CS 229, Winter 2025
Problem Set #2

Due Wednesday, February 5 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible.

(2) If you have a question about this homework, we encourage you to post your question on our Ed forum, at `https://edstem.org/us/courses/37893/discussion/`.

(3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work.

(4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted.

(5) The due date is Wednesday, February 5 at 11:59 pm. If you submit after Wednesday, February 5 at 11:59 pm, you will begin consuming your late days. The late day policy can be found in the course website: Course Logistics and FAQ.

All students must submit an electronic PDF version of the written question including plots generated from the codes. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup. Please make sure that your PDF file and zip file are submitted to the corresponding Gradescope assignments respectively. We reserve the right to not give any points to the written solutions if the associated code is not submitted.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solution independently, and without referring to written notes from the joint session. Each student must understand the solution well enough in order to reconstruct it by him/herself. It is an honor code violation to copy, refer to, or look at written or code solutions from a previous year, including but not limited to: official solutions from a previous year, solutions posted online, and solutions you or someone else may have written up in a previous year. Furthermore, it is an honor code violation to post your assignment solutions online, such as on a public git repo. We run plagiarism-detection software on your code against past solutions as well as student submissions from previous years. Please take the time to familiarize yourself with the Stanford Honor Code[1] and the Stanford Honor Code[2] as it pertains to CS courses.

[1] https://communitystandards.stanford.edu/policies-and-guidance/honor-code
[2] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164/handouts/honor-code.pdf

1. **[35 points] Implicit Regularization**

Recall that in the overparameterized regime (where the number of parameters is larger than the number of samples), typically there are infinitely many solutions that can fit the training dataset perfectly, and many of them cannot generalize well (that is, they have large validation errors). However, in many cases, the particular optimizer we use (e.g., GD, SGD with particular learning rates, batch sizes, noise, etc.) tends to find solutions that generalize well. This phenomenon is called implicit regularization effect (also known as algorithmic regularization or implicit bias).

In this problem, we will look at the implicit regularization effect on two toy examples in the overparameterized regime: linear regression and a quadratically parameterized model. For linear regression, we will show that gradient descent with zero initialization will always find the minimum norm solution (instead of an arbitrary solution that fits the training data), and in practice, the minimum norm solution tends to generalize well. For a quadratically parameterized model, we will show that initialization and batch size also affect generalization.

(a) [3 points] Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \leq i \leq n$. We assume the dataset is generated by a linear model without noise. That is, there is a vector $\beta^\star \in \mathbb{R}^d$ such that $y^{(i)} = (\beta^\star)^\top x^{(i)}$ for all $1 \leq i \leq n$. Let $X \in \mathbb{R}^{n \times d}$ be the matrix representing the inputs (i.e., the i-th row of X corresponds to $x^{(i)}$)) and $\vec{y} \in \mathbb{R}^n$ the vector representing the labels (i.e., the i-th row of $\vec{y}$ corresponds to $y^{(i)}$)):

$$X = \begin{bmatrix} - & x^{(1)\top} & - \\ - & x^{(2)\top} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)\top} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Then in matrix form, we can write $\vec{y} = X\beta^\star$. We assume that the number of examples is less than the number of parameters (that is, $n < d$).

We use the least-squares cost function to train a linear model:

$$J(\beta) = \frac{1}{2n}\|X\beta - \vec{y}\|_2^2. \tag{1}$$

In this sub-question, we characterize the family of global minimizers to Eq. (1). We assume that $XX^\top \in \mathbb{R}^{n \times n}$ is an invertible matrix. **Prove that** β achieves zero cost in Eq. (1) if and only if

$$\beta = X^\top (XX^\top)^{-1}\vec{y} + \zeta \tag{2}$$

for some ζ in the subspace orthogonal to all the data (that is, for some ζ such that $\zeta^\top x^{(i)} = 0, \forall 1 \leq i \leq n$.)

Note that this implies that there is an infinite number of β's such that Eq. (1) is minimized. We also note that $X^\top (XX^\top)^{-1}$ is the pseudo-inverse of X, but you don't necessarily need this fact for the proof.

(b) [3 points] We still work with the setup of part (a). Among the infinitely many optimal solutions of Eq. (1), we consider the *minimum norm* solution. Let $\rho = X^\top (XX^\top)^{-1}\vec{y}$. In the setting of (a), **prove that** for any β such that $J(\beta) = 0$, $\|\rho\|_2 \leq \|\beta\|_2$. In other words, ρ is the minimum norm solution.

