CS 229, Autumn 2016
Problem Set #4 Solutions: Unsupervised learning & RL

Due Wednesday, December 7 at 11:00 am 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/autumn2016/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 problems that require programming, please include in your submission a printout of your code (with comments) and any figures that you are asked to plot.

If you are scanning your document by cellphone, please check the Piazza forum for recommended cellphone scanning apps and best practices.

1. **[11 points] EM for MAP estimation**

   The EM algorithm that we talked about in class was for solving a maximum likelihood estimation problem in which we wished to maximize

$$\prod_{i=1}^{m} p(x^{(i)};\theta) = \prod_{i=1}^{m} \sum_{z^{(i)}} p(x^{(i)}, z^{(i)};\theta),$$

   where the $z^{(i)}$'s were latent random variables. Suppose we are working in a Bayesian framework, and wanted to find the MAP estimate of the parameters θ by maximizing

$$\left(\prod_{i=1}^{m} p(x^{(i)}|\theta)\right) p(\theta) = \left(\prod_{i=1}^{m} \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}|\theta)\right) p(\theta).$$

   Here, $p(\theta)$ is our prior on the parameters. Generalize the EM algorithm to work for MAP estimation. You may assume that $\log p(x, z|\theta)$ and $\log p(\theta)$ are both concave in θ, so that the M-step is tractable if it requires only maximizing a linear combination of these quantities. (This roughly corresponds to assuming that MAP estimation is tractable when x, z is fully observed, just like in the frequentist case where we considered examples in which maximum likelihood estimation was easy if x, z was fully observed.)

   Make sure your M-step is tractable, and also prove that $\prod_{i=1}^{m} p(x^{(i)}|\theta)p(\theta)$ (viewed as a function of θ) monotonically increases with each iteration of your algorithm.

   Answer: We will derive the EM updates the same way as done in class for maximum likelihood estimation. Monotonic increase with every iteration is guaranteed because of the same reason: in the E-step we compute a lower bound that is tight at the current estimate of θ, in the M-step we optimize θ for this lower bound, so we are guaranteed to improve the actual objective function.

$$
\begin{aligned}
\log \prod_{i=1}^{m} p(x^{(i)}|\theta)p(\theta) &= \log p(\theta) + \sum_{i=1}^{m} \log p(x^{(i)}|\theta) \\
&= \log p(\theta) + \sum_{i=1}^{m} \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}|\theta) \\
&= \log p(\theta) + \sum_{i=1}^{m} \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)}|\theta)}{Q_i(z^{(i)})} \\
&\geq \log p(\theta) + \sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}|\theta)}{Q_i(z^{(i)})},
\end{aligned}
$$

where we just did straightforward substitutions and rewritings, and the last step is given by Jensen's inequality. Requiring the inequality to be tight, gives us the E-step:

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \theta).$$

For the M-step we maximize the lower bound, i.e.

$$\theta = \arg\max_{\theta} \left[\log p(\theta) + \sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}|\theta)}{Q_i(z^{(i)})} \right].$$

The M-step is tractable, since it only requires maximizing a linear combination of tractable concave terms $\log p(x, z|\theta)$ and $\log p(\theta)$.

2. **[22 points] EM application**

Consider the following problem. There are P papers submitted to a machine learning conference. Each of R reviewers reads each paper, and gives it a score indicating how good he/she thought that paper was. We let $x^{(pr)}$ denote the score that reviewer r gave to paper p. A high score means the reviewer liked the paper, and represents a recommendation from that reviewer that it be accepted for the conference. A low score means the reviewer did not like the paper.

We imagine that each paper has some "intrinsic," true value that we denote by μ_p, where a large value means it's a good paper. Each reviewer is trying to estimate, based on reading the paper, what μ_p is; the score reported $x^{(pr)}$ is then reviewer r's guess of μ_p.

However, some reviewers are just generally inclined to think all papers are good and tend to give all papers high scores; other reviewers may be particularly nasty and tend to give low scores to everything. (Similarly, different reviewers may have different amounts of variance in the way they review papers, making some reviewers more consistent/reliable than others.) We let ν_r denote the "bias" of reviewer r. A reviewer with bias ν_r is one whose scores generally tend to be ν_r higher than they should be.

All sorts of different random factors influence the reviewing process, and hence we will use a model that incorporates several sources of noise. Specifically, we assume that reviewers' scores are generated by a random process given as follows:

$$
\begin{aligned}
y^{(pr)} &\sim \mathcal{N}(\mu_p, \sigma_p^2), \\
z^{(pr)} &\sim \mathcal{N}(\nu_r, \tau_r^2), \\
x^{(pr)}|y^{(pr)}, z^{(pr)} &\sim \mathcal{N}(y^{(pr)} + z^{(pr)}, \sigma^2).
\end{aligned}
$$

The variables $y^{(pr)}$ and $z^{(pr)}$ are independent; the variables (x, y, z) for different paper-reviewer pairs are also jointly independent. Also, we only ever observe the $x^{(pr)}$'s; thus, the $y^{(pr)}$'s and $z^{(pr)}$'s are all latent random variables.

We would like to estimate the parameters $\mu_p, \sigma_p^2, \nu_r, \tau_r^2$. If we obtain good estimates of the papers' "intrinsic values" μ_p, these can then be used to make acceptance/rejection decisions for the conference.

