CS 229, Fall 2018
Problem Set #4: EM, DL, & RL

Due Wednesday, Dec 05 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 Piazza forum, at `http://piazza.com/stanford/fall2018/cs229`. (3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on Handout #1 (available from 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 listed on Gradescope is Dec 08 at 11:59 pm. If you submit after Dec 05, you will begin consuming your late days. If you wish to submit on time, submit before Dec 05 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX. If you are scanning your document by cell phone, please check the Piazza forum for recommended scanning apps and best practices. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. In order to pass the auto-grader tests, 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 when running `p01_nn.py`, `p04_ica.py` and `p06_cartpole.py`. Your submission will be evaluated by the auto-grader using a private test set.

1. **[30 points] Neural Networks: MNIST image classification**

In this problem, you will implement a simple convolutional 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 with only a single channel. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image.

The following shows some example images from the MNIST dataset: [1]

The data for this problem can be found in the data folder as `images_train.csv`, `images_test.csv`, `labels_train.csv` and `labels_test.csv`.

The code for this assignment can be found within `p01_nn.py` within the src folder.

The starter code splits the set of 60,000 training images and labels into a sets of 59,600 examples as the training set and 400 examples for dev set.

To start, you will implement a simple convolutional neural network and cross entropy loss, and train it with the provided data set.

The architecture is as follows:

(a) The first layer is a convolutional layer with 2 output channels with a convolution size of 4 by 4.

(b) The second layer is a max pooling layer of stride and width 5 by 5.

(c) The third layer is a ReLU activation layer.

(d) After the four layer, the data is flattened into a single dimension.

(e) The fith layer is a single linear layer with output size 10 (the number of classes).

(f) The sixth layer is a softmax layer that computes the probabilities for each class.

(g) Finally, we use a cross entropy loss as our loss function.

We have provided all of the forward functions for these different layers so there is an unambigious definition of them in the code. Your job in this assignment will be to implement functions that

[1] https://commons.wikimedia.org/wiki/File:MnistExamples.png

compute the gradients for these layers. However, here is some additional text that might be helpful in understanding the forward functions.

We have discussed convolutional layers on the exam, but as a review, the following equation defines what we mean by a 2d convolution:

$$\begin{aligned} output[out\_channel, x, y] = convolution\_bias[out\_channel] + \\ \sum_{di,dj,in\_channel} input[in\_channel, x + di, y + dy] * \\ convolution\_weights[out\_channel, in\_channel, di, dj] \end{aligned}$$

di and dj iterate through the convolution width and height respectively.

The output of a convolution is of size (# output channels, input width - convolution width + 1, output height - convolution height + 1). Note that the dimension of the output is smaller due to padding issues.

Max pooling layers simply take the maximum element over a grid.

It's defined by the following function

$$output[out\_channel, x, y] = \max_{di,dj} input[in\_channel, x * pool\_width + di, y * pool\_height + dy]$$

The ReLU (rectified linear unit) is our activation function. The ReLU is simply $max(0, x)$ where x is the input.

We use cross entropy loss as our loss function. 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).

We are also doing mini-batch gradient descent with a batch size of 16. Normally we would iterater over the data multiple times with multiple epochs, but for this assignment we only do 400 batches to save time.

(a) **[20 points]**
Implement the following functions within `p01_nn.py`. We recommend that you start at the top of the list and work your way down:
   i. `backward_softmax`
   ii. `backward_relu`
   iii. `backward_cross_entropy_loss`
   iv. `backward_linear`
   v. `backward_convolution`
   vi. `backward_max_pool`

(b) **[10 points]** Now implement a function that computes the full backward pass.
   i. `backward_prop`

2. **[15 points] Off Policy Evaluation And Causal Inference**

In class we have discussed Markov decision processes (MDPs), methods for learning MDPs from data, and ways to compute optimal policies from that MDP. However, before we use that policy, we often want to get an estimate of the its performance. In some settings such as games or simulations, you are able to directly implement that policy and directly measure the performance, but in many situations such as health care implementing and evaluating a policy is very expensive and time consuming.

