MATH 173A: Homework #5

Due on Mov 19, 2024 at 23:59pm

Professor Cloninger

Ray Tsai

A16848188

Problem 1

Consider the function $f(x_1, x_2) = (2x_1 - 1)^4 + (x_1 + x_2 - 1)^2$. Recall the minimizer is $(\frac{1}{2}, \frac{1}{2})$ for this problem. Starting at $x^{(0)} = (0, 0)$, perform two steps of Newton's method to find $x^{(2)}$. Show your work.

Proof. Note that

$$\nabla f(x_1, x_2) = \begin{bmatrix} 8(2x_1 - 1)^3 + 2(x_1 + x_2 - 1) \\ 2(x_1 + x_2 - 1) \end{bmatrix},$$
$$\nabla^2 f(x_1, x_2) = \begin{bmatrix} 48(2x_1 - 1)^2 + 2 & 2 \\ 2 & 2 \end{bmatrix}.$$

Starting at $x^{(0)} = (0, 0)$, we have

$$[\nabla^2 f(x^{(0)})]^{-1} = \frac{1}{48} \begin{bmatrix} 1 & -1 \\ -1 & 25 \end{bmatrix}, \quad \nabla f(x^{(0)}) = \begin{bmatrix} -10 \\ -2 \end{bmatrix},$$
$$x^{(1)} = x^{(0)} - [\nabla^2 f(x^{(0)})]^{-1} \nabla f(x^{(0)}) = \frac{1}{6} \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$
$$[\nabla^2 f(x^{(1)})]^{-1} = \frac{1}{64} \begin{bmatrix} 3 & -3 \\ -3 & 35 \end{bmatrix}, \quad \nabla f(x^{(1)}) = \frac{1}{48} \begin{bmatrix} -\frac{64}{27} \\ 0 \end{bmatrix},$$
$$x^{(2)} = x^{(1)} - [\nabla^2 f(x^{(1)})]^{-1} \nabla f(x^{(1)}) = \frac{1}{18} \begin{bmatrix} 5 \\ 13 \end{bmatrix}.$$

$\square$

Question 2

We'll consider the function $f(x_1, x_2) = 200(x_2 - x_1^2)^2 + (1 - x_1)^2$.

In [50]:
```
# import statements
import numpy as np
from matplotlib import pyplot as plt
```

In [ ]:
```
def f(x):
    return 200 * (x[1] - x[0] ** 2)**2 + (1 - x[0])**2

def df(x):
    return np.array([-800 * x[0] * (x[1] - x[0] ** 2) - 2 * (1 - x[0]),
                     200 * (x[1] - x[0] ** 2)])

def hf(x):
    return np.array([[2400 * x[0]**2 - 800 * x[1] + 2, -800 * x[0]],
                     [-800 * x[0], 400]])

minimum = np.array([1, 1])
T = 5000
```

Part 2A

Write a computer program to run Newton's method on this problem.

In [52]:
```
def newton(x, mu=1e-1):
    return x - mu * np.linalg.solve(hf(x), df(x))

newton_x_values = []
newton_fx_values = []

x = np.array([0, 0])

for i in range(T):
    newton_x_values.append(np.linalg.norm(x - minimum))
    newton_fx_values.append(f(x) - f(minimum))
    x = newton(x)
```

Part 2B

Write a computer program to run Gradient Descent with a fixed step size $\mu = 10^{-3}$ on this problem.

In [53]:
```
def gd(x, mu = 1e-3):
    return x - mu * df(x)

gd_x_values = []
gd_fx_values = []

x = np.array([0, 0])

for i in range(T):
    gd_x_values.append(np.linalg.norm(x - minimum))
    gd_fx_values.append(f(x) - f(minimum))
    x = gd(x)
```