We will estimate the parameters by maximizing the marginal likelihood of the data $\{x^{(pr)}; p = 1, \ldots, P, r = 1, \ldots, R\}$. This problem has latent variables $y^{(pr)}$ and $z^{(pr)}$, and the maximum likelihood problem cannot be solved in closed form. So, we will use EM. Your task is to derive the EM update equations. Your final E and M step updates should consist only of addition/subtraction/multiplication/division/log/exp/sqrt of scalars; and addition/subtraction/multiplication/inverse/determinant of matrices. For simplicity, you need to treat only $\{\mu_p, \sigma_p^2; p = 1 \ldots P\}$ and $\{\nu_r, \tau_r^2; r = 1 \ldots R\}$ as parameters. I.e. treat σ^2 (the conditional variance of $x^{(pr)}$ given $y^{(pr)}$ and $z^{(pr)}$) as a fixed, known constant.

(a) In this part, we will derive the E-step:

(i) The joint distribution $p(y^{(pr)}, z^{(pr)}, x^{(pr)})$ has the form of a multivariate Gaussian density. Find its associated mean vector and covariance matrix in terms of the parameters $\mu_p, \sigma_p^2, \nu_r, \tau_r^2$, and σ^2.

[Hint: Recognize that $x^{(pr)}$ can be written as $x^{(pr)} = y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}$, where $\epsilon^{(pr)} \sim \mathcal{N}(0, \sigma^2)$ is independent Gaussian noise.]

(ii) Derive an expression for $Q_{pr}(y^{(pr)}, z^{(pr)}) = p(y^{(pr)}, z^{(pr)}|x^{(pr)})$ (E-step), using the rules for conditioning on subsets of jointly Gaussian random variables (see the notes on Factor Analysis).

(b) Derive the M-step updates to the parameters $\{\mu_p, \nu_r, \sigma_p^2, \tau_r^2\}$. [Hint: It may help to express the lower bound on the likelihood in terms of an expectation with respect to $(y^{(pr)}, z^{(pr)})$ drawn from a distribution with density $Q_{pr}(y^{(pr)}, z^{(pr)})$.]

Remark. In a recent machine learning conference, John Platt (whose SMO algorithm you've seen) implemented a method quite similar to this one to estimate the papers' true scores μ_p. (There, the problem was a bit more complicated because not all reviewers reviewed every paper, but the essential ideas are the same.) Because the model tried to estimate and correct for reviewers' biases ν_r, its estimates of μ_p were significantly more useful for making accept/reject decisions than the reviewers' raw scores for a paper.

Answer:

Let Θ denote the whole set of parameters we are estimating, then the EM steps for our problem are (at a high level):

(a) (E-step) For each p, r, set $Q_{pr}(y^{(pr)}, z^{(pr)}) = p(y^{(pr)}, z^{(pr)}|x^{(pr)}; \theta)$.

(b) (M-step) Set $\Theta = \arg\max_\Theta \sum_{p=1}^{P} \sum_{r=1}^{R} E_{Q_{pr}(Y^{(pr)}, Z^{(pr)})} \log p(x^{(pr)}, Y^{(pr)}, Z^{(pr)}; \Theta)$.

Now it's a matter of working out how these updates can actually be computed.

For the E-step, if we use Bayes's Rule to compute $p(y^{(pr)}, z^{(pr)}|x^{(pr)})$, then we'll get integrals of Gaussians in the denominator, which are tough to compute. Instead, observe that

$$p(y^{(pr)}, z^{(pr)}, x^{(pr)}) = p(y^{(pr)}, z^{(pr)})p(x^{(pr)}|y^{(pr)}, z^{(pr)}) = p(y^{(pr)})p(z^{(pr)})p(x^{(pr)}|y^{(pr)}, z^{(pr)})$$

is the product of three Gaussian densities, so it is itself a multivariate Gaussian density. Therefore, the joint distribution $p(y^{(pr)}, z^{(pr)}, x^{(pr)})$ is some type of normal distribution so we can use the rules for conditioning Gaussians to compute the conditional. To get a form for the joint density, we'll exploit the fact that a multivariate Gaussian density is fully parameterized by its mean vector and covariance matrix.

- To compute the mean vector, we'll rewrite the $x^{(pr)}$ in the following way: $x^{(pr)} = y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}$, where $\epsilon^{(pr)} \sim \mathcal{N}(0, \sigma^2)$ is independent Gaussian noise.[1] Then, $E[y^{(pr)}] = \mu_p$, $E[z^{(pr)}] = \nu_r$ and

$$\begin{aligned} E[x^{(pr)}] &= E[y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}] = E[y^{(pr)}] + E[z^{(pr)}] + E[\epsilon^{(pr)}] \\ &= \mu_p + \nu_r + 0 = \mu_p + \nu_r. \end{aligned}$$

- To compute the covariance matrix, observe that $\mathrm{Var}(y^{(pr)}) = \sigma_p^2$, $\mathrm{Var}(z^{(pr)}) = \tau_r^2$, and $\mathrm{Cov}(y^{(pr)}, z^{(pr)}) = \mathrm{Cov}(z^{(pr)}, y^{(pr)}) = 0$ (since $y^{(pr)}$ and $z^{(pr)}$ are independent). Also, since $y^{(pr)}$, $z^{(pr)}$, and $\epsilon^{(pr)}$ are independent, we have

$$\begin{aligned} \mathrm{Var}(x^{(pr)}) = \mathrm{Var}(y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}) &= \mathrm{Var}(y^{(pr)}) + \mathrm{Var}(z^{(pr)}) + \mathrm{Var}(\epsilon^{(pr)}) \\ &= \sigma_p^2 + \tau_r^2 + \sigma^2. \end{aligned}$$

