CS 229, Summer 2019
Problem Set #2 Solutions

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Monday, July 29 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 Piazza forum, at `http://piazza.com/stanford/summer2019/cs229`. (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) To account for late days, the due date is Monday, July 29 at 11:59 pm. If you submit after Monday, July 29 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 29 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. 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.

1. **[15 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).

We have provided an implementation of logistic regression in `src/stability/stability.py`, and two labeled datasets A and B in `src/stability/ds1_a.csv` and `src/stability/ds1_b.csv`.

Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on A and B. You can run the code by simply executing `python stability.py` in the `src/stability` directory.

(a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

Answer: Training A is much faster then training B.

(b) [5 points] 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.

Answer: Firstly, let's investigate dots, that lie under and above the line $(-1+x_1^{(i)}+x_2^{(i)}=0)$. In dataset A, in positive class, 41 dots are above the line, while 5 are under. In dataset B, in positive class, 55 dots are above the line, while 0 are under. In negative class, 45 dots are under the line. Lets assume $\alpha\nabla_\theta(J(\theta)) = \alpha\sum_{i=1}^{n}(y^{(i)}-\sigma(\theta^T x^{(i)}))x^{(i)}$

The true split is a line $(-1+x_1^{(i)}+x_2^{(i)}=0)$ Lets calculate projection of the gradient to the normalized vector of true $\theta : (-1,1,1)$

$$Proj_{n_{\vec{grad}}}\alpha\nabla_\theta(J(\theta)) = (-\frac{1}{\sqrt(3)}, \frac{1}{\sqrt(3)}, \frac{1}{\sqrt(3)})\alpha\sum_{i=1}^{n}(y^{(i)}-\sigma(\theta^T x^{(i)}))x^{(i)} =$$

$$\alpha\sum_{i=1}^{n}(y^{(i)}-\sigma(\theta^T x^{(i)}))x^{(i)}(-\frac{1}{\sqrt(3)}+\frac{1}{\sqrt(3)}x_1^{(i)}+\frac{1}{\sqrt(3)}x_2^{(i)}) =$$

$$\frac{\alpha}{\sqrt(3)}\sum_{i=1}^{n}(y^{(i)}-\sigma(b(-1+x_1^{(i)}+x_2^{(i)})))(-1+x_1^{(i)}+x_2^{(i)})$$

For only B dataset: we can see that if class is negative, it is under the line, consequently, $-1+x_1^{(i)}+x_2^{(i)}<0$, then the right term is negative, while the left term is slightly fewer then zero, and gradient increases. The, if class is positive, it is above the line, consequently $-1+x_1^{(i)}+x_2^{(i)}>0$, then the right term is positive, while the left term is slightly bigger then zero, and gradient increases.

Therefore, the series of only positive terms can only increase, while the alternating sign series can coverges faster.

(c) [5 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.

Answer:

i. The above equations state that for any constant α for any ideally separabale dataset θ increases uncontrollably.

ii. So,

$$\theta_{t+1} = \theta_t + \frac{\alpha}{t^2}\sum_{i=1}^{n}(y^{(i)} - \sigma(\theta^T x^{(i)}))x^{(i)}) = \theta_t + \frac{\alpha}{t^2}\sum_{i=1}^{n} a_i x^{(i)}$$

$-1 \leq a_i \leq 1$. Let's consider the edge case $|a_i| = 1$

$$\theta_t = \frac{\alpha}{t^2}\sum_{i=1}^{n} x^{(i)} = \frac{\alpha}{t^2}C$$

$$\sum_{t=1}^{n}\frac{\alpha}{t^2} = \alpha\frac{\pi^2}{6}$$

iii. Linear scaling tends to transform separating line to $Ax_1^{(i)} + Bx_2^{(i)} - 1 = 0$
That transformation simply changes the slope of separating line. The problem will remain.

iv.