Part 2C

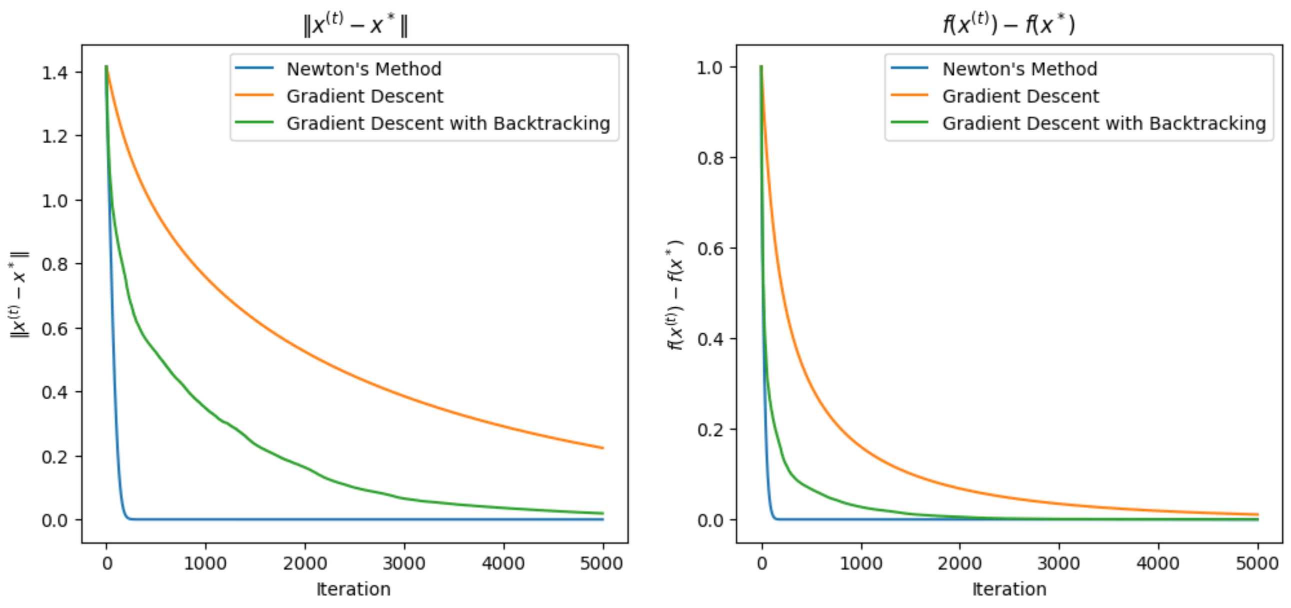
Write a computer program to run Gradient Descent with backtracking line- search for this problem (you may set β and γ as you wish).

In [54]:
```
def backtrack(x, beta=0.5, gamma=0.8):
    mu = 1e-2
    while f(gd(x, mu)) > f(x) - gamma * mu * np.linalg.norm(df(x))**2:
        mu *= beta
    return mu

gdb_x_values = []
gdb_fx_values = []

x = np.array([0, 0])

for i in range(T):
    gdb_x_values.append(np.linalg.norm(x - minimum))
    gdb_fx_values.append(f(x) - f(minimum))
    mu = backtrack(x)
    x = gd(x, mu)
```

Part 2D

Starting at the same $x^{(0)}$, run each of the three algorithms and plot $||x^{(t)} - x^*||$ for each, in the same figure. In a separate figure, plot $f(x^{(t)}) - f(x^*)$ for each of them. Comment on the performance. Note that $x^* = (1, 1)$ for this problem.

In [55]:
```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(T), newton_x_values, label='Newton\'s Method')
plt.plot(range(T), gd_x_values, label='Gradient Descent')
plt.plot(range(T), gdb_x_values, label='Gradient Descent with Backtracking')
plt.xlabel('Iteration')
plt.ylabel(r'$\|x^{(t)} - x^*\|$')
plt.legend()
plt.title(r'$\|x^{(t)} - x^*\|$')

plt.subplot(1, 2, 2)
plt.plot(range(T), newton_fx_values, label='Newton\'s Method')
plt.plot(range(T), gd_fx_values, label='Gradient Descent')
plt.plot(range(T), gdb_fx_values, label='Gradient Descent with Backtracking'
plt.xlabel('Iteration')
plt.ylabel(r'$f(x^{(t)}) - f(x^*)$')
plt.legend()
plt.title(r'$f(x^{(t)}) - f(x^*)$')

plt.show()
```



Comment

Newton's Method converges significantly faster than the other two and has significantly lower runtime compare to gradident descent with backtracking. Gradident descent with backtracking performs better than the ordinary gradient descent.

