hw3.5

November 6, 2024

1 Overview: HW3 - Question 4

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

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

For this problem, I would suggest using functions to prepare the dataset, run gradient descent, and return classification error. By doing this, you only have to write the code one time and just use the functions to return results for part (4c).

2 Loading MNIST Data

In this section, you will learn to load MNIST data. If you do not have tensorflow available on your jupyter notebook, uncomment the next cell, run it, restart the kernel, and comment the next cell once more.

[1]:
```
#!pip3 install sklearn
!pip3 install scikit-learn
```

```
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) (2.0.1)
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)
```

[2]:
```
# import statements
import pandas as pd
import numpy as np
```

```
from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
```

[4]:
```
!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
```

```
/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(

zsh:1: no such file or directory: /Applications/Python 3.12.4/Install
Certificates.command
```

[5]:
```
# 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)
```

```
X shape: (70000, 784) y shape: (70000,)
```

[7]:
```
# 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 )
```

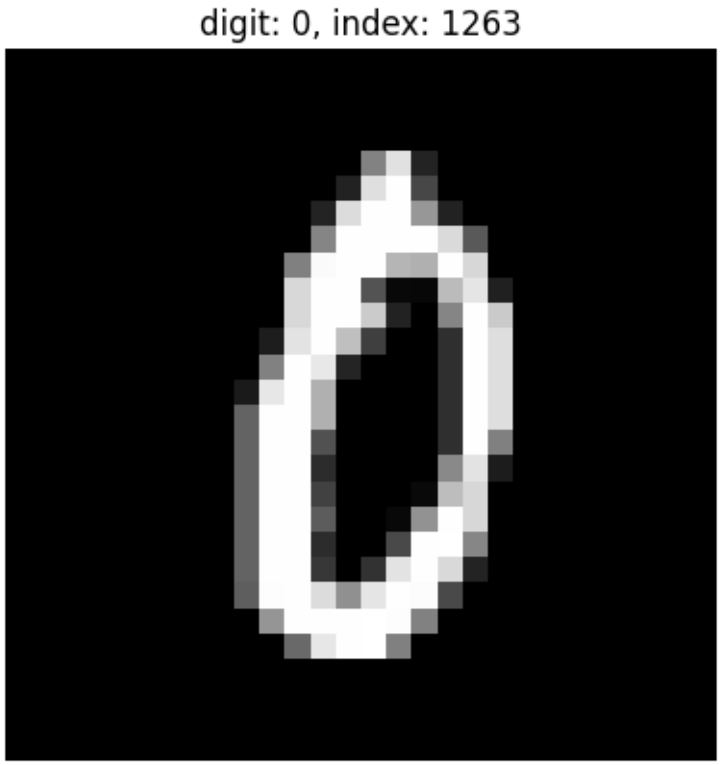
```
Shape of data with label 0 : (6903, 28, 28)
Shape of data with label 1 : (7877, 28, 28)
Shape of data with label 2 : (6990, 28, 28)
Shape of data with label 3 : (7141, 28, 28)
Shape of data with label 4 : (6824, 28, 28)
Shape of data with label 5 : (6313, 28, 28)
Shape of data with label 6 : (6876, 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)
```