$$J(\theta) = \sum_{i=1}^{n} y^{(i)}\log(\sigma(x^{(i)})) + (1 - y^{(i)})\log(1 - \sigma(x^{(i)})) - \frac{1}{2}||\theta||_2^2$$

$$\nabla_\theta(J(\theta)) = \sum_{i=1}^{n}(y^{(i)} - \sigma(x^{(i)}))x^{(i)}) - \theta$$

$$Proj_{n_{\vec{grad}}}\alpha\nabla_\theta(J(\theta)) = (-\frac{1}{\sqrt(3)}, \frac{1}{\sqrt(3)}, \frac{1}{\sqrt(3)})\alpha[\sum_{i=1}^{n}(y^{(i)} - \sigma(\theta^T x^{(i)}))x^{(i)} - \theta] =$$

$$\frac{\alpha}{\sqrt(3)}[\sum_{i=1}^{n}(y^{(i)} - \sigma(x^{(i)})(-1 + x_1^{(i)} + x_2^{(i)})) - (-\theta_0 + \theta_1 + \theta_2)] =$$

$$\frac{\alpha}{\sqrt(3)}[\sum_{i=1}^{n}(y^{(i)} - \sigma(x^{(i)})(-1 + x_1^{(i)} + x_2^{(i)})) - (-b + b + b)]$$

θ_t can be written as: $\theta_t = (-1, 1, 1) * b, b \in \mathbb{R}$ b scales θ. When b is very big, the gradient tends to be very negative, while b is very negative, the gradient tends to be very positive.

v. Zero mean gaussian will shuffle a bit dots from both of classes, so that will make algorithm to converge.

(d) [3 points] Are support vector machines, vulnerable to datasets like B? Why or why not? Give an informal justification.

Answer: SVMs aren't vulnerable to the problems such a dataset B. It's because, of we write update rule for β, we will find that update term can be positive and negative.

Complete the `get_top_five_naive_bayes_words` function within the provided code using the above formula in order to obtain the 5 most indicative tokens.

Report the top five words in your writeup.

Answer: ['claim', 'won', 'prize', 'tone', 'urgent!']

(d) [2 points] Support vector machines (SVMs) are an alternative machine learning model that we discussed in class. We have provided you an SVM implementation (using a radial basis function (RBF) kernel) within `src/spam/svm.py` (You should not need to modify that code).

One important part of training an SVM parameterized by an RBF kernel (a.k.a Gaussian kernel) is choosing an appropriate kernel radius parameter.

Complete the `compute_best_svm_radius` by writing code to compute the best SVM radius which maximizes accuracy on the validation dataset. Report the best kernel radius you obtained in the writeup.

Answer: 0.1

3. [18 points] Constructing kernels

In class, we saw that by choosing a kernel $K(x,z) = \phi(x)^T\phi(z)$, we can implicitly map data to a high dimensional space, and have a learning algorithm (e.g SVM or logistic regression) 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. I.e., suppose we have a function $K(x,z)$ that we think gives an appropriate similarity measure for our learning problem, and we are considering plugging K into the SVM as the kernel function. However for $K(x,z)$ 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 $K(x,z)$ is a (Mercer) kernel if and only if for any finite set $\{x^{(1)},\ldots,x^{(n)}\}$, the square matrix $K \in \mathbb{R}^{n\times n}$ whose entries are given by $K_{ij} = K(x^{(i)},x^{(j)})$ is symmetric and positive semidefinite. You can find more details about Mercer's theorem in the notes, though the description above is sufficient for this problem.

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 $f : \mathbb{R}^d \mapsto \mathbb{R}$ be a real-valued function, let $\phi : \mathbb{R}^d \to \mathbb{R}^p$ be a function mapping from $\mathbb{R}^d$ to $\mathbb{R}^p$, let K_3 be a kernel over $\mathbb{R}^p \times \mathbb{R}^p$, and let $p(x)$ a polynomial over x with *positive* coefficients.

For each of the functions K below, state whether it is necessarily a kernel. If you think it is, prove it; if you think it isn't, give a counter-example.

