CS 229, Spring 2023
Problem Set #4 Solutions

Anthony Weng (`ad2weng`)

Due Thursday, June 8 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/37893/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 Thursday, June 8 at 11:59 pm. If you submit after Thursday, June 8 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.

[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

1. **[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 (CDF), and g' is the probability density function (PDF) 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 case 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.

Answer: To derive a closed form expression for W in terms of X, we will evaluate the gradient of $\ell(W)$ with respect to W, set the resulting quantity equal to 0, and solve for W as follows:

$$\ell(W) = \sum_{i=1}^{n} \left(\log|W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}) \right) \tag{1}$$

$$\nabla_W \ell(W) = \nabla_W \sum_{i=1}^{n} \left(\log|W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}) \right) \tag{2}$$

$$= \sum_{i=1}^{n} \left(\nabla_W \log|W| + \sum_{j=1}^{d} \nabla_W \log\left(\frac{1}{\sqrt{2\pi}} \times \exp\left(-\frac{(w_j^\top x^{(i)})^2}{2}\right)\right) \right) \tag{3}$$

$$= \sum_{i=1}^{n} \left(\frac{1}{|W|} \times (|W|(W^{-1})^\top) + \sum_{j=1}^{d} \nabla_W \log\left(\frac{1}{\sqrt{2\pi}}\right) + \nabla_W \log\left(\exp\left(-\frac{(w_j^\top x^{(i)})^2}{2}\right)\right) \right) \tag{4}$$

$$= \sum_{i=1}^{n} \left((W^{-1})^\top + \sum_{j=1}^{d} 0 + \nabla_W - \frac{(w_j^\top x^{(i)})^2}{2} \right) \tag{5}$$

$$= n(W^{-1})^\top + \sum_{i=1}^{n} \sum_{j=1}^{d} -w_j^\top (x^{(i)} x^{(i)\top}) \tag{6}$$

$$= n(W^{-1})^\top + \sum_{i=1}^{n} -W(x^{(i)} x^{(i)\top}) \tag{7}$$

$$0 = n(W^{-1})^\top - WX^\top X \tag{8}$$

$$WX^\top X = n(W^{-1})^\top \tag{9}$$

$$W^\top W X^\top X = nW^\top (W^{-1})^\top = n(W^{-1}W)^\top = nI \tag{10}$$

$$W^\top W = n(X^\top X)^{-1} \tag{11}$$

Now, let R be an arbitrary orthogonal (i.e., rotation/reflection) matrix such that $RR^\top = R^\top R = I$ and let $W = RW'$ where W' is a rotated version of W. Observe that:

$$W^\top W = n(X^\top X)^{-1} \tag{12}$$
$$(RW')^\top RW' = n(X^\top X)^{-1} \tag{13}$$
$$W'^\top R^\top RW' = n(X^\top X)^{-1} \tag{14}$$
$$W'^\top IW' = n(X^\top X)^{-1} \tag{15}$$
$$W'^\top W' = W^\top W = n(X^\top X)^{-1} \tag{16}$$

Eq. (16) shows that for any candidate unmixing matrix W for our data X, there exists some rotated version of W, W', which is an equally good candidate for being the unmixing matrix – thereby highlighting the ambiguity in computing W from X in terms of the rotational invariance, as desired ■.

(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

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

Answer: The generic gradient ascent update rule is given by:

$$W := W + \alpha \nabla_W \ell(W).$$

For a single example $x^{(i)}$, the likelihood function $\ell(W)$ is:

$$\ell(W) = \log|W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}).$$

We can evaluate $\nabla_W \ell(W)$ as follows:

$$\ell(W) = \log|W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}) \tag{17}$$

$$\nabla_W \ell(W) = \nabla_W \left(\log|W| + \sum_{j=1}^{d} \log g'(w_j^\top x^{(i)}) \right) \tag{18}$$

$$= \nabla_W \log|W| + \sum_{j=1}^{d} \nabla_W \log \frac{1}{2}\exp(-|w_j^\top x^{(i)}|) \tag{19}$$

$$= \frac{1}{|W|} \times |W|(W^{-1})^\top + \sum_{j=1}^{d} \nabla_W \log\frac{1}{2} + \nabla_W \log\exp(-|w_j^\top x^{(i)}|) \tag{20}$$

$$= (W^{-1})^\top + \sum_{j=1}^{d} 0 + \nabla_W(-|w_j^\top x^{(i)}|) \tag{21}$$

$$= (W^{-1})^\top - \sum_{j=1}^{d} \frac{w_j^\top x^{(i)}}{|w_j^\top x^{(i)}|} x^{(i)\top} \tag{22}$$

$$= (W^{-1})^\top - \sum_{j=1}^{d} \mathsf{sign}(w_j^\top x^{(i)}) x^{(i)\top} \tag{23}$$

$$= (W^{-1})^\top - \begin{bmatrix} \mathsf{sign}(w_1^\top x^{(i)}) \\ \cdots \\ \mathsf{sign}(w_d^\top x^{(i)}) \end{bmatrix} x^{(i)\top} \tag{24}$$