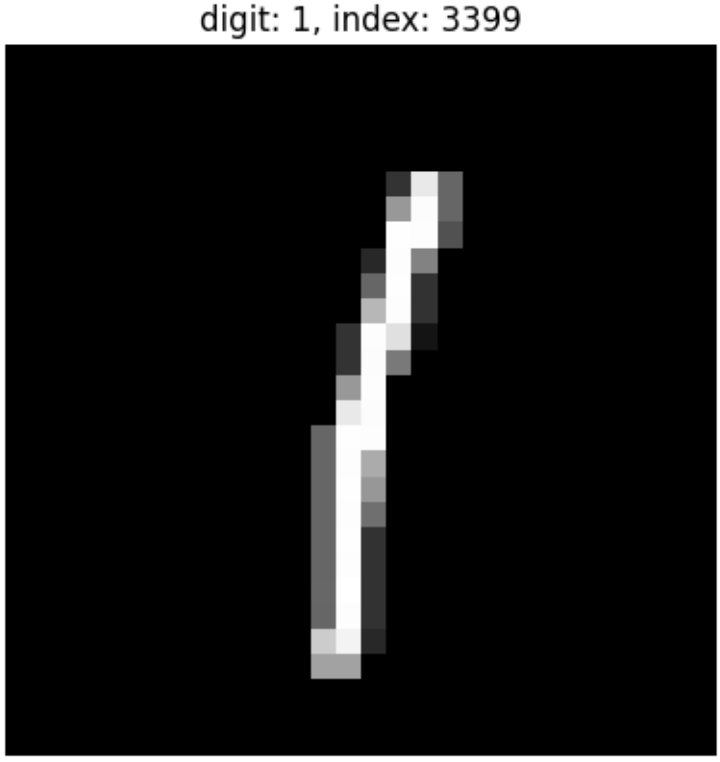
[8]:
```
# this cell would stack 100 examples from each class together
# this cell also ensures that each pixel is a flot between 0 and 1 instead of␣
 ↪an int between 0 and 255
data = []
for i in range(10):
    flattened_images = digits[i][:100].reshape(100,-1)
    data.append(flattened_images)

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

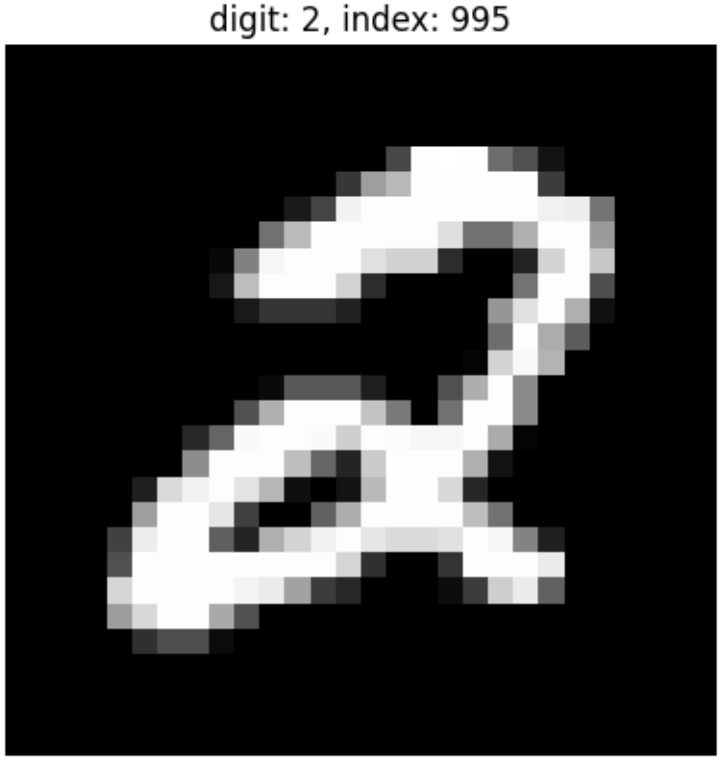
3 (4a) Plotting

Display one randomly selected image from your training data for each digit class. Provide the index number for each image.