Finally,

$$\begin{aligned} \mathrm{Cov}(y^{(pr)}, x^{(pr)}) &= \mathrm{Cov}(x^{(pr)}, y^{(pr)}) \\ &= \mathrm{Cov}(y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}, y^{(pr)}) \\ &= \mathrm{Cov}(y^{(pr)}, y^{(pr)}) + \mathrm{Cov}(z^{(pr)}, y^{(pr)}) + \mathrm{Cov}(\epsilon^{(pr)}, y^{(pr)}) \\ &= \sigma_p^2 + 0 + 0 = \sigma_p^2. \end{aligned}$$

where the second to last equality follows from independence of $y^{(pr)}$, $z^{(pr)}$ and $\epsilon^{(pr)}$. Similarly, we can show that $\mathrm{Cov}(z^{(pr)}, x^{(pr)}) = \mathrm{Cov}(x^{(pr)}, z^{(pr)}) = \tau_r^2$.

This allows us to write

$$y^{(pr)}, z^{(pr)}, x^{(pr)} \sim \mathcal{N}\left(\begin{bmatrix} \mu_p \\ \nu_r \\ \mu_p + \nu_r \end{bmatrix}, \begin{bmatrix} \sigma_p^2 & 0 & \sigma_p^2 \\ 0 & \tau_r^2 & \tau_r^2 \\ \sigma_p^2 & \tau_r^2 & \sigma_p^2 + \tau_r^2 + \sigma^2 \end{bmatrix} \right)$$

Now we can use the standard results for conditioning on subsets of variables for Gaussians (from the Factor Analysis notes) to obtain:

$$Q_{pr}(y^{(pr)}, z^{(pr)}) = \mathcal{N}\left(\begin{bmatrix} \mu_{pr,Y} \\ \mu_{pr,Z} \end{bmatrix}, \begin{bmatrix} \Sigma_{pr,YY} & \Sigma_{pr,YZ} \\ \Sigma_{pr,ZY} & \Sigma_{pr,ZZ} \end{bmatrix} \right)$$

[1] To see why this follows from the definition in the problem statement, observe that the probability that $\epsilon^{(pr)} = x^{(pr)} - y^{(pr)} - z^{(pr)}$ takes on any specific value ϵ is $p(\epsilon^{(pr)} = \epsilon | y^{(pr)}, z^{(pr)}) = p(x^{(pr)} - y^{(pr)} - z^{(pr)} = \epsilon | y^{(pr)}, z^{(pr)}) = p(x^{(pr)} = \epsilon + y^{(pr)} + z^{(pr)} | y^{(pr)}, z^{(pr)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2\sigma^2}\epsilon^2)$ which does not depend on either $y^{(pr)}$ or $z^{(pr)}$; hence $\epsilon^{(pr)}$ can be regarded as independent zero-mean Gaussian noise with σ^2 variance.

上面求的是边缘分布，让我们考虑一下条件分布的问题，也就是$x_1|x_2$的问题。根据多元高斯分布的定义，$x_1|x_2 \sim N(\mu_{1|2}, \Sigma_{1|2})$。

且

$$\begin{aligned} \mu_{1|2} &= \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2), & (1) \\ \Sigma_{1|2} &= \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. & (2) \end{aligned}$$

where

$$\mu_{pr} = \begin{bmatrix} \mu_{pr,Y} \\ \mu_{pr,Z} \end{bmatrix} = \begin{bmatrix} \mu_p + \frac{\sigma_p^2}{\sigma^2 + \sigma_p^2 + \tau_r^2}(x^{(pr)} - \mu_p - \nu_r) \\ \nu_r + \frac{\tau_r^2}{\sigma^2 + \sigma_p^2 + \tau_r^2}(x^{(pr)} - \mu_p - \nu_r) \end{bmatrix} \quad (1)$$

$$\Sigma_{pr} = \begin{bmatrix} \Sigma_{pr,YY} & \Sigma_{pr,YZ} \\ \Sigma_{pr,ZY} & \Sigma_{pr,ZZ} \end{bmatrix} = \frac{1}{\sigma_p^2 + \tau_r^2 + \sigma^2} \begin{bmatrix} \sigma_p^2(\tau_r^2 + \sigma^2) & -\sigma_p^2\tau_r^2 \\ -\sigma_p^2\tau_r^2 & \tau_r^2(\sigma_p^2 + \sigma^2) \end{bmatrix}. \quad (2)$$

For the M-step, an important realization is that the Q_{pr} distribution is defined in terms of Θ^t, while we want to choose the parameters for the next time step, Θ^{t+1}. This means that the parameters of the Q_{pr} distributions are constant in terms of the parameters we wish to maximize. Maximizing the expected log-likelihood, we have (letting $E_Q[\cdot]$ denote expectations with respect to $Q_{pr}(y^{(pr)}, z^{(pr)})$ for each p and r, respectively),

