ROB313: Assignment 3

Sidhanth Moolayil

March 28, 2021

1. Objectives

The objectives of this assignment is to learn the methods of optimization, particularly full batch and stochastic gradient descent, and how to implement them through gradient calculations beforehand, as well as using the autograd auto-differentiation library. These optimizations are applied through statistical perspectives, specifically the MAP and ML estimation procedure to find model parameters, and exploring how the mentioned optimization methods can be applied to maximize the posterior or likelihoods respectively.

2. Code Structure

The code is organized with helper functions answering each question, or sub question (commented appropriately in the code), and the main scripts at the bottom executing the full project.

3. Discussion

3.1 Question 1

3.1.A Part A

The log likelihood when the model outputs 1, but the true data is 0, will be negative infinity. This is reasonable as the output is the farthest it can be from the true result, and thus receives the maximum penalization (negative infinity). While this may cause computational issues, this output would likely not occur as the weights would have to be incredibly large values to equal 1.

3.1.B Part B

Given a zero mean gaussian prior, the log prior will be Pr(w) = -0.5\*wTw. The log likelihood and gradient of the log likelihood are provided in the assignment sheet, and from the log prior above, the gradient was found to be -0.5w. Since the motive of MAP estimation is to maximize the posterior, a gradient of the posterior would be required. Knowing that posterior = likelihood\*prior, taking the log on both sides yields log(posterior) = log(likelihood) + log(prior). By converting to an addition of the log(likelihood) and log(prior), taking the gradient of the log(posterior) is simply the sum of the gradients of the log(likelihood) and log(prior), based on the linearity of the gradient operation. Since the gradients of the log(likelihood) and log(prior) are known, the gradient of the log(posterior) is known. The steepest gradient update rule thereby becomes: wi+1 = wi + (learning\_rate)\*grad(log(posterior)), or in its expanded form:

wi+1 = wi + (learning\_rate)\* [ Text, letter

Description automatically generated – 0.5\*w]

3.1.C Part C

Figure 1 below pictures the training of various models with the learning rate hyperparameter. Of these, the model at a particular epoch with the lowest seen loss was evaluated on the testing data to determine if a given flower was an iris versicolour or not. The test accuracy and test log likelihood is reported in Data Table 1.

Chart

Description automatically generatedChart, line chart

Description automatically generated

Figure 1. Convergence Trends of Full Batch GD and SGB at listed learning rates (in upper right legends).

Data Table 1. Test Accuracy and Test log likelihood

|  |  |
| --- | --- |
| Test Accuracy | Test log likelihood |
| 0.733 | -9.93 |

The test log likelihood is a preferable metric as it also considers how strongly the model predicts inaccurately/accurately, demonstrating when the model may confidently output incorrect results, as well as when the model is not as confident about correct results. Essentially, the log likelihood accounts for the continuous nature of prediction, whereas the accuracy solely accounts for a discrete correctness percentage.

3.2 Question 2

3.2.A Part A

Refer to Code Appendix A.

3.2.B Part B

Refer to Code Appendix A.

3.2.C Part C

A picture containing graphical user interface

Description automatically generated

Figure 2. Stochastic Estimate of Training Set Log Likelihood and Validation Set Log Likelihood vs Epoch

The training negative log likelihood monotonically decreased, while the validation negative log likelihood reached a local minima. This represents the model overfitting on the training set, and loosing the ability to predict on general data (in this case the validation set). For this reason, early stopping was utilized, detecting when the validation negative log likelihood began to increase and truncating the model training at that point. This model was then saved, and tested on testing data, for which the accuracy and negative log likelihoods are reported in Data Table 2.

