Homework 6
Optimization and Machine Learning

CMU 15-780: Graduate AI (Spring 2016)
OUT: April 9, 2016
DUE: 11:59pm April 19, 2016

Instructions

Collaboration/Academic Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. NO DOWNLOADING OR COPYING OF CODE OR OTHER ANSWERS IS ALLOWED. If you use a string of at least 5 words from some source, you must cite the source; however you cannot just copy the solution from some source but instead you should write it up in your own words.

Submission

Please create a tar archive of your answers and submit to Homework 6 on autolab. You should have two files in your archive: a completed `problems.py` for the programming portion, and a PDF for your answers to the written component. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation. Please put your Andrew ID somewhere on the first page of your written answers.

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

1 Written

1.1 Minimizing 0/1 loss [15 points]

In class, we mentioned that we cannot efficiently find a linear classifier that minimizes 0/1 loss. That is, we cannot easily solve the optimization problem

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{m} \mathbf{1}\{y^{(i)} \cdot \theta^T x^{(i)} \leq 0\} \tag{1}$$

with optimization variables θ, training data $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, m$ and where $\mathbf{1}$ denotes the indicator function (one if the argment is true and zero otherwise).

Show that we can, however, solve this problem as a binary mixed integer program. That is, write an optimization problem over the variables θ and z with whatever (convex) objective and constraints you want, plus the additional constraint that $z_i \in \{0, 1\}$, such that the solution of this optimization problem gives the θ the minimizes 0/1 loss.

1.2 Gradient and Hessian for regularized logistic regression [10 pts]

Recall the problem of mimizing a regularized logistic loss function

$$\underset{\theta}{\text{minimize}} \; L(\theta) \equiv \sum_{i=1}^{m} \log\left(1 + \exp\{-y^{(i)} \cdot \theta^T x^{(i)}\}\right) \tag{2}$$

with optimization variables θ, and training data $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, m$. Show that the gradient and Hessian of the objective are given by

$$\begin{aligned} \nabla_\theta L(\theta) &= -\sum_{i=1}^{m} y^{(i)} x^{(i)} z_i \\ \nabla^2_\theta L(\theta) &= \sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T z_i (1 - z_i) \end{aligned} \tag{3}$$

where in both cases

$$z_i \equiv \frac{1}{1 + \exp(y^{(i)} \cdot \theta^T x^{(i)})}. \tag{4}$$

2 Programming

In this section, you will develop a few different multi-class classifiers to classify 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, called the softmax or cross-entropy loss. In our new setting, we have a training set 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), where $y_j^{(i)} = 1$ 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 class 4, then corresponding $y^{(i)}$ is simply

$$y^{(i)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{5}$$

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 relative size of the $\hat{y}_j$ corresponds roughly to how likely we believe that the output is really class j (this will become more concrete when we formally define the class function). For instance, the (hypothetical) output

$$\hat{y} = h_\theta(x^{(i)}) = \begin{bmatrix} 0.1 \\ -0.2 \\ 2.0 \\ 5.0 \\ 0.1 \\ -1.0 \\ -5.0 \\ 1.0 \\ 0.4 \\ 0.2 \end{bmatrix} \tag{6}$$

would correspond to a predict that the example $x^{(i)}$ is probably from class 4 (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 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[1]

$$\ell(\hat{y}, y) = \log\left(\sum_{j=1}^{k} e^{\hat{y}_j}\right) - \hat{y}^T y. \tag{9}$$

This loss function has the gradient

$$\nabla_{\hat{y}} \ell(\hat{y}, y) = \frac{e^{\hat{y}}}{\sum_{j=1}^{k} e^{\hat{y}_j}} - y \tag{10}$$

where the exponent $e^{\hat{y}}$ is taken elementwise.