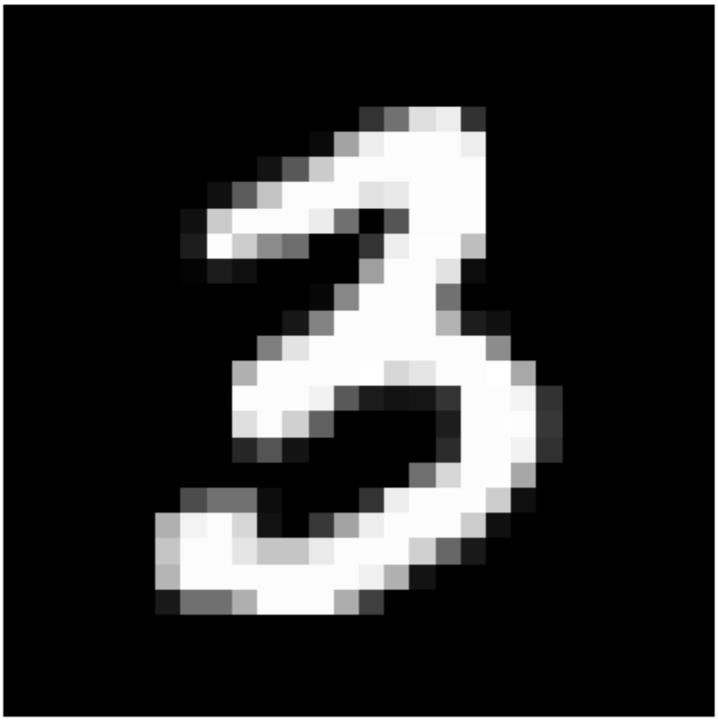
[108]:
```
import random
import matplotlib.pyplot as plt
for i in range(10):
    image = digits[i][image_index]
    image_index = random.randint(0, len(digits[digit_to_display]))
    plt.imshow(image, cmap='gray')
    plt.title(f"digit: {i}, index: {image_index}")
    plt.axis('off')
    plt.show()
```


digit: 0, index: 1263

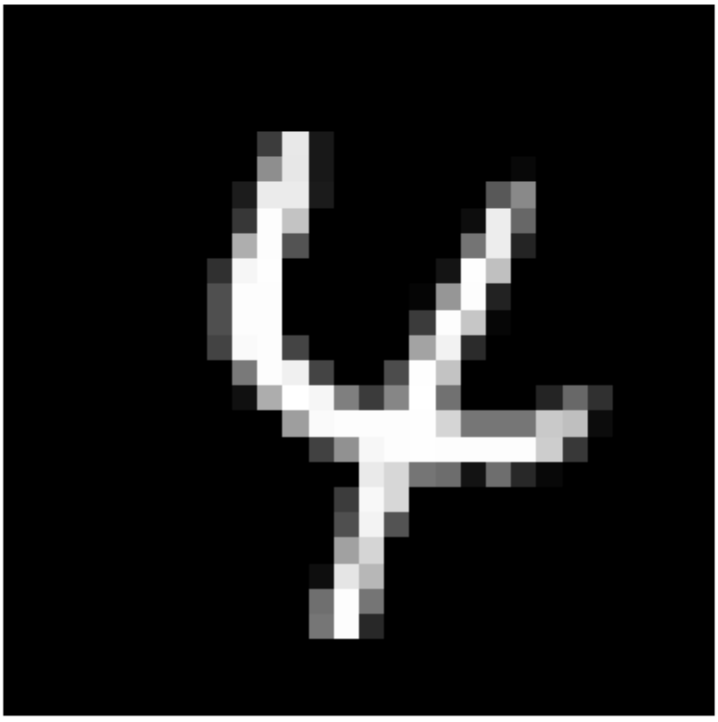

digit: 1, index: 3399

