CS 229, Public Course
Problem Set #4 Solutions: Unsupervised Learning and Reinforcement Learning

---

1. **EM for supervised learning**

   In class we applied EM to the unsupervised learning setting. In particular, we represented $p(x)$ by marginalizing over a latent random variable

   $$p(x) = \sum_z p(x, z) = \sum_z p(x|z)p(z).$$

   However, EM can also be applied to the supervised learning setting, and in this problem we discuss a "mixture of linear regressors" model; this is an instance of what is often call the Hierarchical Mixture of Experts model. We want to represent $p(y|x)$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, and we do so by again introducing a discrete latent random variable

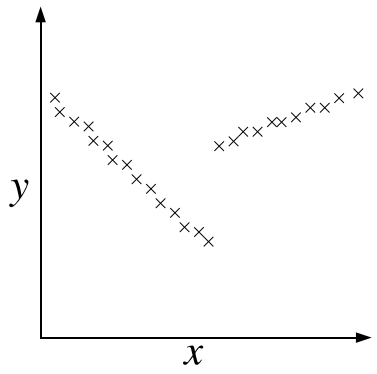
   $$p(y|x) = \sum_z p(y, z|x) = \sum_z p(y|x, z)p(z|x).$$

   For simplicity we'll assume that z is binary valued, that $p(y|x, z)$ is a Gaussian density, and that $p(z|x)$ is given by a logistic regression model. More formally

   $$\begin{aligned} p(z|x;\phi) &= g(\phi^T x)^z (1 - g(\phi^T x))^{1-z} \\ p(y|x, z = i;\theta_i) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y - \theta_i^T x)^2}{2\sigma^2}\right) \quad i = 1, 2 \end{aligned}$$

   where σ is a known parameter and $\phi, \theta_0, \theta_1 \in \mathbb{R}^n$ are parameters of the model (here we use the subscript on θ to denote two different parameter vectors, not to index a particular entry in these vectors).

   Intuitively, the process behind model can be thought of as follows. Given a data point x, we first determine whether the data point belongs to one of two hidden classes $z = 0$ or $z = 1$, using a logistic regression model. We then determine y as a linear function of x (different linear functions for different values of z) plus Gaussian noise, as in the standard linear regression model. For example, the following data set could be well-represented by the model, but not by standard linear regression.

(a) Suppose x, y, and z are all observed, so that we obtain a training set $\{(x^{(1)}, y^{(1)}, z^{(1)}), \ldots, (x^{(m)}, y^{(m)}, z^{(m)})\}$. Write the log-likelihood of the parameters, and derive the maximum likelihood estimates for ϕ, θ_0, and θ_1. Note that because $p(z|x)$ is a logistic regression model, there will not exist a closed form estimate of ϕ. In this case, derive the gradient and the Hessian of the likelihood with respect to ϕ; in practice, these quantities can be used to numerically compute the ML esimtate.

Answer: The log-likelihood is given by

$$\begin{aligned}
\ell(\phi, \theta_0, \theta 1) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, z^{(i)}; \theta_0, \theta_1) p(z^{(i)}|x^{(i)}; \phi) \\
&= \sum_{i:z^{(i)}=0} \log \left((1 - g(\phi^T x)) \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y^{(i)} - \theta_0^T x^{(i)})^2}{2\sigma^2} \right) \right) \\
&\quad + \sum_{i:z^{(i)}=1} \log \left((g(\phi^T x) \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y^{(i)} - \theta_1^T x^{(i)})^2}{2\sigma^2} \right) \right)
\end{aligned}$$

Differentiating with respect to θ_1 and setting it to 0,

$$\begin{aligned}
0 &\overset{\text{set}}{=} \nabla_{\theta_0} \ell(\phi, \theta_0, \theta_1) \\
&= \nabla_\theta \sum_{i:z^{(i)}=0} -(y^{(i)} - \theta_0^T x^{(i)})^2
\end{aligned}$$

But this is just a least-squares problem on a subset of the data. In particular, if we let X_0 and $\vec{y}_0$ be the design matrices formed by considering only those examples with $z^{(i)} = 0$, then using the same logic as for the derivation of the least squares solution we get the maximum likelihood estimate of θ_0,

$$\theta_0 = (X_0^T X_0)^{-1} X_0^T \vec{y}_0.$$

The derivation for θ_1 proceeds in the identical manner.