Hint: As a intermediate step, you can prove that for any β in the form of Eq. (2),

$$\|\beta\|_2^2 = \|\rho\|_2^2 + \|\zeta\|_2^2.$$

(c) [5 points] **Coding question: minimum norm solution generalizes well**
For this sub-question, we still work with the setup of parts (a) and (b). We use the following datasets:

`src/implicitreg/ds1_train.csv,ds1_valid.csv`

Each file contains $d+1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.
Using the formula in sub-question (b), **compute** the minimum norm solution using the training dataset. Then, **generate** three other different solutions with zero costs and different norms using the formula in sub-question (a). The starter code is in `src/implicitreg/linear.py`. **Plot** the validation error of these solutions (including the minimum norm solution) in a scatter plot. Use the norm of the solutions as x-axis, and the validation error as y-axis. For your convenience, the plotting function is provided as the method `generate_plot` in the starter code. Your plot is expected to demonstrate that the minimum norm solution generalizes well.

(d) [5 points] For this sub-question, we work with the setup of part (a) and (b). In this sub-question, you will prove that the gradient descent algorithm with *zero initialization* always converges to the minimum norm solution. Let $\beta^{(t)}$ be the parameters found by the GD algorithm at time step t. Recall that at step t, the gradient descent algorithm update the parameters in the following way

$$\beta^{(t)} = \beta^{(t-1)} - \eta \nabla J(\beta^{(t-1)}) = \beta^{(t-1)} - \frac{\eta}{n} X^\top (X\beta^{(t-1)} - \vec{y}). \tag{3}$$

As in sub-question (a), we also assume $XX^\top$ is an invertible matrix. **Prove** that if the GD algorithm with zero initialization converges to a solution $\hat{\beta}$ satisfying $J(\hat{\beta}) = 0$, then $\hat{\beta} = X^\top (XX^\top)^{-1}\vec{y} = \rho,$, that is, $\hat{\beta}$ is the minimum norm solution.
Hint: As a first step, you can prove by induction that if we start with zero initialization, $\beta^{(t)}$ will always be a linear combination of $\{x^{(1)}, x^{(2)}, \cdots, x^{(n)}\}$ for any $t \geq 0$. Then, for any $t \geq 0$, you can write $\beta^{(t)} = X^\top v^{(t)}$ for some $v^{(t)} \in \mathbb{R}^n$. As a second step, you can prove that if $\hat{\beta} = X^\top v^{(t)}$ for some $v^{(t)}$ and $J(\hat{\beta}) = 0$, then we have $\hat{\beta} = \rho$.
You don't necessarily have to follow the steps in this hint. But if you use the hint, you need to prove the statements in the hint.

(e) [3 points] In the following sub-questions, we consider a slightly more complicated model called quadratically parameterized model. A quadratically parameterized model has two sets of parameters $\theta, \phi \in \mathbb{R}^d$. Given a d-dimensional input $x \in \mathbb{R}^d$, the output of the model is

$$f_{\theta,\phi}(x) = \sum_{k=1}^{d} \theta_k^2 x_k - \sum_{k=1}^{d} \phi_k^2 x_k. \tag{4}$$

Note that $f_{\theta,\phi}(x)$ is linear in its input x, but non-linear in its parameters θ, ϕ. Thus, if the goal was to learn the function, one should simply just re-parameterize it with a linear model and use linear regression. However, here we insist on using the parameterization above in Eq. (4) in order to study the implicit regularization effect in models that are nonlinear in the parameters.
Notations: To simplify the equations, we define the following notations. For a vector $v \in \mathbb{R}^d$, let $v^{\odot 2}$ be its element-wise square (that is, $v^{\odot 2}$ is the vector $[v_1^2, v_2^2, \cdots, v_d^2] \in \mathbb{R}^d$.) For two

vectors $v, w \in \mathbb{R}^d$, let $v \odot w$ be their element-wise product (that is, $v \odot w$ is the vector $[v_1 w_1, v_2 w_2, \cdots, v_d w_d] \in \mathbb{R}^d$.) Then our model can be written as

$$f_{\theta,\phi}(x) = x^\top(\theta^{\odot 2} - \phi^{\odot 2}). \tag{5}$$

Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \le i \le n$, and

$$y^{(i)} = (x^{(i)})^\top((\theta^\star)^{\odot 2} - (\phi^\star)^{\odot 2})$$

for some $\theta^\star, \phi^\star \in \mathbb{R}^d$. Similarly, we use $X \in \mathbb{R}^{n \times d}$ and $\vec{y} \in \mathbb{R}^n$ to denote the matrix/vector representing the inputs/labels respectively:

$$X = \begin{bmatrix} - & x^{(1)\top} & - \\ - & x^{(2)\top} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)\top} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Let $J(\theta, \phi) = \frac{1}{4n}\sum_{i=1}^n (f_{\theta,\phi}(x^{(i)}) - y^{(i)})^2$ be the cost function.