We can substitute Eq. (24) into the generic gradient ascent update rule to obtain the Laplace sources-update rule as follows:

$$W := W + \alpha \left((W^{-1})^\top - \begin{bmatrix} \mathsf{sign}(w_1^\top x^{(i)}) \\ \cdots \\ \mathsf{sign}(w_d^\top x^{(i)}) \end{bmatrix} x^{(i)\top} \right).$$

(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 in 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 to 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).

Answer: Obtained unmixing matrix W (also included as `W.txt` in the code submission):

$$W = \begin{bmatrix} 52.833 & 16.795 & 19.941 & -10.198 & -20.897 \\ -9.933 & -0.979 & -4.680 & 8.044 & 1.790 \\ 8.311 & -7.477 & 19.315 & 15.174 & -14.326 \\ -14.667 & -26.645 & 2.441 & 21.382 & -8.421 \\ -0.269 & 18.374 & 9.312 & 9.103 & 30.594 \end{bmatrix}$$

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 $k = 1$.

Answer: First, we will re-express the projection of point x onto a given unit-length vector u, i.e., $f_u(x)$. $f_u(x)$ is given as $f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2$ where $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$. Observe that, for some given u, x, the minimization in $f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2$ is solely a function of α as $v = \alpha u$.

Therefore, computing the projection of x onto u is equivalent to finding the minimizing α^* for the expression $f_u(x) = ||x - v^*||^2 = ||x - \alpha^* u||^2$, which we can do by evaluating this expression's gradient with respect to α, setting it equal to 0, and solving for α^*. We do so as follows:

$$\begin{aligned}
f_u(x) &= ||x - v^*||^2 && (25)\\
&= ||x - \alpha^* u||^2 && (26)\\
&= (x - \alpha u)^\top (x - \alpha u) && (27)\\
&= x^\top x - 2\alpha x^\top u + \alpha^2 u^\top u && (28)\\
\nabla_\alpha f_u(x) &= \nabla_\alpha (x^\top x - 2\alpha x^\top u + \alpha^2 u^\top u) && (29)\\
0 &= 0 - 2x^\top u + 2\alpha u^\top u && (30)\\
2\alpha u^\top u &= 2x^\top u && (31)\\
\alpha^* &= \frac{x^\top u}{u^\top u} = \frac{u \cdot x}{u^\top u} && (32)
\end{aligned}$$

Given this value of the $f_u(x) = ||x - v^*||^2$-minimizing α^*, we can now re-express the projection $f_u(x)$ as follows:

$$f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2 = v^* = \alpha^* u = \left(\frac{u \cdot x}{u^\top u}\right) u.$$

Now, suppose we have some unit-length vector u^* which minimizes the mean squared error between

the projections $f_u(x^{(i)})$ and $x^{(i)}$; i.e., u^* such that:

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

To see that u^* is equal to the first principal component of the data, observe the following:

$$u^* = \arg\min_{u:u^Tu=1} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2 \tag{33}$$

$$= \arg\min_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2 \tag{34}$$

$$= \arg\max_{u:u^Tu=1} -\left(\frac{1}{n} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2\right) \tag{35}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} -\|x^{(i)} - f_u(x^{(i)})\|_2^2 \tag{36}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} -(x^{(i)} - f_u(x^{(i)}))^\top (x^{(i)} - f_u(x^{(i)})) \tag{37}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} (f_u(x^{(i)}) - x^{(i)})^\top (x^{(i)} - f_u(x^{(i)})) \tag{38}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} 2f_u(x^{(i)})^\top x^{(i)} - f_u(x^{(i)})^\top f_u(x^{(i)}) - x^{(i)\top} x^{(i)} \tag{39}$$

