Optimization via Gradient Descent

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
#np.seterr(all='raise')
```

. For each of the functions above, run the GD method with and without the backtracking, trying different values for the step size α > 0 when you are not using backtracking. Observe the different behavior of GD.

. To help visualization, it is convenient to plot the error vector that contains the ||∇f(xk)||2, to check that it goes to zero. Compare the convergence speed (in terms of the number of iterations k) in the different cases.

. For each of the points above, fix x0= (0, 0, . . . , 0)T, kmax = 100, while choose your values for tolf and tolx. It is recommended to also plot the error ||xk − x∗||2 varying k when the true x∗ is available.

. Only for the non-convex function defined in 5, plot it in the interval [−3, 3] and test the convergence point of GD with different values of x0 and different step-sizes. Observe when the convergence point is the global minimum and when it stops on a local minimum or maximum.

. Hard (optional): For the functions 1 and 2, plot the contour around the minimum and the path defined by the iterations (following the example seen during the lesson). See plt.contour to do that.

In [ ]:
```
def backtracking(f, grad_f, x):
    alpha = 1
    c = 0.8
    tau = 0.25
    # f(x) - c * alpha * np.linalg.norm(grad_f(x), 2) ** 2 represents the estimate of th
    while f(x - alpha * grad_f(x)) > f(x) - c * alpha * np.linalg.norm(grad_f(x), 2) **
        alpha = tau * alpha

        if alpha < 1e-3: #lower bound for the value of alpha
            break
    return alpha
```

In [ ]:
```
def gd(fn, grad_fn, x0, k_max, tol_f, tol_x, alpha=None):
    curr_x, prev_x = x0, np.inf
    curr_k = 0
    grad_x0, curr_grad = grad_fn(x0), grad_fn(curr_x)
    history_x = [x0]
    history_f = [fn(x0)]
    history_grad = [grad_x0]
    history_err = [np.linalg.norm(grad_x0, 2)]
    use_backtracking = alpha is None # bool that is true if alpha is None

    while (curr_k < k_max and
            not (np.linalg.norm(curr_grad, 2) < tol_f*np.linalg.norm(grad_x0, 2)) and #h
            not (np.linalg.norm(curr_x - prev_x, 2) < tol_x)): # how much the current an
        if use_backtracking:
            alpha = backtracking(fn, grad_fn, curr_x)
        prev_x = curr_x
        curr_x = curr_x - alpha*grad_fn(curr_x)

        curr_grad = grad_fn(curr_x)
        curr_k += 1

        history_x.append(curr_x)
```

```
        history_f.append(fn(curr_x))
        history_grad.append(curr_grad)
        history_err.append(np.linalg.norm(curr_grad, 2))

    return history_x, curr_k, history_f, history_grad, history_err
```

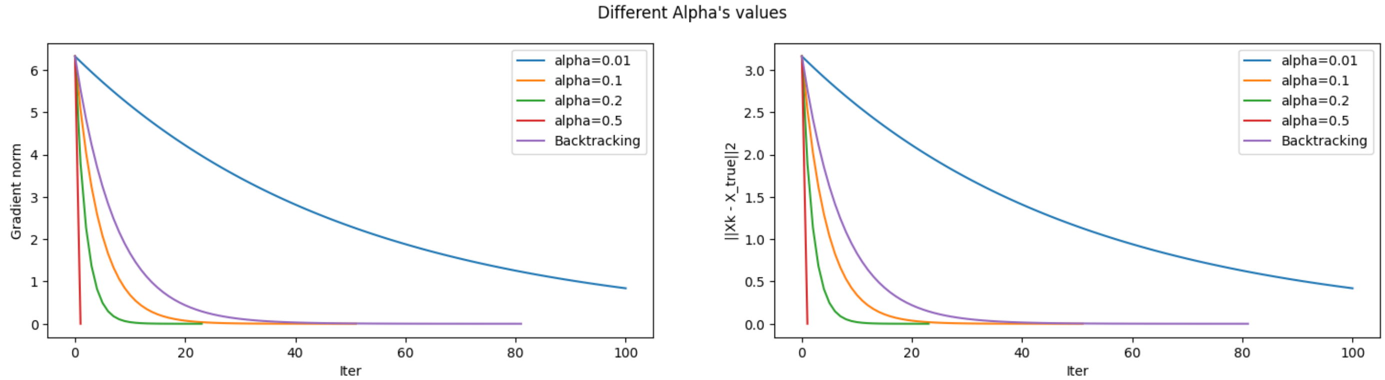
In [ ]:
```
def testVaryingAlpha(fn, grad_fn, input_size, x_true, k_max=100, try_alpha=[0.01, 0.1, 0
    plt.figure(figsize=(18, 4))
    plt.suptitle("Different Alpha's values")
    ax1 = plt.subplot(1, 2, 1)
    ax2 = plt.subplot(1, 2, 2)
    for alpha in try_alpha:
        try:
            history_x, curr_k, history_f, history_grad, history_err = gd(fn, grad_fn, np
            label_alpha = "Backtracking" if alpha is None else f"alpha={alpha}"
            ax1.plot(range(0, len(history_err)), history_err, label=label_alpha) # plot
            ax2.plot(range(0, len(history_x)), [np.linalg.norm(x - x_true, 2) for x in h
        except Exception as e:
            print(f"alpha={alpha}: {e}")
    ax1.set_xlabel("Iter")
    ax1.set_ylabel("Gradient norm")
    ax1.legend()
    ax2.set_xlabel("Iter")
    ax2.set_ylabel("||Xk - X_true||2")
    ax2.legend()
    plt.show()