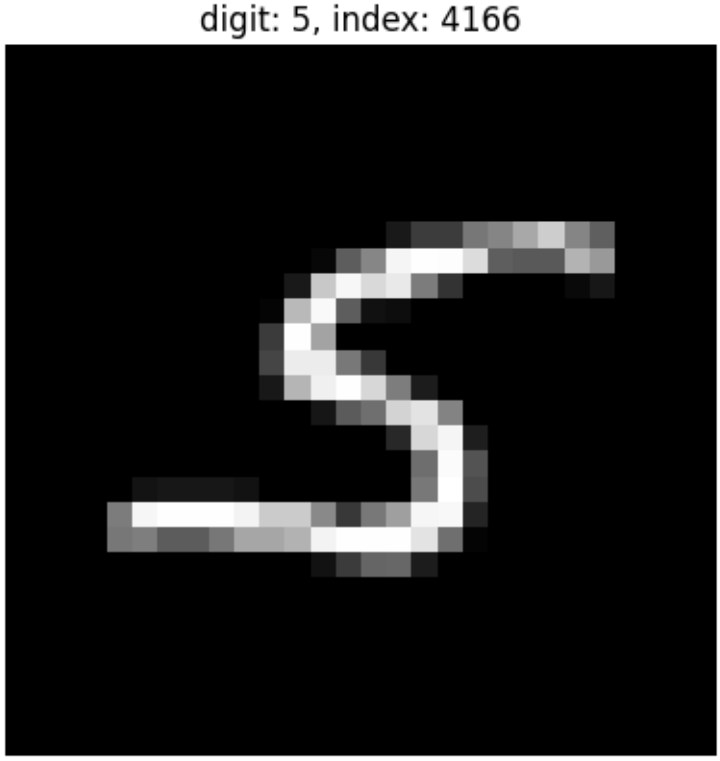

digit: 2, index: 995

digit: 3, index: 15

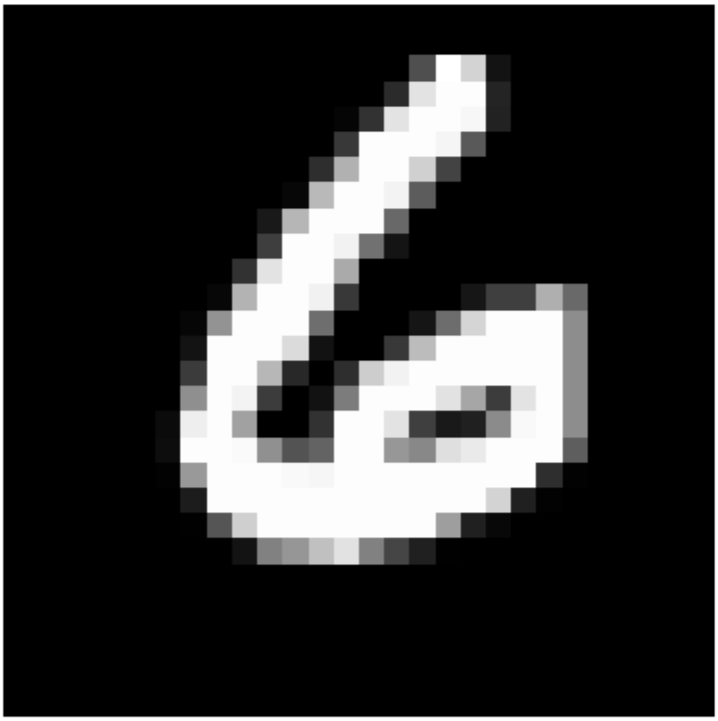


digit: 4, index: 4168

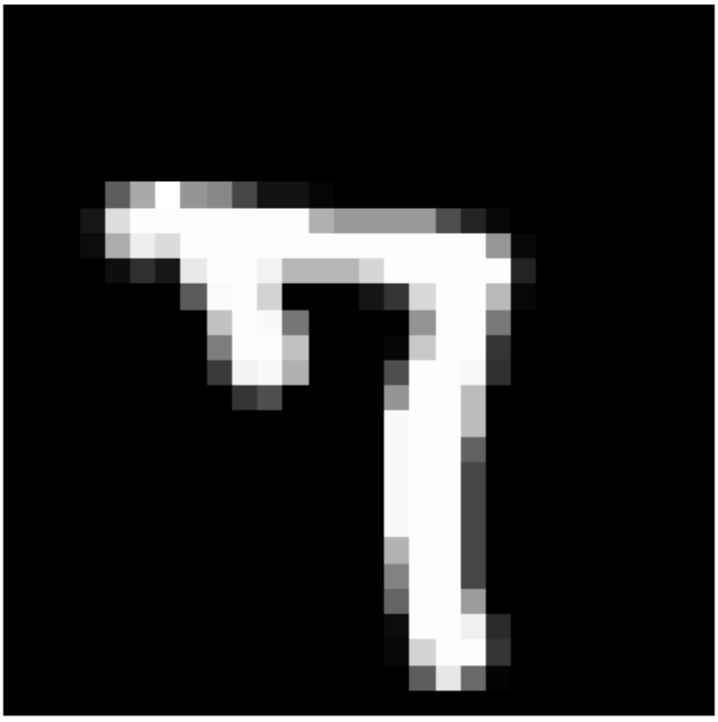


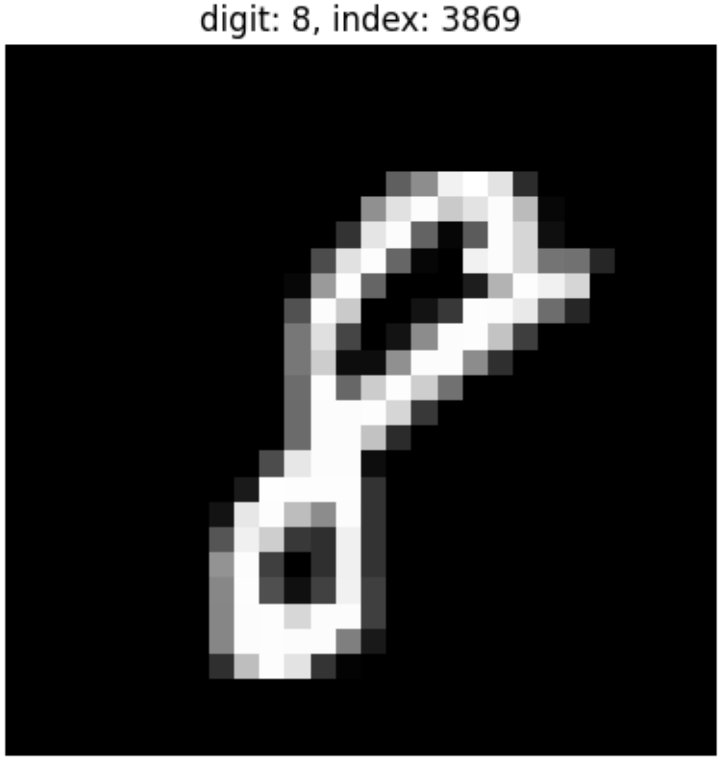

digit: 5, index: 4166

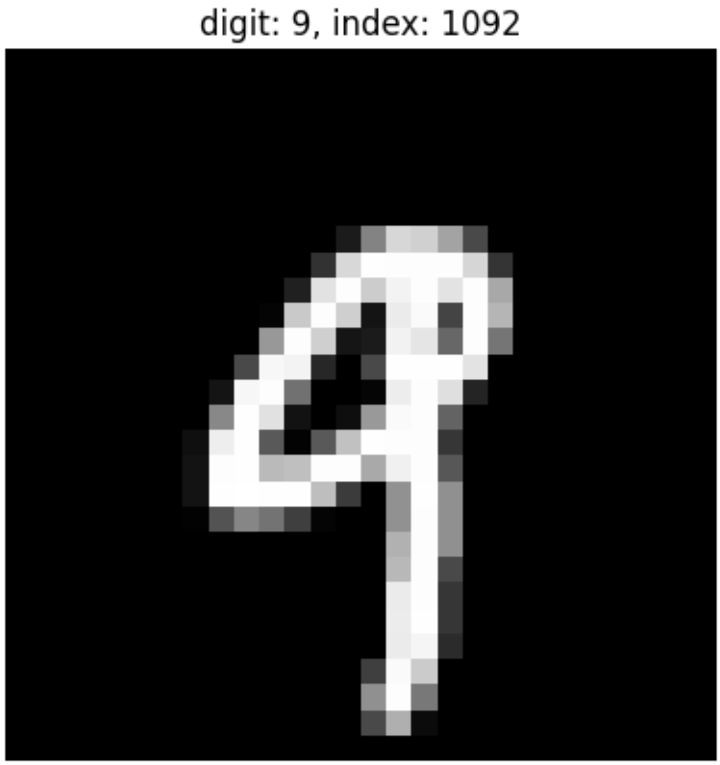
digit: 6, index: 4685



digit: 7, index: 3006




digit: 8, index: 3869



4 (4b) Label data

Select the first 500 examples of 0's and 1's for this example, those will form the training data $(x_i, y_i) \in \mathbb{R}^{784} \times \{-1, 1\}, i = 1, ..., 1000$. Assign label $y_i = 1$ for 1s and $y_i = -1$ for 0s. Also, renormalize your x_i so that the pixel values are floats between 0 and 1, instead of ints from 0 to 255. You can do this by augmenting the code given above for stacking data from different classes.

[ ]:

[128]:
```
# 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
map = {((-1)** (i+1)):[digit.flatten() for digit in digits[i][:nbr_examples]]␣
 ↪for i in range(2)}
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
```