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 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 unzip them. Using the functions `parse_images` and `parse_labels` provided in the included `problem.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 28 by 28 image of a digit. Similarly, `y_train` is a 60000×10 array where each row is an indicator of which digit is present in the image (ordered 0 through 9). If you have the matplotlib library installed on your machine, you can view some of the loaded images with the following code:

[1] It won't be relevant to the implementation in this problem, but for those curious where this loss function comes from, softmax regression can be interpreted as a probabilistic model where

$$p(y = j|\hat{y}) = \frac{e^{\hat{y}_j}}{\sum_{l=1}^{k} e^{\hat{y}_l}}. \tag{7}$$

If we look at maximum likelihood estimation under this true model, we get the optimization problem

$$\underset{\theta}{\text{minimize}} - \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \;\equiv\; \underset{\theta}{\text{minimize}} \sum_{i=1}^{m} \log\left(\sum_{j=1}^{k} e^{h_\theta(x^{(i)})_j}\right) - h_\theta(x^{(i)})^T y^{(i)}. \tag{8}$$

```
import matplotlib.pyplot as plt
M,N = 10,20
fig, ax = plt.subplots(figsize=(N,M))
digits = np.vstack([np.hstack([np.reshape(X_train[i*N+j,:],(28,28))
                               for j in range(N)]) for i in range(M)])
ax.imshow(255-digits, cmap=plt.get_cmap('gray'))
```

2.2 Linear softmax regression [30 points]

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

$$h_\theta(x^{(i)}) = \Theta \begin{bmatrix} x^{(i)} \\ 1 \end{bmatrix} \tag{11}$$

where $\Theta \in \mathbb{R}^{10\times 785}$ is our vector of parameters. Using a simple application of the same chain rule we applied to derive backpropagation (you can try to verify this for yourself), we can compute the gradient of our loss function for this hypothesis class

$$\nabla_\Theta \ell(h_\theta(x^{(i)}, y) = \nabla_{\hat{y}} \ell(\hat{y}, y) \begin{bmatrix} x^{(i)T} & 1 \end{bmatrix}. \tag{12}$$

where $\hat{y} \equiv h_\theta(x^{(i)})$ and where the gradient $\nabla_{\hat{y}} \ell(\hat{y}, y))$ is given in (10) above (to make this a bit simplier, in practice you can just create a new X matrix with an additional column of all ones added).

Gradient descent Implement the gradient descent algorithm to solve this optimization problem. Recall from the slides that the gradient descent algorithm is given by:

function $\theta = \text{Gradient-Descent}(\{(x^{(i)}, y^{(i)})\}, h_\theta, \ell, \alpha)$
  Initialize: $\theta \leftarrow 0$
  For $t = 1, \ldots, T$:
    $g \leftarrow 0$
    For $i = 1, \ldots, m$:
      $g \leftarrow g + \frac{1}{m} \nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$
    $\theta \leftarrow \theta - \alpha g$
  return θ

i.e., we compute the gradient with respect to the loss function for all the examples, divide it by the total number of examples, and take a small step in this direction (you can leave all the step sizes α at their default values for the programming part). Each pass over the whole dataset is referred to as an *epoch*.

Specifically, you'll implement the softmax_gd function:

```
def softmax_gd(X, y, Xt, yt, epochs=10, alpha = 0.5):
    """
    Run gradient descent to solve linear softmax regression.

    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        epochs: number of passes to make over the whole training set
        alpha: step size

    Outputs:
        Theta: 10 x 785 numpy array of trained weights
    """
```

In addition to outputing the trained parameters, your function should output the error (computed by the included `error` function) on the test and training sets, and the after softmax loss on the test and training sets (again using the provided function, which will help you in computing the gradient). You should report all these errors computed on the training set at each iteration (epoch) *before* adjusting the parameters for that iteration. For example, our implementation of gradient descent outputs the following:

```
Test Err  |Train Err |Test Loss |Train Loss
    0.9020|    0.9013|    2.3026|    2.3026
    0.3192|    0.3276|    1.8234|    1.8318
    0.2101|    0.2228|    1.4983|    1.5141
    0.2142|    0.2246|    1.2830|    1.3018
    0.1844|    0.1935|    1.1341|    1.1555
    0.1816|    0.1924|    1.0260|    1.0490
    0.1702|    0.1785|    0.9463|    0.9697
    0.1670|    0.1754|    0.8826|    0.9072
    0.1613|    0.1681|    0.8334|    0.8576
    0.1559|    0.1653|    0.7909|    0.8160
```

Stochastic gradient descent Implement the stochastic gradient descent algorithm for linear softmax regression. 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, rather than computing the average gradient over all the examples.

You'll implement the following function:

```
def softmax_sgd(X, y, Xt, yt, epochs=10, alpha = 0.5):
    """
    Run stochastic gradient descent to solve linear softmax regression.

    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        epochs: number of passes to make over the whole training set
        alpha: step size

    Outputs:
        Theta: 10 x 785 numpy array of trained weights
    """
```

You should output the same information (train/test error/loss) as for the above function, but importantly only output this information *once* per outer iteration, before you iterate over all the examples.