Differentiating with respect to ϕ, and ignoring terms that do not depend on ϕ

$$\begin{aligned}
\nabla_\phi \ell(\phi, \theta_0, \theta_1) &= \nabla_\phi \sum i : z^{(i)} = 0 \log(1 - g(\phi^T x)) + \sum i : z^{(i)} = 1 \log g(\phi^T x) \\
&= \nabla_\phi \sum_{i=1}^{m} (1 - z^{(i)}) \log(1 - g(\phi^T x)) + z^{(i)} \log g(\phi^T x)
\end{aligned}$$

This is just the standard logistic regression objective function, for which we already know the gradient and Hessian

$$\nabla_\phi \ell(\phi, \theta_0, \theta_1) = X^T(\vec{z} - \vec{h}), \quad \vec{h}_i = g(\phi^T x^{(i)}),$$

$$H = X^T D X, \quad D_{ii} = g(\phi^T x^{(i)})(1 - g(\phi^T x^{(i)})).$$

(b) Now suppose z is a latent (unobserved) random variable. Write the log-likelihood of the parameters, and derive an EM algorithm to maximize the log-likelihood. Clearly specify the E-step and M-step (again, the M-step will require a numerical solution, so find the appropriate gradients and Hessians).

Answer: The log likelihood is now:

$$\begin{aligned}\ell(\phi,\theta_0,\theta_1) &= \log\prod_{i=1}^{m}\sum_{z^{(i)}} p(y^{(i)}|x^{(i)},z^{(i)};\theta_1,\theta_2)p(z^{(i)}|x^{(i)};\phi)\\ &= \sum_{i=1}^{m}\log\left((1-g(\phi^T x^{(i)}))^{1-z^{(i)}}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(\frac{-(y^{(i)}-\theta_0^T x^{(i)})^2}{2\sigma^2}\right)\right.\\ &\qquad\left. + g(\phi^T x^{(i)})^{z^{(i)}}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(\frac{-(y^{(i)}-\theta_1^T x^{(i)})^2}{2\sigma^2}\right)\right)\end{aligned}$$

In the E-step of the EM algorithm we compute

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)},y^{(i)};\phi,\theta_0,\theta_1) = \frac{p(y^{(i)}|x^{(i)},z^{(i)};\theta_0,\theta_1)p(z^{(i)}|x^{(i)};\phi)}{\sum_z p(y^{(i)}|x^{(i)},z;\theta_0,\theta_1)p(z|x^{(i)};\phi)}$$

Every propability in this term can be computed using the probability densities defined in the problem, so the E-step is tractable.

For the M-step, we first define $w_j^{(i)} = p(z^{(i)}=j|x^{(i)},y^{(i)};\phi,\theta_0,\theta_1)$ for $j=0,1$ as computed in the E-step (of course we only need to compute one of these terms in the real E-step, since $w_0^{(i)} = 1-w_1^{(i)}$, but we define both to simplify the expressions). Differentiating our lower bound on the likelihood with respect to θ_0, removing terms that don't depend on θ_0, and setting the expression equal to zero, we get

$$\begin{aligned}0 &\overset{\text{set}}{=} \nabla_{\theta_0}\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log\frac{p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)p(z^{(i)}=j|x^{(i)};\phi)}{w_j^{(i)}}\\ &= \nabla_{\theta_0}\sum_{i=1}^{m} w_0^{(i)}\log p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)\\ &= \nabla_{\theta_0}\sum_{i=1}^{m} -w_0^{(i)}(y^{(i)}-\theta_0^T x^{(i)})^2\end{aligned}$$

This is just a weighted least-squares problem, which has solution

$$\theta_0 = (X_0^T W X_0)^{-1}X_0^T W\vec{y}_0, \qquad W = \text{diag}(w_0^{(1)},\ldots,w_0^{(m)}.$$

The derivation for θ_1 proceeds similarly.

