MATH 173A - HW5

Victor Pekkari, epekkari@ucsd.edu

November 20, 2024

Problem 1

$$f(x) = (2x_1 - 1)^4 + (x_1 + x_2 - 1)^2$$

$$\nabla f(x) = \begin{bmatrix} 8(2x_1 - 1)^3 + 2(x_1 + x_2 - 1) \\ 2(x_1 + x_2 - 1) \end{bmatrix} \qquad \nabla^2 f(x) = \begin{bmatrix} (48(2x_1 - 1)^2 + 2) & 2 \\ 2 & 2 \end{bmatrix} \qquad x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$x_{t+1} = x_t - \nabla^2 f(x_t)^{-1} \cdot \nabla f(x_t)$$

$$x_1 = x_0 - \nabla^2 f(x_0)^{-1} \cdot \nabla f(x_0) = \begin{bmatrix} -\frac{1}{6} \\ -\frac{5}{6} \end{bmatrix} \tag{1}$$

$$x_2 = x_1 - \nabla^2 f(x_1)^{-1} \cdot \nabla f(x_1) \approx \begin{bmatrix} 0.0078 \\ -0.4245 \end{bmatrix} \tag{2}$$

Answer: $x_2 = \begin{bmatrix} 0.0078 \\ -0.4245 \end{bmatrix}$

```python
import numpy as np
import matplotlib.pyplot as plt
nbr_iterations = 5000

def plot_function():
    x0 = np.linspace(-2, 2, 400)
    x1 = np.linspace(-1, 3, 400)
    X0, X1 = np.meshgrid(x0, x1)
    Z = np.zeros_like(X0)

    for i in range(X0.shape[0]):
        for j in range(X0.shape[1]):
            Z[i, j] = function(np.array([X0[i, j], X1[i, j]]))

    plt.figure(figsize=(8, 6))
    cp = plt.contourf(X0, X1, Z, levels=50, cmap='viridis')
    plt.colorbar(cp)
    plt.title('Contour plot of the function')
    plt.xlabel('x0')
    plt.ylabel('x1')
    plt.show()

def function(x: np.ndarray) -> float:
    return 200 * ((x[1] - (x[0]**2)) ** 2) + ((1 - x[0])**2)

def gradient(x: np.ndarray) -> np.ndarray:
    return np.array([
        -800 * (x[1] - x[0]**2) * x[0] - 2 * (1 - x[0]),
        400 * (x[1] - x[0]**2)
    ])

def hessian(x: np.ndarray, inverse=False, regularization=1e-8) -> np.ndarray:
    hessian = np.array([
        [1600 * x[0]**2 - 800 * (x[1] - (x[0]**2)) + 2, -800 * x[0]],
        [-800 * x[0], 400]
    ])
    if inverse:
        # hessian += np.eye(2) * regularization  # Add regularization term
        return np.linalg.inv(hessian)
    return hessian

def newtons(x: np.ndarray, stopping_criteria=1e-5, iterations:int=nbr_iterations) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for i in range(iterations):
        x -= np.dot(hessian(x, inverse=True), gradient(x))
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(x)
    return x_values, y_values

def armijo_condition(x:np.ndarray, lr:float, sigma) -> bool:
    grad = gradient(x)
    left = function(x - lr*grad)
    right = function(x) - sigma * lr * np.linalg.norm(grad)**2
    return left <= right

def backtracking(x: np.ndarray, iterations:int=nbr_iterations, sigma=1e-1, beta=0.9, lr=8e-4) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for _ in range(iterations):
        learning_rate = lr
        while not armijo_condition(x,lr=learning_rate, sigma=sigma):
            learning_rate *= beta
        x -= lr * gradient(x)
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(f'x: {x}, f(x): {function(x)}')
    return x_values, y_values

def gradient_descent(x: np.ndarray, iterations:int=nbr_iterations, lr=1e-3) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for i in range(iterations):
        x -= lr * gradient(x)
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(f'x: {x}, f(x): {function(x)}')

    return x_values, y_values

iterations = [i for i in range(1, nbr_iterations+2)]

x_new, y_new = newtons(np.array([0.5, 0.5],dtype=float))
x_gd, y_gd = gradient_descent(np.array([0.5, 0.5],dtype=float))
x_back, y_back = backtracking(np.array([0.5, 0.5],dtype=float))

plt.plot(iterations, x_new, label='Newton\'s', c='r')
plt.plot(iterations, x_gd, label='Gradient Descent', c='b')
plt.plot(iterations, x_back, label='Backtracking', c='g')
plt.title("|| x_i - x_* ||")
plt.xlabel("iterations")
plt.ylabel("dist from optimal x_*")
plt.show()

plt.plot(iterations, y_new, label='Newton\'s', c='r')
plt.plot(iterations, y_gd, label='Gradient Descent', c='b')
plt.plot(iterations, y_back, label='Backtracking', c='g')
plt.title("|| f(x_i) ||")
plt.xlabel("iterations")
plt.ylabel("|| f(x_i) ||")
plt.show()
```

