XCS229i Problem Set 4

Due Sunday, February 7 at 11:59pm PT.

Guidelines

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 Slack channel, at http://xcs229i-scpd.slack.com/
3. Familiarize yourself with the collaboration and honor code policy before starting work.
4. For the coding problems, you may not use any libraries except those defined in the provided started code. In particular, ML-specific libraries such as `scikit-learn` are not permitted.

Submission Instructions

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. Your code will be autograded online using `src/grader.py`, which is provided for you in the `src/` subdirectory. You can also run this autograder on your local computer, although some of the tests will be skipped (since they require the instructor solution code for comparison).

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, 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.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases. These tests so not require the instructor solution code and can therefore be run on your local computer.
- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. Since these tests require the instructor solution code to verify results, only the setup and inputs are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder. When your run the autograder online, these tests will run and you will receive feedback on any errors that might occur.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

---

1. **Spam classification**

In this problem, we will use the Naive Bayes algorithm and an SVM to build a spam classifier.

In recent years, spam on electronic media has been a growing concern. Here, we'll build a classifier to distinguish between real messages, and spam messages. For this class, we will be building a classifier to detect SMS spam messages. We will be using an SMS spam dataset developed by Tiago A. Almedia and José María Gómez Hidalgo which is publicly available on http://www.dt.fee.unicamp.br/~tiago/smsspamcollection [1]

We have split this dataset into training and testing sets and have included them in this assignment as:

- `src-spam/spam_train.tsv`
- `src-spam/spam_test.tsv`

See `src-spam/spam_readme.txt` for more details about this dataset. Please refrain from redistributing these dataset files. The goal of this assignment is to build a classifier from scratch that can tell the difference the spam and non-spam messages using the text of the SMS message.

(a) **[8 points (Coding)]** Implement code for processing the the spam messages into numpy arrays that can be fed into machine learning models. Do this by completing the `get_words()`, `create_dictionary()`, and `transform_text()` functions within our provided `src-spam/submission.py`. Do note the corresponding comments for each function for instructions on what specific processing is required.

The autograder test case `1a-4-basic` will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`.

(b) **[3 points (Coding)]** In this question you are going to implement a Naive Bayes classifier for spam classification with **multinomial event model** and Laplace smoothing.

Code your implementation by completing the `fit_naive_bayes_model()` and `predict_from_naive_bayes_model()` functions in `src-spam/submission.py`.

Now the functions in `src-spam/submission.py` should be able to train a Naive Bayes model. Use autograder test case `1b-1-basic` to compute your prediction accuracy and then save your resulting predictions to `spam_naive_bayes_predictions_(soln)`.

Remark. If you implement Naive Bayes the straightforward way, you will find that the computed $p(x|y) = \prod_i p(x_i|y)$ often equals zero. This is because $p(x|y)$, which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called "underflow.") You'll have to find a way to compute Naive Bayes' predicted class labels without explicitly representing very small numbers such as $p(x|y)$. [**Hint:** Think about using logarithms.]

(c) **[3 points (Coding)]** Intuitively, some tokens may be particularly indicative of an SMS being in a particular class. We can try to get an informal sense of how indicative token i is for the SPAM class by looking at:

$$\log \frac{p(x_j = i|y = 1)}{p(x_j = i|y = 0)} = \log \left(\frac{P(\text{token } i|\text{email is SPAM})}{P(\text{token } i|\text{email is NOTSPAM})} \right).$$

Complete the `get_top_five_naive_bayes_words()` function within the provided code using the above formula. Run autograder test case `1c-1-basic` to obtain the 5 most indicative tokens.

(d) **[6 points (Coding)]** 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.

[1] Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.

2. **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/submission.py`
- `src-mnist/images_train.csv` (unzip `Archive.zip` to access this file)
- `src-mnist/labels_train.csv` (unzip `Archive.zip` to access this file)
- `src-mnist/images_test.csv` (unzip `Archive.zip` to access this file)
- `src-mnist/labels_test.csv` (unzip `Archive.zip` to access this file)

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) **[12 points (Coding)]**

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.