Finally, as before, we can't compute the M-step update for ϕ in closed form, so we instead find the gradient and Hessian. However, to do this we note that

$$\nabla_\phi\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log\frac{p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)p(z^{(i)}=j|x^{(i)};\phi)}{w_j^{(i)}} =$$

$$\nabla_\phi\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log p(z^{(i)}=j|x^{(i)};\phi) = \sum_{i=1}^{m}\left(w_0^{(i)}\log g(\phi^T x) + (1-w_0^{(i)})\log(1-g(\phi^T x^{(i)}))\right)$$

This term is the same as the objective for logistic regression task, but with the $w^{(i)}$ quantity replacing $y^{(i)}$. Therefore, the gradient and Hessian are given by

$$\nabla_\phi \sum_{i=1}^{m} \sum_{j=0,1} w_j^{(i)} \log p(z^{(i)} = j | x^{(i)}; \phi) = X^T(\vec{w} - \vec{h}), \quad \vec{h}_i = g(\phi^T x^{(i)}),$$

$$H = X^T D X, \quad D_{ii} = g(\phi^T x^{(i)})(1 - g(\phi^T x^{(i)})).$$

2. **Factor Analysis and PCA**

In this problem we look at the relationship between two unsupervised learning algorithms we discussed in class: Factor Analysis and Principle Component Analysis.

Consider the following joint distribution over (x, z) where $z \in \mathbb{R}^k$ is a latent random variable

$$\begin{aligned} z &\sim \mathcal{N}(0, I) \\ x|z &\sim \mathcal{N}(Uz, \sigma^2 I). \end{aligned}$$

where $U \in \mathbb{R}^{n \times k}$ is a model parameters and σ^2 is assumed to be a known constant. This model is often called Probabilistic PCA. Note that this is nearly identical to the factor analysis model except we assume that the variance of $x|z$ is a known scaled identity matrix rather than the diagonal parameter matrix, Φ, and we do not add an additional μ term to the mean (though this last difference is just for simplicity of presentation). However, as we will see, it turns out that as $\sigma^2 \to 0$, this model is equivalent to PCA.

For simplicity, you can assume for the remainder of the problem that $k = 1$, i.e., that U is a column vector in $\mathbb{R}^n$.

(a) Use the rules for manipulating Gaussian distributions to determine the joint distribution over (x, z) and the conditional distribution of $z|x$. [Hint: for later parts of this problem, it will help significantly if you simplify your soluting for the conditional distribution using the identity we first mentioned in problem set #1: $(\lambda I + BA)^{-1}B = B(\lambda I + AB)^{-1}$.]

Answer: To compute the joint distribution, we compute the means and covariances of x and z. First, $E[z] = 0$ and

$$E[x] = E[Uz + \epsilon] = UE[z] + E[\epsilon] = 0, \quad (\text{where } \epsilon \sim \mathcal{N}(0, \sigma^2 I)).$$

Since both x and z have zero mean

$$\begin{aligned} \Sigma_{zz} &= E[zz^T] = I \quad (= 1, \text{ since } z \text{ is a scalar when } k = 1) \\ \Sigma_{zx} &= E[(Uz + \epsilon)z^T] = UE[zz^T] + E[\epsilon z^T] = U \\ \Sigma_{xx} &= E[(Uz + \epsilon)(Uz + \epsilon)^T] = E[Uzz^TU^T + \epsilon z^T U^T + Uz\epsilon^T + \epsilon\epsilon^T] \\ &= UE[zz^T]U^T + E[\epsilon\epsilon^T] = UU^T + \sigma^2 I \end{aligned}$$