hw5q3

November 18, 2024

1 set-up

[1]:
```
!pip3 install scikit-learn
# import statements
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
!pip install --upgrade certifi
!brew install openssl
!brew link openssl --force
import certifi
import ssl
import os

os.environ['SSL_CERT_FILE'] = certifi.where()
import requests

url = 'https://example.com'
response = requests.get(url, verify=False)
!/Applications/Python\ 3.12.4/Install\ Certificates.command
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
# 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␣
 ↪from online

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␣
 ↪following first:
X = X.values # this converts X from a pandas dataframe to a numpy array

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␣
 ↪one numpy array
for j in range(10):
    print('Shape of data with label', j, ':', digits[j].shape )
    # this cell would stack 100 examples from each class together
# this cell also ensures that each pixel is a flot between 0 and 1 instead of␣
 ↪an int between 0 and 255
data = []
for i in range(10):
    flattened_images = digits[i][:100].reshape(100,-1)
    data.append(flattened_images)

data = np.vstack(data)
data = data.astype('float32') / 255.0
```

```
Requirement already satisfied: scikit-learn in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(1.5.2)
Requirement already satisfied: numpy>=1.19.5 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.14.1)
Requirement already satisfied: joblib>=1.2.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (3.5.0)
Requirement already satisfied: certifi in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(2024.8.30)
==> Downloading https://formulae.brew.sh/api/formula.jws.json
######################################################################## 100.0%
==> Downloading https://formulae.brew.sh/api/cask.jws.json
######################################################################## 100.0%
Warning: openssl@3 3.4.0 is already installed and up-to-date.
To reinstall 3.4.0, run:
  brew reinstall openssl@3
Warning: Already linked: /opt/homebrew/Cellar/openssl@3/3.4.0
To relink, run:
  brew unlink openssl@3 && brew link openssl@3
zsh:1: no such file or directory: /Applications/Python 3.12.4/Install
Certificates.command

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/urllib3/connectionpool.py:1099: InsecureRequestWarning: Unverified
```

```
HTTPS request is being made to host 'example.com'. Adding certificate
verification is strongly advised. See:
https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls-warnings
  warnings.warn(

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)
```

2 Extract 4s and 9s

[2]:
```
# create dataset here (essentially just create a numpy array of 1's and -1's␣
  ↪for the labels)

#zeros = digits[0][:500]
#ones = digits[1][:500]
nbr_examples = 500
numbers = [4,9]
map = {((-1)** (i+1)):[digit.flatten() for digit in digits[i][:nbr_examples]]␣
  ↪for i in numbers}
for key in [-1, 1]:
    x = map[key]
    for i in range(len(x)):
        x_i = (x[i] - x[i].min()) / (x[i].max() - x[i].min())
        map[key][i] = x_i
```