(a) [1 points] $K(x,z) = K_1(x,z) + K_2(x,z)$

(b) [1 points] $K(x,z) = K_1(x,z) - K_2(x,z)$

(c) [1 points] $K(x,z) = aK_1(x,z)$

(d) [1 points] $K(x,z) = -aK_1(x,z)$

(e) [5 points] $K(x,z) = K_1(x,z)K_2(x,z)$

(f) [3 points] $K(x,z) = f(x)f(z)$

(g) [3 points] $K(x,z) = K_3(\phi(x),\phi(z))$

(h) [3 points] $K(x,z) = p(K_1(x,z))$

[**Hint:** For part (e), the answer is that K *is* indeed a kernel. You still have to prove it, though. (This one may be harder than the rest.) This result may also be useful for another part of the problem.]

Answer:

(a) (Symm)$K(x,z) = K_1(x,z) + K_2(x,z) = K_1(z,x) + K_2(z,x) = K(z,x)$
(PSD) $x^TK_1x + x^TK_2x = x^T(K_1+K_2)x \geq 0$

(b) (not PSD) $x^T\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}x - x^T\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}x = x^T\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}x = -(x_1^2+x_2^2) \leq 0$

(c) (Symm)$K(x,z) = aK_1(x,z) = aK_1(z,x) = K(z,x)$
(PSD) $x^TK(x,z)x = ax^TK_1(x,z)x \geq 0$

(d) (not PSD) $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$ - kernel $\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}$ - not a kernel

(e) (Symm)$K(x,z) = K_1(x,z)K_2(x,z) = K_1(z,x)K_2(z,x) = K(z,x)$

(PSD) $K(x^{(i)}, x^{(j)}) = K_1(x^{(i)}, x^{(j)})K_2(x^{(i)}, x^{(j)}) = (\sum_{p,q} \phi_1^{(p)}(x^{(i)})\phi_1^{(q)}(x^{(j)}))(\sum_{p,q} \phi_2^{(p)}(x^{(i)})\phi_2^{(q)}(x^{(j)})) = \sum_{p,q} \phi_1^{(p)}(x^{(i)})\phi_1^{(q)}(x^{(i)})\phi_2^{(p)}(x^{(i)})\phi_2^{(q)}(x^{(i)})$ $x^TKx = \sum_{p,q}\sum_{i,j} x^{(i)}\phi_1^{(p)}(x^{(i)})\phi_1^{(q)}(x^{(i)})\phi_2^{(p)}(x^{(i)})\phi_2^{(q)}(x^{(i)})x^{(j)} = \sum_{p,q}(\sum_i x^{(i)}\phi_1^{(p)}(x^{(i)})\phi_2^{(q)}(x^{(i)}))^2$

(f) (Symm)$K(x,z) = f(x)f(z) = f(z)f(x) = K(z,x)$

(PSD) $x^TKx = \sum_{i,j} x^{(i)}f(x^{(i)})x^{(j)}f(x^{(j)}) = (\sum_i x^{(i)}f(x^{(i)}))^2 \geq 0$

(g) (Symm) $K(x,z) = K_3(\phi(x),\phi(z)) = K_3(\phi(z),\phi(x)) = K(z,x)$

(PSD) $x^TKx = \sum_{i,j} x^{(i)}f(\phi(x^{(i)}))x^{(j)}f(\phi(x^{(j)})) = (\sum_i x^{(i)}f(\phi(x^{(i)})))^2 \geq 0$

(h) (Symm) $K(x,z) = p(K_1(x,z)) = p(K_1(z,x)) = K(z,x)$

(PSD) $K = a_0 \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} + a_1 \begin{bmatrix} K_1(x^{(1)}, x^{(1)}) & \dots & K_1(x^{(1)}, x^{(n)}) \\ \vdots & \ddots & \vdots \\ K_1(x^{(n)}, x^{(1)}) & \dots & K_1(x^{(n)}, x^{(n()}) \end{bmatrix} +$
$a_2 \begin{bmatrix} K_1(x^{(1)}, x^{(1)})^2 & \dots & K_1(x^{(1)}, x^{(n)})^2 \\ \vdots & \ddots & \vdots \\ K_1(x^{(n)}, x^{(1)})^2 & \dots & K_1(x^{(n)}, x^{(n()})^2 \end{bmatrix} + ...$