Therefore,

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & U^T \\ U & UU^T + \sigma^2 I \end{bmatrix} \right).$$

Using the rules for conditional Gaussian distributions, $z|x$ is also Gaussian with mean and covariance

$$\begin{aligned} \mu_{z|x} &= U^T(UU^T + \sigma^2 I)^{-1} x = \frac{U^T x}{U^T U + \sigma^2} \\ \Sigma_{z|x} &= 1 - U^T(UU^T + \sigma^2 I)^{-1} U = 1 - \frac{U^T U}{U^T U + \sigma^2} \end{aligned}$$

where in both cases the last equality comes from the identity in the hint.

(b) Using these distributions, derive an EM algorithm for the model. Clearly state the E-step and the M-step of the algorithm.

Answer: Even though $z^{(i)}$ is a scalar value, in this problem we continue to use the notation ${z^{(i)}}^T$, etc, to make the similarities to the Factor anlysis case obvious.

For the E-step, we compute the distribution $Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; U)$ by computing $\mu_{z^{(i)}|x^{(i)}}$ and $\Sigma_{z^{(i)}|x^{(i)}}$ using the above formulas.

For the M-step, we need to maximize

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, |z^{(i)}; U) p(z^{(i)})}{Q_i(z^{(i)})}$$

$$= \sum_{i=1}^{m} E_{z^{(i)} \sim Q_i} \left[\log p(x^{(i)}|z^{(i)}; U) + \log p(z^{(i)}) - \log Q_i(z^{(i)}) \right].$$

Taking the gradient with respect to U equal to zero, dropping terms that don't depend on U, and omitting the subscript on the expectation, this becomes

$$\begin{aligned}
\nabla_U \sum_{i=1}^{m} E\left[\log p(x^{(i)}|z^{(i)}; U)\right] &= \nabla_U \sum_{i=1}^{m} E\left[-\frac{1}{2\sigma^2}(x^{(i)} - Uz^{(i)})^T(x^{(i)} - Uz^{(i)})\right] \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{m} \nabla_U E\left[\mathrm{tr} {z^{(i)}}^T U^T U z^{(i)} - 2\mathrm{tr} {z^{(i)}}^T U^T x^{(i)}\right] \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{m} E\left[U z^{(i)} {z^{(i)}}^T - x^{(i)} {z^{(i)}}^T\right] \\
&= \frac{1}{2\sigma^2} \sum_{i=1}^{m} \left[-U E[z^{(i)} {z^{(i)}}^T] + x^{(i)} E[{z^{(i)}}^T]\right]
\end{aligned}$$

using the same reasoning as in the Factor Analysis class notes. Setting this derivative to zero gives

$$\begin{aligned}
U &= \left(\sum_{i=1}^{m} x^{(i)} E[{z^{(i)}}^T]\right) \left(\sum_{i=1}^{m} E[z^{(i)} {z^{(i)}}^T]\right)^{-1} \\
&= \left(\sum_{i=1}^{m} x^{(i)} \mu_{z^{(i}|x^{(i)}}^T\right) \left(\sum_{i=1}^{m} \Sigma_{z^{(i)}|x^{(i)}} + \mu_{z^{(i}|x^{(i)}} \mu_{z^{(i}|x^{(i)}}^T\right)^{-1}
\end{aligned}$$

All these terms were calcuated in the E step, so this is our final M step update.

(c) As $\sigma^2 \to 0$, show that if the EM algorithm convergences to a parameter vector $U^\star$ (and such convergence is guarenteed by the argument presented in class), then $U^\star$ must be an eigenvector of the sample covariance matrix $\Sigma = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T$ — i.e., $U^\star$ must satisfy

$$\lambda U^\star = \Sigma U^\star.$$

[Hint: When $\sigma^2 \to 0$, $\Sigma_{z|x} \to 0$, so the E step only needs to compute the means $\mu_{z|x}$ and not the variances. Let $w \in \mathbb{R}^m$ be a vector containing all these means,

$w_i = \mu_{z^{(i)}|x^{(i)}}$, and show that the E step and M step can be expressed as

$$w = \frac{XU}{U^T U}, \quad U = \frac{X^T w}{w^T w}$$

respectively. Finally, show that if U doesn't change after this update, it must satisfy the eigenvector equation shown above.]