#just for function4
def testVaryingLambda(fn, input_size, x_true, n=5, k_max=100, try_lambda=[0, 0.1, 0.25,
    plt.figure(figsize=(18, 4))
    plt.suptitle(f"Lambda's values (alpha defined using backtracking)")
    ax1 = plt.subplot(1, 2, 1)
    ax2 = plt.subplot(1, 2, 2)
    for lamb in try_lambda:
        f, grad_f, x_true, input_size = fn(n=n, lamb=lamb)
        history_x, curr_k, history_f, history_grad, history_err = gd(f, grad_f, np.zeros
        ax1.plot(range(0, len(history_err)), history_err, label=f"lamb={lamb}")
        ax2.plot(range(0, len(history_x)), [np.linalg.norm(x - x_true, 2) for x in histo
    ax1.set_xlabel("Iter")
    ax1.set_ylabel(f"Gradient norm")
    ax1.legend()
    ax2.set_xlabel("Iter")
    ax2.set_ylabel("||Xk - X_true||2")
    ax2.legend()
    plt.show()


def showContour(fn, grad_fn, x_true, x0, k_max, tol_f, tol_x, alpha=None, contour_area=(
    history_x, curr_k, history_f, history_grad, history_err = gd(fn, grad_fn, x0, k_max,
    x = np.linspace(contour_area[0], contour_area[1], 1000)
    y = np.linspace(contour_area[0], contour_area[1], 1000)
    x_contour, y_contour = np.meshgrid(x, y)
    z_contour = fn((x_contour, y_contour))

    history_f.sort()
    to_visualize_levels = [history_f[0]] + [history_f[i] for i in range(1, len(history_f
    # to_visualize_levels = history_f
    contour_graph = plt.contour(x, y, z_contour, levels=to_visualize_levels)
    plt.clabel(contour_graph, inline=1, fontsize=10)
    plt.scatter([a[0] for a in history_x], [a[1] for a in history_x], marker="o") #gradi
    plt.scatter(x_true[0], x_true[1], marker="x", c="red", label="Optima") # solution
    plt.legend()
    plt.show()
```

In [ ]:
```
def function1():
```

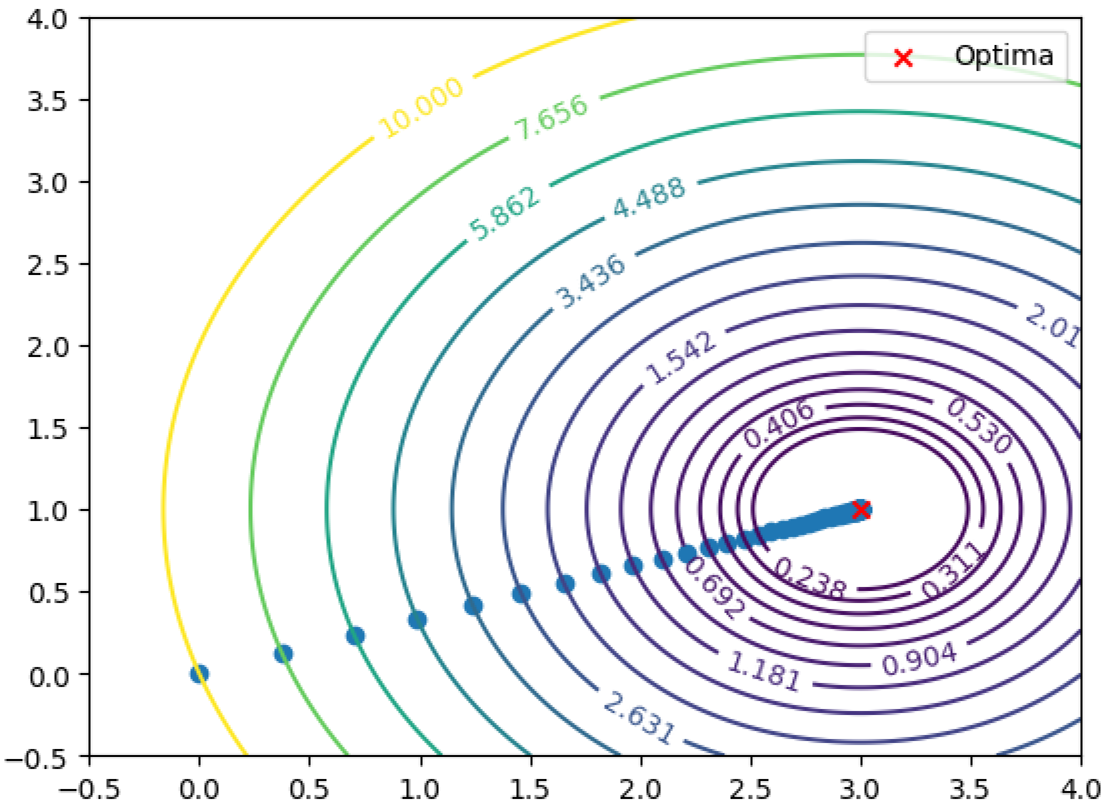
```
    def f(x):
        x1, x2 = x
        return (x1 - 3)**2 + (x2 - 1)**2

