Homework 4
Optimization & ML

15-381/781: Artificial Intelligence (Fall 2016)
OUT: Oct. 30, 2016
DUE: Nov. 14, 2016 at 11:59pm

Instructions

Homework Policy

Homework is due on autolab by the posted deadline. Assignments submitted past the deadline will incur the use of late days.

You have 6 late days, but cannot use more than 2 late days per homework. No credit will be given for homework submitted more than 2 days after the due date. After your 6 late days have been used you will receive 20% off for each additional day late.

You can discuss the exercises with your classmates, but you should write up your own solutions. If you find a solution in any source other than the material provided on the course website or the textbook, you must mention the source. You can work on the programming questions in pairs, but theoretical questions are always submitted individually. Make sure that you include a README file with your andrew id and your collaborator's andrew id.

Submission

Please create a tar archive of your answers and submit to Homework 4 on autolab. You should have four files in your archive: a completed `problems.py` and a completed `bbproblems.py` for the programming portion, a PDF for your answers to the written component, and a README file with your Andrew ID and your collaborators' Andrew IDs. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation. There is a `sample.txt` and a `bbsamples.txt` that contains sample inputs and outputs for reference.

1 Written

1.1 Convexity (20 points)

Which of the following mathematical programming problems are convex? Prove your statements. It might be helpful to try sketching the objective functions or the sets we are optimizing over.

(a) (6 points) The optimization variables are $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$.

$$\begin{array}{ll} \text{minimize} & 3x_1 - 5x_2 \\ \text{subject to} & x_1^2 + x_2^2 \leq 1 \end{array}$$

(b) (6 points) The optimization variables are $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$.

$$\begin{array}{ll} \text{minimize} & 3x_1 - 5x_2 \\ \text{subject to} & x_1^2 - x_2^2 \leq 1 \end{array}$$

(c) (8 points) The optimization variables are $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Also A is an $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$.

$$\begin{array}{ll} \text{minimize} & \exp\left(\sqrt{\sum_{i=1}^n x_i^2}\right) \\ \text{subject to} & A\boldsymbol{x} \leq \boldsymbol{b} \end{array}$$

1.2 Neural Networks (20 points)

In this question, you will explore the representation power of neural networks, and how multiple layers can affect it. We will assume the input $x \in \{0,1\}^n$ are binary vectors of length n. We will also use the true binary threshold as the activation function $f(z) = 1$ if $z > 0$ and 0 otherwise. Note that the weights are still allowed to be real numbers. The output will be the result of a single unit and thus be either 0 or 1. We can think of using such a neural network to implement boolean functions.

(a) (4 points) Suppose $n = 2$ i.e. the input is a pair of binary values. Suppose we have a neural network where we don't have any hidden units and just a single output unit i.e. $y = f(w^T x + b)$ is the entire network. What should w, b be if we want to implement boolean AND (i.e. $y = 1$ only when $x = (1,1)$). What about boolean OR?

(b) (1 point) Under the same conditions as above, what boolean function of two variables cannot be represented? You just need to state one, and provide an explanation.

(c) (4 points) Suppose we now allow a single layer of hidden units i.e. $y = f(w^T z + b), z_j = f(w_j^T x + b_j)$. Construct a neural network that can implement the boolean function you mentioned previously that could not be represented before. The number of hidden units is up to you, but try to keep it as simple as possible.

(d) (8 points) It turns out that for any number of input boolean variables, a single hidden layer is enough to represent any boolean function. Describe a general scheme that one can use to construct such a neural network for any boolean function (HINT: consider conjunctive normal form or disjunctive normal form).

(e) (3 points) If a single layer is enough to represent all boolean functions, why would you ever want to use multiple hidden layers? What does this suggest about designing deep neural network structures in practice?

1.3 VC Dimension (20 points)

In this question, you will first prove two properties of VC-Dimension and then find the VC-Dimension of a given concept class. When proving $\text{VCDim}(\mathcal{C}) = d$, make sure that your proof shows a lower-bound ($\text{VCDim}(\mathcal{C}) \geq d$) and a matching upper-bound ($\text{VCDim}(\mathcal{C}) \leq d$). The former is usually established by showing an example of a set of size d that is shattered by $\mathcal{C}$ and the latter is usually established by proving that no set of size at least $d+1$ can be shattered by $\mathcal{C}$.

(a) (5 points) Prove that for any concept class $\mathcal{C}$, $\text{VCDim}(\mathcal{C}) \leq \log(|\mathcal{C}|)$.