```
51
```

5 (4c) Running Gradient Descent

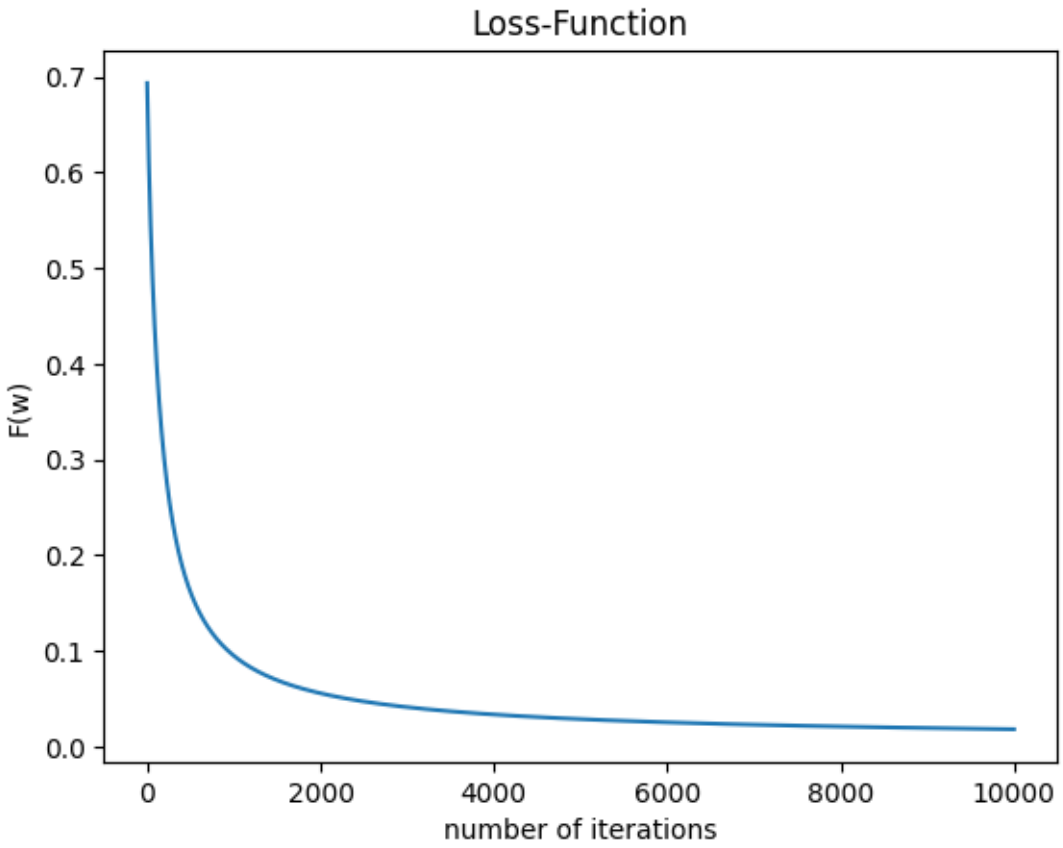
Implement and run a Gradient Descent algorithm, with step-size $\mu = 10^{-4}$, to optimize the function above associated with this setup. You should run your algorithm for at least $T = 10,000$ iterations, but if your computer can handle it try $T = 100,000$ or until a reasonable stopping criterion is satisfied. Provide a plot showing the value of $F(w)$ at each iteration. Also, feel free to adjust μ to be larger / smaller if the plot does not match your expectations.

[129]:
```
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)}")
```

[130]:
```
# implement gradient descent here
import math
lr = 1e-3
iteration_nbr = 10**4