Data Table 2. Test Accuracy and Test Negative log likelihood

|  |  |
| --- | --- |
| Test Accuracy | Test Negative log likelihood |
| 0.948 | 161.416 |

3.2.D Part D

Below are sample data the model had less than 0.49 certainty as any one number:



Figure 3. Individual Data Examples Model was uncertain about

Appendices

Appendix A

import autograd.numpy as np  
from autograd import value\_and\_grad  
import matplotlib.pyplot as plt  
import numpy as np  
from data\_utils import load\_dataset, plot\_digit  
  
### Question 1  
def neg\_log\_posterior(x, y, w):  
 fhat = 1 / (1 + np.exp(-x.dot(w)))  
 return -np.sum(y \* np.log(fhat) + (~y) \* np.log(1 - fhat)) + np.dot(w.T,w)  
  
def posterior\_grad(x, y, w):  
 fhat = 1 / (1 + np.exp(-x.dot(w)))  
 return -np.sum((y - fhat) \* x, axis=0, keepdims=True).T + 0.5\*w  
  
def full\_batch\_GD(x\_train,y\_train):  
 np.random.seed(1)  
 loss\_best = np.inf  
 plt.figure()  
 # plot different learning rates  
 for learning\_rate in [0.001, 0.0001, 0.01]:  
 w = np.zeros((x\_train.shape[1], 1))  
 loss\_curve = [neg\_log\_posterior(x\_train, y\_train, w)]  
 #print('next LR')  
 for i in range(1000):  
 # compute the gradient  
 grad\_w = posterior\_grad(x\_train, y\_train, w)  
 w = w - learning\_rate \* grad\_w  
 [[nll]] = neg\_log\_posterior(x\_train, y\_train, w)  
 loss\_curve.append(nll)  
 #print(nll)  
 if loss\_curve[-1] < loss\_best:  
 loss\_best = nll  
 w\_best = w.copy()  
 best\_learning\_rate = learning\_rate  
 plt.plot(range(len(loss\_curve)), loss\_curve, label=learning\_rate)  
 plt.xlabel("Epoch")  
 plt.ylabel("Negative Log-Likelihood")  
 plt.title("Full Batch Gradient Descent")  
 plt.legend()  
 plt.show()  
 print("Learning Rate: ", best\_learning\_rate)  
 return w\_best, loss\_best  
  
def SGD(x\_train,y\_train):  
 np.random.seed(1)  
 loss\_best = np.inf  
 plt.figure()  
 for learning\_rate in [0.001, 0.0001, 0.01]:  
 w = np.zeros((x\_train.shape[1], 1))  
 loss\_curve = [neg\_log\_posterior(x\_train, y\_train, w)]  
 for i in range(25000):  
 # compute the gradient  
 mini\_batch = np.random.choice(x\_train.shape[0], size=(1,))  
 grad\_w = posterior\_grad(x\_train[mini\_batch], y\_train[mini\_batch], w)  
 w = w - learning\_rate \* grad\_w  
 [[nll]] = neg\_log\_posterior(x\_train, y\_train, w)  
 loss\_curve.append(nll)  
 if loss\_curve[-1] < loss\_best:  
 loss\_best = nll  
 w\_best = w.copy()  
 best\_learning\_rate = learning\_rate  
 plt.plot(range(len(loss\_curve)), loss\_curve, label=learning\_rate)  
 plt.xlabel("Epoch")  
 plt.ylabel("Negative Log-Likelihood")  
 plt.title("Stochastic Gradient Descent")  
 plt.legend()  
 plt.show()  
 print("Learning Rate: ", best\_learning\_rate)  
 return w\_best, loss\_best  
  
  
def model\_testing(w\_best,x\_test,y\_test):  
 fhat\_test = 1 / (1 + np.exp(-x\_test.dot(w\_best)))  
 accuracy = np.mean((fhat\_test > 0.5) == y\_test)  
 print("Test accuracy: ", accuracy)  
 print("Test log-likelihood: ", -neg\_log\_posterior(x\_test, y\_test, w\_best))  
  