0. $x^Ta_0 \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} x \geq 0$

1. $a_1x^TK_1x \geq 0$

2. $a_1x^TK_1 \circ K_1x \geq 0$ (consequence from (e))

3. ...

4. **[15 points] Kernelizing the Perceptron**

Let there be a binary classification problem with $y \in \{0, 1\}$. The perceptron uses hypotheses of the form $h_\theta(x) = g(\theta^T x)$, where $g(z) = \text{sign}(z) = 1$ if $z \geq 0$, 0 otherwise. In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters θ is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))x^{(i+1)}$$

where $\theta^{(i)}$ is the value of the parameters after the algorithm has seen the first i training examples. Prior to seeing any training examples, $\theta^{(0)}$ is initialized to $\vec{0}$.

(a) [3 points] Let K be a Mercer kernel corresponding to some very high-dimensional feature mapping ϕ. Suppose ϕ is so high-dimensional (say, ∞-dimensional) that it's infeasible to ever represent $\phi(x)$ explicitly. Describe how you would apply the "kernel trick" to the perceptron to make it work in the high-dimensional feature space ϕ, but without ever explicitly computing $\phi(x)$.

[**Note:** You don't have to worry about the intercept term. If you like, think of ϕ as having the property that $\phi_0(x) = 1$ so that this is taken care of.] Your description should specify:

i. [1 points] How you will (implicitly) represent the high-dimensional parameter vector $\theta^{(i)}$, including how the initial value $\theta^{(0)} = 0$ is represented (note that $\theta^{(i)}$ is now a vector whose dimension is the same as the feature vectors $\phi(x)$);

ii. [1 points] How you will efficiently make a prediction on a new input $x^{(i+1)}$. I.e., how you will compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, using your representation of $\theta^{(i)}$; and

iii. [1 points] How you will modify the update rule given above to perform an update to θ on a new training example $(x^{(i+1)}, y^{(i+1)})$; *i.e.,* using the update rule corresponding to the feature mapping ϕ:

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)})$$

Answer:

i. $\theta^{(i)} = \alpha \sum_{k=1}^{i} (y^{(i)} - h_{\theta^{(k-1)}}(x^{(k)}))\phi(x^{(k)}) = \sum_{k=1}^{i} \beta_k \phi(x^{(k)})$

ii. $h_{\theta^{(i)}}(x^{(i+1)}) = g((\theta^{(i)})^T \phi(x^{(i)})) = g(\sum_{k=1}^{i} \beta_k (\phi(x^{(k)}))^T \beta_k \phi(x^{(i+1)}))$

iii. $\beta_{i+1} = \alpha(y^{(i+1)} - g(\sum_{k=1}^{i} \beta_k (\phi(x^{(k)}))^T \beta_k \phi(x^{(i+1)}))$

(b) [10 points] Implement your approach by completing the `initial_state`, `predict`, and `update_state` methods of `src/perceptron/perceptron.py`.

We provide two kernels, a dot-product kernel and a radial basis function (RBF) kernel. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

Include the two plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which kernel.

