CPSC-540 Machine Learning 2011

Homework # 5

NAME:_____

Signature:_____

STD. NUM: _____

General guidelines for homeworks:

You are encouraged to discuss the problems with others in the class, but all write-ups are to be done on your own.
Homework grades will be based not only on getting the "correct answer," but also on good writing style and clear presentation of your solution. It is your responsibility to make sure that the graders can easily follow your line of reasoning.
Try every problem. Even if you can't solve the problem, you will receive partial credit for explaining why you got stuck on a promising line of attack. More importantly, you will get valuable feedback that will help you learn the material.
Please acknowledge the people with whom you discussed the problems and what sources you used to help you solve the problem (e.g. books from the library). This won't affect your grade but is important as academic honesty.
When dealing with python exercises, please attach a printout with all your code and show your results clearly.

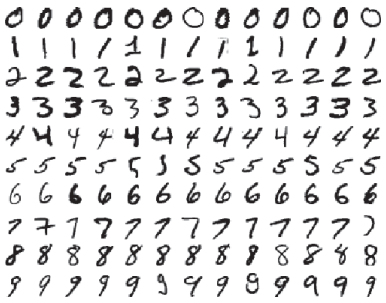

Figure 1: Example images from the MNIST dataset.

1. Multi-layer perceptron

In this question you will implement a multi-layer perceptron (MLP) in order to classify digits from the MNIST dataset. More specifically, you will compute the gradient using back-propagation and perform stochastic gradient descent on the data. For information on the MNIST dataset and how to build a linear classifier for the 10 digits, you may refer to the tutorial on python for large datasets that was handed out in class.

Let us consider a network with n_{hid} hidden units, n_{out} outputs, and inputs of size n_{in}; we can then define the following parameters:

- $\mathbf{W}_{\text{hid}}$, an n_{in}-by-n_{hid} matrix of hidden weights;
- $\mathbf{b}_{\text{hid}}$, an n_{hid} vector of hidden biases;
- $\mathbf{W}_{\text{out}}$, an n_{hid}-by-n_{out} matrix of output weights;
- $\mathbf{b}_{\text{out}}$, an n_{out} vector of output biases.

For this question we will tanh activations for the hidden units instead of sigmoid (logistic) activations. tanh still looks sigmoidal but it varies in the range $(-1, 1)$. To classify the images into the 10 digit categories, we need output softmax activations:

$$\sigma(a_j) = \frac{\exp(a_j)}{\sum_k \exp(a_k)},$$

where $\mathbf{a}$ is the vector of outputs of the hidden layer units and $j = 0, \ldots, 9$. Now, for some input $\mathbf{x}$ we can write the outputs as

$$\mathbf{z} = \tanh(\mathbf{W}_{\text{hid}}^T \mathbf{x} + \mathbf{b}_{\text{hid}}),$$
$$\mathbf{p} = \sigma(\mathbf{W}_{\text{out}}^T \mathbf{z} + \mathbf{b}_{\text{out}}).$$

Now, the outputs p_j can be interpreted as the probability that the input is of class j. Given n data samples (x_i, y_i) let p_{ij} be output j evaluated on the ith data sample, we can then write the negative log-likelihood of this data as

$$-\sum_{i=1}^{n} \sum_{j=1}^{n_{\text{out}}} t_{ij} \log p_{ij}$$

where t_{ij} is an indicator which is one when $y_i = j$, i.e. when the ith data sample is of class j. If we think of the log-likelihood as a function of our parameters **derive the gradient of this function with respect to each parameter**. Note: the derivative of $h(a) = \tanh(a)$ is given by $h'(a) = 1 - h(a)^2$.

2. Implementing the MLP

You will now implement an MLP to classify MNIST digits. First, download the dataset from the course website. We are providing you with simple code to evaluate the outputs of the MLP. We also have some extra code for computing the log-likelihood. Your task is to implement the gradient function.

```
class MLP(object):
    def __init__(self, nin, nout, nhidden):
        # the initial weights of the hidden units should be set in this weird
        # way so that early in the training it's easy to propagate errors
        # up/down. See ``Understanding the difficulty of training deep
        # feedforward neural networks'' by Glorot and Bengio.
        rng = np.random.RandomState(0)
        self.W_hidden = rng.uniform(
                size = (nin, nhidden),
                low  = -np.sqrt(6 / (nin+nhidden)),
                high =  np.sqrt(6 / (nin+nhidden)))

        self.b_hidden = np.zeros(nhidden)
        self.W_out = np.zeros((nhidden, nout))
        self.b_out = np.zeros(nout)
        self.params = (self.W_hidden, self.b_hidden, self.W_out, self.b_out)

    def forward(self, X):
        """
        Given inputs `X` as an (n,d)-array compute and return the hidden unit
        activations and the output unit activations. Return this as a tuple
        (z,p) where these are arrays of size (n,nhidden) and (n,nout)
        respectively.
        """
        # z contains the activations of the hidden units
        z = np.tanh(np.dot(X, self.W_hidden) + self.b_hidden)

        # first compute the inputs to the output units and then we'll compute
        # the activations. But we do this in place since we won't be returning
        # the inputs.
        a = np.dot(z, self.W_out) + self.b_out
        a -= np.log(np.sum(np.exp(a), axis=1)).reshape(-1,1)
        p = np.exp(a, out=a)

        return z, p

    def log_likelihood(self, X, y):
        """
        Compute the log-likelihood of a batch of data (X,y).
        """
        z, p = self.forward(X)
        i = np.arange(0, p.size, p.shape[1]) + y
        return np.mean(np.log(p.flat[i]))

    def error(self, X, y):
        """
```

```
        Compute the mean number of errors given a batch of data (X, y).
        """
        z, p = self.forward(X)
        return np.mean(y != np.argmax(p, axis=1))

    def gradient(self, X, y):
        """
        Evaluate the gradient of the negative log-likelihood on an array of
        inputs `X` and target outputs `y`. Return a tuple of the derivatives
        (dW_hidden, db_hidden, dW_out, db_out).
        """
        raise NotImplementedError
```

