters of the form of $\mathbf{W} \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ and bias parameters of the form of $\mathbf{b} \in \mathbb{R}^{1 \times D_{\text{out}}}$.

As we saw in the lecture, we can compute the output of the layer through a simple matrix multiply operation. Concretely, this can be seen as $\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{B}$. In this equation, we denote the output matrix as $\mathbf{Y} \in \mathbb{R}^{N \times D_{\text{out}}}$ and the bias matrix as $\mathbf{B} \in \mathbb{R}^{N \times D_{\text{out}}}$ where each row of $\mathbf{B}$ is the vector $\mathbf{b}$. (Please note that we are using row-vector notation here to make it compatible with Numpy/PyTorch, but the lecture slides used column-vector notation, which is standard notation for most ML methods. We hope that the distinction is clear from the context.)

Please note, that the matrix multiply operation stated above is generally useful to compute batch outputs (i.e. simultaneously computing the output of N samples in the batch). However, for the purposes of computing the gradient, you might first want to consider the single-sample output operation of this fully-connected layer, which can be stated in row-major notation as $\mathbf{y}^{(n)} = \mathbf{x}^{(n)}\mathbf{W} + \mathbf{b}$ where $\mathbf{y}^{(n)} \in \mathbb{R}^{1 \times D_{out}}$ and $\mathbf{x}^{(n)} \in \mathbb{R}^{1 \times D_{in}}$ for $1 \leq n \leq N$. Here, the index "(n)" in the superscript denotes n-th example, not the layer index, and $X_i^{(n)} = X_{n,i}$ denotes i-th dimension of n-th example $\mathbf{x}^{(n)}$.

Note: it might be fruitful for you to consider this expression as a summation and then calculate the gradient for individual parameters for a single-example case. After this, you can try to extend it to a vector/matrix format for the involved parameters.

Now, **compute the partial derivatives (in matrix form)** $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} \triangleq \nabla_{\mathbf{W}} \mathcal{L} \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$, $\frac{\partial \mathcal{L}}{\partial b} \triangleq \nabla_b \mathcal{L} \in \mathbb{R}^{1 \times D_{\text{out}}}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{X}} \triangleq \nabla_{\mathbf{X}} \mathcal{L} \in \mathbb{R}^{N \times D_{\text{in}}}$ **in terms of** $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \triangleq \nabla_{\mathbf{Y}} \mathcal{L} \in \mathbb{R}^{N \times D_{\text{out}}}$.[4] For technical correctness, you might want to start with writing the gradient with non-vectorized form involving $\frac{\partial \mathcal{L}}{\partial Y_j^{(n)}} \in \mathbb{R}$, where $\frac{\partial \mathcal{L}}{\partial Y_j^{(n)}}$ is the gradient with respect to j-th element of n-th sample in $\mathbf{Y}$ with $1 \leq n \leq N$ and $1 \leq j \leq D_{\text{out}}$.

Hint for $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$
Please note that we use a "matrix/vector" notation $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ to denote a matrix in $\mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ where the element in i-th row and j-th column is $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$ and $W_{i,j}$ is the i-th row, j-th column element of $\mathbf{W}$ with $1 \leq i \leq D_{\text{in}}$ and $1 \leq j \leq D_{\text{out}}$. Here, you would probably want to calculate the gradient $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$ using the formula $\frac{\partial \mathcal{L}}{\partial W_{i,j}} = \sum_{n=1}^{N} \sum_{m=1}^{D_{\text{out}}} \frac{\partial \mathcal{L}}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial W_{i,j}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial Y_j^{(n)}} \frac{\partial Y_j^{(n)}}{\partial W_{i,j}}$ (note: for the last expression, you can use the fact that $W_{i,j}$ will only affect $Y_j^{(n)}$'s, for $n = 1, ..., N$, i.e., the summation over m can be removed by fixing $m = j$; you don't need to prove this.), and then vectorize $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$ to get $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$.

[4] $\triangleq$ means 'equal to by definition'

Hint for $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$
Please note that $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$ is a vector in $\mathbb{R}^{1 \times D_{\text{out}}}$. Ideally, you might want to start by using the formula $\frac{\partial \mathcal{L}}{\partial b_j} = \sum_{n=1}^{N} \sum_{m=1}^{D_{\text{out}}} \frac{\partial \mathcal{L}}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial b_j}$ and then move on to derive $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$.

Hint for $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$
Please note that $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ is a matrix in $\mathbb{R}^{N \times D_{in}}$. In order to derive this gradient, you can start by using the formula $\frac{\partial \mathcal{L}}{\partial X_i^{(n)}} = \sum_{n'=1}^{N} \sum_{m=1}^{D_{\text{out}}} \frac{\partial \mathcal{L}}{\partial Y_m^{(n')}} \frac{\partial Y_m^{(n')}}{\partial X_i^{(n)}}$ Following this, you can say that $\frac{\partial \mathcal{L}}{\partial X_i^{(n)}} = \sum_{m=1}^{D_{\text{out}}} \frac{\partial \mathcal{L}}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial X_i^{(n)}}$ because the n-th sample in X is only related with the n-th sample in $\mathbf{Y}$ and $\frac{Y_m^{(n')}}{\partial X_i^{(n)}}$=0 for all $n' \neq n$.

(b) **[3 points] ReLU**

Let $\mathbf{X} \in \mathbb{R}^{N \times D}$ be a 2-D array and $\mathbf{Y} = \text{ReLU}(\mathbf{X})$. In this case, ReLU is applied to $\mathbf{X}$ in an element-wise fashion. For an element $x \in \mathbf{X}$, the corresponding output is $y = \text{ReLU}(x) = \max(0, x)$. You can observe that $\mathbf{Y}$ will have the same shape as $\mathbf{X}$. **Express $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ in terms of $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$**, where $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ has the same shape as $\mathbf{X}$.

Hint: you may need to use the element-wise product to express $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$. Please feel free to introduce the indicator function $\mathbb{I}[\cdot]$.

(c) **[14 points]** (Autograder) Now, we will implement the corresponding forward and backward passes in `two_layer_net.py`. More importantly, please ensure that you use vectorization to make your layer implementations as efficient as possible. You should use `two_layer_net.ipynb` to help you in checking the correctness of the your implementation. Note that for each layer, the ipython notebook just checks the result for *one specific example*. Passing these tests does not necessarily mean that your implementation is 100% correct. Once you implement the layers properly, please implement two fully-connected layer based classifier with a Softmax loss in `two_layer_net.py` and test it on MNIST dataset, following the guide in `two_layer_net.ipynb`. All your implementations should go to `two_layer_net.py`.

(d) **[1 points]** Please report the training loss with train/test accuracy plot of MNIST dataset, stored as `two_layer_net.png` from `two_layer_net.ipynb`, in your **writeup**.

(e) **[1 points]** Please report the accuracy obtained on the test set of MNIST dataset in your **writeup**.