CS 229, Winter 2024
Problem Set #2

Due Wednesday, February 7 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/51342/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 7 at 11:59 pm. If you submit after Wednesday, February 7 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.

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 solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years,

[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

we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

Regarding Notation: The notation used in this problem set matches the notation used in the lecture notes. Some notable differences from lecture notation:

- The superscript "(i)" represents an index into the training set – for example, $x^{(i)}$ is the i-th feature vector and $y^{(i)}$ is the i-th output variable. In lecture notation, these would instead be expressed as x_i and y_i.
- The subscript j represents an index in a vector – in particular, $x_j^{(i)}$ represents the j-th feature in the feature vector $x^{(i)}$. In lecture notation, $x_j^{(i)}$ would be $h_j(x_i)$ or $x_i[j]$.
- The vector that contains the weights parameterizing a linear regression is expressed by the variable θ, whereas lectures use the variable $\mathbf{w}$. As such, $\theta_0 = w_0$, $\theta_1 = w_1$, and so on.

An overview of this notation is also given at the beginning of the lecture notes (pages 6-7).

1. **[25 points] Poisson Regression**

In this question we will construct another kind of a commonly used GLM, which is called Poisson Regression. In a GLM, the choice of the exponential family distribution is based on the kind of problem at hand. If we are solving a classification problem, then we use an exponential family distribution with support over discrete classes (such as Bernoulli, or Categorical). Simiarly, if the output is real valued, we can use Gaussian or Laplace (both are in the exponential family). Sometimes the desired output is to predict counts, for example, predicting the number of emails expected in a day, or the number of customers expected to enter a store in the next hour, etc. based on input features (also called covariates). You may recall that a probability distribution with support over integers (i.e., counts) is the Poisson distribution, and it also happens to be in the exponential family.

In the following sub-problems, we will start by showing that the Poisson distribution is in the exponential family, derive the functional form of the hypothesis, derive the update rules for training models, and finally using the provided dataset to train a real model and make predictions on the test set.

(a) [5 points] Consider the Poisson distribution parameterized by λ:

$$p(y; \lambda) = \frac{e^{-\lambda}\lambda^y}{y!}.$$

(Here y has positive integer values and $y!$ is the factorial of y.) Show that the Poisson distribution is in the exponential family, and clearly state the values for $b(y)$, η, $T(y)$, and $a(\eta)$.

(b) [3 points] Consider performing regression using a GLM model with a Poisson response variable. What is the canonical response function for the family? (You may use the fact that a Poisson random variable with parameter λ has mean λ.)

(c) [7 points] For a training set $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, let the log-likelihood of an example be $\log p(y^{(i)}|x^{(i)}; \theta)$. By taking the derivative of the log-likelihood with respect to θ_j, derive the stochastic gradient ascent update rule for learning using a GLM model with Poisson responses y and the canonical response function.

(d) [10 points] **Coding problem**

Consider a website that wants to predict its daily traffic. The website owners have collected a dataset of past traffic to their website, along with some features which they think are useful in predicting the number of visitors per day. The dataset is split into train/valid sets and the starter code is provided in the following files:

- `src/poisson/{train,valid}.csv`
- `src/poisson/poisson.py`

We will apply Poisson regression to model the number of visitors per day. Note that applying Poisson regression in particular assumes that the data follows a Poisson distribution whose natural parameter is a linear combination of the input features (*i.e.,* $\eta = \theta^T x$). In `src/poisson/poisson.py`, implement Poisson regression for this dataset and use *full batch gradient ascent* to maximize the log-likelihood of θ. For the stopping criterion, check if the change in parameters has a norm smaller than a small value such as 10^{-5}.

Using the trained model, predict the expected counts for the **validation set**, and create a scatter plot between the true counts vs predicted counts (on the validation set). In the

scatter plot, let x-axis be the true count and y-axis be the corresponding predicted expected count. Note that the true counts are integers while the expected counts are generally real values.

2. **[15 points] Convexity of Generalized Linear Models**

In this question we will explore and show some nice properties of Generalized Linear Models, specifically those related to its use of Exponential Family distributions to model the output.