2.3 Deep neural network [30 points]

Here you will implement a deep neural network to classify MNIST digits, trained by stochastic gradient decent. You can implement the final `nn_sgd` function any way you like, but it may be easier if you use

the conventions that we set up here. Specifically, we recomment that you implement the following functions:

```
def nn(x, W, b, f):
    """
    Compute output of a neural network.

    Input:
        x: numpy array of input
        W: list of numpy arrays for W parameters
        b: list of numpy arraos for b parameters
        f: list of activation functions for each layer

    Output:
        z: list of activationsn, where each element in the list is a tuple:
           (z_i, z'_i)
           for z_i and z'_i as defined in the class slides
    """

def nn_loss(x, y, W, b, f):
    """
    Compute loss of a neural net prediction, plus gradients of parameters

    Input:
        x: numpy array of input
        y: numpy array of output
        W: list of numpy arrays for W parameters
        b: list of numpy arraos for b parameters
        f: list of activation functions for each layer

    Output tuple: (L, dW, db)
        L: softmax loss on this example
        dW: list of numpy arrays for gradients of W parameters
        db: list of numpy arrays for gradients of b parameters
    """

def nn_sgd(X,y, Xt, yt, W, b, f, epochs=10, alpha = 0.005):
    """
    Run stoachstic gradient descent to solve linear softmax regression.

    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        W: list of W parameters (with initial values)
        b: list of b parameters (with initial values)
        f: list of activation functions
        epochs: number of passes to make over the whole training set
        alpha: step size

    Output: None (you can directly update the W and b inputs in place)
    """
```

The size of the inputs `W,b,f` determine the effective size of the neural network. We have included starter code that constructs the network in such a format, so you just need to implement the above function. In particular, a deep network for the MNIST problem, initialized with random weights, with rectified linear units in all layers excpect just a linear unit in the last layer, is constructed by the code:

```
np.random.seed(0)
layer_sizes = [784, 200, 100, 10]
W = [0.1*np.random.randn(n,m) for m,n in zip(layer_sizes[:-1], layer_sizes[1:])]
b = [0.1*np.random.randn(n) for n in layer_sizes[1:]]
f = [f_relu]*(len(layer_sizes)-2) + [f_lin]
```

In particular, this creates a deep network with 4 total layers (2 hidden layers), of sizes 784 (the size of the input), 200, 100, and 10 (the size of the output) respectively. This means, for instance, that there will be 3 W terms: $W_1 \in \mathbb{R}^{200\times 784}$, $W_2 \in \mathbb{R}^{100\times 200}$, and $W_3 \in \mathbb{R}^{10\times 100}$.

You'll use virtually the same SGD procedure as in the previous question, so the main challenge will be just to compute the network output and the gradients in the `nn` and `nn_loss` functions, using the backpropagation algorithm as specified in the class slides. We have included the various activation functions, which return both the non-linear function f and its elementsize subgradient f'.

Note that training a neural network with pure stochastic gradient descent (i.e., no minibatches) can be fairly slow, so we recommend you test your system on a small subset of (say) the first 1000 training and testing examples, and only run it on the final data set after you have debugged its performance on a smaller data set. As a reference point, our (quite unoptimized) implementation takes about a minute per epoch, and achieves around 2.5% error.

2.4 Written portion [10 points]

In addition to the code that you'll submit (which will be evaluated on simpler toy examples to check for correctness), also include with your submission

1. A figure showing the training error and testing error versus epoch for the two linear softmax regression algorithms you implemented plus the neural network, as measured on the full MNIST digit classification data set.
2. A figure showing the average training and testing loss versus epoch for all three algorithms, again as measured on the MNIST problem.

2.5 Experimentation with neural networks [5 points]

The neural network code you implemented in the previous section is quite flexible (i.e., easy to define different fully connected architectures, different activations, etc). Experiment with some different settings of the algorithms (different numbers of layers, hidden units, activations, functions, step sizes, etc). This part is open ended, and you can get full credit by simply trying some additional architectures and providing similar charts as in the previous portion. But we also encourage you to experiment to see how well you can perform on the MNIST dataset, which for years was a typical benchmark for ML algorithms.