Answer: For the E step, when $\sigma^2 \to 0$, $\mu_{z^{(i)}|x^{(i)}} = \frac{U^T x^{(i)}}{U^T U}$, so using w as defined in the hint we have

$$w = \frac{XU}{U^T U}$$

as desired.

As mentioned in the hint, when $\sigma^2 \to 0$, $\Sigma_{z^{(i)}|x^{(i)}} = 0$, so

$$\begin{aligned} U &= \left(\sum_{i=1}^m x^{(i)} \mu_{z^{(i}|x^{(i)}}^T\right) \left(\sum_{i=1}^m \Sigma_{z^{(i)}|x^{(i)}} + \mu_{z^{(i}|x^{(i)}} \mu_{z^{(i}|x^{(i)}}^T\right)^{-1} \\ &= \left(\sum_{i=1}^m x^{(i)} w_i\right) (\sum_{i=1}^m w_i w_i)^{-1} = \frac{X^T w}{w^T w} \end{aligned}$$

For U to remain unchanged after an update requires that

$$U = \frac{X^T \frac{XU}{U^T U}}{\frac{U^T X^T}{U^T U} \frac{XU}{U^T U}} = X^T XU \frac{U^T U}{U^T X^T XU} = X^T XU \frac{1}{\lambda}$$

proving the desired equation.

3. **PCA and ICA for Natural Images**

In this problem we'll apply Principal Component Analysis and Independent Component Analysis to images patches collected from "natural" image scenes (pictures of leaves, grass, etc). This is one of the classical applications of the ICA algorithm, and sparked a great deal of interest in the algorithm; it was observed that the bases recovered by ICA closely resemble image filters present in the first layer of the visual cortex.

The `q3/` directory contains the data and several useful pieces of code for this problem. The raw images are stored in the *images/* subdirectory, though you will not need to work with these directly, since we provide code for loading and normalizing the images.

Calling the function `[X_ica, X_pca] = load_images;` will load the images, break them into 16x16 images patches, and place all these patches into the columns of the matrices `X_ica` and `X_pca`. We create two different data sets for PCA and ICA because the algorithms require slightly different methods of preprocessing the data.[1]

For this problem you'll implement the `ica.m` and `pca.m` functions, using the PCA and ICA algorithms described in the class notes. While the PCA implementation should be straightforward, getting a good implementation of ICA can be a bit trickier. Here is some general advice to getting a good implementation on this data set:

[1] Recall that the first step of performing PCA is to subtract the mean and normalize the variance of the features. For the image data we're using, the preprocessing step for the ICA algorithm is slightly different, though the precise mechanism and justification is not imporant for the sake of this problem. Those who are curious about the details should read Bell and Sejnowki's paper "The 'Independent Components' of Natural Scenes are Edge Filters," which provided the basis for the implementation we use in this problem.

- Picking a good learning rate is important. In our experiments we used $\alpha = 0.0005$ on this data set.
- Batch gradient descent doesn't work well for ICA (this has to do with the fact that ICA objective function is not concave), but the pure stochastic gradient described in the notes can be slow (There are about 20,000 16x16 images patches in the data set, so one pass over the data using the stochastic gradient rule described in the notes requires inverting the 256x256 W matrix 20,000 times). Instead, a good compromise is to use a hybrid stochastic/batch gradient descent where we calculate the gradient with respect to several examples at a time (100 worked well for us), and use this to update W. Our implementation makes 10 total passes over the entire data set.
- It is a good idea to randomize the order of the examples presented to stochastic gradient descent before each pass over the data.
- Vectorize your Matlab code as much as possible. For general examples of how to do this, look at the Matlab review session.