Question 3

In this coding question, you'll implement a classifier with logistic regression

$$F(w) = \frac{1}{N}\sum_{i=1}^{N} \log(1 + e^{-\langle w, x_i \rangle y_i}).$$

In [27]:
```
# import statements
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
```

Load MNIST Data

In [28]:
```
# !pip3 install scikit-learn
# this cell will take a minute to run depending on your internet connection
X, y = fetch_openml('mnist_784', version=1, return_X_y=True) # getting data
print('X shape:', X.shape, 'y shape:', y.shape)
X, y = fetch_openml('mnist_784', version=1, return_X_y=True) # getting data
print('X shape:', X.shape, 'y shape:', y.shape)
# this cell processes some of the data

# if this returns an error of the form "KeyError: 0", then try running the f
# X = X.values # this converts X from a pandas dataframe to a numpy array

X = X.values
digits = {j:[] for j in range(10)}
for j in range(len(y)): # takes data assigns it into a dictionary
    digits[int(y[j])].append(X[j].reshape(28,28))
digits = {j:np.stack(digits[j]) for j in range(10)} # stack everything to be
for j in range(10):
    print('Shape of data with label', j, ':', digits[j].shape )
```

```
X shape: (70000, 784) y shape: (70000,)
X shape: (70000, 784) y shape: (70000,)
Shape of data with label 0 : (6903, 28, 28)
Shape of data with label 1 : (7877, 28, 28)
Shape of data with label 2 : (6990, 28, 28)
Shape of data with label 3 : (7141, 28, 28)
Shape of data with label 4 : (6824, 28, 28)
Shape of data with label 5 : (6313, 28, 28)
Shape of data with label 6 : (6876, 28, 28)
Shape of data with label 7 : (7293, 28, 28)
Shape of data with label 8 : (6825, 28, 28)
Shape of data with label 9 : (6958, 28, 28)
```

Data PreProcess

In [29]:
```
x_4 = digits[4][:500].reshape(500,-1)
x_9 = digits[9][:500].reshape(500,-1)

x_4_test = digits[4][500:1000].reshape(500,-1)
x_9_test = digits[9][500:1000].reshape(500,-1)

x_train = np.vstack((x_4, x_9))
x_train = x_train.astype('float32') / 255.0

x_test = np.vstack((x_4_test, x_9_test))
x_test = x_test.astype('float32') / 255.0

y_train = np.hstack((-1 * np.ones(500), np.ones(500)))
y_test = np.hstack((-1 * np.ones(500), np.ones(500)))
```

Define $F(w)$ and $\nabla F(w)$

In [30]:
```
def F(w):
    sum = 0
    N = len(x_train)
    for i in range(N):
        sum += np.log(1 + np.exp(-y_train[i] * np.dot(w, x_train[i])))
    return sum / N

def dF(w):
    sum = 0
    N = len(x_train)
    for i in range(N):
        sum += -y_train[i] * np.exp(-y_train[i] * np.dot(w, x_train[i])) * x
    return sum / N
```

