MATH173B, HW3

Victor Pekkari — epekkari@ucsd.edu

February 8, 2025

Problem 1

SGD with constant step size $\alpha = 10^{-5}$.

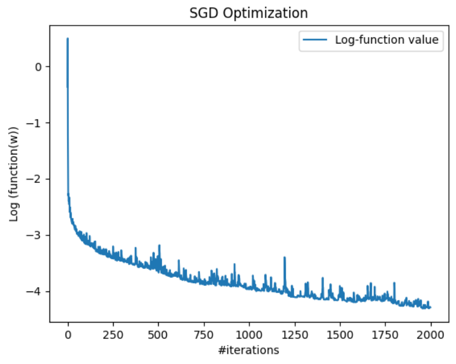


Figure 1: batch size = 30

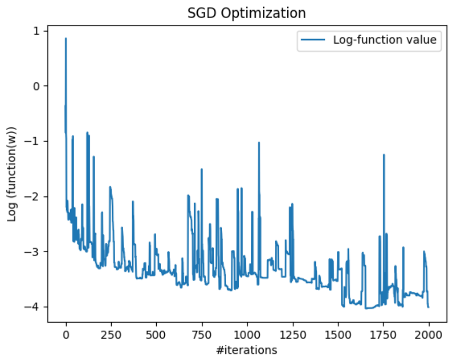


Figure 2: batch size = 5

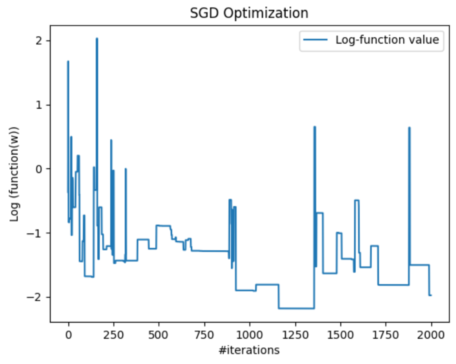


Figure 3: batch size = 1

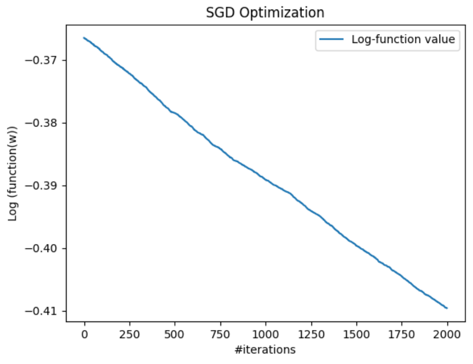


Figure 4: batch size = 1, with normalized input

Comment: (on batch size=1, unnormalized input)

- The log function value $log\left[F(w)\right]$ does not decrease at every iteration. This is because we don't update our weight in the direction of the negative gradient, we can therefore not guarantee descent in every iteration.
- This SGD fails to converge to a fixed w^*, likely because the gradients we use to update w are too irregular.

These results are very consistent with the theory we have learned in class. We learned that "on average" the function should decrease, but it is not guaranteed to decrease at every iteration.

The variance of $g_j(w) = \left[\frac{1}{m}\sum_{k=1}^{m} f_{i_k}(w_t)\right]_j$ decreases as the batch size increases in the plots above which makes sense:

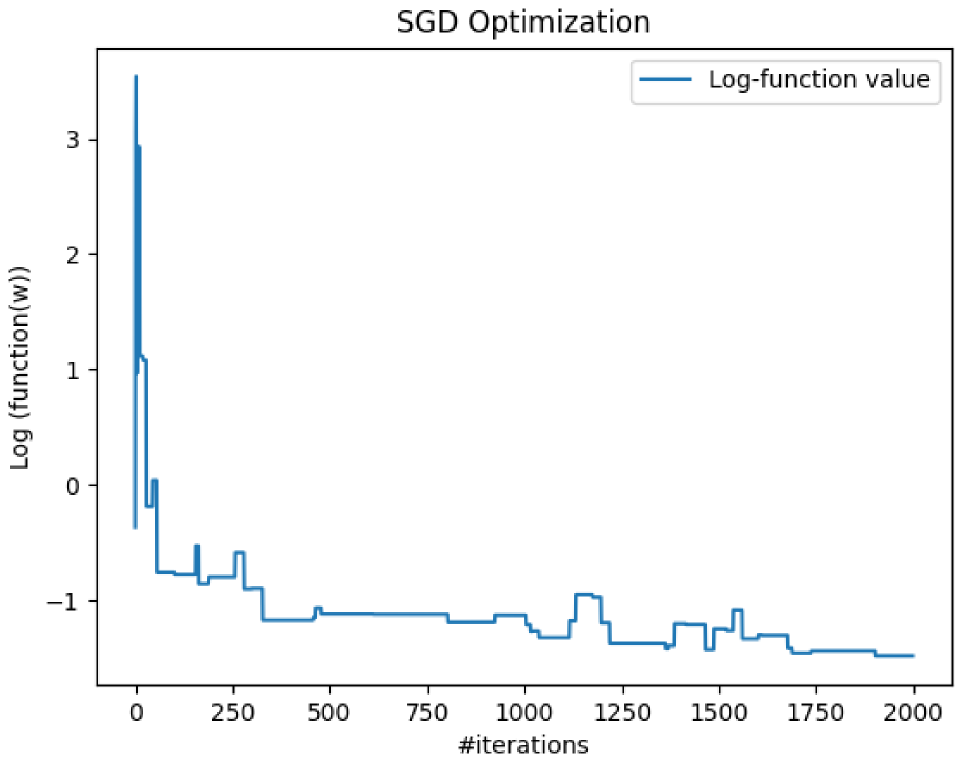
$$\begin{aligned}
Var(g_j(x)) &= \mathbb{E}\left[g_j(x)^2\right] - \left(\frac{\partial F}{\partial x_j}(x)\right)^2 \\
&= \frac{1}{m}\mathbb{E}\left[\left(\frac{\partial f_{i_k}}{\partial x_j}(x)\right)^2\right] + \frac{m-1}{m}\left(\frac{\partial F}{\partial x_j}(x)\right)^2 - \left(\frac{\partial F}{\partial x_j}(x)\right)^2 \\
&= \frac{1}{m}\mathbb{E}\left[\left(\frac{\partial f_{i_k}}{\partial x_j}(x)\right)^2\right] - \frac{1}{m}\left(\frac{\partial F}{\partial x_j}(x)\right)^2 \\
&= \frac{1}{m}\cdot\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial f_i}{\partial x_j}(x)\right)^2 - \frac{1}{m}\left(\frac{\partial F}{\partial x_j}(x)\right)^2.
\end{aligned}$$

(c)

Answer: Error rate was 2% on the test-data and 1.5% on the training-data.

Problem 2

SGD with batch size = 1, and step size: $\alpha_t = 10^{-4} \cdot \sqrt{\frac{1}{1+t}}$



Comment:

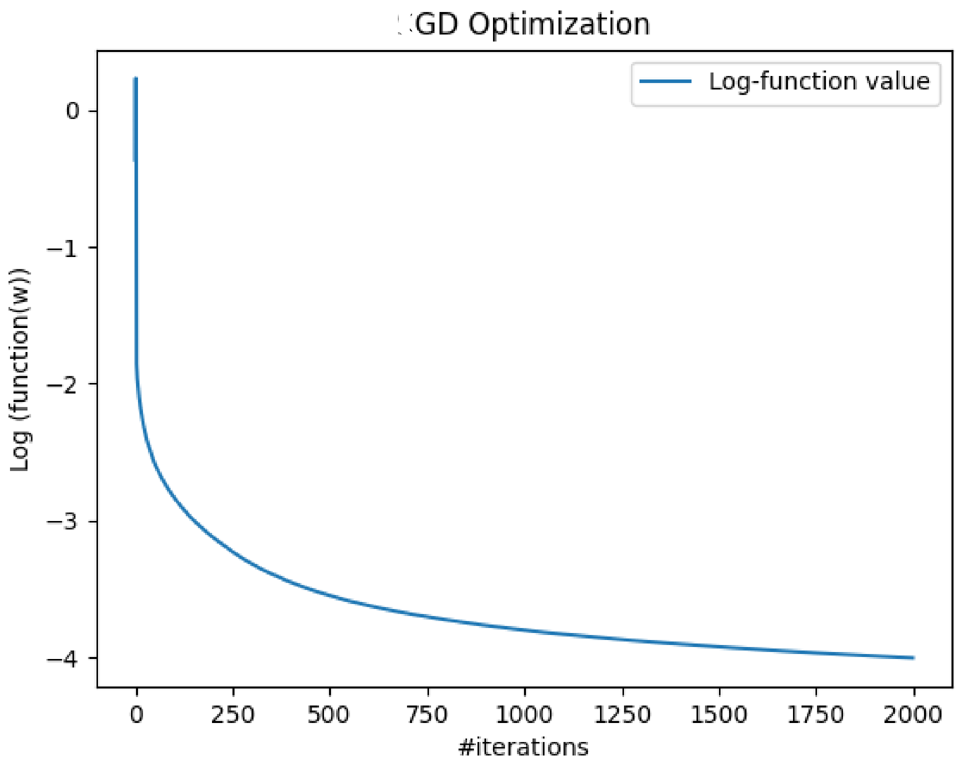
- The log function value $log\,[F(w)]$ does not decrease at every iteration.
- w seems to converge to a fixed w^* as we can see a decreasing function value in the plot, despite fluctuations. This is probably because we decrease our step size after each iteration. Since our step size goes to zero, our updates will converge to zero as well, which means that w will converge to a fixed value to, we just have to hope we are close to a "good" w when we start converging.

(c)

Answer: Error rate was 1% on the test-data and 1.4% on the training-data.

Problem 3

GD with step size: $\alpha_t = 10^{-4} \cdot \sqrt{\frac{1}{1+t}}$



Comment:

- The log function value $log\,[F(w)]$ now decreases at every iteration. That is because we always update the weight in the direction of the negative gradient, unlike in SGD when that is not a guarantee.
- w seems to converge to a fixed w^* as we can in all plots see a decreasing function value, despite fluctuations.
- It is possible to see how there are smaller and smaller changes to the value of the function. Likely due to the decrease of the step size and the gradient

Answer: Error rate was 0.06% on the test-data and 0.05% on the training-data.

Problem 4

(a)

Answer: Say we have a problem where we want to run T number of iterations, we have d number of features for $w \in R^d$, and we have N number of datapoints.

SGD(batch size=1): here we have a time complexity of

$$O(T \cdot d)$$

to run the algorithm. Since we have to do T iterations, and every iteration we have to calculate the gradient, to calculate the gradient we have N terms (data-points) to compute with, and every term has d dimensions.

GD: here we have a time complexity of

$$O(T \cdot N \cdot d)$$

to run the algorithm. Since we have to do T iterations, and every iteration we have to calculate the gradient, to calculate the gradient we have 1 term (data-points) to compute, and every this term has d dimensions.

Conclusion: The big upside with SGD compared to GD computationally, is that SGD becomes a lot faster when N is huge,

(b & c)

Answer: (b)

(1): $F(w) = 0.1563$, (2): $F(w) = 0.2448$, (3): $F(w) = 0.01792$

Gradient descent (3) got closest to the minimizer, it does it at a much more expensive computational cost however. The decay in step size might not let SGD with adaptive step size converge in time.

Comment: Gradient descent got the highest accuracy on both the training data-set and the test-set. SGD with adaptive step size can achieve better accuracy but a worse function value because accuracy depends only on correct classification (a discrete measure), while loss also depends on prediction confidence (a continuous measure). As the adaptive step size shrinks, updates become too small to fully minimize the loss, even if the decision boundary is already well-positioned for high accuracy. In contrast, SGD with a constant step size continues optimizing loss but may not generalize as well.

It is very likely that the SGD with adaptive step size would have overtaken the other SGD in loss function value if we only ran it for more iterations, since a smaller step size requiers us to do more updates.

