

CS 229, Spring 2021

Problem Set #4 Solutions

YOUR NAME HERE (YOUR SUNET HERE)

Due Friday, May 28 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 Ed forum, at <https://edstem.org/us/courses/5279/discussion/>. (3) This quarter, Spring 2021, 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) If you submit after Friday, May 28 at 11:59pm, you will begin consuming your late days. No assignment will be accepted after Monday, May 31 at 11:59pm. Please make sure to reserve sufficient time for potentially compiling, scanning, and uploading your homework questions.

Submission: (1) All students must submit an electronic PDF version of the written questions. You will receive one bonus point (as in, a point on the same scale as in this problem set) for typesetting the homework using L^AT_EX. Bonuses will be applied only after course grade cutoffs have been determined. This is a very minor incentive, but we also think you will find it more convenient overall to typeset. You may submit a non-typeset or even handwritten version, but if you do, please write neatly – illegible responses may receive no credit!

(2) Please make sure to tag your solutions properly on Gradescope. The graders reserve the right to penalize incorrectly tagged solutions by 0.5 points per question.

(3) 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 an auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

1. [24 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 $\{x^{(1)}, \dots, x^{(n)}\}$ 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) [8 points] Gaussian source

For this sub-question, we assume sources are distributed according to a standard normal distribution, i.e $s_j \sim \mathcal{N}(0, 1)$, $j = \{1, \dots, d\}$. 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 .

Let g be the CDF of the standard normal distribution. Derive the relationship between W and X in the simplest form, aiming at a closed-form expression for W in terms of X . If you cannot find one, explain why such a closed form expression does not exist. (Hint: Think in terms of rotational invariance.)

Answer:

(b) [8 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 to solve ICA using gradient ascent¹, in the form

$$W := W + \alpha(\dots)$$

Answer:

(c) [8 points] Cocktail Party Problem

¹In 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.

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`.

First implement the `update_W` and `unmix` functions in `src/ica/ica.py`. Then, you can run `ica.py` 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.

Finally, include the full unmixing matrix W (5×5) that you obtained in `W.txt` in your write-up.

Notes:

- i. Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated. If your implementation is correct, your output `split_0.wav` should sound similar to the file `correct_split_0.wav` included with the source code.
- ii. If your media player reports decoding error, try using the VLC media player.
- iii. Mac users: iTunes may play the original mixed versions even if the program runs correctly. If your program doesn't seem to be doing anything, consider renaming the saved sounds to slightly different filenames to double-check.

Answer:

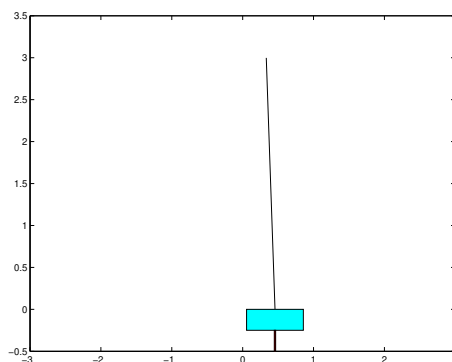
2. [25 points] Reinforcement Learning: The inverted pendulum

Problem Setup

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.²

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.

²The dynamics are adapted from <http://www-anw.cs.umass.edu/rlr/domains.html>

Task

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. There are several details (e.g. convergence criteria) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

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
- Act greedily with respect to this value function

Initially, the estimated reward of each state is 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 should solve Bellman's equation via value iteration to obtain a new value function and optimal policy. When you do the value iteration, use a convergence criterion to check if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, consider the entire learning procedure as converged when 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.

After you implement and execute the reinforcement learning algorithm as specified, include the following items in your write-up:

- Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. The starter code already includes plotting codes.
- Without changing the random seed, how many trials did it take for the algorithm to converge? (i.e. how many trials does it take before the log number of steps before failure becomes stable?)
Hint: if your solution is correct, the red line on the plot should start to flatten out at about 60 iterations.
- Find the line of code that says `np.random.seed`. Set the seed to 1, 2, and 3 and rerun the code for each seed. Describe your observations (you don't need to include the graphs). What do they imply about the algorithm?

Answer:

3. [2 points] Final Survey

https://docs.google.com/forms/d/e/1FAIpQLSevvGvpJPu5oRo2QjoqRoKuyvg0QPysgHZpveZwe3N4-elVsw/viewform?usp=sf_link

This final survey is specific to how we can improve future iterations of 229. We would appreciate your feedback. Even if you only have time to check boxes instead of leaving written comments, that would be valuable to us.

The survey is optional and you will get the 2 points for free regardless, so no need to write anything!

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 :)