For reference, computing the ICA W matrix for the entire set of image patches takes about 5 minutes on a 1.6 Ghz laptop using our implementation.

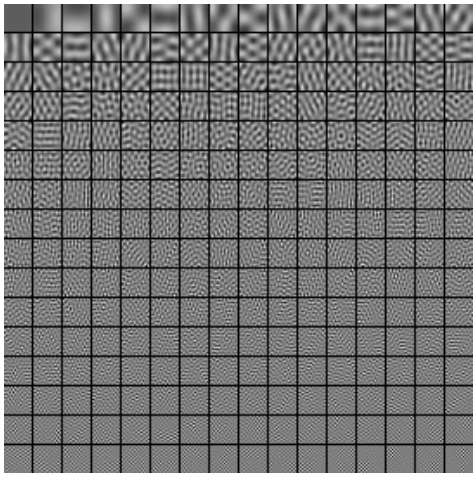
After you've learned the U matrix for PCA (the columns of U should contain the principal components of the data) and the W matrix of ICA, you can plot the basis functions using the `plot_ica_bases(W);` and `plot_pca_bases(U);` functions we have provide. Comment briefly on the difference between the two sets of basis functions.

Answer: The following are our implementations of `pca.m` and `ica.m`:

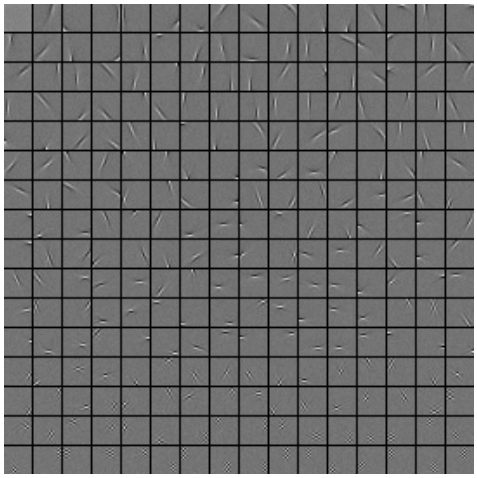
```
function U = pca(X)

[U,S,V] = svd(X*X');
```

```
function W = ica(X)

[n,m] = size(X);
chunk = 100;
alpha = 0.0005;
W = eye(n);

for iter=1:10,
  disp([num2str(iter)]);
  X = X(:,randperm(m));
  for i=1:floor(m/chunk),
    Xc = X(:,(i-1)*chunk+1:i*chunk);
    dW = (1 - 2./(1+exp(-W*Xc)))*Xc' + chunk*inv(W');
    W = W + alpha*dW;
  end
end
```

PCA produces the following bases:



while ICA produces the following bases



The PCA bases capture global features of the images, while the ICA bases capture more local features.

4. **Convergence of Policy Iteration**

   In this problem we show that the Policy Iteration algorithm, described in the lecture notes, is guarenteed to find the optimal policy for an MDP. First, define B^π to be the Bellman operator for policy π, defined as follows: if $V' = B(V)$, then

$$V'(s) = R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s')V(s').$$

(a) Prove that if $V_1(s) \leq V_2(s)$ for all $s \in \mathbb{S}$, then $B(V_1)(s) \leq B(V_2)(s)$ for all $s \in \mathbb{S}$.
Answer:

$$\begin{aligned} B(V_1)(s) &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V_1(s') \\ &\leq R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V_2(s') = B(V_2)(s) \end{aligned}$$

where the inequality holds because $P_{s\pi(s)}(s') \geq 0$.