(b) (5 points) Prove that for any two concept classes $\mathcal{C}' \subseteq \mathcal{C}$, $\text{VCDim}(\mathcal{C}') \leq \text{VCDim}(\mathcal{C})$.

(c) (10 points) Let $\mathcal{X} = \{0,1\}^n$ be the set of n-bit binary vectors. For a set $I \subseteq \{1,\ldots,n\}$, define h_I as follows. On a binary vector $\vec{x} = (x_1,\ldots,x_n) \in \mathcal{X}$,

$$h_I(\vec{x}) = \left(\sum_{i\in I} x_i\right) \mod 2 \ .$$

What is the VC-dimension of the class $\mathcal{C} = \{h_I | I \subseteq \{1,\ldots,n\}\}$? Prove your claim. *Hint: You might find part (a) useful for this question.*

1.4 Graduate Problem: Integer Programming (30 points)

Consider the following *Vertex Cover* problem. We are given as input an undirected graph $G = (V, E)$, along with "costs" $c_v \geq 0$ for all $v \in V$. We define a *vertex cover* to be a subset $S \subseteq V$ such that, for each edge $e \in E$, at least one of e's endpoints is in S. The *cost* of a vertex cover S is $\text{cost}(S) = \sum_{v\in S} c_v$. The goal is to output a vertex cover S of minimum cost.

(a) (6 points) Formulate this problem as an IP.

(b) (4 points) Give the natural LP relaxation of the IP you gave in part (a).

(c) (10 points) Consider the following instance. $G = K_n$, the complete graph on n vertices. That is, $V = [n]$ and for each $i, j \in [n]$ with $i \neq j$, the edge ij is in the graph. Also, let $c_v = 1$ for all $v \in V$. Let OPT denote the minimum cost of a vertex cover on this instance, and let OPT_{LP} denote the value of the LP on this instance. Compare OPT_{LP} and OPT. There's no need to compute them exactly – just try to see if one is bigger than the other.

(d) (10 points) When one is given a solution to an LP relaxation of a problem, it is often a good idea to try to "round it" to a solution to the original combinatorial problem. For example, suppose $\boldsymbol{x} \in \mathbb{R}^V$ is an optimal solution to the LP (which we can find in polynomial time). Define the subset $S_{\boldsymbol{x}} \subseteq V$ by $S_{\boldsymbol{x}} = \{v \in V : x_v \geq 1/2\}$. Show that $S_{\boldsymbol{x}}$ is vertex cover. Then, show that $\text{cost}(S_{\boldsymbol{x}}) \leq 2\text{OPT}_{LP}$, where OPT_{LP} is defined as in the previous problem. From this, conclude that $S_{\boldsymbol{x}} \leq 2\text{OPT}$, where OPT is also defined as in the previous problem.

2 Programming

2.1 Handwritten Digit Classification [20 points]

We will use the numpy library (`http://www.numpy.org/`) for python to handle matrices and matrix operations. We will use `numpy.ndarray` to store 2D arrays that represent matrices. Matrix multiplication can be done with `A.dot(B)` where `A,B` are both 2D `numpy.ndarray` objects. Be careful when mixing 2D arrays and 1D arrays since 1D arrays may represent either column or row vectors. For more details consult the online documentation.

2.1.1 Loss Function

You will develop a linear softmax classifier to classify handwritten digits from the MNIST data set. We will extend a bit the notation used in the class, and use a loss function that *directly* captures a k-class classification tasks (as opposed to training k different binary classifiers), called the softmax or cross-entropy loss. In our new setting, we have a training set of m example points of the form $(x^{(i)}, y^{(i)})$, $i = 1,\ldots,m$, with $y^{(i)} \in \{0,1\}^k$ (remember that k is the number of classes we're trying to predict). Remember that both $x^{(i)}, y^{(i)}$ are column vectors. The output $y_j^{(i)} = 1$ when j is the target class, and 0 otherwise. That is, if

output values can take on one of 10 classes (as will be the case in the digit classification task), and the target class for this example is the fourth class, then corresponding $y^{(i)}$ is simply

$$y^{(i)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{1}$$

This is sometimes called a "one-hot" encoding of the output class.