First, when $n < d$ and $XX^\top$ is invertible, **prove** that there exists infinitely many optimal solutions with zero cost.

Hint: Find a mapping between the parameter β in linear model and the parameter θ, ϕ in quadratically parameterized model. Then use the conclusion in sub-question (a).

(f) [10 points] **Coding question: implicit regularization of initialization**

We still work with the setup in part (e). For this sub-question, we use the following datasets:

`src/implicitreg/ds2_train.csv,ds2_valid.csv`

Each file contains $d + 1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.

First of all, the gradient of the loss has the following form:

$$\nabla_\theta J(\theta, \phi) = \frac{1}{n}\sum_{i=1}^n ((x^{(i)})^\top(\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\theta \odot x^{(i)}), \tag{6}$$

$$\nabla_\phi J(\theta, \phi) = -\frac{1}{n}\sum_{i=1}^n ((x^{(i)})^\top(\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\phi \odot x^{(i)}). \tag{7}$$

You don't need to prove these two equations. They can be verified directly using the chain rule.

Using the formula above, run gradient descent with initialization $\theta = \alpha\mathbf{1}, \phi = \alpha\mathbf{1}$ with $\alpha \in \{0.1, 0.03, 0.01\}$ (where $\mathbf{1} = [1, 1, \cdots, 1] \in \mathbb{R}^d$ is the all-1's vector) and learning rate 0.08. We provide the starter code in `src/implicitreg/qp.py`. **Plot** the curve of training error and validation error with different α. Use the number of gradient steps as x-axis, and training/validation error as y-axis. Include your plot in the writeup and **answer** the following two questions based on your plot: which models can fit the training set? Which initialization achieves the best validation error?

Remark: Your plot is expected to demonstrate that the initialization plays an important role in the generalization performance—different initialization can lead to different global minimizers with different generalization performance. In other words, the initialization has an implicit regularization effect.

(g) [6 points] **Coding question: implicit regularization of batch size**

We still work with the setup in part (e). For this sub-question, we use the same dataset and starter code as in sub-question (f). We will show that the noise in the training process also induces implicit regularization. In particular, the noise introduced by *stochastic* gradient descent in this case helps generalization. **Implement** the SGD algorithm, and **plot** the training and validation errors with batch size $\{1, 5, 40\}$, learning rate 0.08, and initialization $\alpha = 0.1$. Similarly, use the number of gradient steps as x-axis, and training/validation error as y-axis. For simplicity, the code for selecting a batch of examples is already provided in the starter code. **Compare** the results with those in sub-question (f) with the same initialization. Does SGD find a better solution?

Your plot is expected to show that the stochasticity in the training process is also an important factor in the generalization performance — in our setting, SGD finds a solution that generalizes better. In fact, a conjecture is that stochasticity in the optimization process (such as the noise introduced by a small batch size) helps the optimizer to find a solution that generalizes better. This conjecture can be proved in some simplified cases, such as the quadratically parameterized model in this sub-question (adapted from the paper HaoChen et al., 2020), and can be observed empirically in many other cases.

2. **[12 points] Double Descent on Linear Models**

Background: In this question, you will empirically observe the sample-wise double descent phenomenon. That is, the validation losses of some learning algorithms or estimators do not monotonically decrease as we have more training examples, but instead have a curve with two U-shaped parts. The double descent phenomenon can be observed even for simple linear models. In this question, we consider the following setup. Let $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ be the training dataset. Let $X \in \mathbb{R}^{n \times d}$ be the matrix representing the inputs (i.e., the i-th row of X corresponds to $x^{(i)}$)), and $\vec{y} \in \mathbb{R}^n$ the vector representing the labels (i.e., the i-th row of $\vec{y}$ corresponds to $y^{(i)}$)):

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Similarly, we use $X_v \in \mathbb{R}^{m \times d}, \vec{y}_v \in \mathbb{R}^m$ to represent the validation dataset, where m is the size of the validation dataset. We assume that the data are generated with $d = 500$.