3 Function and gradients....

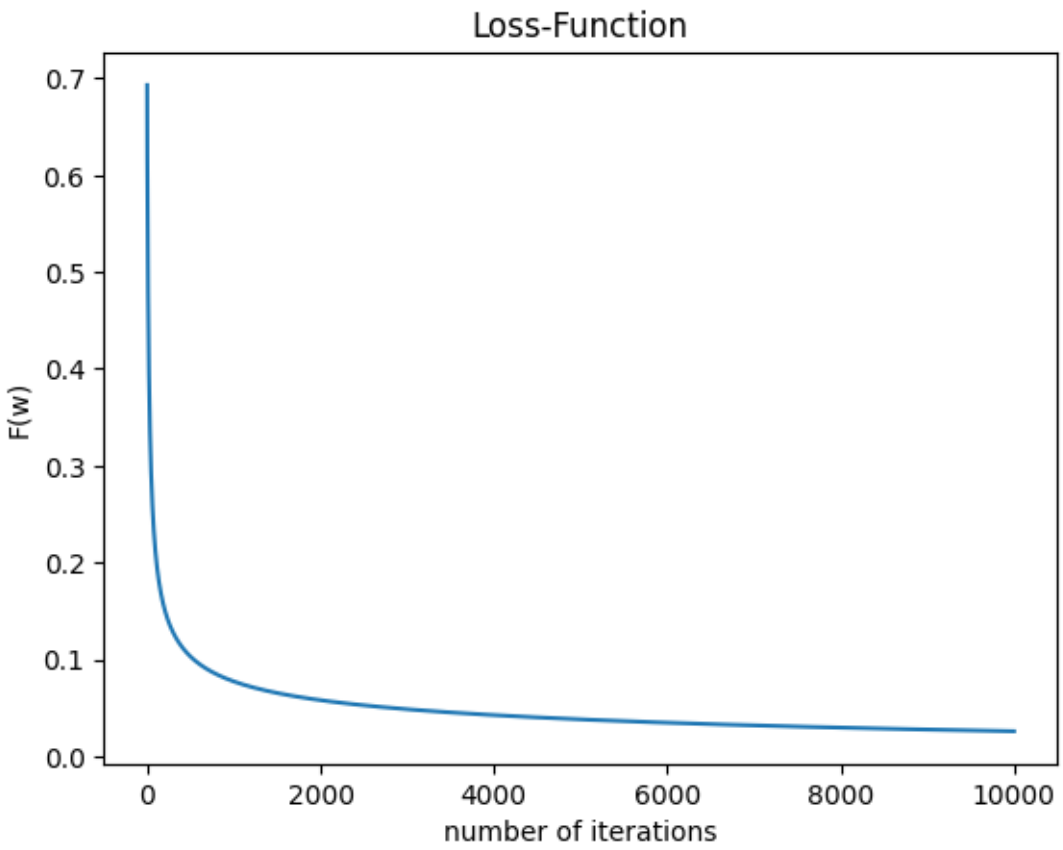
[3]:
```
def function(weights:[float], map:{int:np.array}) -> float:
    sum = 0
    for i in [-1,1]:
        for x_i in map[i]:
            exponent = float(np.dot(weights, x_i)) * i * (-1)
            inner_expression = 1 + math.exp(exponent)
            sum += math.log(inner_expression)
    return sum / (len(map[-1]) + len(map[1]))
```

```
def gradient(weights:[float], map:{int:np.array}) -> np.array:
    gradient_array = np.zeros(len(weights))
    N = len(weights)
    for w_index in range(N):
        for i in [-1,1]:
            for x_i in map[i]:
                exponent = float(np.dot(weights, x_i)) * i * (-1)
                numerator = (-i) * x_i[w_index] * math.exp(exponent)
                denominator = 1 + math.exp(exponent)
                gradient_array[w_index] += numerator / denominator
        gradient_array[w_index] /= N
    return gradient_array

def optimized_gradient(weights:[float], map:{int:np.array}) -> np.array:
    gradient_array = np.zeros(len(weights))
    N = len(weights)
    for i in [-1,1]:
        for x_i in map[i]:
            exponent = float(np.dot(weights, x_i)) * i * (-1)
            numerator = (-i) * math.exp(exponent)
            denominator = 1 + math.exp(exponent)
            for w_index in range(N):
                gradient_array[w_index] += numerator * x_i[w_index] /␣
 ↪denominator
    return np.array([grad / N for grad in gradient_array])

def printf(weights:[float], map:{int:np.array}, i:int):
    if (i % 1000) == 0:
        print(f"iteration : {i}, F(w) = {function(weights, map)}")
    #else:
     #   pass
        #print(f"iteration : {i}, F(w) = {function(weights, map)}")
def get_P(weights:[float], map:{int:np.array}):
    fw = optimized_gradient(weights, map)
    return np.sign(fw) * np.sum(np.abs(fw))

def get_P1(weights: np.array, map: {int: np.array}, count_vector):
    fw = optimized_gradient(weights, map)
    j_star = np.argmax(np.abs(fw))
    if count_vector is not None:
        count_vector[j_star] += 1
    p_t = np.zeros_like(fw)
    p_t[j_star] = np.sign(fw[j_star]) * np.max(np.abs(fw))

    return p_t
```