$$\begin{aligned} \Theta &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} E_Q \log p(x^{(pr)}, y^{(pr)}, z^{(pr)}; \Theta) \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} E_Q \log \left[\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x^{(pr)} - y^{(pr)} - z^{(pr)})^2} \frac{1}{\sqrt{2\pi}\sigma_p} e^{-\frac{1}{2\sigma_p^2}(y^{(pr)} - \mu_p)^2} \frac{1}{\sqrt{2\pi}\tau_r} e^{-\frac{1}{2\tau_r^2}(z^{(pr)} - \nu_r)^2} \right] \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} E_Q \left[\log \frac{1}{(2\pi)^{3/2}\sigma\sigma_p\tau_r} - \frac{1}{2\sigma^2}(x^{(pr)} - y^{(pr)} - z^{(pr)})^2 - \frac{1}{2\sigma_p^2}(y^{(pr)} - \mu_p)^2 - \frac{1}{2\tau_r^2}(y^{(pr)} - \nu_r)^2 \right] \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} E_Q \left[\log \frac{1}{\sigma_p\tau_r} - \frac{1}{2\sigma_p^2}(y^{(pr)} - \mu_p)^2 - \frac{1}{2\tau_r^2}(z^{(pr)} - \nu_r)^2 \right] \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} E_Q \left[\log \frac{1}{\sigma_p\tau_r} - \frac{1}{2\sigma_p^2}((y^{(pr)})^2 - 2y^{(pr)}\mu_p + \mu_p^2) - \frac{1}{2\tau_r^2}((z^{(pr)})^2 - 2z^{(pr)}\nu_r + \nu_r^2) \right] \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} \left[\log \frac{1}{\sigma_p\tau_r} - \frac{1}{2\sigma_p^2}(E_Q[(y^{(pr)})^2] - 2E_Q[y^{(pr)}]\mu_p + \mu_p^2) - \frac{1}{2\tau_r^2}(E_Q[(z^{(pr)})^2] - 2E_Q[z^{(pr)}]\nu_r + \nu_r^2) \right] \\ &= \arg\max_{\Theta} \sum_{p=1}^{P}\sum_{r=1}^{R} \left[\log \frac{1}{\sigma_p\tau_r} - \frac{1}{2\sigma_p^2}(\Sigma_{pr,YY} + \mu_{pr,Y}^2 - 2\mu_{pr,Y}\mu_p + \mu_p^2) - \frac{1}{2\tau_r^2}(\Sigma_{pr,ZZ} + \mu_{pr,Z}^2 - 2\mu_{pr,Z}\nu_r + \nu_r^2) \right]. \end{aligned}$$

where the equality in the last line follows from $E_Q[y^{(pr)}] = \mu_{pr,Y}$ and $E_Q[(y^{(pr)})^2] = (E_Q[(y^{(pr)})^2] - E_Q[y^{(pr)}]^2) + E_Q[y^{(pr)}]^2 = \Sigma_{pr,YY} + \mu_{pr,Y}^2$ (and similarly for $E_Q[z^{(pr)}]$ and

$E_Q[(z^{(pr)})^2]$). Setting derivatives w.r.t. parameters $\mu_p, \nu_r, \sigma_p, \tau_r$ to 0,

$$-\frac{1}{2\sigma_p^2}\sum_{r=1}^{R}(2\mu_p - 2\mu_{pr,Y}) = 0 \quad \Longrightarrow \quad \mu_p = \frac{1}{R}\sum_{r=1}^{R}\mu_{pr,Y} \tag{3}$$

$$-\frac{1}{2\tau_r^2}\sum_{p=1}^{P}(2\nu_r - 2\mu_{pr,Z}) = 0 \quad \Longrightarrow \quad \nu_r = \frac{1}{P}\sum_{p=1}^{P}\mu_{pr,Z} \tag{4}$$

$$\sum_{r=1}^{R}\left[-\frac{1}{\sigma_p} + \frac{1}{\sigma_p^3}(\Sigma_{pr,YY} + \mu_{pr,Y}^2 - 2\mu_{pr,Y}\mu_p + \mu_p^2)\right] = 0 \quad \Longrightarrow \quad \sigma_p^2 = \frac{1}{R}\sum_{r=1}^{R}(\Sigma_{pr,YY} + \mu_{pr,Y}^2 - 2\mu_{pr,Y}\mu_p + \mu_p^2) \tag{5}$$

$$\sum_{p=1}^{P}\left[-\frac{1}{\tau_r} + \frac{1}{\tau_r^3}(\Sigma_{pr,ZZ} + \mu_{pr,Z}^2 - 2\mu_{pr,Z}\nu_r + \nu_r^2)\right] = 0 \quad \Longrightarrow \quad \tau_r^2 = \frac{1}{P}\sum_{p=1}^{P}(\Sigma_{pr,ZZ} + \mu_{pr,Z}^2 - 2\mu_{pr,Z}\nu_r + \nu_r^2) \tag{6}$$

Using the above results, we can restate our E and M steps in terms of actual computations:

(a) (E-step) For each p, r, compute μ_{pr}, Σ_{pr} using equations (1),(2)

(b) (M-step) Compute $\mu_p, \nu_r, \sigma_p^2, \tau_r^2$ using equations (3), (4), (5), (6).

3. **[14 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

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

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 $k = 1$.

Answer: First note we have $f_u(x^{(i)}) = u^T x u$.[2] So we have to solve the following problem:

$$
\begin{aligned}
\arg\min_{u:u^Tu=1} \sum_{i=1}^{m} \|x^{(i)} - f_u(x^{(i)})\|_2^2 &= \arg\min_{u:u^Tu=1} \sum_{i=1}^{m} \|x^{(i)} - u^T x^{(i)} u\|_2^2 \\
&= \arg\min_{u:u^Tu=1} \sum_{i=1}^{m} (x^{(i)} - u^T x^{(i)} u)^T (x^{(i)} - u^T x^{(i)} u) \\
&= \arg\min_{u:u^Tu=1} \sum_{i=1}^{m} (x^{(i)T} x^{(i)} - 2(u^T x^{(i)})^2 + u^T u (u^T x^{(i)})^2) \\
&= \arg\min_{u:u^Tu=1} \sum_{i=1}^{m} (x^{(i)T} x^{(i)} - 2(u^T x^{(i)})^2 + (u^T x^{(i)})^2) \\
&= \arg\min_{u:u^Tu=1} \sum_{i=1}^{m} -(u^T x^{(i)})^2 \\
&= \arg\max_{u:u^Tu=1} u^T \left(\sum_{i=1}^{m} x^{(i)} x^{(i)T} \right) u
\end{aligned}
$$

And the last line corresponds to the optimization problem that defines the first principal component.

