CS 229, Fall 2020
Problem Set #4

Due Wednesday, November 18 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/fall2020/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 Wednesday, November 18 at 11:59pm. If you submit after Wednesday, November 18 at 11:59pm, you will begin consuming your late days. If you wish to submit on time, submit before Wednesday, November 18 at 11:59pm. Please make sure to reserve sufficient time for potentially compiling, scanning, and uploading your homework questions.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX, and we will award one bonus point for submissions typeset in LaTeX. 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. 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. **[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

$$f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||_2^2.$$

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

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

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 $k = 1$.

2. **[25 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) [10 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.

Deduce the relationship between W and X in the simplest form when g is the standard normal CDF. Then, either derive a closed form expression for W in terms of X when g is the standard normal CDF, or explain why such a closed form expression does not exist. (Hint: Think in terms of rotational invariance.)

(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.
Note: Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated.
Note: If your media player reports decoding error, try using the VLC media player.

Include the full unmixing matrix W (5×5) that you obtained in W.txt in your write-up and code submission.

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).

3. **[20 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) **[15 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.")

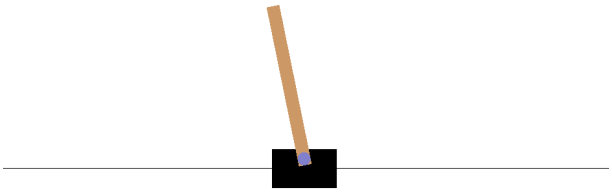
Remark: The result you proved in part(a) implies that value iteration converges geometrically (i.e. exponentially) to the optimal value function V^*.

(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.

4. **[35 points] Reinforcement Learning: Policy Gradient**

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.[1] 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 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

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

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 $\sigma(\cdot)$.

(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.

(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?

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 :)