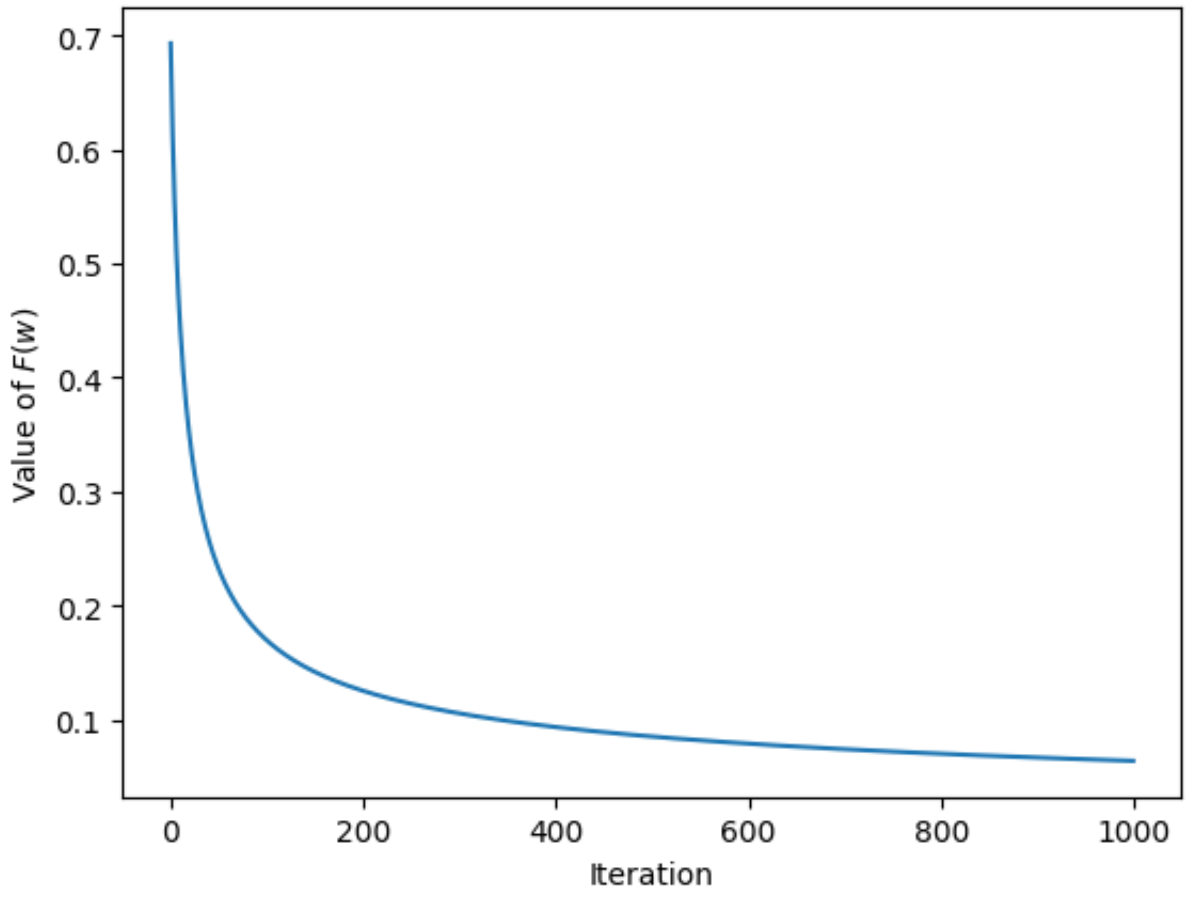
Problem Statement

We will consider the MNIST coding question from HW4. In this question, we run these questions for *differentiating 4's and 9's*. You can reuse the template from the previous homework for loading / formatting MNIST. Implement the following two methods and plot $F(w)$ per iteration for each. You need to submit (i) the code for the algorithms and plots, and (ii) the plots.

Gradient descent with backtracking line search

At each iteration t, initialize the step size $\mu = 10^{-1}$, and use $\gamma = 0.5$ and $\beta = 0.8$ to determine the correct $\mu^{(t)}$. Run your algorithm for at least 10,000 iterations and plot the loss curve $F(w^{(t)})$ as a function of t.

In [31]:
```
T = 1000

w_values = []
iterations = range(T)

def gd(w, mu = 1e-1):
    return w - mu * dF(w)

def backtrack(w, gamma=0.5, beta=0.8):
    mu = 1e-1
    while F(gd(w, mu)) > F(w) - gamma * mu * np.linalg.norm(dF(w))**2:
        mu *= beta
    return mu

w = np.zeros(x_train.shape[1])
for i in iterations:
    w_values.append(F(w))
    mu = backtrack(w)
    w = gd(w, mu)

plt.plot(iterations, w_values)
plt.xlabel("Iteration")
plt.ylabel(r"Value of $F(w)$")
plt.show()
```



Error Rate

In [32]:
```
error = 0

for i in range(1000):
    if np.dot(w, x_test[i]) > 0:
        y_test[i] = 1
    else:
        y_test[i] = -1
    error += (y_test[i] != y_train[i])

print("Error rate:", error / 1000 * 100, "%")
```