4. **[12 points] Independent components analysis**

For this question you will implement the Bell and Sejnowski ICA algorithm, as covered in class. The files you'll need for this problem are in `/afs/ir/class/cs229/ps/ps4/q4`. The file `mix.dat` contains a matrix with 5 columns, with each column corresponding to one of the mixed signals x_i. The file `bellsej.m` contains starter code for your implementation.

Implement and run ICA, and report what was the W matrix you found. Please make your code clean and very concise, and use symbol conventions as in class. To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap in the sources may be present, but the different sources should be pretty clearly separated.)

Note: In our implementation, we **annealed** the learning rate α (slowly decreased it over time) to speed up learning. We briefly describe in `bellsej.m` what we did, but you should feel free to play with things to make it work best for you. In addition to using the variable learning rate to speed up convergence, one thing that we also tried was choosing 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); this is something that you could try, too.

Answer:

```
%--------------------------------------------------------
% ICA
```

[2] To see why, observe that

$$f_u(x) = u \cdot \left(\arg\min_{\alpha} ||x - \alpha u||^2 \right) = u \cdot \left(\arg\min_{\alpha} (x^T x - 2\alpha x^T u + \alpha^2 u^T u) \right) = u \cdot \left(\frac{2x^T u}{2u^T u} \right) = u x^T u$$

where the third equality follows from the fact that the minimum of a convex quadratic function $ax^2 + bx + c$ is given by $x = -\frac{b}{2a}$, and the last equality follows from the fact that u is a unit-length vector.

```
load mix.dat % load mixed sources
Fs = 11025; %sampling frequency being used

% listen to the mixed sources
normalizedMix = 0.99 * mix ./ (ones(size(mix,1),1)*max(abs(mix)));

% handle writing in both matlab and octave
v = version;
if (v(1) <= '3') % assume this is octave
  wavwrite('mix1.wav', normalizedMix(:, 1), Fs, 16);
  wavwrite('mix2.wav', normalizedMix(:, 2), Fs, 16);
  wavwrite('mix3.wav', normalizedMix(:, 3), Fs, 16);
  wavwrite('mix4.wav', normalizedMix(:, 4), Fs, 16);
  wavwrite('mix5.wav', normalizedMix(:, 5), Fs, 16);
else
  wavwrite(normalizedMix(:, 1), Fs, 16, 'mix1.wav');
  wavwrite(normalizedMix(:, 2), Fs, 16, 'mix2.wav');
  wavwrite(normalizedMix(:, 3), Fs, 16, 'mix3.wav');
  wavwrite(normalizedMix(:, 4), Fs, 16, 'mix4.wav');
  wavwrite(normalizedMix(:, 5), Fs, 16, 'mix5.wav');
end

W=eye(5); % initialize unmixing matrix

% this is the annealing schedule I used for the learning rate.
% (We used stochastic gradient descent, where each value in the
% array was used as the learning rate for one pass through the data.)
% Note: If this doesn't work for you, feel free to fiddle with learning
%  rates, etc. to make it work.
anneal = [0.1 0.1 0.1 0.05 0.05 0.05 0.02 0.02 0.01 0.01 ...
      0.005 0.005 0.002 0.002 0.001 0.001];

for iter=1:length(anneal)

   %%%% here comes your code part

   m = size(mix, 1);
   order = randperm(m);
   for i = 1:m
      x = mix(order(i), :)';
      g = 1 ./ (1 + exp(-W * x));
      W = W + anneal(iter) * ((1 - 2 * g) * x' + inv(W'));
   end

end;

%%%% After finding W, use it to unmix the sources.  Place the unmixed sources
%%%% in the matrix S (one source per column).  (Your code.)
```

```
S = mix * W';

S=0.99 * S./(ones(size(mix,1),1)*max(abs(S)));  % rescale each column to have maximum absolute

% now have a listen --- You should have the following five samples:
% * Godfather
% * Southpark
% * Beethoven 5th
% * Austin Powers
% * Matrix (the movie, not the linear algebra construct :-)

v = version;
if (v(1) <= '3') % assume this is octave
  wavwrite('unmix1.wav', S(:, 1), Fs, 16);
  wavwrite('unmix2.wav', S(:, 2), Fs, 16);
  wavwrite('unmix3.wav', S(:, 3), Fs, 16);
  wavwrite('unmix4.wav', S(:, 4), Fs, 16);
  wavwrite('unmix5.wav', S(:, 5), Fs, 16);
else
  wavwrite(S(:, 1), Fs, 16, 'unmix1.wav');
  wavwrite(S(:, 2), Fs, 16, 'unmix2.wav');
  wavwrite(S(:, 3), Fs, 16, 'unmix3.wav');
  wavwrite(S(:, 4), Fs, 16, 'unmix4.wav');
  wavwrite(S(:, 5), Fs, 16, 'unmix5.wav');
end
```

5. **[16 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) **[12 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.")