Under the model, our hypothesis function $\hat{y} = h_\Theta(x)$ will now output *vectors* in $\mathbb{R}^k$, where the value of an entry of $\hat{y}_j$ corresponds roughly to how likely we believe that the output is really class j (this will become more concrete when we formally define the function class). For instance, the (hypothetical) output

$$\hat{y} = h_\Theta(x^{(i)}) = \begin{bmatrix} 0.1 \\ -0.2 \\ 2.0 \\ 5.0 \\ 0.1 \\ -1.0 \\ -5.0 \\ 1.0 \\ 0.4 \\ 0.2 \end{bmatrix} \tag{2}$$

would correspond to a prediction that the point $x^{(i)}$ is probably from the fourth class (the element with the largest entry). Analogous to binary classification (where, if we wanted binary prediction we would simply take the sign of the hypothesis function), if we want the to predict a single class label for the output, we simply predict class j for which $\hat{y}_j$ takes on the largest value.

Our loss function is now defined as a function $\ell : \mathbb{R}^k \times \{0,1\}^k \to \mathbb{R}_+$ that quantifies how good a prediction is. In the "best" case, the predictions $\hat{y}_j$ would be $+\infty$ for the true class (i.e. for the element where $y_j^{(i)} = 1$) and $-\infty$ otherwise (of course, we usually won't make infinite predictions, because we would then suffer very high loss if we ever made a mistake, and we won't get such predictions with most classifiers if we include a regularization term). The loss function we use is the softmax loss, given by

$$\ell(\hat{y}, y) = \log\left(\sum_{j=1}^{k} e^{\hat{y}_j}\right) - \hat{y}^T y. \tag{3}$$

This loss function has the gradient

$$\nabla_{\hat{y}} \ell(\hat{y}, y) = \frac{e^{\hat{y}}}{\sum_{j=1}^{k} e^{\hat{y}_j}} - y \tag{4}$$

where the exponent $e^{\hat{y}}$ is taken elementwise. Note that the gradient is a vector of size k because $e^{\hat{y}}$ and y are both vectors of size k and the denominator in the fraction is just a scalar.

In practice, you would probably want to implemented regularized loss minimization, but for the sake of this problem set, we'll just consider minimizing loss without any regularization (at the expense of overfitting a little bit).

2.1.2 Linear Classifier

In this section, you'll implement a linear classification model to classify digits. That is, our hypothesis function will be

$$h_\Theta(x^{(i)}) = \Theta \begin{bmatrix} x^{(i)} \\ 1 \end{bmatrix} \tag{5}$$

where $\Theta \in \mathbb{R}^{10\times 785}$ is our matrix of parameters. Note that here we are taking the column vector $x^{(i)}$ and adding a new row at the bottom that is just the constant 1. This is for convenience since this is equivalent to $Wx^{(i)} + b$ with the parameters W, b put together into a single matrix Θ. Using a simple application of the chain rule we can compute the gradient of our loss function for this hypothesis class

$$\nabla_\Theta \ell(h_\Theta(x^{(i)}, y) = \nabla_{\hat{y}} \ell(\hat{y}, y) \begin{bmatrix} {x^{(i)}}^T & 1 \end{bmatrix}. \tag{6}$$

where $\hat{y} \equiv h_\Theta(x^{(i)})$ and where the gradient $\nabla_{\hat{y}}(\ell(\hat{y}, y))$ is given in (4) above. Note that this gradient is multiplying a column vector with a row vector resulting in a matrix of the same dimension as Θ (this is also known as the outer product of two vectors).

2.1.3 Downloading and setting up data set

Download the MNIST data set from http://yann.lecun.com/exdb/mnist/. You'll want to specifically download all the files

```
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
```

and uncompress them. Using the functions `parse_images` and `parse_labels` provided in the included `problems.py` file, you can read these files using the code

```
X_train = parse_images("train-images.idx3-ubyte")
y_train = parse_labels("train-labels.idx1-ubyte")
X_test = parse_images("t10k-images.idx3-ubyte")
y_test = parse_labels("t10k-labels.idx1-ubyte")
```

After running these functions `X_train`, for example, will be a 60000×784 numpy array where each row corresponds to a flattened 28 by 28 greyscale image of a digit. Each pixel is a single floating point number between 0 and 1 representing a grey. Similarly, `y_train` is a 60000×10 array where each row is a one-hot encoding of which digit is present in the image (ordered 0 through 9). Note that the examples are represented by rows (as opposed to columns normally used in the notation) due to efficiency reasons.

2.1.4 Gradient Computation [5 points]

Here you will compute the gradient of the parameters Θ given a single example point $x^{(i)}, y^{(i)}$. You'll implement the following function:

```
def grad(Theta, x, y):
    """
    Compute the gradient given input x and output y, and current parameters Theta
    Note that this assumes the constant 1 has already been appended to the input

    Arguments:
        Theta: 10 x 785 numpy array of current parameters
        x: 785 sized 1D numpy array of input
        y: 10 sized 1D numpy array of output

    Return:
        A 10 x 785 numpy array gradient
    """
```

You may use the helper function `softmax_loss(yp,y)` for the gradient of the loss function.