Most commonly, GLMs are trained by using the negative log-likelihood (NLL) as the loss function. This is mathematically equivalent to Maximum Likelihood Estimation (*i.e.,* maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood). In this problem, our goal is to show that the NLL loss of a GLM is a *convex* function w.r.t the model parameters. As a reminder, this is convenient because a convex function is one for which any local minimum is also a global minimum, and there is extensive research on how to optimize various types of convex functions efficiently with various algorithms such as gradient descent or stochastic gradient descent.

To recap, an exponential family distribution is one whose probability density can be represented as

$$p(y;\eta) = b(y)\exp(\eta^T T(y) - a(\eta)),$$

where η is the *natural parameter* of the distribution. Moreover, in a Generalized Linear Model, η is modeled as $\theta^T x$, where $x \in \mathbb{R}^d$ are the input features of the example, and $\theta \in \mathbb{R}^d$ are learnable parameters. In order to show that the NLL loss is convex for GLMs, we break down the process into sub-parts, and approach them one at a time. Our approach is to show that the second derivative (*i.e.,* Hessian) of the loss w.r.t the model parameters is Positive Semi-Definite (PSD) at all values of the model parameters. We will also show some nice properties of Exponential Family distributions as intermediate steps.

For the sake of convenience we restrict ourselves to the case where η is a scalar. Assume $p(Y|X;\theta) \sim \text{ExponentialFamily}(\eta)$, where $\eta \in \mathbb{R}$ is a scalar, and $T(y) = y$. This makes the exponential family representation take the form

$$p(y;\eta) = b(y)\exp(\eta y - a(\eta)).$$

Note that the above probability density is for a single example (x, y).

(a) [5 points] Derive an expression for the mean of the distribution. Show that $\mathbb{E}[Y;\eta] = \frac{\partial}{\partial \eta} a(\eta)$ (note that $\mathbb{E}[Y;\eta] = \mathbb{E}[Y|X;\theta]$ since $\eta = \theta^T x$). In other words, show that the mean of an exponential family distribution is the first derivative of the log-partition function with respect to the natural parameter.

Hint: Start with observing that $\frac{\partial}{\partial \eta}\int p(y;\eta)dy = \int \frac{\partial}{\partial \eta} p(y;\eta)dy$.

(b) [5 points] Next, derive an expression for the variance of the distribution. In particular, show that $\text{Var}(Y;\eta) = \frac{\partial^2}{\partial \eta^2} a(\eta)$ (again, note that $\text{Var}(Y;\eta) = \text{Var}(Y|X;\theta)$). In other words, show that the variance of an exponential family distribution is the second derivative of the log-partition function w.r.t. the natural parameter.

Hint: Building upon the result in the previous sub-problem can simplify the derivation.

(c) [5 points] Finally, write out the loss function $\ell(\theta)$, the NLL of the distribution, as a function of θ. Then, calculate the Hessian of the loss w.r.t θ, and show that it is always PSD. This concludes the proof that NLL loss of GLM is convex.

Hint 1: Use the chain rule of calculus along with the results of the previous parts to simplify your derivations.

Hint 2: Recall that variance of any probability distribution is non-negative.

Remark: The main takeaways from this problem are:

- Any GLM model is convex in its model parameters.
- The exponential family of probability distributions are mathematically nice. Whereas calculating mean and variance of distributions in general involves integrals (hard), surprisingly we can calculate them using derivatives (easy) for exponential family.

3. **[25 points] Logistic Regression: Training stability**

In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

(a) [10 points] **Coding question**

In lecture we saw the average empirical loss for logistic regression:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right),$$

where $y^{(i)} \in \{0,1\}$, $h_\theta(x) = g(\theta^T x)$ and $g(z) = 1/(1+e^{-z})$.