Answer: First we observe that $|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$. To see why, define $a_f = \arg\max_a f(a)$ and $a_g = \arg\max_a g(a)$, respectively. Then,

$$f(a_f) - g(a_g) \leq f(a_f) - g(a_f) \leq |f(a_f) - g(a_f)| \leq \max_a |f(a) - g(a)|$$
$$g(a_g) - f(a_f) \leq g(a_g) - f(a_g) \leq |g(a_g) - f(a_g)| \leq \max_a |g(a) - f(a)|,$$

where the first inequality in each line follows from the fact that a_g and a_f are the maximizers of g and f, respectively. Combining the results from the two lines, it follows that $|f(a_f) - g(a_g)| \leq \max_a |f(a) - g(a)|$, which is the equivalent to $|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$.

Then, we have

$$\begin{aligned}
\|B(V_1) - B(V_2)\|_\infty &= \max_{s\in S} \left| \gamma \max_{a\in A} \sum_{s'\in S} P_{sa}(s')V_1(s') - \gamma \max_{a\in A} \sum_{s''\in S} P_{sa}(s'')V_2(s'') \right| \\
&\leq \gamma \max_{s\in S} \max_{a\in A} \left| \sum_{s'\in S} P_{sa}(s')(V_1(s') - V_2(s')) \right| \\
&\leq \gamma \max_{s\in S} \max_{a\in A} \sum_{s'\in S} |P_{sa}(s')(V_1(s') - V_2(s'))| \\
&= \gamma \max_{s\in S} \max_{a\in A} \sum_{s'\in S} P_{sa}(s') \, |(V_1(s') - V_2(s'))| \\
&\leq \gamma \max_{s\in S} \max_{a\in A} \max_{s'\in S} |(V_1(s') - V_2(s'))| \\
&= \gamma \max_{s'\in S} |V_1(s') - V_2(s')| \\
&= \gamma \|V_1 - V_2\|_\infty .
\end{aligned}$$

The first equality uses the definition of the Bellman operator (after noticing that $R(s)$ cancels). The second inequality comes from the fact that $|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$ (and some simplification). The third inequality comes from the triangle inequality. The fourth equality comes from the fact that probabilities are nonnegative. The fifth equality follows from the fact that an expectation of a random variable is necessarily less than its maximum value. The sixth equality involves removing maximizations which play no role, and the final equality uses the definition of the max-norm.

(b) [**4 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.

Answer: Suppose that V_1 and V_2 are 2 fixed points of B. We proved that $\|B(V_1) - B(V_2)\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$, but $B(V_1) = V_1$ and $B(V_2) = V_2$ so

$$\|V_1 - V_2\|_\infty \leq \gamma \|V_1 - V_2\|_\infty \quad \Longrightarrow \quad (1-\gamma)\|V_1 - V_2\|_\infty \leq 0.$$

Since $0 \leq \gamma < 1$, then the coefficient $1 - \gamma$ is positive. Dividing through by $1 - \gamma$, and observing that the max-norm is always nonnegative, it follows that $\|V_1 - V_2\|_\infty = 0$, i.e. $V_1 = V_2$.

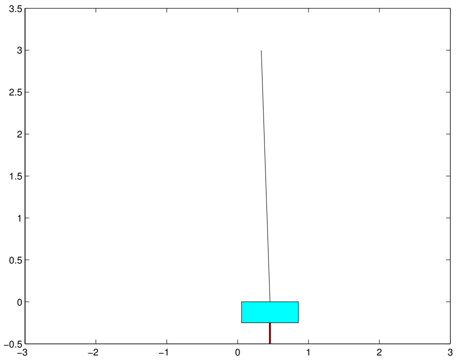
6. **[25 points] Reinforcement Learning: The inverted pendulum**

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.[3]

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

[3] The dynamics are adapted from `http://www-anw.cs.umass.edu/rlr/domains.html`



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 Matlab 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'd 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 1 to `NUM_STATES`. 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 1 and 2 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 `/afs/ir/class/cs229/ps/ps4/q6`. Most of the the code has already been written for you, and you need to make changes only to `control.m` in the places specified. This file can be run in Matlab 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.[4]

(a) 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.

[4] Note that the routine for drawing the cart does not work in Octave.

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 `control.m`, 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?