Thus we need methods for evaluating policies without actually implementing them. This task is usually referred to as off-policy evaluation or causal inference. In this problem we will explore different ways of estimating off policy performance and prove some of the properties of those estimators.

Most of the methods we discuss apply to general MDPs, but for the sake of this problem, we will consider MDPs with a single timestep. We consider a universe consisting of states S, actions A, a reward function $R(s, a)$ where s is a state and a is an action. One important factor is that we often only have a subset of a in our dataset. For example, each state s could represent a patient, each action a could represent which drug we prescribe to that patient and $R(s, a)$ be their lifespan after prescribing that drug.

A policy is defined by a function $\pi_i(s, a) = p(a|s, \pi_i)$. In other words, $\pi_i(s, a)$ is the conditional probability of an action given a certain state and a policy.

We are given an observational dataset consisting of $(s, a, R(s, a))$ tuples.

Let $p(s)$ denote the probability density function for the distribution of state s values within that dataset. Let $\pi_0(s, a) = p(a|s)$ within our observational data. π_0 corresponds to the baseline policy present in our observational data. Going back to the patient example, $p(s)$ would be the probability of seeing a particular patient s and $\pi_0(s, a)$ would be the probability of a patient receiving a drug in the observational data.

We are also given a target policy $\pi_1(s, a)$ which gives the conditional probability $p(a|s)$ in our optimal policy that we are hoping to evaluate. One particular note is that even though this is a distribution, many of the policies that we hope to evaluate are deterministic such that given a particular state s_i, $p(a|s_i) = 1$ for a single action and $p(a|s) = i$ for the other actions.

Our goal is to compute the expected value of $R(s, a)$ in the same population as our observational data, but with a policy of π_1 instead of π_0. In other words, we are trying to compute:

$$\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} R(s, a)$$

Important Note About Notation And Simplifying Assumptions:

We haven't really covered expected values over multiple variables such as $\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} R(s, a)$ in class yet. For the purpose of this question, you may make the simplifying assumption that our states and actions are discrete distributions. This expected value over multiple variables simply indicates that we are taking the expected value over the joint pair (s, a) where s comes from $p(s)$ and a comes from $\pi_1(s, a)$. In other words, you have a $p(s, a)$ term which is the probabilities of observing that pair and we can factorize that probability to $p(s)p(a|s) = p(s)\pi_1(s, a)$. In math notation, this can be written as:

$$\begin{aligned}\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} R(s,a) &= \sum_{(s,a)} R(s,a)p(s,a) \\ &= \sum_{(s,a)} R(s,a)p(s)p(a|s) \\ &= \sum_{(s,a)} R(s,a)p(s)\pi_1(s,a)\end{aligned}$$

Unfortunately, we cannot estimate this directly as we only have samples created under policy π_0 and not π_1. For this problem, we will be looking at formulas that approximate this value using expectations under π_0 that we can actually estimate.

We will make one additional assumption that each action has a non-zero probability in the observed policy $\pi_0(s,a)$. In other words, for all actions a and states s, $\pi_0(s,a) > 0$.

Regression: The simplest possible estimator is to directly use our learned MDP parameters to estimate our goal. This is usually called the regression estimator. While training our MDP, we learn an estimator $\hat{R}(s,a)$ that estimates $R(s,a)$. We can now directly estimate

$$\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} R(s,a)$$

with

$$\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} \hat{R}(s,a)$$

If $\hat{R}(s,a) = R(s,a)$, then this estimator is trivially correct.

We will now consider alternative approaches and explore why you might use one estimator over another.

(a) [2 points] **Importance Sampling:** One commonly used estimator is known as the importance sampling estimator. Let $\hat{\pi}_0$ be an estimate of the true π_0. The importance sampling estimator uses that $\hat{\pi}_0$ and has the form:

$$\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_0(s,a)}} \frac{\pi_1(s,a)}{\hat{\pi}_0(s,a)} R(s,a)$$

Please show that if $\hat{\pi}_0 = \pi_0$, then the importance sampling estimator is equal to:

$$\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_1(s,a)}} R(s,a)$$

Note that this estimator only requires us to model π_0 as we have the $R(s,a)$ values for the items in the observational data.