Implement logistic regression using gradient descent in `src/logreg_stability/logreg.py`. Starting with $\theta = \vec{0}$, run gradient descent until the updates to θ are small: Specifically, train until the first iteration k such that $\|\theta_k - \theta_{k-1}\|_1 < \epsilon$, where $\epsilon = 1 \times 10^{-5}$, or for a maximum of 10^5 iterations. Then, perform logistic regression on dataset A in `src/logreg_stability/ds1_a.csv`. You can run the code by simply executing `python logreg.py` in the `src/logreg_stability` directory.

Include a plot of the training data with x_1 on the horizontal axis and x_2 on the vertical axis. To visualize the two classes, use a different symbol for examples $x^{(i)}$ with $y^{(i)} = 0$ than for those with $y^{(i)} = 1$. On the same figure, plot the decision boundary found by logistic regression (i.e., line corresponding to $p(y|x) = 0.5$).

Note: If you want to print the loss during training, you may encounter some numerical instability issues. Recall that the loss function on an example (x, y) is defined as

$$y\log(h_\theta(x)) + (1-y)\log(1-h_\theta(x)),$$

where $h_\theta(x) = (1+\exp(-x^\top\theta))^{-1}$. Technically speaking, $h_\theta(x) \in (0,1)$ for any $\theta, x \in \mathbb{R}^d$. However, in Python a real number only has finite precision. So it is possible that in your implementation, $h_\theta(x) = 0$ or $h_\theta(x) = 1$, which makes the loss function ill-defined. A typical solution to the numerical instability issue is to add a small perturbation. In this case, you can compute the loss function using

$$y\log(h_\theta(x)+\epsilon) + (1-y)\log(1-h_\theta(x)+\epsilon),$$

instead, where ϵ is a very small perturbation (for example, $\epsilon = 10^{-5}$).

(b) [7 points] Perform logistic regression on dataset B in `src/logreg_stability/ds1_a.csv`, and include the same plot as for dataset A in the previous subquestion. What is the most notable difference in training the logistic regression model on datasets A and B? Investigate why the training procedure behaves unexpectedly on dataset B, but not on A. Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to A.

Hint: The issue is not a numerical rounding or over/underflow error.

(c) [3 points] For each of these possible modifications, state whether or not it would lead to the provided training algorithm converging on datasets such as B. Justify your answers.

i. Using a different constant learning rate.

ii. Decreasing the learning rate over time (e.g. scaling the initial learning rate by $1/t^2$, where t is the number of gradient descent iterations thus far).

iii. Linear scaling of the input features.

iv. Adding a regularization term $\|\theta\|_2^2$ to the loss function.

v. Adding zero-mean Gaussian noise to the training data or labels.

(d) [5 points] **Coding question** To attempt to fix the issues brought up in the previous subequestions, implement L2 regularization in `src/logreg_stability/logreg.py`. Specifically, modify the gradient update to reflect the following loss function:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right) + \frac{1}{2}\lambda\|\theta\|_2^2. \tag{1}$$

Run the same code as in the previous subquestion using $\lambda = 0.01$. For both datasets, print the final value of θ, and compare it to the final value of θ without regularization. Additionally, include a plot of the decision boundary and training data for both datasets and comment on the results qualitatively. How does the decision boundary compare to the one obtained without regularization, especially with respect to the worst misclassified example; in what cases may this be desirable?

4. **[25 points] Learning Imbalanced dataset**

In this problem, we study how to learn a classifier from an imbalanced dataset, where the marginal distribution of the classes/labels are imbalanced. Imbalanced datasets are ubiquitous in real-world applications. For example, in the spam detection problem, the training dataset usually has only a small fraction of spam emails (positive examples) but a large fraction of ordinary emails (negative examples). For simplicity, we consider binary classification problem where the labels are in $\{0, 1\}$ and the number of positive examples is much smaller than the number of negative examples.

(a) [10 points] **Evaluation metrics**

In this sub-question, we discuss the common evaluation metrics for imbalanced dataset. Suppose we have a validation dataset and for some $\rho \in (0, 1)$, we assume that ρ fraction of the validation examples are positive examples (with label 1), and $1 - \rho$ fraction of them are negative examples (with label 0).