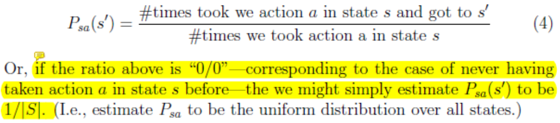
Answer: The number of trials needed varies a good deal, but in the example run shown in the reference solution answer to part (b), 160 trials were needed.

```
%%%%%%% CS 229 Machine Learning %%%%%%%%%%%
%%%%%%% Programming Assignment 4 %%%%%%%%%%
%%%
%%% Parts of the code (cart and pole dynamics, and the state
%%% discretization) are adapted from code available at the RL repository
%%% http://www-anw.cs.umass.edu/rlr/domains.html
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This file controls the pole-balancing simulation. You need to write
% code in places marked "CODE HERE" only.
% Briefly, the main simulation loop in this file calls cart_pole.m for
% simulating the pole dynamics, get_state.m for discretizing the
% otherwise continuous state space in discrete states, and show_cart.m
% for display.

% Some useful parameters are listed below.

% NUM_STATES: Number of states in the discretized state space
% You must assume that states are numbered 1 through NUM_STATES. The
% state numbered NUM_STATES (the last one) is a special state that marks
% the state when the pole has been judged to have fallen (or when the
% cart is out of bounds). However, you should NOT treat this state any
% differently in your code. Any distinctions you need to make between
% states should come automatically from your learning algorithm.

% After each simulation cycle, you are supposed to update the transition
% counts and rewards observed. However, you should not change either
% your value function or the transition probability matrix at each
% cycle.

% Whenever the pole falls, a section of your code below will be
% executed. At this point, you must use the transition counts and reward
% observations that you have gathered to generate a new model for the MDP
% (i.e., transition probabilities and state rewards). After that, you
% must use value iteration to get the optimal value function for this MDP
% model.

% TOLERANCE: Controls the convergence criteria for each value iteration
% run
% In the value iteration, you can assume convergence when the maximum
% absolute change in the value function at any state in an iteration
% becomes lower than TOLERANCE.

% You need to write code that chooses the best action according
% to your current value function, and the current model of the MDP. The
% action must be either 1 or 2 (corresponding to possible directions of
% pushing the cart).

% Finally, we assume that the simulation has converged when
% 'NO_LEARNING_THRESHOLD' consecutive value function computations all
% converged within one value function iteration. Intuitively, it seems
% like there will be little learning after this, so we end the simulation
% here, and say the overall algorithm has converged.


% Learning curves can be generated by calling plot_learning_curve.m (it
% assumes that the learning was just executed, and the array
% time_steps_to_failure that records the time for which the pole was
% balanced before each failure are in memory). num_failures is a variable
% that stores the number of failures (pole drops / cart out of bounds)
% till now.

% Other parameters in the code are described below:

% GAMMA: Discount factor to be used

% The following parameters control the simulation display; you dont
% really need to know about them:

% pause_time: Controls the pause between successive frames of the
% display. Higher values make your simulation slower.
% min_trial_length_to_start_display: Allows you to start the display only
% after the pole has been successfully balanced for at least this many
% trials. Setting this to zero starts the display immediately. Choosing a
% reasonably high value (around 100) can allow you to rush through the
% initial learning quickly, and start the display only after the
% performance is reasonable.

%%%%%%%%%% Simulation parameters %%%%%%%%%%

pause_time = 0.001;
min_trial_length_to_start_display = 0;
display_started = min_trial_length_to_start_display == 0;

NUM_STATES = 163;

GAMMA=0.995;

TOLERANCE=0.01;

NO_LEARNING_THRESHOLD = 20;

%%%%%%%%%%  End parameter list   %%%%%%%%%%

% Time cycle of the simulation
time=0;

% These variables perform bookkeeping (how many cycles was the pole
% balanced for before it fell). Useful for plotting learning curves.
time_steps_to_failure=[];
num_failures=0;
time_at_start_of_current_trial=0;

max_failures=500; % You should reach convergence well before this.

% Starting state is (0 0 0 0)
% x, x_dot, theta, theta_dot represents the actual continuous state vector
x = 0.0; x_dot = 0.0; theta = 0.0; theta_dot = 0.0;
```

```
% state is the number given to this state - you only need to consider
% this representation of the state
state = get_state(x, x_dot, theta, theta_dot);

if display_started==1
  show_cart(x, x_dot, theta, theta_dot, pause_time);
end

%%% CODE HERE: Perform all your initializations here %%%

% Assume no transitions or rewards have been observed
% Initialize the value function array to small random values (0 to 0.10,
% say)
% Initialize the transition probabilities uniformly (ie, probability of
% transitioning for state x to state y using action a is exactly
% 1/NUM_STATES). Initialize all state rewards to zero.

transition_counts = zeros(NUM_STATES, NUM_STATES, 2);
transition_probs = ones(NUM_STATES, NUM_STATES, 2) / NUM_STATES;
reward_counts = zeros(NUM_STATES, 2);
reward = zeros(NUM_STATES, 1);
value = rand(NUM_STATES, 1) * 0.1;

%%%% END YOUR CODE %%%%%%%%%%%%%%%%%%%%%%%%%%%%




%%% CODE HERE (while loop condition) %%%
% This is the criterion to end the simulation
% You should change it to terminate when the previous
% 'NO_LEARNING_THRESHOLD' consecutive value function computations all
% converged within one value function iteration. Intuitively, it seems
% like there will be little learning after this, so end the simulation
% here, and say the overall algorithm has converged.

consecutive_no_learning_trials = 0;
while (consecutive_no_learning_trials < NO_LEARNING_THRESHOLD)

  %%% CODE HERE: Write code to choose action (1 or 2) %%%

  % This action choice algorithm is just for illustration. It may
  % convince you that reinforcement learning is nice for control
  % problems!  Replace it with your code to choose an action that is
  % optimal according to the current value function, and the current MDP
  % model.
```

```
score1 = transition_probs(state, :, 1) * value;
score2 = transition_probs(state, :, 2) * value;
if (score1 > score2)
  action = 1;
elseif (score2 > score1)
  action = 2;
else
  if (rand < 0.5)
    action = 1;
  else
    action = 2;
  end
end

%%% END YOUR CODE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Get the next state by simulating the dynamics
[x, x_dot, theta, theta_dot] = cart_pole(action, x, x_dot, theta, theta_dot);

% Increment simulation time
time = time + 1;

% Get the state number corresponding to new state vector
new_state = get_state(x, x_dot, theta, theta_dot);

if display_started==1
  show_cart(x, x_dot, theta, theta_dot, pause_time);
end

% Reward function to use - do not change this!
if (new_state==NUM_STATES)
  R=-1;
else
  %R=-abs(theta)/2.0;
  R=0;
end


%%% CODE HERE: Perform updates %%%%%%%%%

% A transition from 'state' to 'new_state' has just been made using
% 'action'. The reward observed in 'new_state' (note) is 'R'.
% Write code to update your statistics about the MDP - i.e., the
% information you are storing on the transitions and on the rewards
% observed. Do not change the actual MDP parameters, except when the
% pole falls (the next if block)!

transition_counts(state, new_state, action) = ...
```