Implement the gradient you derived in the previous question. Note, in order to make this reasonably quick you should make sure to vectorize this expression and avoid using for-loops. Note, for example the use of `i` in the implementation of log-likelihood, which uses a flat indexing of the outputs `p`. You can then train the neural network using the following code:

```
def train_model(model, data, alpha, batchsize, nepochs):
    # grab all the dataset splits.
    (Xtrain,ytrain), (Xvalid,yvalid) = data

    # compute number of minibatches for training, validation and testing
    ntrain = int(Xtrain.shape[0] / batchsize)
    nvalid = int(Xvalid.shape[0] / batchsize)

    # this just creates a lambda expression to makes it easier to reference
    # batches of the data. X[b(i)] is the data for batch i, which can
    # equivalently be written as X[i*batchsize:(i+1)*batchsize].
    b = lambda i: slice(i*batchsize, (i+1)*batchsize)

    for epoch in xrange(nepochs):
        for i in xrange(ntrain):
            for (p,g) in zip(model.params, model.gradient(Xtrain[b(i)], ytrain[b(i)])):
                p -= alpha * g
        print 'epoch %2d, validation error: %.4f' % \
            (epoch,
             np.mean([model.error(Xvalid[b(i)], yvalid[b(i)]) for i in xrange(nvalid)]))

if __name__ == '__main__':
    print '... loading data'
    data = pickle.load(gzip.open('mnist.pkl.gz', 'rb'))

    print '... building/training the model'
    model = MLP(nin=28*28, nout=10, nhidden=500)
    train_model(model, data, alpha=0.01, batchsize=20, nepochs=100)
```

Here we have used 500 hidden units, a learning-rate of 0.01, and a batch-size of 20. This code trains the neural network by looping over small "mini-batches" of size 20 and taking a gradient step based on this data. An entire loop over the training data is called an *epoch*. After running this code for 100 epochs, **display 20 instances of the digit "4" that are misclassified.** Note, this may take some time to train (around 1–2 hour(s)) so make sure you don't wait until the last minute!

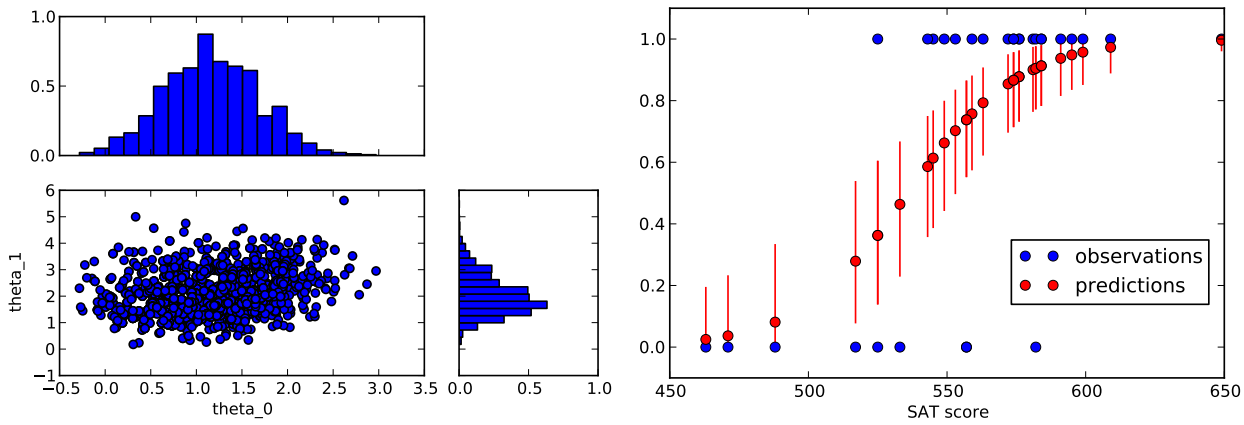

Figure 2: The left plot shows a histogram displaying the distribution of $\boldsymbol{\theta}$ and the right plot shows the predictive distribution.

3. Bayesian logistic regression with sampling

In this question you will implement Bayesian logistic regression in order to classify whether someone passes a class based upon their SAT score. First, download the data from the course website.

We can first load the data and preprocess this to normalize the features and add a constant feature.

```
data = np.loadtxt('sat.data', skiprows=1, delimiter=',')
n = data.shape[0]
X = data[:,0].reshape(n,-1)
y = np.asarray(data[:,1], dtype=int)

X_ = X.copy()
Xmu = np.mean(X, axis=0)
X -= Xmu
Xstd = np.std(X, axis=0)
X /= Xstd
X = np.c_[np.ones(n), X]
```

Implement Bayesian logistic regression using importance sampling. You should use a Gaussian prior and proposal distribution. Justify your choices.

Produce a plot which displays a histogram of the data as well as a plot which shows the predictive distribution and the 5–95th percentiles. Your plots should look something like those of Figure 2.