### Question 2  
  
# Part A  
def forward\_pass(W1, W2, W3, b1, b2, b3, x):  
 *"""  
 forward-pass for an fully connected neural network with 2 hidden layers of M neurons  
 Inputs:  
 W1 : (M, 784) weights of first (hidden) layer  
 W2 : (M, M) weights of second (hidden) layer  
 W3 : (10, M) weights of third (output) layer  
 b1 : (M, 1) biases of first (hidden) layer  
 b2 : (M, 1) biases of second (hidden) layer  
 b3 : (10, 1) biases of third (output) layer  
 x : (N, 784) training inputs  
 Outputs:  
 Fhat : (N, 10) output of the neural network at training inputs  
 """* H1 = np.maximum(0, np.dot(x, W1.T) + b1.T) # layer 1 neurons with ReLU activation, shape (N, M)  
 H2 = np.maximum(0, np.dot(H1, W2.T) + b2.T) # layer 2 neurons with ReLU activation, shape (N, M)  
 Fhat = np.dot(H2, W3.T) + b3.T # layer 3 (output) neurons with linear activation, shape (N, 10)  
 # #######  
 # Note that the activation function at the output layer is linear!  
 # You must impliment a stable log-softmax activation function at the ouput layer  
 # #######  
 Fhatmax = Fhat.max(axis=1, keepdims=True)  
 return Fhat - (Fhatmax + np.log(np.sum(np.exp(Fhat - Fhatmax), axis=1, keepdims=True)))  
  
# Part B  
def negative\_log\_likelihood(W1, W2, W3, b1, b2, b3, x, y):  
 *"""  
 computes the negative log likelihood of the model `forward\_pass`  
 Inputs:  
 W1, W2, W3, b1, b2, b3, x : same as `forward\_pass`  
 y : (N, 10) training responses  
 Outputs:  
 nll : negative log likelihood  
 """* Fhat = forward\_pass(W1, W2, W3, b1, b2, b3, x)  
 # ########  
 # Note that this function assumes a Gaussian likelihood (with variance 1)  
 # You must modify this function to consider a categorical (generalized Bernoulli) likelihood  
 # ########  
 nll = -np.sum(Fhat[y])  
 return nll  
  
  
nll\_gradients = value\_and\_grad(negative\_log\_likelihood, argnum=[0,1,2,3,4,5])  
"""  
 returns the output of `negative\_log\_likelihood` as well as the gradient of the   
 output with respect to all weights and biases  
 Inputs:  
 same as negative\_log\_likelihood (W1, W2, W3, b1, b2, b3, x, y)  
 Outputs: (nll, (W1\_grad, W2\_grad, W3\_grad, b1\_grad, b2\_grad, b3\_grad))  
 nll : output of `negative\_log\_likelihood`  
 W1\_grad : (M, 784) gradient of the nll with respect to the weights of first (hidden) layer  
 W2\_grad : (M, M) gradient of the nll with respect to the weights of second (hidden) layer  
 W3\_grad : (10, M) gradient of the nll with respect to the weights of third (output) layer  
 b1\_grad : (M, 1) gradient of the nll with respect to the biases of first (hidden) layer  
 b2\_grad : (M, 1) gradient of the nll with respect to the biases of second (hidden) layer  
 b3\_grad : (10, 1) gradient of the nll with respect to the biases of third (output) layer  
 """  
  
# Part C  
def run\_example(learning\_rate, max\_epoch, M):  
 *"""  
 This example demonstrates computation of the negative log likelihood (nll) as  
 well as the gradient of the nll with respect to all weights and biases of the  
 neural network. We will use 50 neurons per hidden layer and will initialize all  
 weights and biases to zero.  
 """* # load the MNIST\_small dataset  
 from data\_utils import load\_dataset  
 x\_train, x\_valid, x\_test, y\_train, y\_valid, y\_test = load\_dataset('mnist\_small')  
  
 # initialization of weights and biases (weights initialized randomly, biases to 0)  
 W1 = np.random.randn(M, 784)  
 W2 = np.random.randn(M, M)  
 W3 = np.random.randn(10, M)  
 b1 = np.zeros((M, 1))  
 b2 = np.zeros((M, 1))  
 b3 = np.zeros((10, 1))  
  
 # initialization  
 min\_valid\_nll = np.inf  
 nll\_train = []  
 nll\_validation = []  
 iterations = range(max\_epoch)  
 #print(x\_train.shape)  
  