In this question, we consider *regularized* linear regression. For a regularization level $\lambda \geq 0$, define the regularized cost function

$$J_\lambda(\beta) = \frac{1}{2}\|X\beta - \vec{y}\|_2^2 + \frac{\lambda}{2}\|\beta\|_2^2,$$

and its minimizer $\hat{\beta}_\lambda = \arg\min_{\beta \in \mathbb{R}^d} J_\lambda(\beta)$.

(a) [2 points] In this sub-question, we derive the closed-form solution of $\hat{\beta}_\lambda$. **Prove** that when $\lambda > 0$,

$$\hat{\beta}_\lambda = (X^\top X + \lambda I_{d\times d})^{-1} X^\top \vec{y} \tag{8}$$

(recall that $I_{d\times d} \in \mathbb{R}^{d\times d}$ is the identity matrix.)

Note: $\lambda = 0$ is a special case here. When $\lambda = 0$, $(X^\top X + \lambda I_{d\times d})$ could be singular. Therefore, there might be more than one solutions that minimize $J_0(\beta)$. In this case, we define $\hat{\beta}_0$ in the following way:

$$\hat{\beta}_0 = (X^\top X)^+ X^\top \vec{y}. \tag{9}$$

where $(X^\top X)^+$ denotes the Moore-Penrose pseudo-inverse of $X^\top X$. You don't need to prove the case when $\lambda = 0$, but this definition is useful in the following sub-questions.

(b) [5 points] **Coding question: the double descent phenomenon for unregularized models**

In this sub-question, you will empirically observe the double descent phenomenon.You are given 13 training datasets of sample sizes $n = 200, 250, \ldots, 750$, and 800, and a validation dataset, located at

- `src/doubledescent/train200.csv`, `train250.csv`, etc.
- `src/doubledescent/validation.csv`

For each training dataset $(X, \vec{y})$, compute the corresponding $\hat{\beta}_0$, and evaluate the mean squared error (MSE) of $\hat{\beta}_0$ on the validation dataset. The MSE for your estimators $\hat{\beta}$ on a validation dataset $(X_v, \vec{y}_v)$ of size m is defined as:

$$\text{MSE}(\hat{\beta}) = \frac{1}{2m}\|X_v\hat{\beta} - \vec{y}_v\|_2^2.$$

Complete the `regression` method of `src/doubledescent/doubledescent.py` which takes in a training file and a validation file, and computes $\hat{\beta}_0$. You can use `numpy.linalg.pinv` to compute the pseudo-inverse.

In your writeup, include a line plot of the validation losses. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. You should observe that the validation error increases and then decreases as we increase the sample size.

Note: When $n \approx d$, the test MSE could be very large. For better visualization, it is okay if the test MSE goes out of scope in the plot for some points.

(c) [5 points] **Coding question: double descent phenomenon and the effect of regularization.**

In this sub-question, we will show that regularization mitigates the double descent phenomenon for linear regression. We will use the same datasets as specified in sub-question (b). Now consider using various regularization strengths. For $\lambda \in \{0, 1, 5, 10, 50, 250, 500, 1000\}$, you will compute the minimizer of $J_\lambda(\beta)$.

Complete the `ridge_regression` method of `src/doubledescent/doubledescent.py` which takes in a training file and a validation file, computes the $\hat{\beta}_\lambda$ that minimizes the training objective under different regularization strengths, and returns a list of validation errors (one for each choice of λ).

In your writeup, include a plot of the validation losses of these models. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. Draw one line for each choice of λ connecting the validation errors across different training dataset sizes. Therefore, the plot should contain 8×13 points and 8 lines connecting them.

You should observe that for some small λ's, the validation error may increase and then decrease as we increase the sample size. However, double descent does not occur for a relatively large λ.

Remark: If you want to learn more about the double descent phenomenon and the effect of regularization, you can start with this paper Nakkiran, et al. 2020.

3. **[25 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(n)}\}$ be a dataset of n samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

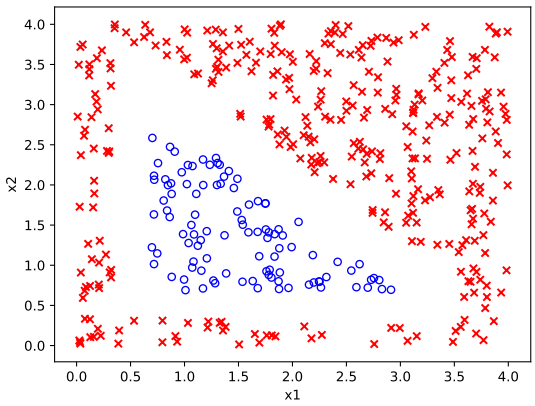

Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "×" and examples in class 0 are marked as "○". We want to perform binary classification using a simple neural network with the architecture shown in Figure 2:

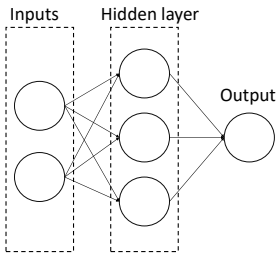

Figure 2: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{n} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

(b) [10 points] Now, suppose instead of using the sigmoid function for the activation function for h_1, h_2, h_3 and o, we instead used the step function $f(x)$, defined as

$$f(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$$

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_step_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `step_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint 1: There are three sides to a triangle, and there are three neurons in the hidden layer.

Hint 2: A solution can be found where all weight and bias parameters take values only in $\{-1, -0.5, 0, 1, 3, 4\}$. You are free to come up with other solutions as well.

(c) [10 points] Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same step function as before.

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_linear_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `linear_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint: The hints from the previous sub-question might or might not apply.

4. **[36 points] Neural Networks: MNIST image classification**

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images are shown below.

The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv`
- `src/mnist/labels_train.csv`
- `src/mnist/images_test.csv`
- `src/mnist/labels_test.csv`

The starter code splits the set of 60,000 training images and labels into a set of 50,000 examples as the training set, and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. You will use the sigmoid function as activation for the hidden layer and use the cross-entropy loss for multi-class classification. Recall that for a single example (x, y), the cross entropy loss is:

$$\ell_{\text{CE}}(\bar{h}_\theta(x), y) = -\log\left(\frac{\exp(\bar{h}_\theta(x)_y)}{\sum_{s=1}^{k}\exp(\bar{h}_\theta(x)_s)}\right),$$

where $\bar{h}_\theta(x) \in \mathbb{R}^k$ is the logits, i.e., the output of the the model on a training example x, $\bar{h}_\theta(x)_y$ is the y-th coordinate of the vector $\bar{h}_\theta(x)$ (recall that $y \in \{1, \ldots, k\}$ and thus can serve as an index.)

We have labeled data $(x^{(i)}, y^{(i)})_{i=1}^n$, where $x^{(i)} \in \mathbb{R}^d$, and $y^{(i)} \in \{1, \ldots, k\}$ is ground truth label. Note that, in this specific application, we pass in the *flattened* images as input. In other words, if each original input image is in $\mathbb{R}^{r \times r}$, we have $x^{(i)} \in \mathbb{R}^{r^2}$.

Additionally, let m be the number of hidden units in the neural network, so that weight matrices $W^{[1]} \in \mathbb{R}^{d \times m}$ and $W^{[2]} \in \mathbb{R}^{m \times k}$.[3] We also have biases $b^{[1]} \in \mathbb{R}^m$ and $b^{[2]} \in \mathbb{R}^k$. The parameters of the model θ is $(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]})$.

We define the following neural network with a single hidden layer, which we refer to as the **Linear Layer NN**. For a single input $x^{(i)}$, the forward propagation equations are:

$$\begin{aligned} a^{(i)} &= \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^m \\ \bar{h}_\theta(x^{(i)}) &= {W^{[2]}}^\top a^{(i)} + b^{[2]} \in \mathbb{R}^k \\ h_\theta(x^{(i)}) &= \text{softmax}(\bar{h}_\theta(x^{(i)})) \in \mathbb{R}^k \end{aligned}$$

where σ is the sigmoid function.

For n training examples, we average the cross entropy loss over the n examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^n \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)}) = -\frac{1}{n}\sum_{i=1}^n \log\left(\frac{\exp(\bar{h}_\theta(x^{(i)})_{y^{(i)}})}{\sum_{s=1}^k \exp(\bar{h}_\theta(x^{(i)})_s)}\right).$$

Suppose $e_y \in \mathbb{R}^k$ is the one-hot embedding/representation of the discrete label y, where the y-th entry is 1 and all other entries are zeros. We can also write the loss function in the following way:

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = -\frac{1}{n}\sum_{i=1}^n e_{y^{(i)}}^\top \log\left(h_\theta(x^{(i)})\right).$$

Here $\log(\cdot)$ is applied entry-wise to the vector $h_\theta(x^{(i)})$. The starter code already converts labels into one-hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. Concretely, we randomly sample B examples $(x^{(i_k)}, y^{(i_k)})_{k=1}^B$ from $(x^{(i)}, y^{(i)})_{i=1}^n$. In this case, the mini-batch cost function with batch-size B is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{k=1}^B \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)})$$

where B is the batch size, i.e., the number of training examples in each mini-batch.