(b) [2 points] **Weighted Importance Sampling:** One variant of the importance sampling estimator is known as the weighted importance sampling estimator. The weighted importance sampling estimator has the form:

$$\frac{\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_0(s,a)}} \frac{\pi_1(s,a)}{\hat{\pi}_0(s,a)} R(s,a)}{\mathbb{E}_{\substack{s\sim p(s)\\ a\sim\pi_0(s,a)}} \frac{\pi_1(s,a)}{\hat{\pi}_0(s,a)}}$$

Please show that if $\hat{\pi}_0 = \pi_0$, then the weighted importance sampling estimator is equal to:

$$\mathbb{E}_{\substack{s\sim p(s) \\ a\sim\pi_1(s,a)}} R(s,a)$$

(c) [2 points] One issue with the weighted importance sampling estimator is that it can be biased in many finite sample situations. In finite samples, we replace the expected value with a sum over the seen values in our observational dataset. Please show that the weighted importance sampling estimator is biased in these situations.

Hint: Consider the case where there is only a single data element in your observational dataset.

(d) [7 points] **Doubly Robust**: One final commonly used estimator is the doubly robust estimator. The doubly robust estimator has the form:

$$\mathbb{E}_{\substack{s\sim p(s) \\ a\sim\pi_0(s,a)}} ((\mathbb{E}_{a\sim\pi_1(s,a)}\hat{R}(s,a)) + \frac{\pi_1(s,a)}{\hat{\pi}_0(s,a)}(R(s,a) - \hat{R}(s,a)))$$

One advantage of the doubly robust estimator is that it works if either $\hat{\pi}_0 = \pi_0$ or $\hat{R}(s,a) = R(s,a)$

i. [4 points] Please show that the doubly robust estimator is equal to $\mathbb{E}_{\substack{s\sim p(s) \\ a\sim\pi_1(s,a)}} R(s,a)$ when $\hat{\pi}_0 = \pi_0$

ii. [3 points] Please show that the doubly robust estimator is equal to $\mathbb{E}_{\substack{s\sim p(s) \\ a\sim\pi_1(s,a)}} R(s,a)$ when $\hat{R}(s,a) = R(s,a)$

(e) [2 points] We will now consider several situations where you might have a choice between the importance sampling estimator and the regression estimator. Please state whether the importance sampling estimator or the regression estimator would probably work best in each situation and explain why it would work better. In all of these situations, your states s consist of patients, your actions a represent the drugs to give to certain patients and your $R(s,a)$ is the lifespan of the patient after receiving the drug.

i. [1 points] Drugs are randomly assigned to patients, but the interaction between the drug, patient and lifespan is very complicated.

ii. [1 points] Drugs are assigned to patients in a very complicated manner, but the interaction between the drug, patient and lifespan is very simple.

3. **[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 $\{x^{(1)}, \ldots, x^{(m)}\}$. 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

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

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 $k = 1$.

4. **[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 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 `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 `p04_ica.py`.

Implement the `update_W` and `unmix` functions in `p04_ica.py`.

You can then run `p04_ica.py` in order to split the mixed audio into its components. The mixed audio tracks are written to `midex_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.)

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).

5. **[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.")

(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.

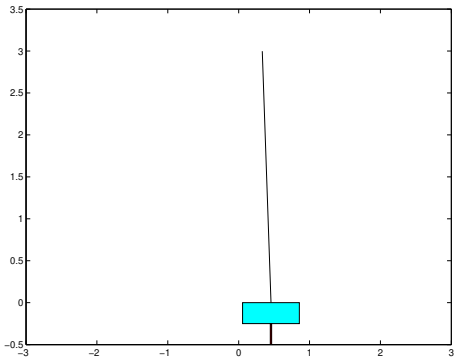
Remark: The result you proved in part(a) implies that value iteration converges geometrically to the optimal value function V^*. That is, after k iterations, the distance between V and V^* is at most γ^k.

6. **[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.[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.

The files for this problem are in `src` directory. Most of the the code has already been written for you, and you need to make changes only to `p06_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.

[2]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 `p06_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?