 # training  
 for iteration in range(max\_epoch):  
  
 # shuffle training set  
 epoch\_order = np.random.permutation(x\_train.shape[0])  
  
 # 250 mini-batch size  
 for mini\_batch in epoch\_order.reshape((-1, 250)):  
  
 # gradient calculation for mini-batch  
 (nll, (W1\_grad, W2\_grad, W3\_grad, b1\_grad, b2\_grad, b3\_grad)) = nll\_gradients(W1, W2, W3, b1, b2, b3, x\_train[mini\_batch], y\_train[mini\_batch])  
  
 # calc nll for validation set  
 valid\_nll = negative\_log\_likelihood(W1, W2, W3, b1, b2, b3, x\_valid, y\_valid)  
  
 # store training and validation nll for plots  
 nll\_train.append(nll)  
 nll\_validation.append(valid\_nll)  
  
 # store parameters and iteration number with minimum validation nll  
 if valid\_nll < min\_valid\_nll:  
 min\_valid\_nll = valid\_nll  
 min\_parameters= [i.copy() for i in [W1, W2, W3, b1, b2, b3]]  
  
 # parameter update  
 W1 = W1 - learning\_rate \* W1\_grad  
 W2 = W2 - learning\_rate \* W2\_grad  
 W3 = W3 - learning\_rate \* W3\_grad  
 b1 = b1 - learning\_rate \* b1\_grad  
 b2 = b2 - learning\_rate \* b2\_grad  
 b3 = b3 - learning\_rate \* b3\_grad  
  
 #print(min\_iteration)  
 [W1, W2, W3, b1, b2, b3] = min\_parameters  
  
 # plot training and validation negative log likelihoods  
 plt.figure()  
 plt.plot(iterations, np.array(nll\_train) / 250 \* x\_train.shape[0], 'r', "Training set")  
 plt.plot(iterations, np.array(nll\_validation), 'b', "Validation set")  
 plt.legend()  
 plt.xlabel("Epoch")  
 plt.ylabel("Negative Log Likelihood")  
  
 # testing accuracy  
 y\_test\_pred = forward\_pass(W1, W2, W3, b1, b2, b3, x\_test)  
 test\_accuracy = np.mean(np.argmax(y\_test\_pred, axis=1) == np.argmax(y\_test, axis=1))  
 print("Test accuracy: ", test\_accuracy)  
 test\_nll = negative\_log\_likelihood(W1, W2, W3, b1, b2, b3, x\_test, y\_test)  
 print("Test negative log likelihood: ", test\_nll)  
  
  
 # Part D  
 max\_prob = np.max(np.exp(forward\_pass(W1, W2, W3, b1, b2, b3, x\_test)), axis=1)  
 for i,idx in enumerate(np.where(max\_prob < 0.49)[0]):  
 plot\_digit(x\_test[idx])  
 if i>3:  
 break  
  
# Question 1  
x\_train, x\_valid, x\_test, y\_train, y\_valid, y\_test = load\_dataset("iris")  
y\_train, y\_valid, y\_test = y\_train[:, (1,)], y\_valid[:, (1,)], y\_test[:, (1,)]  
# merge training and validation data  
x\_train = np.vstack([x\_valid, x\_train])  
y\_train = np.vstack([y\_valid, y\_train])  
# append biases  
x\_train = np.hstack([np.ones((x\_train.shape[0], 1)), x\_train])  
x\_test = np.hstack([np.ones((x\_test.shape[0], 1)), x\_test])  
  
# initialization  
w\_best = [0,0]  
loss\_best = [0,0]  
  
w\_best[0], loss\_best[0] = full\_batch\_GD(x\_train,y\_train)  
w\_best[1], loss\_best[1] = SGD(x\_train,y\_train)  
  
model\_testing(w\_best[np.argmin(loss\_best)],x\_test,y\_test)  
  
# Question 2  
  
# hyperparameters  
learning\_rate = 0.001  
max\_epoch = 1000 # max training iterations  
M = 100 # hidden layer nodes  
run\_example(learning\_rate, max\_epoch, M)