4 Backtracking line search

[38]:
```
# 4 a
import math
iteration_nbr = 10**4

weights = np.array([0 for _ in range(784)], dtype=np.float64)

def armijo_condition(x:np.ndarray, lr:float, gamma,grad) -> bool:
    left = function(x + lr*grad,map)
    right = function(x,map) - gamma * lr * np.linalg.norm(grad)**2
    return left <= right

def backtracking(x: np.ndarray, iterations:int=iteration_nbr, gamma=0.5, beta=0.
 ↪8, lr=0.1) -> np.ndarray:
    x_values = [0]
    y_values = [function(x, map)]
    for i in range(int(iterations)):
        printf(weights, map, n)
        learning_rate = lr
        grad = optimized_gradient(x,map)
        function_value = function(x,map)
        while not armijo_condition(x,lr=learning_rate, gamma=gamma,grad=grad):
            learning_rate *= beta
        x -= lr * gamma * grad
        x_values.append(i)
        y_values.append(function(x,map))
    #print(f'x: {x}, f(x): {function(x,map)}')
    printf(weights, map, 10000)
    return x_values, y_values

start_value = np.array([0 for _ in range(784)], dtype=np.float64)
x_values, y_values = backtracking(start_value)

plt.plot(x_values, y_values)
plt.xlabel('number of iterations')
plt.ylabel('F(w)')
plt.title('Loss-Function')
#plt.xticks(range(1, iteration_nbr + 1, 500))
plt.savefig('F(w).png', format='png', dpi=300)
plt.show()
```

```
iteration : 10000, F(w) = 0.6931471805599322
```

