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*$w^T w$. 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: $w_{i+1} = w_i$ + (learning_rate)*grad(log(posterior)), or in its expanded form:

$$w_{i+1} = w_i + (\text{learning\_rate})* \left[\sum_{i=1}^{N} \left(y^{(i)} - \hat{f}(\mathbf{x}^{(i)}; \mathbf{w})\right) \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_D^{(i)} \end{bmatrix} - 0.5*w \right]$$

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.

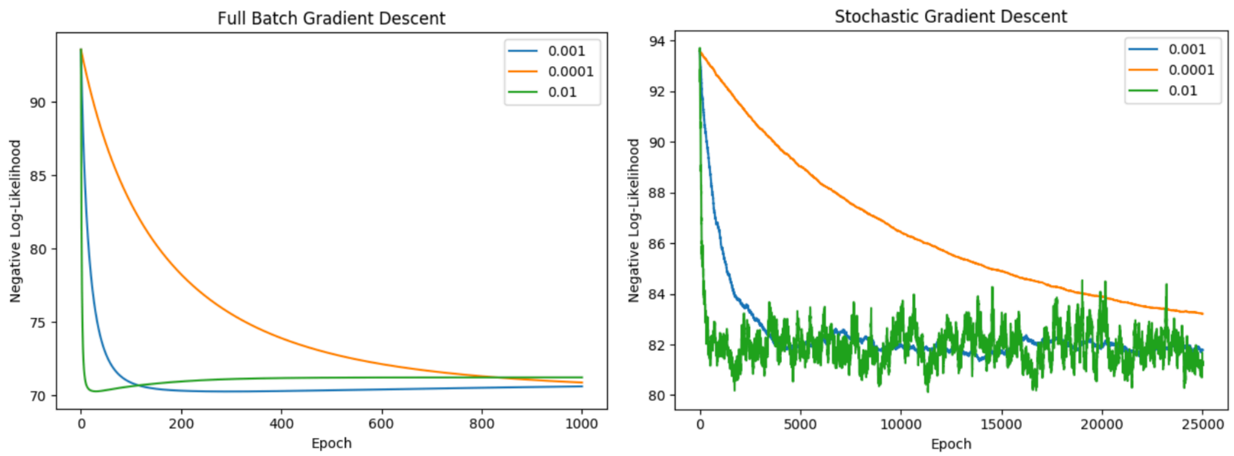

Figure 1. Convergence Trends of Full Batch GD and SGB at listed learning rates (in upper right legends).

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

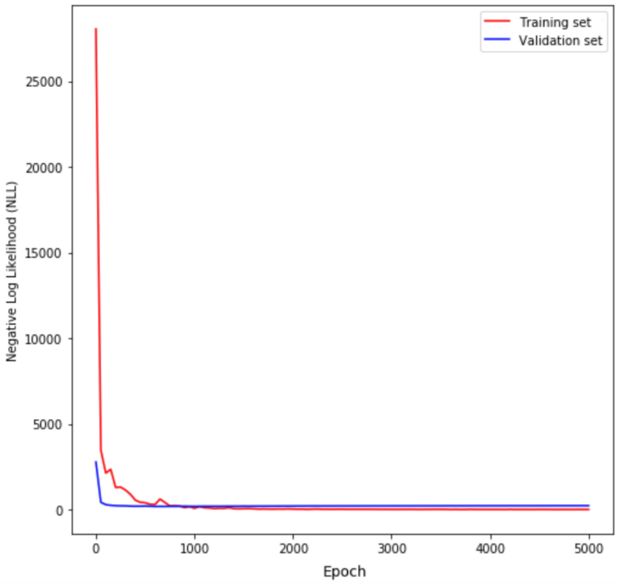

Figure 2. Stochastic Estimate of Training Set Log Likelihood and Validation Set Log Likelihood vs Epoch

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)
```