2.1.5 Stochastic gradient descent [15 points]

Implement the stochastic gradient descent algorithm. Recall from the notes that the SGD algorithm is:

function $\Theta = \text{SGD}(\{(x^{(i)}, y^{(i)})\}, h_\Theta, \ell, \alpha)$
  Initialize: $\Theta \leftarrow 0$
  For $t = 1, \ldots, T$:
    For $i = 1, \ldots, m$:
      $\Theta \leftarrow \Theta - \alpha \nabla_\Theta \ell(h_\Theta(x^{(i)}), y^{(i)})$
  return Θ

That is, we take small gradient steps on *each* example.

You'll implement the following function:

```
def softmax_sgd(X,y, Xt, yt, epochs=10, alpha = 0.01):
    """
    Run stochastic gradient descent to solve linear softmax regression.

    Arguments:
        X: numpy array where each row is a training example input of length 784
        y: numpy array where each row is a training output of length 10
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        epochs: number of passes T to make over the whole training set
        alpha: step size

    Return:
        A list of tuples (Train Err, Train Loss, Test Error, Test Loss) for each epoch
        These should be computed at the end of each epoch
    """
```

You may use the included helper function `get_errors(yp, y, ypt, yt)` to compute the losses and errors.

2.2 Branch and Bound [20 points]

You will use branch and bound in order to solve binary integer programs of the form: minimize Cx under the constraint $Ax \leq b$. Branch and bound searches for an optimal assignment to the variables using linear programming relaxations to prune branches of the search tree.

2.2.1 Binary Branch and Bound

We will search for an optimal assignment using depth-first search, where we assign values to the variables in index order. When a node is selected for expansion we check if the node's subtree can be pruned (i.e. whether we need to continue assigning to the rest of the variables). If the value of the LP relaxation is greater than the best assignment so far then the there is no need to further expand the node, since the LP is a lower bound which means no matter how we assign to the remaining variables there is no way to attain a smaller value. The subtree can also be pruned if the LP assignment consists of binary values for all the variables and is the best seen so far. Then we update the best assignment seen so far and continue searching. Otherwise we expand the children of the node i.e. assign the next variable in index order.

In this assignment the i-th variable will be assigned at depth i in the search tree. So the root has no assignments. The first level assigns to x_0. Since we are pruning parts of the tree make sure to explore the $x_i = 0$ assignment before $x_i = 1$. This is necessary since we ask you to output the nodes expanded by your branch and bound. The DFS exploration is the same method as was discussed in lecture.

2.2.2 LP Relaxation

The LP relaxaion of an integer program is simply removing the restriction that the variables must have integer assignments. The relaxations have known polynomial time solutions. As discussed in class when minimizing an objective the optimal value of the LP relaxation will be a lower bound on the optimal integer assignment. Hence, the LP relaxation is used to prune the search tree.

2.2.3 cvxopt and numpy

You will use cvxopt to solve the linear programming relaxations. numpy will also be useful for manipulating the constraint matrices. Several useful code fragments are listed below.

```
#import cvxopt stuff
from cvxopt import matrix, solvers

#minimize Cx given Ax <= b
#these take matrices as input (see below)
solution = solvers.lp(C, A, b)

#get the variable assignments
var = solution['x']

#get the value of the assignment
val = solution['primal objective']

#check if an optimal solution was found
status = solution['status'] == 'optimal'

#remove some of the output from the solver
solvers.options['show_progress'] = False

#import numpy
import numpy as np

#create numpy array from python list
np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]])

#create cvxopt matrix from numpy array
matrix(np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]]))
```

The input binary IP problems have the form minimize Cx with the constraints $Ax \leq b$ and $\forall i.\ x_i \in \{0,1\}$. C is a list of n coefficients. A is a list of lists with m rows and n columns. b is a list of m values. A does not restrict the domain of the variables x to be binary. You must add these constraints. See `bbsamples.txt` for input samples.

You will output a tuple consisting of `(value, assignment, expandlist)`. Value is the value of the objective for an optimal assignment (`None` if no feasible solution). `assignment` is an optimal assignment list (None if no feasible solution). `expandlist` is the list of nodes expanded by branch and bound. Nodes are labeled by the assignments to the variables. Thus the root is `[]`. The left child of the root with $x_0 = 0$ is `[0]` and the right child is [1]. See `bbsamples.txt` for sample inputs and outputs.

Complete `solve(A,B,C)` in `bbproblems.py`.