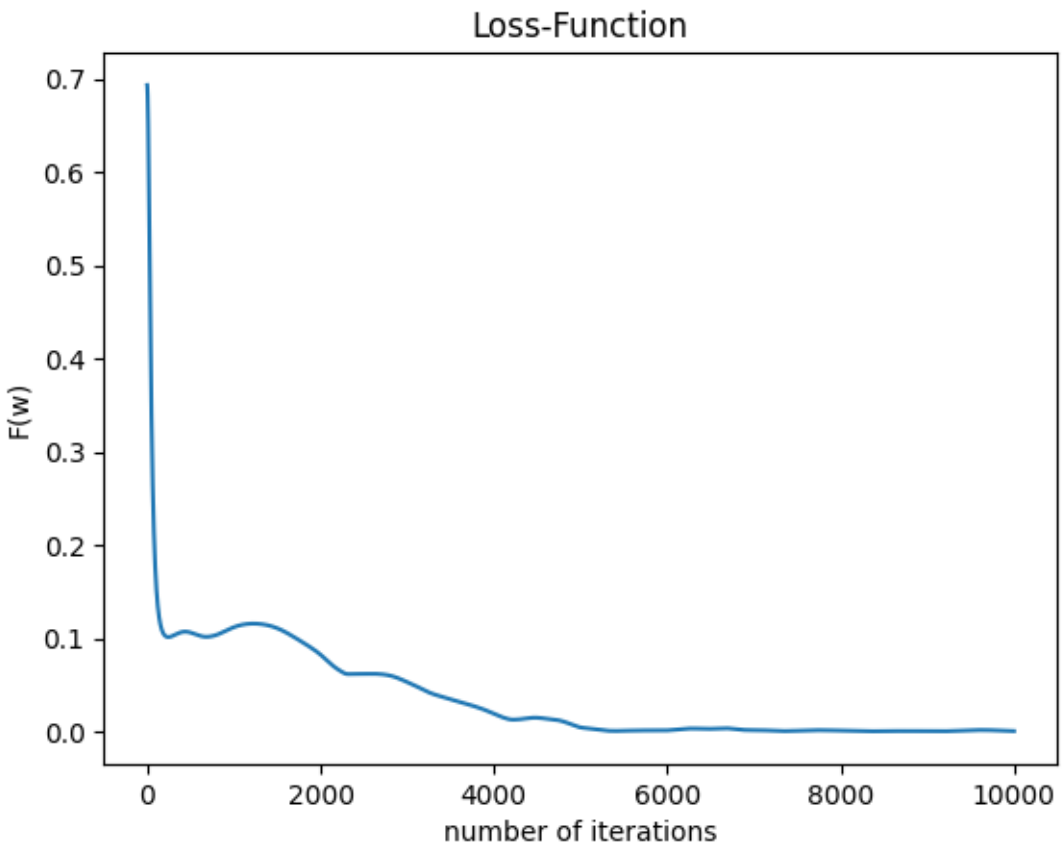


5 Gradient descent with nesterov's acceleration

[36]:
```
# 4 a
import math
lr = 1e-3
iteration_nbr = 10**4
beta = 1
xprev = np.array([0 for _ in range(784)], dtype=np.float64)

weights = np.array([0 for _ in range(784)], dtype=np.float64)

x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
    x_values.append(n)
    y_values.append(function(weights, map))
    x_save = weights
```

```
    weights = weights - lr * optimized_gradient((weights + beta * (weights -␣
 ↪xprev)), map) + beta * (weights - xprev)
    xprev = x_save
printf(weights, map, 10000)

plt.plot(x_values, y_values)
plt.xlabel('number of iterations')
plt.ylabel('F(w)')
plt.title('Loss-Function')
plt.ticklabel_format(style='plain', axis='y')  # Plain formatting for y-axis
#plt.xticks(range(1, iteration_nbr + 1, 500))
plt.savefig('F(w).png', format='png', dpi=300)
plt.show()
```

```
iteration : 0, F(w) = 0.6931471805599322
iteration : 1000, F(w) = 0.11172350057316358
iteration : 2000, F(w) = 0.08227191594481818
iteration : 3000, F(w) = 0.052877920857782625
iteration : 4000, F(w) = 0.018923473819430183
iteration : 5000, F(w) = 0.004191385416169663
iteration : 6000, F(w) = 0.0011126185653139691
iteration : 7000, F(w) = 0.0013404244476149909
iteration : 8000, F(w) = 0.0008098591758488051
iteration : 9000, F(w) = 0.00024442335332544314
iteration : 1000, F(w) = 0.00034703225744883584
```