Python code for question 1,2 and 3:

```python
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

csv_file = "mnist_train.csv"
test_csv = "mnist_test.csv"
data = pd.read_csv(csv_file)

labels = data.iloc[:, 0].to_numpy()
pixels = data.iloc[:, 1:].to_numpy()
ones_column = np.ones((pixels.shape[0], 1))
pixels = np.hstack((pixels, ones_column))
normalize = False

mask = (labels == 1) | (labels == 2)
X = pixels[mask].astype(float)
y = labels[mask]

y = np.where(y == 1, 1, -1)

if normalize:
    X /= 255.0

X_1 = X[y == 1][:2000]
X_2 = X[y == -1][:2000]
y_1 = y[y == 1][:2000]
y_2 = y[y == -1][:2000]

X = np.concatenate((X_1, X_2), axis=0)
y = np.concatenate((y_1, y_2), axis=0)


lr = 1e-5

N = 4000

# 1a
# w = sgd(lr='a', save_fig=False, nbr_iterations=2000)

# 2a
# w = sgd(lr='b', save_fig=False, nbr_iterations=2000)

# 3a
# w = sgd(lr='b', save_fig=True, nbr_iterations=2000, batch_size=4000)

# print(function(weights=w))
# print(classify(w=w))


def gradient(weights:np.ndarray[float], batch_size:int=1) -> np.ndarray[float]:
    indices = np.random.choice(np.arange(0, 4000), size=batch_size, replace=True)
    grad = np.zeros_like(weights)
    for index in indices:
        x_multiplied_y = (-y[index]) * X[index]
        denominator = 1 + np.exp(-np.dot(weights, X[index]) * y[index])
        grad += (1 / batch_size) * x_multiplied_y * (1 - (1 / denominator))
    return grad

def load_test_data():
    test_data = pd.read_csv(test_csv)
    test_labels = test_data.iloc[:, 0].to_numpy()
    test_pixels = test_data.iloc[:, 1:].to_numpy()

    ones_column = np.ones((test_pixels.shape[0], 1))
    test_pixels = np.hstack((test_pixels, ones_column))

    test_mask = (test_labels == 1) | (test_labels == 2)
    test_X = test_pixels[test_mask].astype(float)[:500]

    test_y = test_labels[test_mask][:500]

    test_y = np.where(test_y == 1, 1, -1)

    if normalize:
        test_X /= 255.0
    return test_X, test_y

def classify(w):
    x_test, y_test = load_test_data()
    print("test_data: ", classify_test(x_test, y_test, weights=w))
    print("training_data: ", classify_test( X, y, weights=w))

def classify_test(test_X, test_y, weights:np.ndarray[float]) -> float:

    predictions = np.dot(test_X, weights)
    predicted_labels = np.sign(predictions)
    summation = 0
    for index in range(test_y.shape[0]):
        if predicted_labels[index] == test_y[index]:
            summation += 1
    return summation / test_y.shape[0]

def log_function(weights:np.ndarray[float]) -> float:
    return np.log(function(weights))

def function(weights:np.ndarray[float]) -> float:
    #return (1 / N) * np.sum(np.log(1 + np.exp(-y[:N] * np.dot(X[:N], weights))))
    return (1 / N) * np.sum(np.log(1 + np.exp(-y[i] * np.dot(weights, X[i]))) for i in range(N))

def learning_rate(kind, t) -> float:
    if kind == 'a':
        return 1e-5
    else:
        return 1e-4 * np.sqrt(1/(1+t))

def sgd(lr:str, save_fig:bool=False, nbr_iterations:int = 2000, batch_size=1) -> None:
    weights = np.zeros(X.shape[1])
    function_list = []
    iteration_list = []
    weigh_list = []
    for i in range(nbr_iterations):
        iteration_list.append(i)
        function_list.append(log_function(weights))
        weights -= learning_rate(kind=lr, t=i) * gradient(weights, batch_size=batch_size)

    if save_fig:
        plt.plot(iteration_list, function_list, label='Log-function value')
        plt.xlabel('#iterations')
        plt.ylabel('Log (function(w))')
        plt.title('SGD Optimization')
        plt.legend()
        plt.savefig('sgd_optimization_plot2.png')
        plt.show()
    print(function_list[-1])
    return weights
```