# CS 229, Fall 2022
# Problem Set #4

---

**Due Friday, December 2 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/30055/discussion/`. (3) This quarter, Fall 2022, all homework assignments must be submitted individually. 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 Friday, December 2 at 11:59 pm. If you submit after Friday, December 2 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Friday, December 2 at 11:59 pm.

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] KL Divergence, Fisher Information, and the Natural Gradient**

As seen before, the Kullback-Leibler divergence between two distributions is an asymmetric measure of how different two distributions are. Consider two distributions over the same space given by densities $p(x)$ and $q(x)$. The KL divergence between two continuous distributions, $q$ and $p$ is defined as,

$$\begin{aligned} D_{KL}(p||q) &= \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \\ &= \int_{-\infty}^{\infty} p(x) \log p(x) dx - \int_{-\infty}^{\infty} p(x) \log q(x) dx \\ &= \mathbb{E}_{x \sim p(x)}[\log p(x)] - \mathbb{E}_{x \sim p(x)}[\log q(x)]. \end{aligned}$$

A nice property of KL divergence is that it invariant to parametrization. This means, KL divergence evaluates to the same value no matter how we parametrize the distributions $P$ and $Q$. For e.g, if $P$ and $Q$ are in the exponential family, the KL divergence between them is the same whether we are using natural parameters, or canonical parameters, or any arbitrary reparametrization.

Now we consider the problem of fitting model parameters using gradient descent (or stochastic gradient descent). As seen previously, fitting model parameters using Maximum Likelihood is equivalent to minimizing the KL divergence between the data and the model. While KL divergence is invariant to parametrization, the gradient w.r.t the model parameters (i.e, direction of steepest descent) is *not invariant to parametrization*. To see its implication, suppose we are at a particular value of parameters (either randomly initialized, or mid-way through the optimization process). The value of the parameters correspond to some probability distribution (and in case of regression, a conditional probability distribution). If we follow the direction of steepest descent from the current parameter, take a small step along that direction to a new parameter, we end up with a new distribution corresponding to the new parameters. The non-invariance to reparametrization means, a step of fixed size in the parameter space could end up in a distribution that could either be extremely far away in $D_{KL}$ from the previous distribution, or on the other hand not move very much at all w.r.t $D_{KL}$ from the previous distributions.

This is where the *natural gradient* comes into picture. It is best introduced in contrast with the usual gradient descent. In the usual gradient descent, we *first choose the direction* in the *parameter space* by calculating the gradient of the MLE objective w.r.t the parameters, and then move a magnitude of step size (where size is measured in the *parameter space*) along that direction. Whereas in natural gradient, we *first choose a divergence* amount by which we would like to move, in the $D_{KL}$ sense. This effectively gives us a perimeter (of some arbitrary shape) around the current parameter, such that all points on this perimeter correspond to distributions which are at an equal $D_{KL}$-distance away from the current parameter. Among the set of all distributions on this perimeter, we move to the distribution that maximizes the objective the most (i.e minimize $D_{KL}$ between data and itself the most). This approach makes the optimization process invariant to parametrization. That means, even if we chose a new arbitrary reparametrization, the natural gradient ensures that by starting from a particular distribution, we always descend down the same sequence of distributions towards the optimum. Whereas the usual gradient will choose a path that is specific to the choice of parametrization.

In the rest of this problem, we will construct and derive the natural gradient update rule. For that, we will break down the process into smaller sub-problems, and give you hints to answer them. Along the way, we will encounter important statistical concepts such as the *score function*

and *Fisher Information* (which play a prominent role in Statistical Theory as well). Finally, we will see how this new natural gradient based optimization is actually equivalent to Newton's method for Generalized Linear Models.

Let the distribution of a random variable $Y$ parameterized by $\theta \in \mathbb{R}^d$ be $p(y;\theta)$.

(a) [3 points] **Score function**

The score function associated with $p(y;\theta)$ is defined as $\nabla_\theta \log p(y;\theta)$, which signifies the sensitivity of the likelihood function with respect to the parameters. Note that the score function is actually a vector since it's the gradient of a scalar quantity with respect to the vector $\theta$.