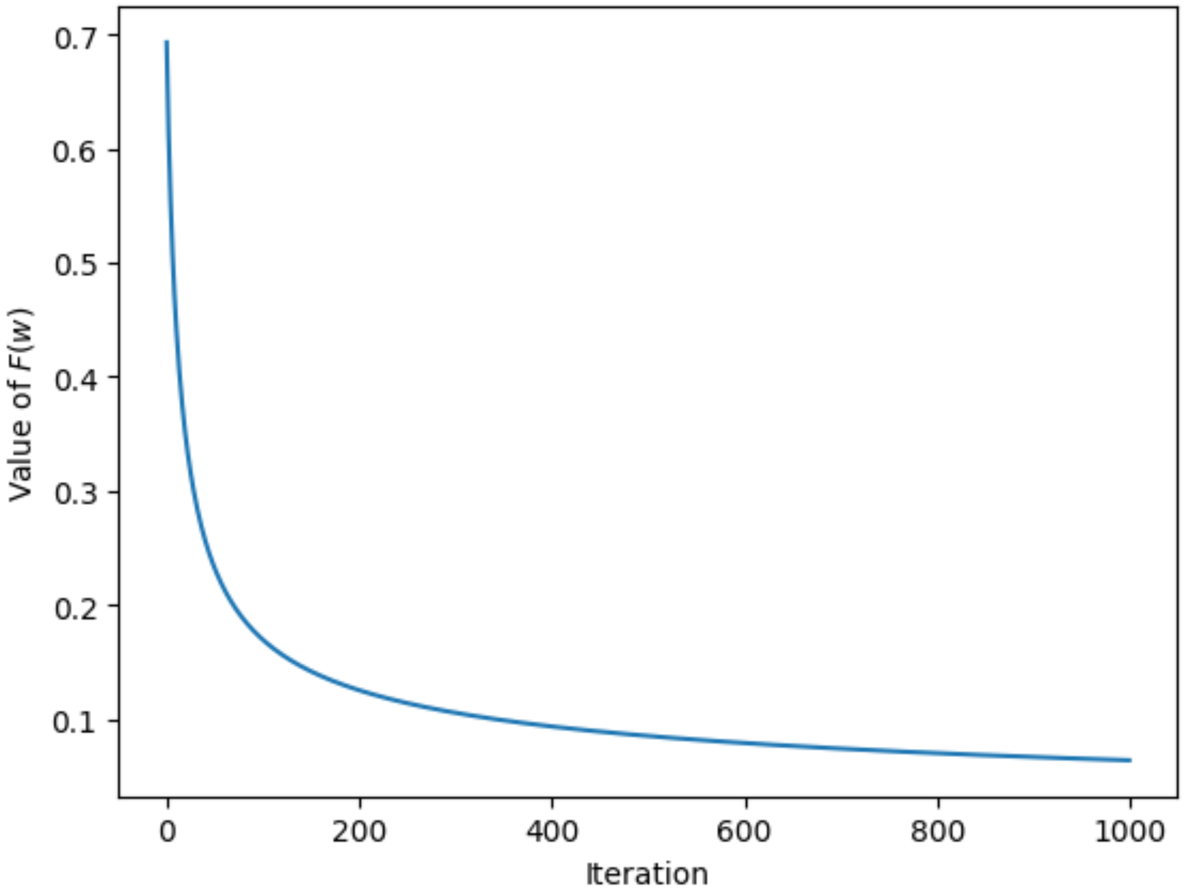
```
Error rate: 4.3 %
```

Gradient Descent with Nesterov acceleration.

You can experiment with the parameters until you find something you like.

In [35]:
```
T = 1000

w_values = []
```

```
iterations = range(T)

def nesterov(w, w_old, mu = 1e-3, beta = 0.9):
    v = w + beta * (w - w_old)
    return v - mu * dF(v)

w = np.zeros(x_train.shape[1])
w_old = w
for i in iterations:
    w_values.append(F(w))
    w_temp = w
    w = nesterov(w, w_old, mu)
    w_old = w

plt.plot(iterations, w_values)
plt.xlabel("Iteration")
plt.ylabel(r"Value of $F(w)$")
plt.show()
```



Error Rate

In [34]:
```
error = 0
```

```
for i in range(1000):
    if np.dot(w, x_test[i]) > 0:
        y_test[i] = 1
    else:
        y_test[i] = -1
    error += (y_test[i] != y_train[i])

print("Error rate:", error / 1000 * 100, "%")
```

```
Error rate: 4.3 %
```