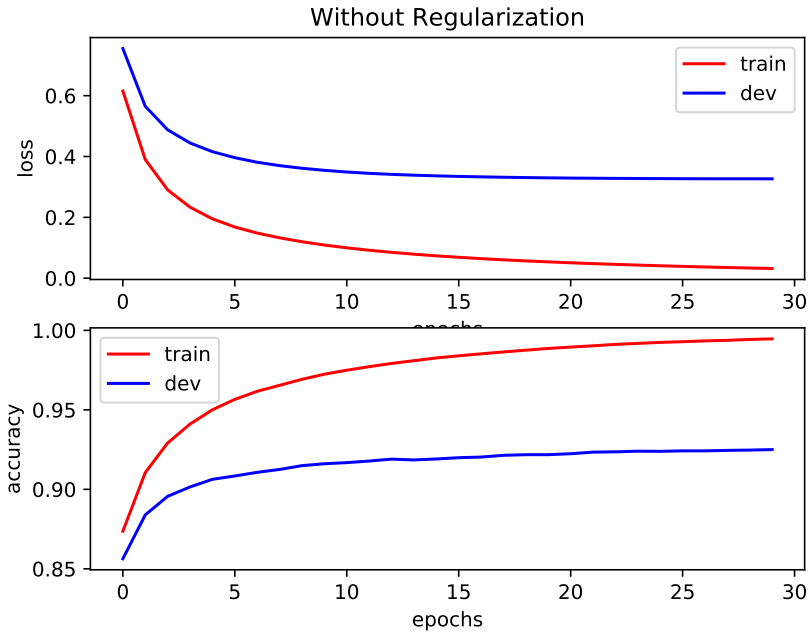
Use autograder test case `2aii-6-basic` to train the model with mini-batch gradient descent as described above. Before running this test case, edit line 188 of `src-mnist/grader.py` to state `skip = False` (model plotting/training is disabled by default to run the auotgrader faster). This will run the training for 30 epochs. At the end of each epoch, it will calculate the value of loss function averaged over the entire training set. It will then plot the average loss (y-axis) against the number of epochs (x-axis). In the same image, it will also plot the value of the loss function averaged over the dev set, and against the number of epochs.

This will also 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, it will plot the accuracy over the dev set versus number of epochs.

Also, at the end of 30 epochs, the autograder will 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.

You plots should look similar to the following (You are not required to submit any plots. These are for your own verification.):



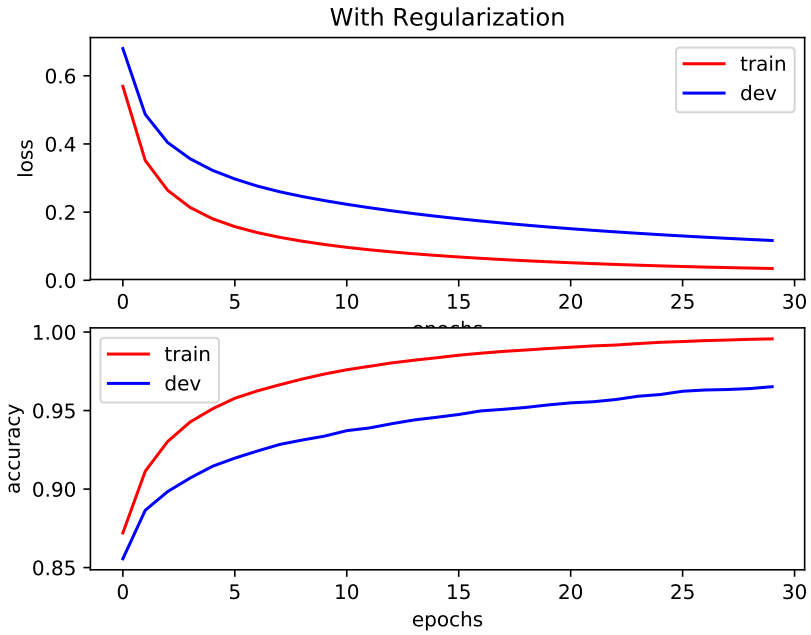
(b) **[6 points (Coding)]** Now add a regularization term to your cross entropy loss by implementing `backward_prop_regularized()`. The loss function will become

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

Autograder test case `2b-2-basic` will perform the same as `2aii-6-basic` (described earlier), except that it utilizes your new regularized backprop function. Before running this test case, edit line 286 of `src-mnist/grader.py` to state `skip = False` (model plotting/training is disabled by default to run the auotgrader faster). It will also plot the same figures as part (a). Note that it does NOT 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).

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

After creating the plots from the previous part, they should look similar to the following (You are not required to submit any plots. These are for your own verification.):



(c) **[2 points (Coding)]** All this while the test cases have avoided 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.

Autograder test case `2c-1-basic` will train your model and then evaluate its performance on the test data for both the regularized and non-regularized training strategies. Before running this test case, edit line 362 of `src-mnist/grader.py` to state `skip = False` (model plotting/training is disabled by default to run the auotgrader faster).

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 have precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

THERE IS NO WRITTEN SUBMISSION FOR THIS ASSIGNMENT.

YOU ARE NOT REQUIRED TO SUBMIT ANYTHING.