CS 229, Autumn 2017
Problem Set #4: EM, DL & RL

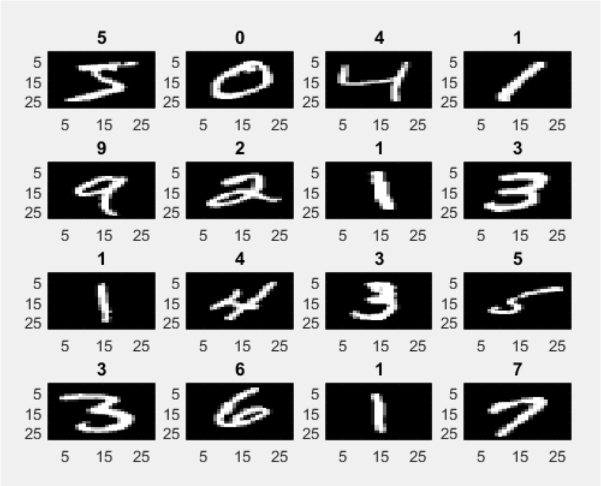
Due Wednesday, Dec 6 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be concise where 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/fall2017/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.

Remember to tag all question parts in Gradescope to avoid docked points. If you are skipping a question, please include it on your PDF/photo, but leave the question blank and tag it appropriately on Gradescope. This includes extra credit problems. If you are scanning your document by cellphone, please see `https://gradescope.com/help#help-center-item-student-scanning` for suggested practices.

1. **[25 points] Neural Networks: MNIST image classification**

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images along with their labels are shown below.



You can download the data and starter code at `http://cs229.stanford.edu/ps/ps4/q1`. The starter code splits the set of 60,000 training images and labels into a sets of 50,000 examples as the training set and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

$$CE(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y_k},$$

where $\hat{y} \in \mathbb{R}^K$ is the vector of softmax outputs from the model for the training example x, and $y \in \mathbb{R}^K$ is the ground-truth vector for the training example x such that $y = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the position of the correct class (also called a "one-hot" representation).

For m training examples, we average the cross entropy loss over the m examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}.$$

The starter code already converts labels into one hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, common practice is to use mini-batch gradient descent for deep learning tasks. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$$

where B is the batch size, i.e. the number of training example in each mini-batch.

(a) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

Submit both the plots along with the code.

Also, at the end of 30 epochs, save the learnt parameters (i.e all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

(b) **[7 points]** Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left(\frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})\right) + \lambda\left(||W^{[1]}||^2 + ||W^{[2]}||^2\right)$$

Be careful not to regularize the bias/intercept term (remember why? from PS2?). Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to calculate the loss value (regularization should only be used to calculate the gradients). Submit the two new plots obtained.

Compare the plots obtained from the regularized version with the plots obtained from the non-regularized version, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

(c) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it whatever value it may be, and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e, the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e, the regularized model).

Report your test accuracy for both regularized model and non-regularized model.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise.

2. **[15 points] EM Convergence**

We again consider the problem of estimating the parameters of a model in the case of partially observed data. Let $x^{(i)}$ indicate an observed example, and $z^{(i)}$ indicate the corresponding latent variable associated with that observed example. Let θ indicate the parameter of the joint model $p(x, z; \theta)$.

Let us denote the log-marginal density of the observed data as

$$\ell(\theta) = \sum_{i=1}^{m} \log p(x^{(i)}; \theta).$$

We saw in class that the EM algorithm monotonically improves $\ell(\theta)$ by iteratively arriving at better estimates of θ, ensuring that

$$\ell(\theta^{(t+1)}) \geq \ell(\theta^{(t)})$$

where t indicates the iteration number. It does so by constructing a lower bound of $\ell(\theta)$ in each iteration (such that the bound is tight at the current estimate of θ), then maximizing this lower bound with respect to θ, and use the resulting θ as the updated estimate. The algorithm stops when we obtain parameter θ^* such that an M-step from that point results in the same parameter θ^*.

This algorithm will always converge. The reason is that every new θ monotonically increases $\ell(\theta)$, and ℓ is a bounded function. So the algorithm cannot go on for ever.

However, it was not shown that when the algorithm converges, we would have completely maximized $\ell(\theta)$. This is a subtle point. Maybe the EM iterations converged to a value of θ^*, but for some reason that θ^* was not a local maxima of $\ell(\theta)$? Remember, we never directly maximized $\ell(\theta)$ and therefore do not have a trivial guarantee that θ^* is at a local maxima of $\ell(\theta)$!

Show that when the EM algorithm converges, we would have indeed maximized (locally) the log-marginal of the data, by showing that $\nabla_\theta \ell(\theta) = 0$ at the converged value of θ^*.

Hint: Be careful and clear about using θ and θ^* in the right places, especially while taking gradients. Remember, while θ represents a variable, θ^* is a constant.

Hint: The following observation can be helpful:

$$\frac{\nabla_\theta f(\theta) \mid_{\theta=\theta^*}}{f(\theta^*)} = \nabla_\theta [\log f(\theta)]_{\theta=\theta^*}$$

Hint: Start from the fact that when EM converges, the M-step would have reached a fixed point (i.e performing M-step from the converged parameter θ^* will return the same parameter θ^*)

3. **[10 points] PCA**

   In class, we showed that PCA finds the "variance maximizing" directions onto which to project the data. In this problem, we find another interpretation of PCA.

   Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(m)}\}$. Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

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

   Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

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

   gives the first principal component.

   Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

4. **[10 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 `http://cs229.stanford.edu/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` (and `bellsej.py`) 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` (and `bellsej.py`) 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.

5. **[15 points] Markov decision processes**

   Consider an MDP with finite state and action spaces, and discount factor $\gamma < 1$. Let B be the Bellman update operator with V a vector of values for each state. I.e., if $V' = B(V)$, then

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

   (a) **[10 points]** Prove that, for any two finite-valued vectors V_1, V_2, it holds true that

   $$||B(V_1) - B(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

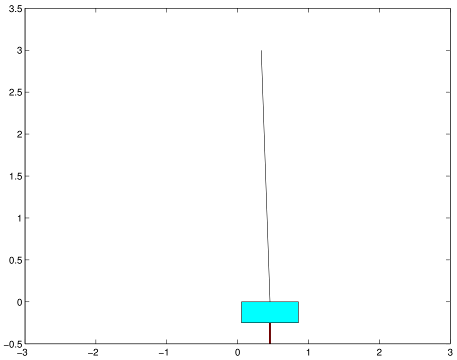
   where

   $$||V||_\infty = \max_{s \in S} |V(s)|.$$

   (This shows that the Bellman update operator is a "γ-contraction in the max-norm.")

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

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



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

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[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 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 `http://cs229.stanford.edu/ps/ps4/q6`. Most of the the code has already been written for you, and you need to make changes only to `control.m` (or `control.py`) in the places specified. This file can be run in Matlab to show a display

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

and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation.[2]

(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.

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 `control.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?

(b) Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Matlab/Octave users just need to execute `plot_learning_curve.m` after `control.m` to get this plot. Python starter code already includes the code to plot.

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