(b) Prove that for any V,

$$\|B^\pi(V) - V^\pi\|_\infty \leq \gamma \|V - V^\pi\|_\infty$$

where $\|V\|_\infty = \max_{s \in \mathcal{S}} |V(s)|$. Intuitively, this means that applying the Bellman operator B^π to any value function V, brings that value function "closer" to the value function for π, V^π. This also means that applying B^π repeatedly (an infinite number of times)

$$B^\pi(B^\pi(\ldots B^\pi(V)\ldots))$$

will result in the value function V^π (a little bit more is needed to make this completely formal, but we won't worry about that here).
[Hint: Use the fact that for any $\alpha, x \in \mathbb{R}^n$, if $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0$, then $\sum_i \alpha_i x_i \leq \max_i x_i$.] **Answer:**

$$\begin{aligned} \|B^\pi(V) - V^\pi\|_\infty &= \max_{s' \in \mathcal{S}} \left| R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V(s') - R(s) - \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V^\pi(s') \right| \\ &= \gamma \max_{s' \in J} \left| \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') \left(V(s') - V^\pi(s') \right) \right| \\ &\leq \gamma \|V - V^\pi\|_\infty \end{aligned}$$

where the inequality follows from the hint above.

(c) Now suppose that we have some policy π, and use Policy Iteration to choose a new policy π' according to

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^\pi(s').$$

Show that this policy will never perform worse that the previous one — i.e., show that for all $s \in \mathcal{S}$, $V^\pi(s) \leq V^{\pi'}(s)$.
[Hint: First show that $V^\pi(s) \leq B^{\pi'}(V^\pi)(s)$, then use the proceeding excercises to show that $B^{\pi'}(V^\pi)(s) \leq V^{\pi'}(s)$.]
Answer:

$$\begin{aligned} V^\pi(s) &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V^\pi(s') \\ &\leq R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^\pi(s') \\ &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi'(s)}(s') V^\pi(s') = B^{\pi'}(V^\pi)(s) \end{aligned}$$

Applying part (a),

$$V^{\pi}(s) \leq B^{\pi'}(V^{\pi})(s) \Rightarrow B^{\pi'}(V^{\pi})(s) \leq B^{\pi'}(B^{\pi'}(V^{\pi}))(s)$$

Continually applying this property, and applying part (b), we obtain

$$V^{\pi}(s) \leq B^{\pi'}(V^{\pi})(s) \leq B^{\pi'}(B^{\pi'}(V^{\pi}))(s) \leq \ldots \leq B^{\pi'}(B^{\pi'}(\ldots B^{\pi'}(V^{\pi})\ldots))(s) = V^{\pi'}(s).$$

(d) Use the proceeding exercises to show that policy iteration will eventually converge (i.e., produce a policy $\pi' = \pi$). Furthermore, show that it must converge to the optimal policy $\pi^{\star}$. For the later part, you may use the property that if some value function satisfies

$$V(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum s' \in \mathcal{S} P_{sa}(s') V(s')$$

then $V = V^{\star}$.

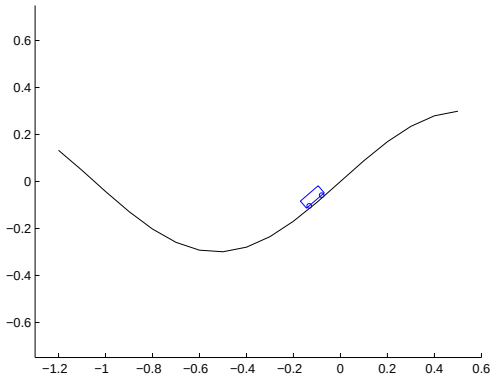
Answer: We know that policy iteration must converge because there are only a finite number of possible policies (if there are $|S|$ states, each with $|A|$ actions, then that leads to a $|S|^{|A|}$ total possible policies). Since the policies are monotonically improving, as we showed in part (c), at some point we must stop generating new policies, so the algorithm must produce $\pi' = \pi$. Using the assumptions stated in the question, it is easy to show convergence to the optimal policy. If $\pi' = \pi$, then using the same logic as in part (c)

$$V^{\pi}(s) = V^{\pi'}(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \int} P_{sa}(s') V^{\pi}(s),$$

So $V = V^{\star}$, and therefore $\pi = \pi^{\star}$.