Answer:

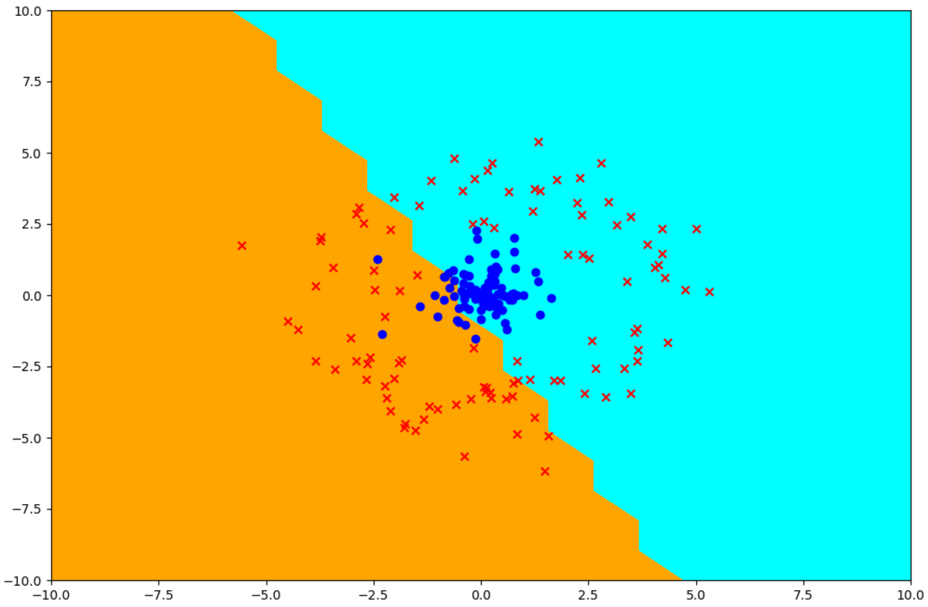


Figure 1: Dot kernel.

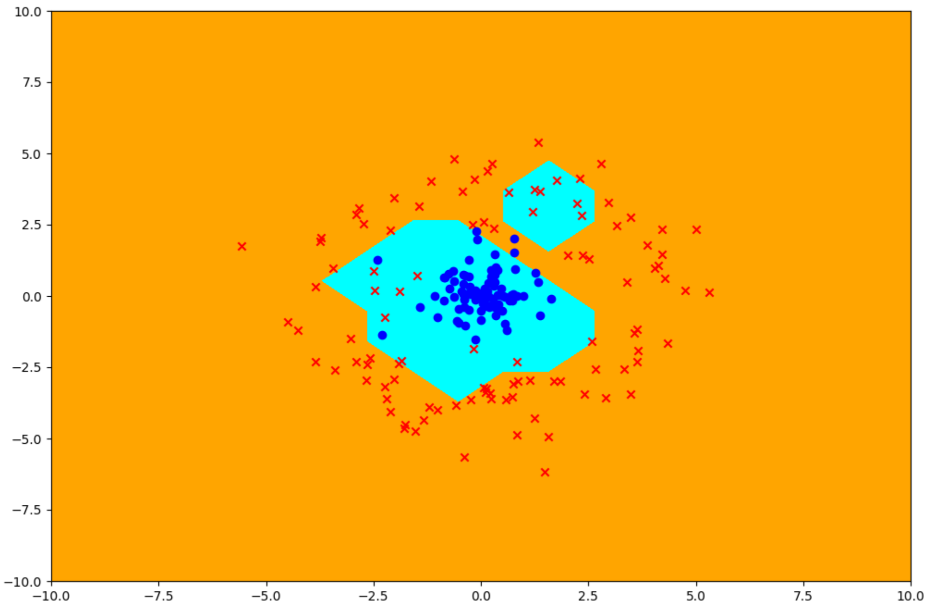


Figure 2: Rbf kernel.

(c) [2 points]

One of the provided kernels performs extremely poorly in classifying the points. Which kernel performs badly and why does it fail?

Answer: Dot kernel performs poorly, because it's descision boundary is almost linear, while the data is organized as circle.

5. **[25 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. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

$$CE(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y_k},$$

where $\hat{y} \in \mathbb{R}^K$ is the vector of softmax outputs from the model for the training example x, and $y \in \mathbb{R}^K$ is the ground-truth vector for the training example x such that $y = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the position of the correct class (also called a "one-hot" representation).