In the preceding expression, observe that the value of $x^{(i)\top} x^{(i)}$ is invariant with choice of u and thus may be dropped from the maximization. We will drop this term, substitute our previously-derived expression of $f_u(x) = (\frac{u \cdot x}{u^\top u})u$, and apply our maximization condition $u^Tu = 1$ as necessary to proceed as follows:

$$u^* = \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} 2\left(\frac{u \cdot x^{(i)}}{u^\top u}\right) u^\top x^{(i)} - \left(\frac{u \cdot x^{(i)}}{u^\top u}\right) u^\top \left(\frac{u \cdot x^{(i)}}{u^\top u}\right) u \tag{40}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} 2\frac{u \cdot x^{(i)}}{1}(u \cdot x^{(i)}) - \frac{u \cdot x^{(i)}}{1} \times \frac{u \cdot x^{(i)}}{1} u^\top u \tag{41}$$

$$= \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} 2(u \cdot x^{(i)})^2 - (u \cdot x^{(i)})^2 \times 1 \tag{42}$$

$$u^* = \arg\max_{u:u^Tu=1} \frac{1}{n} \sum_{i=1}^{n} (u \cdot x^{(i)})^2 \tag{43}$$

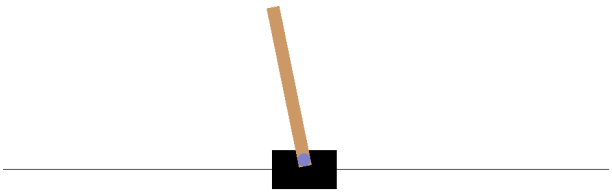
From the preceding equation, it is clear that u^* is precisely the unit-length vector which maximizes the variance of the projections $x^{(i)}$ onto u – i.e., the first principal component, thereby demonstrating that which we wished to prove ■.

3. **[35 points] Reinforcement Learning: Policy Gradient**

Before working on this problem, please run `pip install gym==0.17.3` to install the OpenAI Gym Python dependency.

In this problem you will investigate reinforcement learning, in particular policy gradient methods, as an approach to solving control tasks without explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum problem, also referred to as the pole-balancing problem, provided in the form of the `CartPole-v0` OpenAI Gym environment.[3] The physics setup and details of the MDP are described below, although you do not necessarily have to understand all the details to solve the problem. As shown in the figure below, a thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. Our objective is to develop a controller/policy to balance the pole with these constraints by appropriately having the cart accelerate either left or right. The controller/policy is considered as 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 by going too far left or right).



We have included 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 scalar values: the cart position, the cart velocity, the angle of the pole measured as its deviation from the vertical position, and the angular velocity of the pole. The concatenation of these 4 scalar values is the state s.

At every time step, the controller must choose one of two actions: push (accelerate) the cart left, or push the cart right. (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 according to the underlying dynamics, and provides a new state. If the angle of the pole deviates by more than a certain amount from the vertical position, or if the cart's position goes out of bounds, we conceptually consider that the MDP enters a special "done" state, and once the MDP enters the "done" state, no actions can recover it to any normal state. We choose the horizon T to be 200, meaning, we only take at most 200 actions in each trajectory. Note because once the system enters the "done" state, it stays there

[3] `https://gym.openai.com/envs/CartPole-v0/`

forever, we do not have to simulate the MDP and sample more states anymore after the first time we enter the done state (and all future done states can be ignored because no actions can influence anything). Therefore, in the code, you will only interact with the simulation until the system hits the done state (or reaches a trajectory length of 200), so the effective length of the trajectory can be less than 200. We use $\tilde{T}$ to denote the number of steps with which a trajectory reaches a "done" state or 200 otherwise. The discount factor will be set to $\gamma = 1$ throughout the question.

Our goal is to make the pole and cart stay in bounds without entering the done state for as many steps as possible. To do this, we design the following reward function. For any normal state $s \in \mathbb{R}^4$, we have $R(s) = 1$ (and R does not depend on the action a). When the system is at the "done" state (that is, the pole angle goes beyond a certain limit or when the cart goes too far out), the reward is 0. The reward is given to you in the code as part of the MDP.

The files for this problem are contained within the `src/policy_gradient/` directory. Most of the scaffolding code has already been written for you, and you need to make changes only to `policy_gradient.py` in the places specified. There are also several details that are also clearly outlined inside of the code. This file can then be run to display the behavior of the agent in real time, and to plot a learning curve of the agent at the end.

(a) [6 points] **Policy Gradient**

In this part, we will fully detail the characterization of our policy gradient method and derive the update rule to train our model to solve the `CartPole-v0` environment.

