Assignment 1: ERM, Gradient Descent, and Subsampling

Nima Leclerc (nl475), Ryan Gale (rjg295)
Cornell University
CS 4787
Principles of Large Scale Machine Learning Systems

February 17, 2019

In this work, multinomial logistic regression was used to perform multiclass-classification on handwritten images in the MNIST data set. The hypothesis function h_w was used to to map features $\mathbf{x}_i \in \mathbb{R}^{d\times 1}$ into a multi-class labels $\mathbf{y}_i \in \mathbb{R}^{c\times 1}$. A soft-max implementation of the hypothesis function (1) was used with parameters within the weight matrix, $W \in \mathbb{R}^{c\times d}$. This hypothesis with l_2 regularization is given by (1), such that γ is a regularization parameter.

$$h_W(\mathbf{x}_i) = softmax(W\mathbf{x}_i) + \gamma||W||_F^2 \tag{1}$$

The model was trained using a cross-entropy loss function and gradient descent (GD) optimization, such that a loss function of the form of (2) was optimized.

$$L(h_W(\mathbf{x}_i), \mathbf{y}_i) = -\sum_{j=1}^{c} y_j \log(h_W(\mathbf{x}_i)) \tag{2}$$

Hence, the ability to make predictions given W and $\{(\mathbf{x}_i), \mathbf{y}_i)\} \in D$, such that $D \in \mathbb{R}^{n\times c\times d}$ requires solving the optimization problem presented in (3),

$$\hat{W} = argmin_W[\frac{1}{|D|}\sum_{\{(\mathbf{x}_i),\mathbf{y}_i)\}\in D} L(h_W(\mathbf{x}_i), \mathbf{y}_i)] \tag{3}$$

Given this, GD with the the update step in (4) was implemented (given the ith class).

$$W_{t+1} = W_t - \alpha(softmax(W)_i - y_i)\mathbf{x} + \gamma W_t) \tag{4}$$

Here, α is the stepsize. Given that T iterations were performed, the prediction mapping was performed with the hypothesis function, h_W . In this particular implementation with the MNIST data set, the model was trained on a data set $\mathbf{x}$ with feature dimension $d = 786$, $\mathbf{y}$ output labels with class dimension $c = 10$, and on a training set of dimension $n = 60,000$. The training and test error was monitored over $T = 1000$ iterations, step size of $\alpha = 0.01$, and regularization parameter set to $\gamma = 0.0001$ for the entire test/training data sets and in the corresponding randomly subsampled data sets. Figure 1 and 2 depicts the computed training and test errors over iteration number for the full data set, the 100 and 1000 size subsampled data sets. These results suggest that for smaller give rise to noiser convergence of test and training errors. This suggests that we can achieve resonable accuracy when averaging over the test errors for subsampling, yielding errors comparable to what is observed when training over the full data sets. The computed runtime for the full data set was $t = 0.763$s, while the subsample data set runtimes were $t = 0.0246$ and $t = 0.0136$s for the 100 and 1000 size data sets. This suggest that the subsample runtimes are faster, which is sensible for the training time scales linearly with the data set size. Generally, the algorithm yielded a relatively low training error (on the order of less than 0.1) within convergence, suggesting a successful implementation.

Figure 1: Exact test and training error plotted with iteration number.

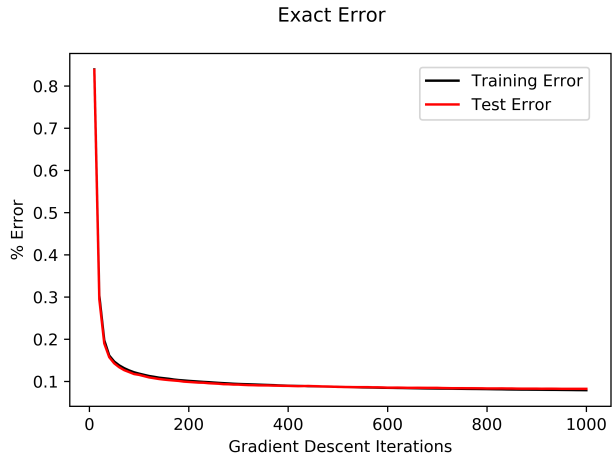
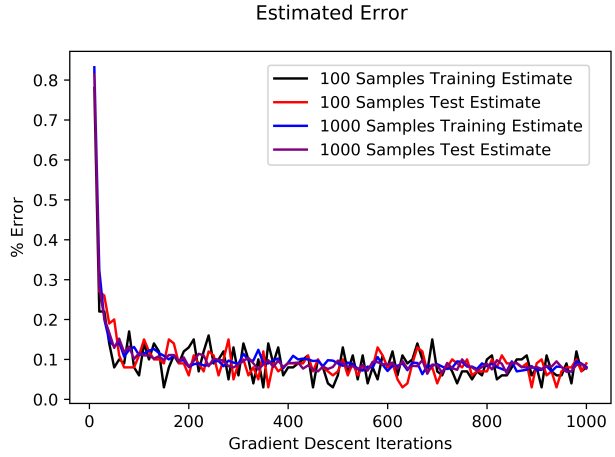


Figure 2: Estimated test and training error plotted with iteration number for subsample sizes of 100 and 1,000.