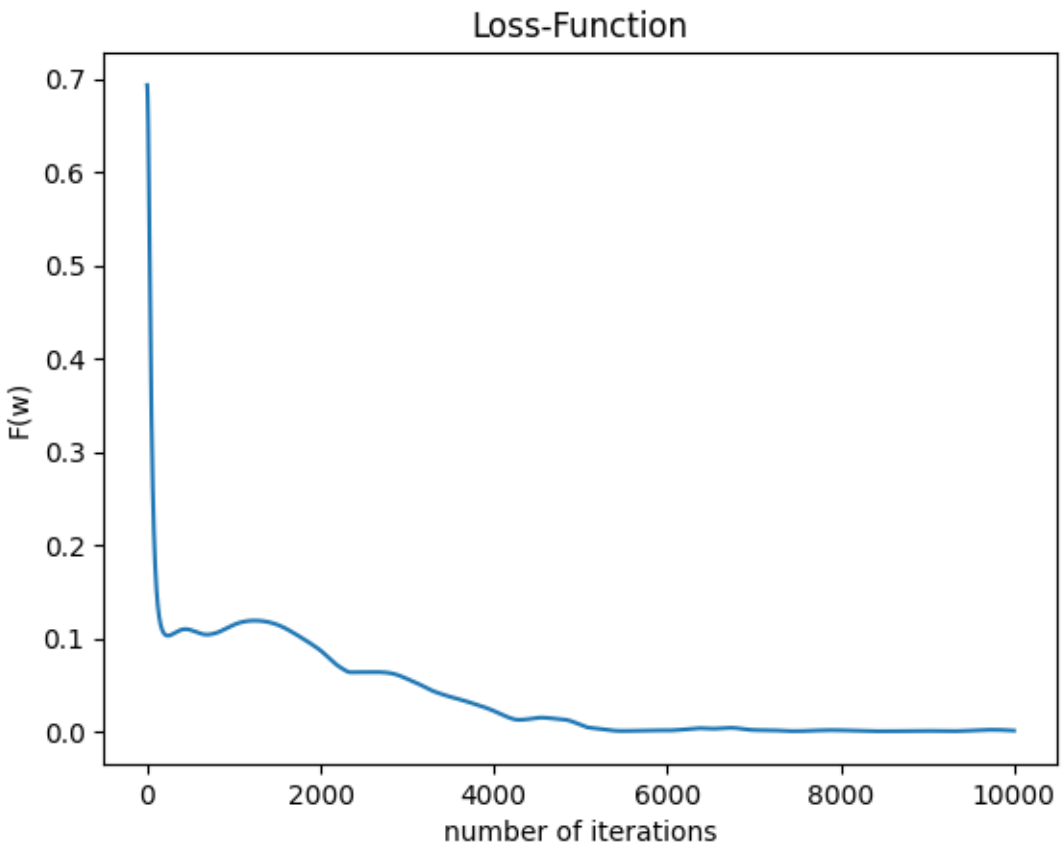


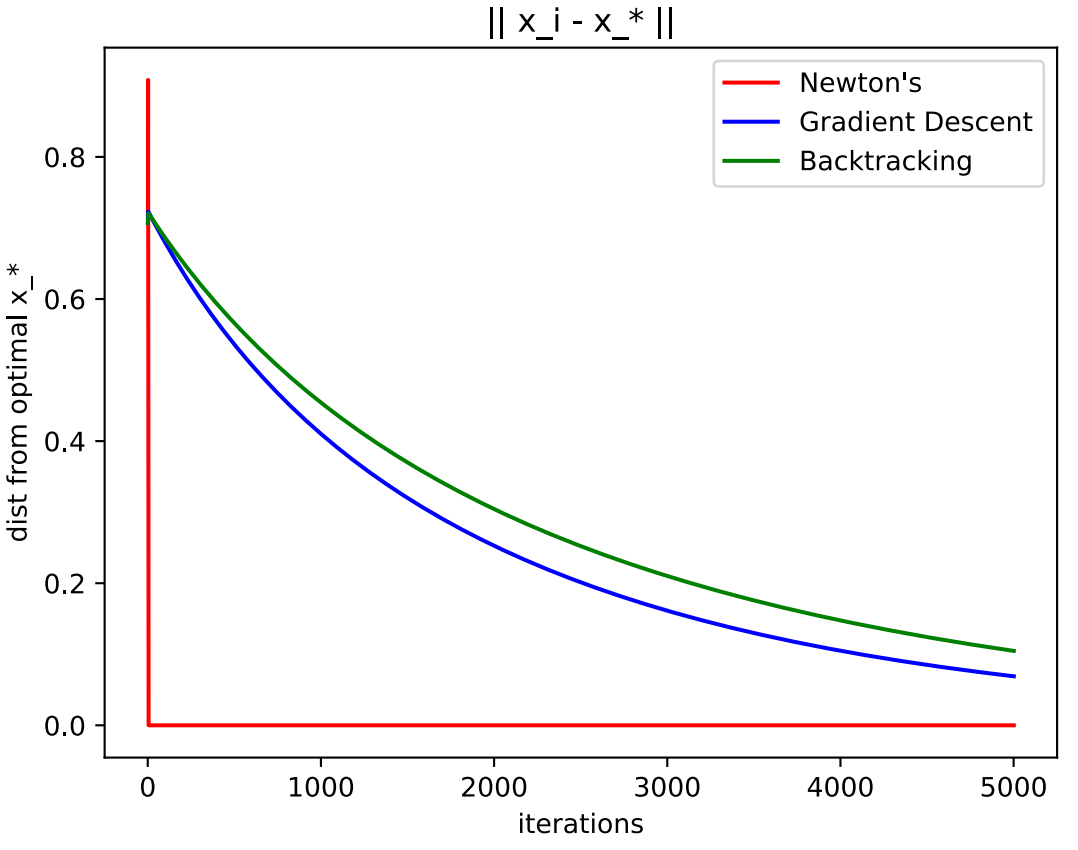
6 Gradient descent with momentum

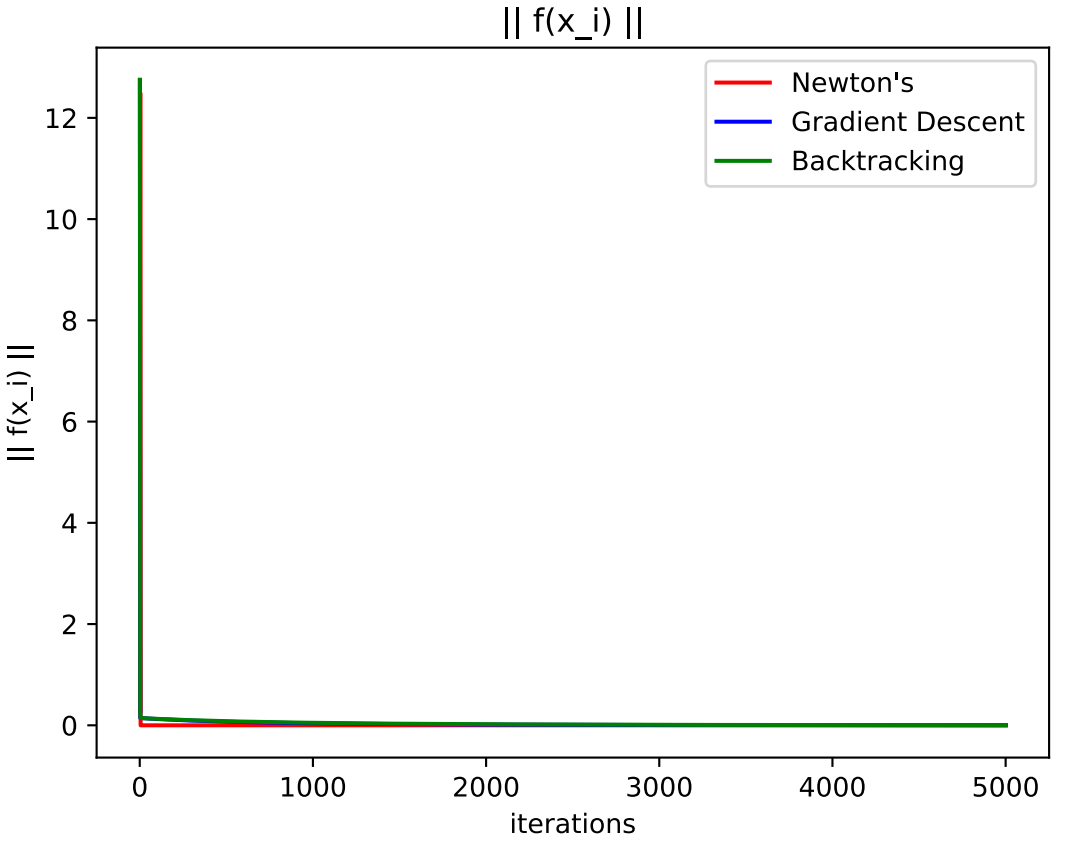
[37]:
```
# 4 a
import math
lr = 1e-3
iteration_nbr = 10**4
beta = 1
xprev = np.array([0 for _ in range(784)], dtype=np.float64)

weights = np.array([0 for _ in range(784)], dtype=np.float64)

x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
    x_values.append(n)
    y_values.append(function(weights, map))
    x_save = weights
```

```
    weights = weights - lr * optimized_gradient(weights, map) + beta * (weights␣
 ↪- xprev)
    xprev = x_save
printf(weights, map, 10000)

plt.plot(x_values, y_values)
plt.xlabel('number of iterations')
plt.ylabel('F(w)')
plt.title('Loss-Function')
#plt.xticks(range(1, iteration_nbr + 1, 500))
plt.savefig('F(w).png', format='png', dpi=300)
plt.show()
```

```
iteration : 0, F(w) = 0.6931471805599322
iteration : 1000, F(w) = 0.11431282483016972
iteration : 2000, F(w) = 0.08673958820295308
iteration : 3000, F(w) = 0.05629395487577995
iteration : 4000, F(w) = 0.02191169896758379
iteration : 5000, F(w) = 0.006880022811634531
iteration : 6000, F(w) = 0.0010065529945642447
iteration : 7000, F(w) = 0.0013990424544088028
iteration : 8000, F(w) = 0.0011878032433038268
iteration : 9000, F(w) = 0.0005675642487008104
iteration : 1000, F(w) = 0.0008088792810672413
```


Loss-Function
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
F(w)
0
2000
4000
6000
8000
10000
number of iterations


|| x_i - x_* ||
Newton's
Gradient Descent
Backtracking
dist from optimal x_*
0.0
0.2
0.4
0.6
0.8
0
1000
2000
3000
4000
5000
iterations


|| f(x_i) ||
Newton's
Gradient Descent
Backtracking
12
10
8
6
4
2
0
|| f(x_i) ||
0
1000
2000
3000
4000
5000
iterations