In this problem we will be learning a *logistic* policy. This means that our policy will be a sigmoid function of a linear function in the state. Recall that the sigmoid function $\sigma(z) = 1/(1+e^{-z})$. Let $\theta \in \mathbb{R}^4$ be the parameters of the policy. The probability of taking action 0 (left) is parameterized by

$$\pi_\theta(0|s) = \sigma(\theta^\top s)$$

Given that our environment only contains two actions, the probability of taking action 1 (right) is simply one minus the probability of taking action 0 (left). To be concrete:

$$\pi_\theta(1|s) = 1 - \pi_\theta(0|s) = \sigma(-\theta^\top s)$$

Now recall the gradient of our objective $\eta(\theta)$ in the context of vanilla policy gradient, which is given by the following expression. This value acts as an estimator for the gradient of the expected cumulative reward with respect to the policy parameters.

$$\nabla_\theta \eta(\theta) = \sum_{t=0}^{\tilde{T}-1} \mathbb{E}_{\tau \sim P_\theta} \left[\nabla_\theta \ln \pi_\theta(a_t|s_t) \cdot \left(\sum_{j=0}^{\tilde{T}-1} R(s_j, a_j) \right) \right]$$

Note that this is slightly different from the formula given in the lecture notes because a) the discount factor $\gamma = 1$ in this question, and b) we dropped everything after time step $\tilde{T}$ because once the trajectory enters the done state, all the rewards become zero and the parameter θ doesn't influence $\eta(\theta)$ anymore.

Before we are able to implement our algorithm, we will need to first derive the expression for $\nabla_\theta \ln \pi_\theta(a|s)$. **Derive this value for each action, namely $\nabla_\theta \ln \pi_\theta(0|s)$ and $\nabla_\theta \ln \pi_\theta(1|s)$.** Both of your answers should be as simplified as possible and in terms of θ, s, and the sigmoid function $\sigma(\cdot)$.

Answer: First, we derive $\nabla_\theta \ln \pi_\theta(0 \mid s)$ as follows:

$$\begin{aligned}
\nabla_\theta \ln \pi_\theta(0 \mid s) &= \nabla_\theta \ln(\sigma(\theta^\top s)) && (44)\\
&= \frac{1}{\sigma(\theta^\top s)} \times \nabla_\theta(\sigma(\theta^\top s)) && (45)\\
&= \frac{1}{\sigma(\theta^\top s)} \times \sigma(\theta^\top s) \times (1 - \sigma(\theta^\top s)) \nabla_\theta(\theta^\top s) && (46)\\
&= (1 - \sigma(\theta^\top s)) \times s. && (47)
\end{aligned}$$

We derive $\nabla_\theta \ln \pi_\theta(1 \mid s)$ in similar fashion:

$$\begin{aligned}
\nabla_\theta \ln \pi_\theta(1 \mid s) &= \nabla_\theta \ln(\sigma(-\theta^\top s)) && (48)\\
&= \frac{1}{\sigma(-\theta^\top s)} \times \nabla_\theta(\sigma(-\theta^\top s)) && (49)\\
&= \frac{1}{\sigma(-\theta^\top s)} \times \sigma(-\theta^\top s) \times (1 - \sigma(-\theta^\top s)) \nabla_\theta(-\theta^\top s) && (50)\\
&= -(1 - \sigma(-\theta^\top s)) \times s. && (51)
\end{aligned}$$

(b) [22 points] **Implementation**

Now that we've derived the gradient which will be used to update our model parameters, follow the instructions in `src/policy_gradient/policy_gradient.py` to implement the algorithm. **In particular, implement the following functions:**

i. `sigmoid(x)`
ii. `policy(state)`
iii. `sample_action(state)`
iv. `grad_log_prob(state)`
v. `compute_weights_full_trajectory(episode_rewards)`
vi. `update(episode_rewards, states, actions)`

Once you've finished implementing the above functions, run the experiment via `python policy_gradient.py`. **Include the generated plot `full_trajectory.png` in your write-up.**

Answer:

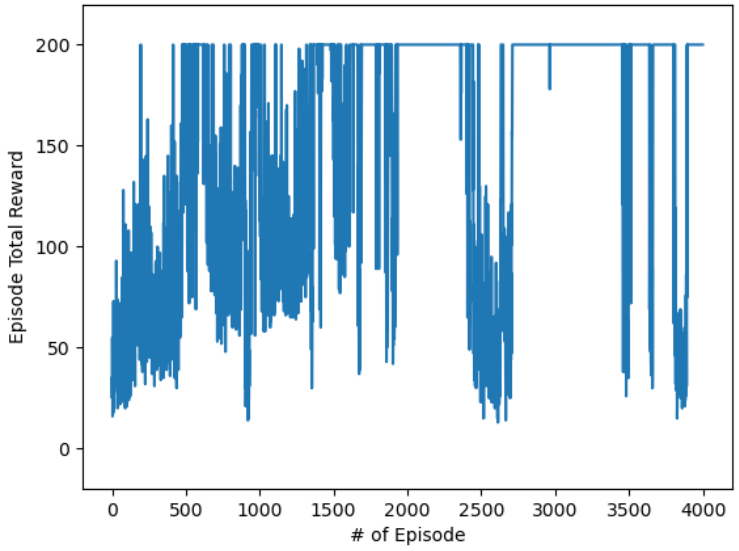


Figure 1: Reward vs. Episode number graph generated using **full trajectory** weights.