weights = np.array([random.uniform(0,1e-5) for _ in range(784)])

x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
    x_values.append(n)
    y_values.append(function(weights, map))
    weights -= lr * optimized_gradient(weights, map)

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) = 0.6932376256527795
iteration : 1000, F(w) = 0.09478854808183304
iteration : 2000, F(w) = 0.05555041681974981
iteration : 3000, F(w) = 0.041061183108227844
iteration : 4000, F(w) = 0.0333654444692731
iteration : 5000, F(w) = 0.02852948478766427
iteration : 6000, F(w) = 0.02517760223183678
iteration : 7000, F(w) = 0.022699761954850368
iteration : 8000, F(w) = 0.02078254762200478
iteration : 9000, F(w) = 0.019247792210869523
```



6 (4d) Discussion about gradient descent

Comment on the resulting plot. In particular, does the shape of $F(w)$ suggest you've successfully converged to a local or global minimum? Does it appear you chose a good stopping criteria? Explain whether your answers to these questions are consistent with the theory we discussed in class (and in the notes). Be specific i.e., point to a specific theorem (or theorems) and indicate why it does or does not explain the behavior of the algorithm. Would the theory dictate a different choice of μ than the one we used?

6.1 Answer:

- I think the weight vector converges to a global minimum.

I know from homework 2 that finding the optimal w-vector is a convex problem since the F is a convex function and w ^784 is a convex set. This implies that if i find a local minimum for the function F it is also a global minimum.

6.1.1 decreasing gradient

As the w gets closer to a local minimizer the gradient will decrease, this will then make updates to the w-vector smaller since they are dependent on the constant learnign-rate we decide in the

beginning, and the gradient which obviously decreases as we get closer to a local minimizer. A smaller update to the w-vector means our functino value F(w) will change less after each iteration. It is possible to see in my plot how the function value F(w) converges to a small number. I also printed the function value after every 1000 iteration and got the following

iteration : 0, F(w) = 0.6932376256527795

iteration : 1000, F(w) = 0.09478854808183304

iteration : 2000, F(w) = 0.05555041681974981

iteration : 3000, F(w) = 0.041061183108227844

iteration : 4000, F(w) = 0.0333654444692731

iteration : 5000, F(w) = 0.02852948478766427

iteration : 6000, F(w) = 0.02517760223183678

iteration : 7000, F(w) = 0.022699761954850368

iteration : 8000, F(w) = 0.02078254762200478

iteration : 9000, F(w) = 0.019247792210869523

(forgot to add final print statement after the 10k iteration)…

it is clear to see here how to function value changes a lot in the beginning, but then as the w-vector converges to the w-optimum the function value doesn't change as much after each iteration anymore.

6.2 Answer 3d)

We dont know how people have labeled the pictures, If someone has just put random labels on the pictures it will be impossible to find a good weight vector and a low loss. It is therefore not possible to look at the final value of our Loss function F(w) to see if we chose a good stoping criteria (since it might be impossible to get a low loss).

What we can do is look at the norm of the gradient. Since we know the function to be convex, we know that it will have a global minimizer, and we know this will be the only place where the gradient is zero, so if we find that our gradient is really close to zero we can stop as we then know we're close to the global mimimizer. the norm of F(w) was small after I ran GD and i therefore know it was a relativly good stopping criteria to stop after 10k iterations.

Worth noting is that I believe the fuctino F(w) to be L-smooth, this means that while we found a good weight, the w(t) converges to w(optimum) as t -> inf. This means that the longer i would have run the GD the lower the F(w(t)) would have been. The step sizer for a L-smooth function should be $0 < \mu <= (1/L)$.

6.2.1 Note:

I chose the learning_rate=1e-3 instead of 1e-4 since it converged faster with lr=1e-3.

Now, use the w you found from part (a) to classify the first 500 *test* data points associated to each of the 0 and 1 handwritten digits. Recall that you need to use the function $y = \text{sign}(w^T x)$ to classify. What was the classification error rate associated with the two digits on the test data (this

should be a number between 0 and 1)? What was it on the training data? Does this relationship make sense?

6.3 Answer 3e)

- Model accuracy on Training-data: 0.9980
- Model accuracy on Test-data: 0.9973

This is the relationship between training-data accuracy and test-data accuracy i expected. Since during our gradient descent we have tweaked our weight-vector to best fit the training data, our only goal when finding the optimal weight vector was to fit the training-data. It is natural that the weight vector still works well on similar data since the test data was also drawn from the same underlying distribution of handwritten digits as the training data. The difference between accuracies suggest that while our model succeeds to generalize to unseen data, it may still encounter minor variations that it doesn't know how to account for.

[164]:
```
# Classify and return the classification error
def calc_accuracy(inputs:{int:np.array}, weights:np.array) -> {int:float}:
    count = 0
    N = len(inputs[-1])
    for target in [0,1]:
        for i in range(N):
            inputs_index = 2*target - 1
            x = inputs[inputs_index][i]
            count += 0.5 + 0.5 * inputs_index * float(np.sign(np.
 ↪dot(weights,x)))
    return count / (N*2)

def print_accuracy(inputs:{int:np.array}, weights:np.array, data_type:str):
    print(f"Model accuracy on {data_type}-data: {calc_accuracy(inputs,␣
 ↪weights)}\n")
```

[165]:
```
training_data = map
test_data = {((-1)** (i+1)):[digit.flatten() for digit in␣
 ↪digits[i][nbr_examples:]] for i in range(2)}

print_accuracy(training_data, weights, "Training")
print_accuracy(test_data, weights, "Test")
```

```
Model accuracy on Training-data: 0.998

Model accuracy on Test-data: 0.9973449945338123
```