Recall that $E_{y\sim p(y)}[g(y)] = \int_{-\infty}^{\infty} p(y)g(y)dy$. Using this fact, show that the expected value of the score is 0, i.e.

$$E_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta')|_{\theta'=\theta}] = 0$$

(b) [2 points] **Fisher Information**

Let us now introduce a quantity known as the Fisher information. It is defined as the covariance matrix of the score function,

$$\mathcal{I}(\theta) = \mathrm{Cov}_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta')|_{\theta'=\theta}]$$

Intuitively, the Fisher information represents the amount of information that a random variable $Y$ carries about a parameter $\theta$ of interest. When the parameter of interest is a vector (as in our case, since $\theta \in \mathbb{R}^d$), this information becomes a matrix. Show that the Fisher information can equivalently be given by

$$\mathcal{I}(\theta) = \mathbb{E}_{y\sim p(y;\theta)}[\nabla_{\theta'} \log p(y;\theta') \nabla_{\theta'} \log p(y;\theta')^\top |_{\theta'=\theta}]$$

Note that the Fisher Information is a function of the parameter. The parameter of the Fisher information is both a) the parameter value at which the score function is evaluated, and b) the parameter of the distribution with respect to which the expectation and variance is calculated.

(c) [5 points] **Fisher Information (alternate form)**

It turns out that the Fisher information can not only be defined as the covariance of the score function, but in most situations it can also be represented as the expected negative Hessian of the log-likelihood.

Show that $\mathbb{E}_{y\sim p(y;\theta)}[-\nabla^2_{\theta'} \log p(y;\theta')|_{\theta'=\theta}] = \mathcal{I}(\theta)$.

**Remark**. The Hessian represents the curvature of a function at a point. This shows that the expected curvature of the log-likelihood function is also equal to the Fisher information matrix. If the curvature of the log-likelihood at a parameter is very steep (i.e. Fisher information is very high), this generally means you need fewer data samples to a estimate that parameter well (assuming data was generated from the distribution with those parameters), and vice versa. The Fisher information matrix associated with a statistical model parameterized by $\theta$ is extremely important in determining how a model behaves as a function of the number of training set examples.

(d) [5 points] **Approximating $D_{KL}$ with Fisher Information**

As we explained at the start of this problem, we are interested in the set of all distributions that are at a small fixed $D_{KL}$ distance away from the current distribution. In order to

calculate $D_{KL}$ between $p(y;\theta)$ and $p(y;\theta+d)$, where $d \in \mathbb{R}^d$ is a small magnitude "delta" vector, we approximate it using the Fisher Information at $\theta$. Eventually $d$ will be the natural gradient update we will add to $\theta$. To approximate the KL-divergence with Fisher Information, we will start with the Taylor Series expansion of $D_{KL}$ and see that the Fisher Information pops up in the expansion.

Show that $D_{KL}(p_\theta||p_{\theta+d}) \approx \frac{1}{2}d^T\mathcal{I}(\theta)d$.

Hint: Start with the Taylor Series expansion of $D_{KL}(p_\theta||p_{\tilde{\theta}})$ where $\theta$ is a constant and $\tilde{\theta}$ is a variable. Later set $\tilde{\theta} = \theta + d$. Recall that the Taylor Series allows us to approximate a scalar function $f(\tilde{\theta})$ near $\theta$ by:

$$f(\tilde{\theta}) \approx f(\theta) + (\tilde{\theta}-\theta)^T\nabla_{\theta'}f(\theta')|_{\theta'=\theta} + \frac{1}{2}(\tilde{\theta}-\theta)^T\left(\nabla^2_{\theta'}f(\theta')|_{\theta'=\theta}\right)(\tilde{\theta}-\theta)$$

(e) [8 points] **Natural Gradient**

Now we move on to calculating the natural gradient. Recall that we want to maximize the log-likelihood by moving only by a fixed $D_{KL}$ distance from the current position. In the previous sub-question we came up with a way to approximate $D_{KL}$ distance with Fisher Information. Now we will set up the constrained optimization problem that will yield the natural gradient update $d$. Let the log-likelihood objective be $\ell(\theta) = \log p(y;\theta)$. Let the $D_{KL}$ distance we want to move by, be some small positive constant $c$. The natural gradient update $d^*$ is

$$d^* = \arg\max_d \ell(\theta+d) \quad \text{subject to} \quad D_{KL}(p_\theta||p_{\theta+d}) = c \tag{1}$$

First we note that we can use Taylor approximation on $\ell(\theta+d) \approx \ell(\theta)+d^T\nabla_{\theta'}\ell(\theta')|_{\theta'=\theta}$. Also note that we calculated the Taylor approximation $D_{KL}(p_\theta||p_{\theta+d})$ in the previous subproblem. We shall substitute both these approximations into the above constrained optimization problem.