Python Implementation

```
import os
import numpy as np
import scipy
import matplotlib
import mnist
import pickle
import time
matplotlib.use('agg')
from matplotlib import pyplot

mnist_data_directory = os.path.join(os.path.dirname(__file__), "data")

def load_MNIST_dataset():
   PICKLE_FILE = os.path.join(mnist_data_directory, "MNIST.pickle")
   try:
      dataset = pickle.load(open(PICKLE_FILE, 'rb'))
```

```
   except:
      # load the MNIST dataset
      mnist_data = mnist.MNIST(mnist_data_directory, return_type="numpy", gz=True)
      Xs_tr, Lbls_tr = mnist_data.load_training();
      Xs_tr = Xs_tr.transpose() / 255.0
      Ys_tr = np.zeros((10, 60000))
      for i in range(60000):
         Ys_tr[Lbls_tr[i], i] = 1.0 # one-hot encode each label
      Xs_te, Lbls_te = mnist_data.load_testing();
      Xs_te = Xs_te.transpose() / 255.0
      Ys_te = np.zeros((10, 10000))
      for i in range(10000):
         Ys_te[Lbls_te[i], i] = 1.0 # one-hot encode each label
      dataset = (Xs_tr, Ys_tr, Xs_te, Ys_te)
      pickle.dump(dataset, open(PICKLE_FILE, 'wb'))
   return dataset

"""Computes the softmax matrix (c * n) corresponding to the 2D matrix X (c * n)"""
def softmax(X):
   c, n = X.shape
   # scale matrix down to avoid overflow
   Z = X - np.amax(X, axis=0)
   mat = np.zeros((c,n))
   sum = np.zeros(n)
   for k in range(c):
      sum = sum + np.exp(Z[k])
   for i in range(c):
      mat[i] = np.true_divide(np.exp(Z[i]), sum)
   return mat

# compute the gradient of the multinomial logistic regression objective, with regularization
#
# Xs       training examples (d * n)
# Ys       training labels (c * n)
# gamma    L2 regularization constant
# W        parameters      (c * d)
#
# returns  the gradient of the model parameters
def multinomial_logreg_grad(Xs, Ys, gamma, W):
   c, d = W.shape
   grad = np.zeros((c,d))
   product = np.matmul(W,Xs)
   grad = np.matmul((softmax(product)-Ys),Xs.T)
   return (np.true_divide(grad,len(Xs[0])) + gamma*W)

# compute the error of the classifier
#
# Xs       examples        (d * n)
# Ys       labels          (c * n)
# W        parameters      (c * d)
#
# returns  the model error as a percentage of incorrect labels

def multinomial_logreg_error(Xs, Ys, W):
   start_logreg_error = time.time()
   c,d = W.shape
   _,n = Xs.shape
   Ys = Ys.astype(int)
   sftmx_preds = softmax(np.matmul(W,Xs))
```

```
   preds_arg = np.argmax(sftmx_preds,axis=0)
   preds = np.zeros([c,n])
   count = 0
   for ii in range(n):
       preds[preds_arg[ii],ii] = int(1.0)
       if np.array_equal(preds[:,ii], Ys[:,ii]):
           count += 1
   error = (n-count)/n;
   assert(error>0)
   end_logreg_error = time.time()
   # print("Logreg Error Time: " + str(end_logreg_error - start_logreg_error))
   return error

# run gradient descent on a multinomial logistic regression objective, with regularization
#
# Xs           training examples (d * n)
# Ys           training labels (d * c)
# gamma        L2 regularization constant
# W0           the initial value of the parameters (c * d)
# alpha        step size/learning rate
# num_iters    number of iterations to run
# monitor_freq how frequently to output the parameter vector
#
# returns      a list of models parameters, one every "monitor_freq" iterations
def gradient_descent(Xs, Ys, gamma, W0, alpha, num_iters, monitor_freq):
   grad = lambda Xs, Ys, gamma, W: multinomial_logreg_grad(Xs, Ys, gamma, W)
   W_update = []
   W = W0
   tt = 0
   while tt < num_iters:
      W = W-alpha*grad(Xs, Ys, gamma, W)
      if tt%monitor_freq == 0:
         W_update.append(W)
         tt+=1
      else:
         tt += 1
   return np.array(W_update)

# estimate the error of the classifier
#
# Xs       examples        (d * n)
# Ys       labels          (c * n)
# gamma    L2 regularization constant
# W        parameters      (c * d)
# nsamples number of samples to use for the estimation
#
# returns  the gradient
def estimate_multinomial_logreg_error(Xs, Ys, W, nsamples):
   start_subsample_error = time.time()
   c,d = W.shape
   _,n = Xs.shape
   sampled_Xs = np.zeros((d, nsamples))
   sampled_Ys = np.zeros((c, nsamples))
   random = np.random.randint(0, high=n, size=nsamples)
   for i in range(nsamples):
      sampled_Xs[:,i] = Xs[:,random[i]]
      sampled_Ys[:,i] = Ys[:,random[i]]
   error = multinomial_logreg_error(sampled_Xs, sampled_Ys, W)
   end_subsample_error = time.time()
```

```
    # print("Subsample Error Time: " + str(end_subsample_error - start_subsample_error))
    return error

if __name__ == "__main__":
    (Xs_tr, Ys_tr, Xs_te, Ys_te) = load_MNIST_dataset()
    gamma = 0.0001 # L2 regualarization constant
    alpha = 1.0   # GD step size
    num_iters = 1000
    monitor_freq = 10
    W0 = np.random.rand(len(Ys_tr[:,0]),len(Xs_tr[:,0]))
    # W0 = np.zeros((len(Ys_tr[:,0]),len(Xs_tr[:,0]))) + 1

    W_iter = gradient_descent(Xs_tr, Ys_tr, gamma, W0, alpha, num_iters, monitor_freq)
```