Define the accuracy as

$$A \triangleq \frac{\#\text{examples that are predicted correctly by the classifier}}{\#\text{examples}}$$

(i) (3 point) Show that for any dataset with ρ fraction of positive examples and $1 - \rho$ fraction of negative examples, there exists a (trivial) classifier with accuracy at least $1 - \rho$.

The statement above suggests that the accuracy is not an ideal evaluation metric when ρ is close to 0. E.g., imagine that for spam detection ρ can be smaller than 1%. The statement suggests there is a trivial classifier that gets more than 99% accuracy. This could be misleading — 99% seems to be almost perfect, but actually you don't need to learn anything from the dataset to achieve it.

Therefore, for imbalanced dataset, we need more informative evaluation metrics. We define the number of true positive, true negative, false positive, false negative examples as

$$TP \triangleq \#\text{ positive examples with a correct (positive) prediction}$$
$$TN \triangleq \#\text{ negative examples with a correct (negative) prediction}$$
$$FP \triangleq \#\text{ negative examples with a incorrect (positive) prediction}$$
$$FN \triangleq \#\text{ positive examples with a incorrect (negative) prediction}$$

Define the accuracy of positive examples as

$$A_1 \triangleq \frac{TP}{TP + FN} = \frac{\#\text{ positive examples with a correct (positive) prediction}}{\#\text{ positive examples}}$$

Define the accuracy of negative examples as

$$A_0 \triangleq \frac{TN}{TN + FP} = \frac{\#\text{ negative examples with a correct (negative) prediction}}{\#\text{ negative examples}}$$

We define the balanced accuracy as

$$\overline{A} \triangleq \frac{1}{2}(A_0 + A_1) \tag{2}$$

With these notations, we can verify that the accuracy is equal to $A = \frac{TP+TN}{TP+TN+FP+FN}$.
(ii) (4 point) Show that

$$\rho = \frac{TP + FN}{TP + TN + FP + FN}$$

and

$$A = \rho \cdot A_1 + (1 - \rho)A_0 \tag{3}$$

Comparing equation (2) and (3), we can see that the accuracy and balanced accuracy are both linear combination of A_0 and A_1 but with different weighting.
(iii) (3 point) Show that the trivial classifier you constructed for part (i) has balanced accuracy 50%.
Partly because of (iii), the balanced accuracy $\overline{A}$ is often a preferable evaluation metric than the accuracy A. Sometimes people also report the accuracies for the two classes (A_0 and A_1) to demonstrate the performance for each class.

(b) [5 points] **Coding problem: vanilla logistic regression**
First, we use the vanilla logistic regression to learn an imbalanced dataset. For the rest of the question, we will use the dataset and starter code provided in the following files:

- `src/imbalanced/{train,validation}.csv`
- `src/imbalanced/imbalanced.py`

Each file contains n examples, one example $(x^{(i)}, y^{(i)})$ per row. x is two-dimensional, i.e., the i-th row contains columns $x_1^{(i)} \in \mathbb{R}$, $x_2^{(i)} \in \mathbb{R}$, and $y^{(i)} \in \{0, 1\}$. Let $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ be our training dataset. $\mathcal{D}$ has ρn examples with label 1 and $(1 - \rho)n$ with label 0. In the dataset we constructed, $\rho = 1/11$.
You will train a linear classifier $h_\theta(x)$ with average empirical loss for logistical regression, where $h_\theta(x) = g(\theta^T x), g(z) = 1/(1 + e^{-z})$, similar to Problem 3:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right),$$

You are encouraged to use your code for problem 3, but you are also allowed to use any standard logistic regression library or package to optimize the objective above. After obtaining the classifier, compute the classifier's accuracy (A), balanced accuracy ($\overline{A}$), accuracies for the two classes (A_0, A_1) on the validation dataset, and report them in the writeup. You are expected to observe that the minority class (positive class) has significantly lower accuracy than the majority class.
Create a plot to visualize the validation set with x_1 on the horizontal axis and x_2 on the vertical axis. Use different symbols for examples $x^{(i)}$ with true label $y^{(i)} = 1$ than those with $y^{(i)} = 0$. On the same figure, plot the decision boundary obtained by your model (i.e, line corresponding to model's predicted probability = 0.5) in red color. Include this plot in your writeup.