(c) [7 points] **Reward-To-Go**

An approach to reducing the variance of the policy gradient is to exploit the fact that our policy cannot impact rewards in the past. This yields the following modified gradient estimator, referred to as the *reward-to-go*, where we multiply $\nabla_\theta \ln \pi(a_t|s_t)$ at each individual time step t by the the sum of future rewards from that time step onward (instead of for all time steps as we did before). The gradient of the objective is given by the following expression:

$$\nabla_\theta \eta(\theta) = \sum_{t=0}^{\tilde{T}-1} \mathbb{E}_{\tau \sim P_\theta} \left[\nabla_\theta \ln \pi(a_t|s_t) \cdot \left(\sum_{j \geq t}^{\tilde{T}-1} R(s_j, a_j) \right) \right]$$

Follow the instructions in `src/policy_gradient/policy_gradient.py` to **implement the function** `compute_weights_reward_to_go(episode_rewards)`. Once you're done, run the new experiment via `python policy_gradient.py --weighting reward_to_go` . **Include the generated plot** `reward_to_go.png` **in your write-up**. Now, **briefly compare the two plots qualitatively** - how does this plot compare to the previous one? Does one estimator of the gradient seem preferable over the other, and what qualities make you say this?

Answer:

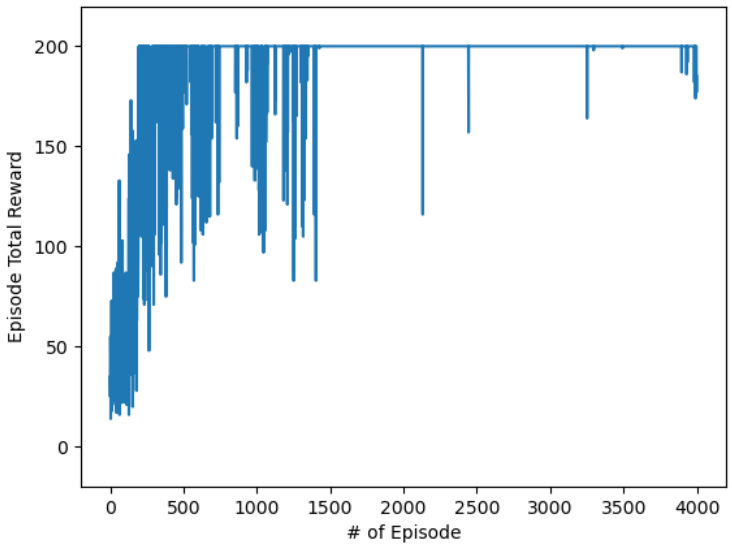


Figure 2: Reward vs. Episode number graph generated using **reward-to-go** weights.

Commentary:

- When comparing the plot from part (c) to that of part (b), the plot from part (c) appears much more *stable* - i.e., after reaching the maximum possible reward (200), there are less fluctuations in the episode total reward as the episode # increases, and any fluctuations are of generally lesser magnitude.

- The **reward-to-go** estimator of the gradient seems preferable over the full trajectory estimator of the gradient.

- Qualities which make me say this include:
  - When using the reward-to-go estimator, errors in states further from the terminal state inform the gradient update more heavily. This quality is desirable as it limits the degree to which our agent over-corrects its policy based on late-in-episode errors; i.e., if our agent makes an error when it has already accumulated an episode reward of 199, said error should induce less "fixing" (i.e., policy adjustment) as compared to an error the agent makes when it has only accumulated an episode reward of 1.

  - Additionally, use of the reward-to-go estimator likely dampens the magnitude of the computed gradients and resulting policy vector θ (since the reward-to-go $\leq$ full trajectory reward at all points in a trajectory). Analogous to regularized regression, a parameter vector with a smaller norm may generalize better, which may explain why using the reward-to-go estimator learns a policy which induces a more stable reward over episodes – i.e., the reward-to-go policy likely generalizes better to unseen states.

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 :)