(a) **[5 points]**
Let $t \in \mathbb{R}^k, y \in \{1, \ldots, k\}$ and $p = \text{softmax}(t)$. Prove that

$$\frac{\partial \ell_{\text{CE}}(t, y)}{\partial t} = p - e_y \in \mathbb{R}^k, \tag{10}$$

[3] Please note that the dimension of the weight matrices is different from those in the lecture notes, but we also multiply ${W^{[1]}}^\top$ instead of $W^{[1]}$ in the matrix multiplication layer. Such a change of notation is mostly for some consistence with the convention in the code.

where $e_y \in \mathbb{R}^k$ is the one-hot embedding of y, (where the y-th entry is 1 and all other entries are zeros.) As a direct consequence,

$$\frac{\partial \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)})}{\partial \bar{h}_\theta(x^{(i)})} = \text{softmax}(\bar{h}_\theta(x^{(i)})) - e_{y^{(i)}} = h_\theta(x^{(i)}) - e_{y^{(i)}} \in \mathbb{R}^k \tag{11}$$

where $\bar{h}_\theta(x^{(i)}) \in \mathbb{R}^k$ is the input to the softmax function, i.e.

$$h_\theta(x^{(i)}) = \text{softmax}(\bar{h}_\theta(x^{(i)}))$$

(Note: in deep learning, $\bar{h}_\theta(x^{(i)})$ is sometimes referred to as the "logits".)

(b) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function $J_{MB} = \frac{1}{B}\sum_{k=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)})$. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.

Also, at the end of 30 epochs, save the learnt parameters (i.e., all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise. For better vectorization, use one-hot label encodings in the code (e_y in part (a)).

(c) **[7 points]** Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left(\frac{1}{B}\sum_{k=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)})\right) + \lambda\left(||W^{[1]}||^2 + ||W^{[2]}||^2\right)$$

Be careful not to regularize the bias/intercept term. Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to measure the loss value for plotting (i.e., regularization should only be used for gradient calculation for the purpose of training).

Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup.

Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

(d) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e., the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e., the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e., the regularized model).

Report your test accuracy for both regularized model and non-regularized model. Briefly (in one sentence) explain why this outcome makes sense. You should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments, which is not the same way as specified in your code. Even if you do not precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

(e) **[6 points]** In this part, we compare the model complexity of the Linear Layer NN model with a convolution-based model. Let the original input image be a grayscale image with height and width r, meaning each image is of shape $(1, r, r)$.

We define a **CNN** (convolutional neural network) version of our original model. We let the number of output classes be k, and we have the following operations:

1. *Convolutional Layer*: We define $a(i)$ as the output of applying a 2D convolutional layer with 32 filters (with bias) of size 3×3, padding 1, stride 1, and ReLU activation on an input image $x^{(i)} \in \mathbb{R}^{1\times r\times r}$.

$$a_1^{(i)} = \text{ReLU}(\text{Conv2D}(x^{(i)}))$$

2. *Flattening*: After applying the convolution, we flatten $a_1^{(i)}$ into a vector.

$$a_2^{(i)} = \text{Flatten}(a_1^{(i)}))$$

3. *Linear Transformation*: We apply a linear transformation to the flattened vector, mapping it to $\mathbb{R}^k$

$$\bar{h}_\theta(x^{(i)}) = {W^{[2]}}^\top a_2^{(i)} + b^{[2]}$$

4. *Softmax Output*: We apply the softmax activation to obtain our final output.

$$h_\theta(x^{(i)}) = \text{softmax}(\bar{h}_\theta(x^{(i)}))$$

i. [1 points] In the CNN model, what are the dimensions of $a(i)$ right after the convolutional layer (before the flattening operation) in terms of r?

ii. [1 points] In the CNN model, how many parameters are there in terms of in terms of the input height/width r and the number of output classes k?

iii. [1 points] In the Linear Layer NN model, how many parameters are there in terms of the input height/width r, the number of hidden units m, and the number of output classes k?

iv. [1 points] What is the maximum value of m for which the Linear NN model has fewer parameters than the CNN model on the MNIST dataset ($r = 28, k = 10$)?

v. [2 points] What is one advantage and one disadvantage of using the CNN compared to the Linear Layer NN?