(c) [5 points] **Re-sampling/Re-weighting Logistic Regression**
The relatively low accuracy for the minority class and the resulting low balanced accuracy are undesirable for many applications. Various methods have been proposed to improve the accuracy for the minority class, and learning imbalanced datasets is an active open research

direction. Here we introduce a simple and classical re-sampling/re-weighting technique that helps improve the balanced accuracy in most of the cases.

We observe that the logistic regression algorithm works well for the accuracy but not for the balanced accuracy. Let $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ denote the existing training dataset. We will create a new dataset $\mathcal{D}'$ such that the accuracy on $\mathcal{D}'$ is the same as the balanced accuracy on $\mathcal{D}$.

Assume $\rho < 1/2$ without loss of generality, and let $\kappa = \frac{\rho}{1-\rho}$. This means the number of positive examples is κ times the number of negative examples in $\mathcal{D}$. Assume for convenience $1/\kappa$ is an integer.[3] Let $\mathcal{D}'$ be the dataset that contain each negative example once and $1/\kappa$ repetitions of each positive example in $\mathcal{D}$. Then we will apply the logistic regression on the new dataset $\mathcal{D}'$.

Prove that for any classifier, the balanced accuracy on $\mathcal{D}$ is equal to the accuracy on $\mathcal{D}'$. Moreover, **show that** the average empirical loss for logistical regression on the dataset $\mathcal{D}'$ is equal to

$$J(\theta) = -\frac{1+\kappa}{2n} \sum_{i=1}^{n} w^{(i)} \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right),$$

where $w^{(i)} = 1$ if $y^{(i)} = 0$ and $w^{(i)} = 1/\kappa$ if $y^{(i)} = 1$.

Observe effectively we are re-weighting the loss function for each example by some constant that depends on the frequency of the class it belongs to.

(d) [5 points] **Coding problem: re-weighting minority class**

In `src/imbalanced/imbalanced.py`, implement the logistic regression algorithm on $\mathcal{D}'$. Compute and report the classifier's accuracy (A), balanced accuracy ($\overline{A}$), accuracies for the two classes (A_0, A_1) on the validation dataset. You are expected to see that the accuracy of minority class (class 1) improved significantly whereas that of the majority class dropped compared to vanilla logistic regression. However, the balanced accuracy is significantly greater than that of the vanilla logistic regression.

Create a plot to visualize the validation set with x_1 on the horizontal axis and x_2 on the vertical axis. Use different symbols for examples $x^{(i)}$ with true label $y^{(i)} = 1$ than those with $y^{(i)} = 0$. On the same figure, plot the decision boundary obtained by your model (i.e, line corresponding to model's predicted probability = 0.5) in red color. Include this plot in your writeup.

[3] otherwise we can round up to the nearest integer with introducing a slight approximation.

5. **[30 points] Neural Networks: MNIST image classification**

Note: This question may requires knowledge on backpropagation covered on Monday of Week 5.

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.)

For clarity, we provide the forward propagation equations below for the neural network with a single hidden layer. 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. 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 $W^{[2]} \in \mathbb{R}^{m \times k}$.[4] We also have biases $b^{[1]} \in \mathbb{R}^m$ and $b^{[2]} \in \mathbb{R}^k$. The parameters of the model θ is $(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]})$. The forward propagation equations for a single input $x^{(i)}$ then 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

$$\frac{\partial \ell_{\text{CE}}(t, y)}{\partial t} = p - e_y \in \mathbb{R}^k, \tag{4}$$

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,

[4] 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.

$$\frac{\partial \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)})}{\partial \bar{h}_\theta(x^{(i)})} = \text{softmax}(\bar{h}_\theta(x^{(i)})) - e_{y^{(i)}} = h_\theta(x^{(i)}) - e_{y^{(i)}} \in \mathbb{R}^k \tag{5}$$

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.