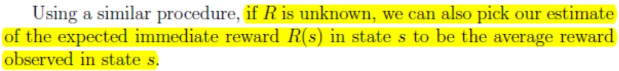
```
 transition_counts(state, new_state, action) + 1;
reward_counts(new_state, 1) = reward_counts(new_state, 1) + R;
reward_counts(new_state, 2) = reward_counts(new_state, 2) + 1;

% Recompute MDP model whenever pole falls
% Compute the value function V for the new model
if (new_state==NUM_STATES)

 % Update MDP model using the current accumulated statistics about the
 % MDP - transitions and rewards.
 % Make sure you account for the case when total_count is 0, i.e., a
 % state-action pair has never been tried before, or the state has
 % never been visited before. In that case, you must not change that
 % component (and thus keep it at the initialized uniform distribution).

 for a = 1:2
  for s = 1:NUM_STATES
   den = sum(transition_counts(s, :, a))
   if (den > 0)
    transition_probs(s, :, a) = transition_counts(s, :, a) / den;
   end
  end
 end
```



$$P_{sa}(s') = \frac{\#\text{times took we action } a \text{ in state } s \text{ and got to } s'}{\#\text{times we took action a in state } s} \qquad (4)$$

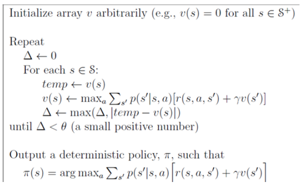
Or, if the ratio above is "0/0"—corresponding to the case of never having taken action a in state s before—the we might simply estimate $P_{sa}(s')$ to be $1/|S|$. (I.e., estimate P_{sa} to be the uniform distribution over all states.)

```
 for s = 1:NUM_STATES
  if (reward_counts(s, 2) > 0)
   reward(s) = reward_counts(s, 1) / reward_counts(s, 2);
  end
 end
```



Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

```
 % Perform value iteration using the new estimated model for the MDP
 % The convergence criterion should be based on TOLERANCE as described
 % at the top of the file.
 % If it converges within one iteration, you may want to update your
 % variable that checks when the whole simulation must end

 iterations = 0;
 new_value = zeros(NUM_STATES, 1);
 while true
  iterations = iterations + 1;
  for s = 1:NUM_STATES
   value1 = transition_probs(s, :, 1) * value;
   value2 = transition_probs(s, :, 2) * value;
   new_value(s) = max(value1, value2);
  end
  new_value = reward + GAMMA * new_value;
  diff = max(abs(value - new_value));
  value = new_value;
  if (diff < TOLERANCE)
```

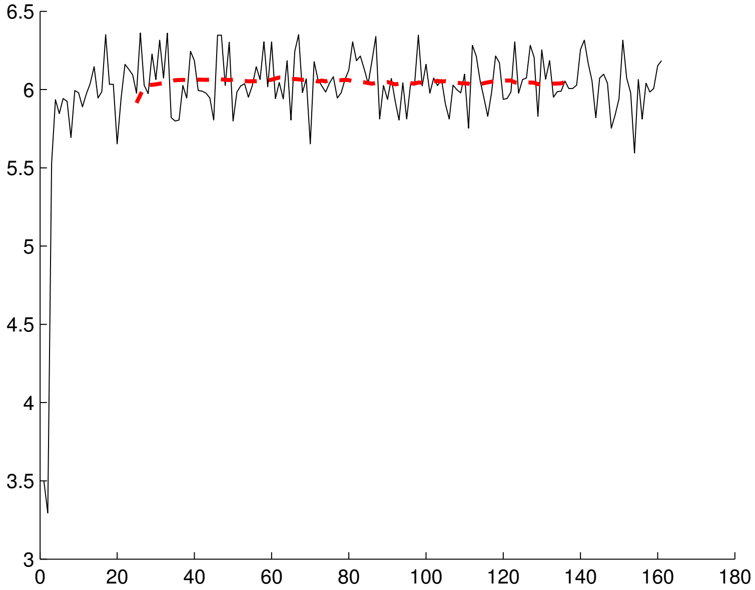


```
Initialize array $v$ arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $temp \leftarrow v(s)$
    $v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
  $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v(s')\Big]$
```

```
        break;
      end
    end

    if (iterations == 1)
      consecutive_no_learning_trials = consecutive_no_learning_trials + 1;
    else
      consecutive_no_learning_trials = 0;
    end

    % pause(0.2); % You can use this to stop for a while!

  end


  %%% END YOUR CODE %%%%%%%%%%%%%%%%%%%


  % Dont change this code: Controls the simulation, and handles the case
  % when the pole fell and the state must be reinitialized
  if (new_state == NUM_STATES)
    num_failures = num_failures+1
    time_steps_to_failure(num_failures) = time - time_at_start_of_current_trial;
    time_at_start_of_current_trial = time;

    time_steps_to_failure(num_failures)

    if (time_steps_to_failure(num_failures) > ...
        min_trial_length_to_start_display)
      display_started=1;
    end

    % Reinitialize state
    x = -1.1 + rand(1)*2.2
    %x=0.0;
    x_dot = 0.0; theta = 0.0; theta_dot = 0.0;
    state = get_state(x, x_dot, theta, theta_dot);
  else
    state=new_state;
  end
end

% Plot the learning curve (time balanced vs trial)
plot_learning_curve
```

(b) Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. You just need to execute `plot_learning_curve.m` after `control.m` to get this plot.

Answer:


6.5
6
5.5
5
4.5
4
3.5
3
0
20
40
60
80
100
120
140
160
180