In order to solve this constrained optimization problem, we employ the *method of Lagrange multipliers*. If you are familiar with Lagrange multipliers, you can proceed directly to solve for $d^*$. If you are not familiar with Lagrange multipliers, here is a simplified introduction. (You may also refer to a slightly more comprehensive introduction in the Convex Optimization section notes, but for the purposes of this problem, the simplified introduction provided here should suffice).

Consider the following constrained optimization problem

$$d^* = \arg\max_d f(d) \quad \text{subject to} \quad g(d) = c$$

The function $f$ is the objective function and $g$ is the constraint. We instead optimize the *Lagrangian* $\mathcal{L}(d,\lambda)$, which is defined as

$$\mathcal{L}(d,\lambda) = f(d) - \lambda[g(d)-c]$$

with respect to both $d$ and $\lambda$. Here $\lambda \in \mathbb{R}_+$ is called the Lagrange multiplier. In order to optimize the above, we construct the following system of equations:

$$\nabla_d\mathcal{L}(d,\lambda) = 0, \tag{a}$$
$$\nabla_\lambda\mathcal{L}(d,\lambda) = 0. \tag{b}$$

So we have two equations (a and b above) with two unknowns ($d$ and $\lambda$), which can be sometimes be solved analytically (in our case, we can).

The following steps guide you through solving the constrained optimization problem:

- Construct the Lagrangian for the constrained optimization problem (1) with the Taylor approximations substituted in for both the objective and the constraint.
- Then construct the system of linear equations (like (a) and (b)) from the Lagrangian you obtained.
- From (a), come up with an expression for $d$ that *involves* $\lambda$.
  At this stage we have already found the "direction" of the natural gradient $d$, since $\lambda$ is only a positive scaling constant. For most practical purposes, the solution we obtain here is sufficient. This is because we almost always include a learning rate hyperparameter in our optimization algorithms, or perform some kind of a line search for algorithmic stability. This can make the exact calculation of $\lambda$ less critical. Let's call this expression $\tilde{d}$ (involving $\lambda$) as the *unscaled natural gradient*. Clearly state what is $\tilde{d}$ as a function of $\lambda$.
  The remaining steps are to figure out the value of the scaling constant $\lambda$ along the direction of $d$, for completeness.
- Plug in that expression for $d$ into (b). Now we have an equation that has $\lambda$ but not $d$. Come up with an expression for $\lambda$ that does *not include* $d$.
- Plug that expression for $\lambda$ (without $d$) back into (a). Now we have an equation that has $d$ but not $\lambda$. Come up with an expression for $d$ that does *not include* $\lambda$.

The expression for $d$ obtained this way will be the desired natural gradient update $d^*$. Clearly state and highlight your final expression for $d^*$. This expression cannot include $\lambda$.

(f) [2 points] **Relation to Newton's Method**

After going through all these steps to calculate the natural gradient, you might wonder if this is something used in practice. We will now see that the familiar Newton's method that we studied earlier, when applied to Generalized Linear Models, is equivalent to natural gradient on Generalized Linear Models. While the two methods (Newton's method and natural gradient) agree on GLMs, in general they need not be equivalent.

Show that the direction of update of Newton's method, and the direction of natural gradient, are exactly the same for Generalized Linear Models. You may want to recall and cite the results you derived in problem set 1 question 4 (Convexity of GLMs). For the natural gradient, it is sufficient to use $\tilde{d}$, the unscaled natural gradient.

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. **[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 "$\gamma$-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 $V^*$.

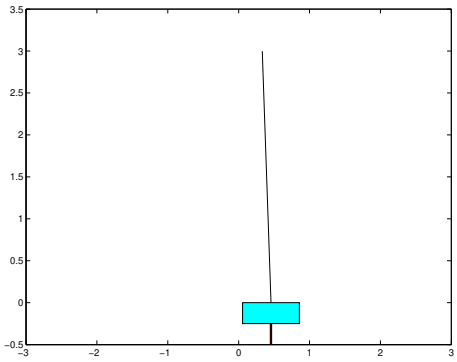
(b) **[5 points]** We say that $V$ is a **fixed point** of $B$ if $B(V) = V$. Using the fact that the Bellman update operator is a $\gamma$-contraction in the max-norm, prove that $B$ has at most one fixed point—i.e., that there is at most one solution to the Bellman equations. You may assume that $B$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

4. **[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 $\theta$ 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 :)**