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

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^{n} CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}.$$

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. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B} \sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$$

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

(a) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function. 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.

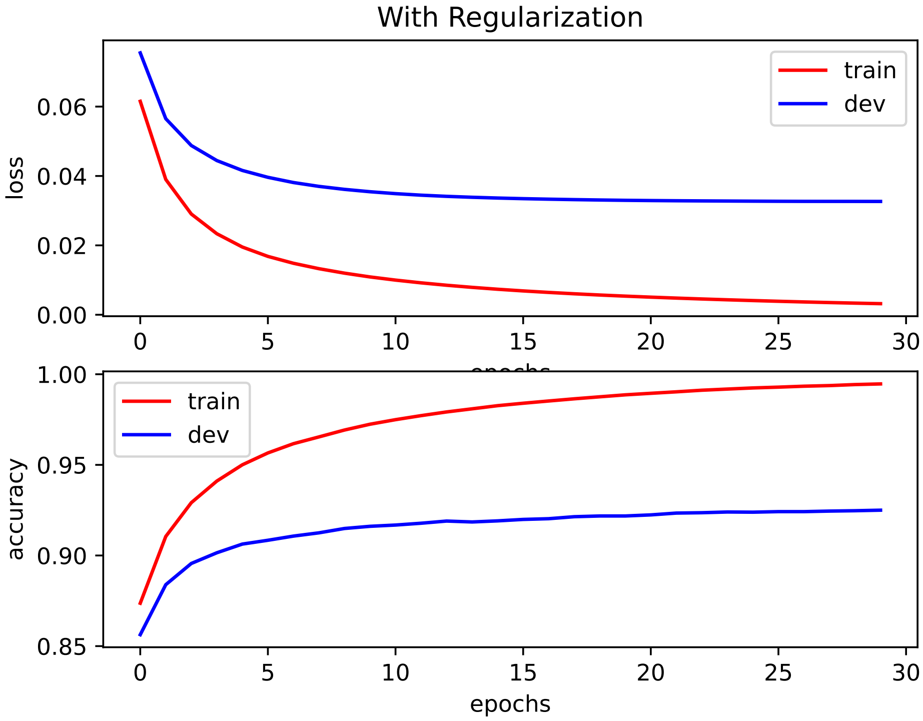
Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the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



Answer:

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

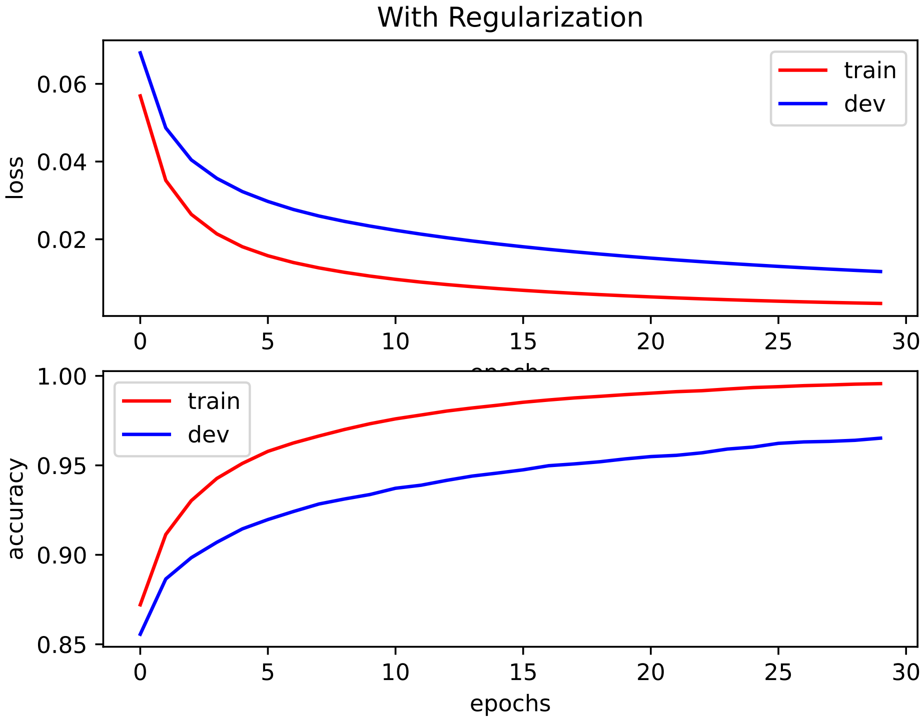
$$J_{MB} = \left(\frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})\right) + \lambda\left(||W^{[1]}||^2 + ||W^{[2]}||^2\right)$$