5. **Reinforcement Learning: The Mountain Car**

In this problem you will implement the Q-Learning reinforcement learning algorithm described in class on a standard control domain known as the Mountain Car.[2] The Mountain Car domain simulates a car trying to drive up a hill, as shown in the figure below.



[2]The dynamics of this domain were taken from Sutton and Barto, 1998.

All states except those at the top of the hill have a constant reward $R(s) = -1$, while the goal state at the hilltop has reward $R(s) = 0$; thus an optimal agent will try to get to the top of the hill as fast as possible (when the car reaches the top of the hill, the episode is over, and the car is reset to its initial position). However, when starting at the bottom of the hill, the car does not have enough power to reach the top by driving forward, so it must first accerlaterate accelerate backwards, building up enough momentum to reach the top of the hill. This strategy of moving away from the goal in order to reach the goal makes the problem difficult for many classical control algorithms.

As discussed in class, Q-learning maintains a table of Q-values, $Q(s, a)$, for each state and action. These Q-values are useful because, in order to select an action in state s, we only need to check to see which Q-value is greatest. That is, in state s we take the action

$$\arg\max_{a \in \mathcal{A}} Q(s, a).$$

The Q-learning algorithm adjusts its estimates of the Q-values as follows. If an agent is in state s, takes action a, then ends up in state s', Q-learning will update $Q(s, a)$ by

$$Q(s, a) = (1 - \alpha)Q(s, a) + \gamma(R(s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a').$$

At each time, your implementation of Q-learning can execute the greedy policy $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$

Implement the `[q, steps_per_episode] = qlearning(episodes)` function in the `q5/` directory. As input, the function takes the total number of episodes (each episode starts with the car at the bottom of the hill, and lasts until the car reaches the top), and outputs a matrix of the Q-values and a vector indicating how many steps it took before the car was able to reach the top of the hill. You should use the `[x, s, absorb] = mountain_car(x, actions(a))` function to simulate one control cycle for the task — the `x` variable describes the true (continuous) state of the system, whereas the `s` variable describes the discrete index of the state, which you'll use to build the Q values.

Plot a graph showing the average number of steps before the car reaches the top of the hill versus the episode number (there is quite a bit of variation in this quantity, so you will probably want to average these over a large number of episodes, as this will give you a better idea of how the number of steps before reaching the hilltop is decreasing). You can also visualize your resulting controller by calling the `draw_mountain_car(q)` function.

Answer: The following is our implementation of `qlearning.m`:

```
function [q, steps_per_episode] = qlearning(episodes)

% set up parameters and initialize q values
alpha = 0.05;
gamma = 0.99;
num_states = 100;
num_actions = 2;
actions = [-1, 1];
q = zeros(num_states, num_actions);

for i=1:episodes,
```

```
  [x, s, absorb] =  mountain_car([0.0 -pi/6], 0);
  [maxq, a] = max(q(s,:));
  if (q(s,1) == q(s,2)) a = ceil(rand*num_actions); end;
  steps = 0;

  while (~absorb)
    % execute the best action or a random action
    [x, sn, absorb] = mountain_car(x, actions(a));
    reward = -double(absorb == 0);

    % find the best action for the next state and update q value
    [maxq, an] = max(q(sn,:));
    if (q(sn,1) == q(sn,2)) an = ceil(rand*num_actions); end
    q(s,a) = (1 - alpha)*q(s,a) + alpha*(reward + gamma*maxq);
    a = an;
    s = sn;
    steps = steps + 1;
  end
  steps_per_episode(i) = steps;
end
```

Within 10000 episodes, the algorithm converges to a policy that usually gets the car up the hill in around 52-53 steps. The following plot shows the number of steps per episode (averaged over 500 episodes) versus the number of episodes. We generated the plot using the following code:

```
for i=1:10,
  [q, ep_steps] = qlearning(10000);
  all_ep_steps(i,:) = ep_steps;
end
plot(mean(reshape(mean(all_ep_steps), 500, 20)));
```