CS 229, Spring 2020
Problem Set #4

Due Wednesday, June 10 at 11:59pm 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 `https://piazza.com/stanford/spring2020/cs229`. (3) This quarter, Spring 2020, students may submit in pairs. 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) No late days can be used.

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. **[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.

(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.

(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. 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.

2. **[10 points] PCA**

In class, we showed that PCA finds the "variance maximizing" directions onto which to project the data. In this problem, we find another interpretation of PCA.

Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(n)}\}$. Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

$$f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2.$$

Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$\arg \min_{u: u^T u = 1} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2 \ .$$

gives the first principal component.

Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

3. **[20 points] Independent components analysis**

While studying Independent Component Analysis (ICA) in class, we made an informal argument about why Gaussian distributed sources will not work. We also mentioned that any other distribution (except Gaussian) for the sources will work for ICA, and hence used the logistic distribution instead. In this problem, we will go deeper into understanding why Gaussian distributed sources are a problem. We will also derive ICA with the Laplace distribution, and apply it to the cocktail party problem.

Reintroducing notation, let $s \in \mathbb{R}^d$ be source data that is generated from d independent sources. Let $x \in \mathbb{R}^d$ be observed data such that $x = As$, where $A \in \mathbb{R}^{d \times d}$ is called the *mixing matrix*. We assume A is invertible, and $W = A^{-1}$ is called the *unmixing matrix*. So, $s = Wx$. The goal of ICA is to estimate W. Similar to the notes, we denote w_j^T to be the j^{th} row of W. Note that this implies that the j^{th} source can be reconstructed with w_j and x, since $s_j = w_j^T x$. We are given a training set $\{x^{(1)}, \ldots, x^{(n)}\}$ for the following sub-questions. Let us denote the entire training set by the design matrix $X \in \mathbb{R}^{n \times d}$ where each example corresponds to a row in the matrix.

(a) [5 points] **Gaussian source**

For this sub-question, we assume sources are distributed according to a standard normal distribution, i.e $s_j \sim \mathcal{N}(0, 1), j = \{1, \ldots, d\}$. The log-likelihood of our unmixing matrix, as described in the notes, is

$$\ell(W) = \sum_{i=1}^{n} \left(\log |W| + \sum_{j=1}^{d} \log g'(w_j^T x^{(i)}) \right),$$

where g is the cumulative distribution function, and g' is the probability density function of the source distribution (in this sub-question it is a standard normal distribution). Whereas in the notes we derive an update rule to train W iteratively, for the cause of Gaussian distributed sources, we can analytically reason about the resulting W.

Try to derive a closed form expression for W in terms of X when g is the standard normal CDF. Deduce the relation between W and X in the simplest terms, and highlight the ambiguity (in terms of rotational invariance) in computing W.

(b) [10 points] **Laplace source.**

For this sub-question, we assume sources are distributed according to a standard Laplace distribution, i.e $s_i \sim \mathcal{L}(0, 1)$. The Laplace distribution $\mathcal{L}(0, 1)$ has PDF $f_{\mathcal{L}}(s) = \frac{1}{2} \exp(-|s|)$. With this assumption, derive the update rule for a single example in the form

$$W := W + \alpha(\ldots).$$

(c) [5 points] **Cocktail Party Problem**

For this question you will implement the Bell and Sejnowski ICA algorithm, but assuming a Laplace source (as derived in part-b), instead of the Logistic distribution covered in class. The file `src/ica/mix.dat` contains the input data which consists of a matrix with 5 columns, with each column corresponding to one of the mixed signals x_i. The code for this question can be found in `src/ica/ica.py`.

Implement the `update_W` and `unmix` functions in `src/ica/ica.py`.

You can then run `ica.py` in order to split the mixed audio into its components. The mixed audio tracks are written to `mixed_i.wav` in the output folder. The split audio tracks are written to `split_i.wav` in the output folder.

To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated.)

Submit the full unmixing matrix W (5×5) that you obtained, by including the W.txt the code outputs along with your code.

If you implemention is correct, your output `split_0.wav` should sound similar to the file `correct_split_0.wav` included with the source code.

Note: In our implementation, we **anneal** the learning rate α (slowly decreased it over time) to speed up learning. In addition to using the variable learning rate to speed up convergence, one thing that we also do is choose a random permutation of the training data, and running stochastic gradient ascent visiting the training data in that order (each of the specified learning rates was then used for one full pass through the data).

4. **[15 points] Markov decision processes**

Consider an MDP with finite state and action spaces, and discount factor $\gamma < 1$. Let B be the Bellman update operator with V a vector of values for each state. I.e., if $V' = B(V)$, then

$$V'(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V(s').$$

(a) **[10 points]** Prove that, for any two finite-valued vectors V_1, V_2, it holds true that

$$||B(V_1) - B(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

where

$$||V||_\infty = \max_{s \in S} |V(s)|.$$

(This shows that the Bellman update operator is a "γ-contraction in the max-norm.")

Remark: The result you proved in part(a) implies that value iteration converges geometrically (i.e. exponentially) to the optimal value function V^*.

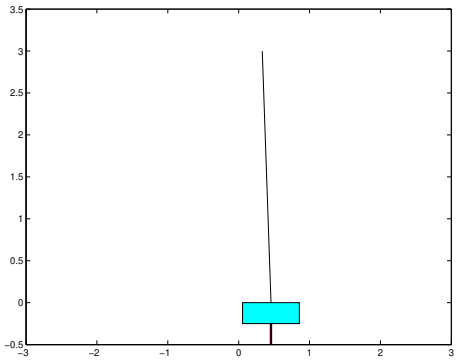
(b) **[5 points]** We say that V is a **fixed point** of B if $B(V) = V$. Using the fact that the Bellman update operator is a γ-contraction in the max-norm, prove that B has at most one fixed point—i.e., that there is at most one solution to the Bellman equations. You may assume that B has at least one fixed point.

5. **[25 points] Reinforcement Learning: The inverted pendulum**

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[1]

Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.



We have written a simple simulator for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it would be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 0 to `NUM_STATES-1`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no *do-nothing* action.) These are represented as actions 0 and 1 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

The files for this problem are in `src/cartpole/` directory. Most of the the code has already been written for you, and you need to make changes only to `cartpole.py` in the places specified. This file can be run to show a display and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation.

[1] The dynamics are adapted from `http://www-anw.cs.umass.edu/rlr/domains.html`

To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO_LEARNING_THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline for this problem is already in `cartpole.py`, and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

Implement the reinforcement learning algorithm as specified, and run it.

- How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged? Hint: if your solution is correct, on the plot the red line indicating smoothed log num steps to failure should start to flatten out at about 60 iterations.
- Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Python starter code already includes the code to plot. Include it in your submission.
- Find the line of code that says np.random.seed, and rerun the code with the seed set to 1, 2, and 3. What do you observe? What does this imply about the algorithm?

If you got here and finished all the above problems, you are done with the final PSet of CS 229! We know these assignments are not easy, so well done :)