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.



Answer:

(c) [**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.

Answer: Accuracies:
(baseline) 0.928700
(regularized) 0.967600

6. **[20 points] Bayesian Interpretation of Regularization**

Background: In Bayesian statistics, almost every quantity is a random variable, which can either be observed or unobserved. For instance, parameters θ are generally unobserved random variables, and data x and y are observed random variables. The joint distribution of all the random variables is also called the *model* (*e.g.,* $p(x, y, \theta)$). Every unknown quantity can be estimated by conditioning the model on all the observed quantities. Such a conditional distribution over the unobserved random variables, conditioned on the observed random variables, is called the *posterior distribution*. For instance $p(\theta|x, y)$ is the posterior distribution in the machine learning context. A consequence of this approach is that we are required to endow our model parameters, *i.e.,* $p(\theta)$, with a *prior distribution*. The prior probabilities are to be assigned *before* we see the data—they capture our prior beliefs of what the model parameters might be before observing any evidence.

In the purest Bayesian interpretation, we are required to keep the entire posterior distribution over the parameters all the way until prediction, to come up with the *posterior predictive distribution*, and the final prediction will be the expected value of the posterior predictive distribution. However in most situations, this is computationally very expensive, and we settle for a compromise that is *less pure* (in the Bayesian sense).

The compromise is to estimate a point value of the parameters (instead of the full distribution) which is the mode of the posterior distribution. Estimating the mode of the posterior distribution is also called *maximum a posteriori estimation* (MAP). That is,

$$\theta_{\text{MAP}} = \arg\max_\theta p(\theta|x, y).$$

Compare this to the *maximum likelihood estimation* (MLE) we have seen previously:

$$\theta_{\text{MLE}} = \arg\max_\theta p(y|x, \theta).$$

In this problem, we explore the connection between MAP estimation, and common regularization techniques that are applied with MLE estimation. In particular, you will show how the choice of prior distribution over θ (*e.g.,* Gaussian or Laplace prior) is equivalent to different kinds of regularization (*e.g.,* L_2, or L_1 regularization). To show this, we shall proceed step by step, showing intermediate steps.

(a) [3 points] Show that $\theta_{\text{MAP}} = \text{argmax}_\theta\, p(y|x, \theta)p(\theta)$ if we assume that $p(\theta) = p(\theta|x)$. The assumption that $p(\theta) = p(\theta|x)$ will be valid for models such as linear regression where the input x are not explicitly modeled by θ. (Note that this means x and θ are marginally independent, but not conditionally independent when y is given.)

Answer: $\theta_{\mathsf{MAP}} = \text{argmax}_\theta \frac{p(y|x,\theta)p(\theta|x)}{p(y|x)} = \text{argmax}_\theta\, p(y|x, \theta)p(\theta)$

(b) [5 points] Recall that L_2 regularization penalizes the L_2 norm of the parameters while minimizing the loss (*i.e.,* negative log likelihood in case of probabilistic models). Now we will show that MAP estimation with a zero-mean Gaussian prior over θ, specifically $\theta \sim \mathcal{N}(0, \eta^2 I)$, is equivalent to applying L_2 regularization with MLE estimation. Specifically, show that

$$\theta_{\text{MAP}} = \arg\min_\theta -\log p(y|x, \theta) + \lambda||\theta||_2^2.$$

Also, what is the value of λ?

Answer: $\theta_{\mathsf{MLE}} = \text{argmax}_\theta\, p(y|x, \theta)p(\theta) = \text{argmax}_\theta[logp(y|x, \theta) + logp(\theta)] =$
$\text{argmin}_\theta[-logp(y|x, \theta) + \lambda||\theta||_2^2]$
$\lambda = \frac{1}{2\eta^2}$

(c) [7 points] Now consider a specific instance, a linear regression model given by $y = \theta^T x + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume that the random noise $\epsilon^{(i)}$ is independent for every training example $x^{(i)}$. Like before, assume a Gaussian prior on this model such that $\theta \sim \mathcal{N}(0, \eta^2 I)$. For notation, let X be the design matrix of all the training example inputs where each row vector is one example input, and $\vec{y}$ be the column vector of all the example outputs.

Come up with a closed form expression for θ_{MAP}.

Answer: $p(\epsilon^{(i)}) = \frac{1}{\sqrt{(2\pi\sigma)}} exp(-\frac{(\epsilon^{(i)})^2}{2\sigma^2})$
$\nabla_\theta (C + \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}) + \frac{1}{2\eta^2}||\theta||_2^2) = 0$
$\nabla_\theta \frac{1}{2\sigma^2}(X\theta - y)^T(X\theta - y) + \frac{1}{2\eta^2}||\theta||_2^2 = \frac{1}{\sigma^2}(X^T X\theta - X^T y) + \frac{1}{\eta^2}\theta = 0$
$\theta_{MAP} = (X^T X + \frac{\sigma^2}{\eta^2} I)^{-1} X^T y$

(d) [5 points] Next, consider the Laplace distribution, whose density is given by

$$f_{\mathcal{L}}(z|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|z - \mu|}{b}\right).$$

As before, consider a linear regression model given by $y = x^T\theta + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume a Laplace prior on this model, where each parameter θ_i is marginally independent, and is distributed as $\theta_i \sim \mathcal{L}(0, b)$.

Show that θ_{MAP} in this case is equivalent to the solution of linear regression with L_1 regularization, whose loss is specified as

$$J(\theta) = ||X\theta - \vec{y}||_2^2 + \gamma||\theta||_1$$

Also, what is the value of γ?

Note: A closed form solution for linear regression problem with L_1 regularization does not exist. To optimize this, we use gradient descent with a random initialization and solve it numerically.

Answer: $\theta_{MAP} = \text{argmax}_\theta\, p(y|x,\theta)p(\theta) = \text{argmin}_\theta[C_1 + \frac{(X\theta - y)^T(X\theta - y)}{2\sigma^2} + C_2 + \frac{||\theta||_1}{b}] =$ $\text{argmin}_\theta[||X\theta - y||^2 + \frac{2\sigma^2||\theta||_1}{b}] = \text{argmin}_\theta[J(\theta)]$
$\gamma = \frac{2\sigma^2}{b}$
Numerical solution:
$\nabla_\theta \frac{1}{2\sigma^2}(X\theta - y)^T(X\theta - y) + \frac{1}{b}||\theta||_1 = 0$
$X^T X\theta - X^T y + \frac{2\sigma^2}{b}\nabla_\theta||\theta||_1 = 0$
$\theta := (X^T X)^{-1}[X^T y - \frac{2\sigma^2}{b}\nabla_\theta||\theta||_1]$

Remark: Linear regression with L_2 regularization is also commonly called *Ridge regression*, and when L_1 regularization is employed, is commonly called *Lasso regression*. These regularizations can be applied to any Generalized Linear models just as above (by replacing $\log p(y|x, \theta)$ with the appropriate family likelihood). Regularization techniques of the above type are also called *weight decay*, and *shrinkage*. The Gaussian and Laplace priors encourage the parameter values to be closer to their mean (*i.e.,* zero), which results in the shrinkage effect.

Remark: Lasso regression (*i.e.,* L_1 regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.