    def grad_f(x):
        x1, x2 = x
        return np.array([ 2*(x1-3), 2*(x2-1) ])

    return f, grad_f, np.array([3, 1]), 2


def function2():
    def f(x):
        x1, x2 = x
        return 10*(x1 - 1)**2 + (x2 - 2)**2

    def grad_f(x):
        x1, x2 = x
        return np.array([ 20*(x1-1), 2*(x2-2) ])

    return f, grad_f, np.array([1, 2]), 2


def function3(n=5):
    x_true = np.ones((n,)) # x_true as a vector of ones
    A = np.vander(np.linspace(0, 1, n))
    b = A @ x_true
    def f(x):
        return (1/2) * np.linalg.norm(A@x - b, 2)**2

    def grad_f(x):
        return (x.T @ A.T @ A - b.T @ A)

    return f, grad_f, x_true, n


def function4(n=5, lamb=0.1):
    x_true = np.ones((n,))
    A = np.vander(np.linspace(0, 1, n))
    b = A @ x_true

    def f(x):
        return (1/2)*np.linalg.norm(A@x - b, 2)**2 + (lamb/2)*np.linalg.norm(x)**2

    def grad_f(x):
        return (x.T @ A.T @ A - b.T @ A) + (lamb*x.T)

    return f, grad_f, x_true, n


def function5():
    def f(x):
        return x**4 + x**3 - 2*x**2 - 2*x

    def grad_f(x):
        return 4*x**3 + 3*x**2 - 4*x - 2

    return f, grad_f, None, 1
```

function 1

In [ ]:
```
f, grad_f, x_true, input_size = function1()
testVaryingAlpha(f, grad_f, input_size, x_true)
```

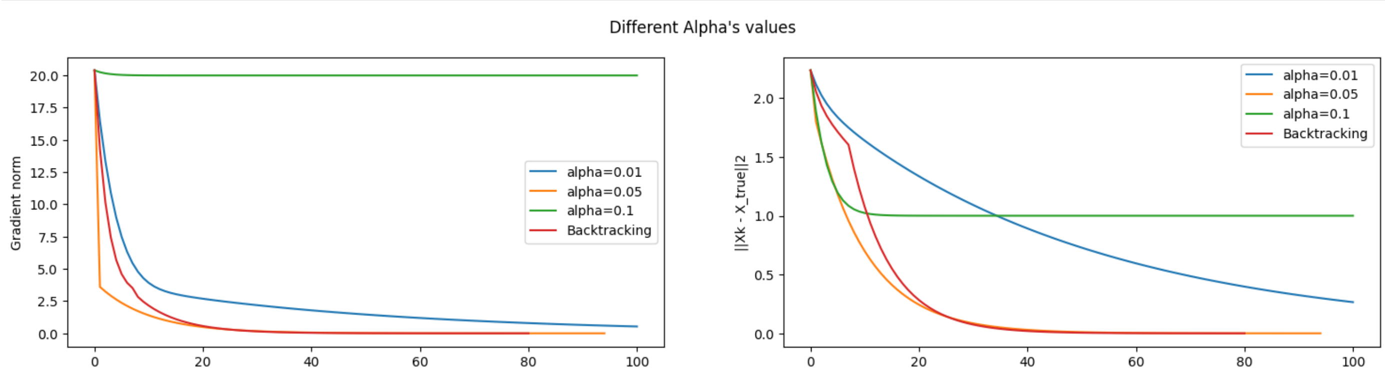


In [ ]:
```
showContour(f, grad_f, x_true,x0 = np.zeros((input_size,)), k_max = 200, tol_f = 1e-5, t
```

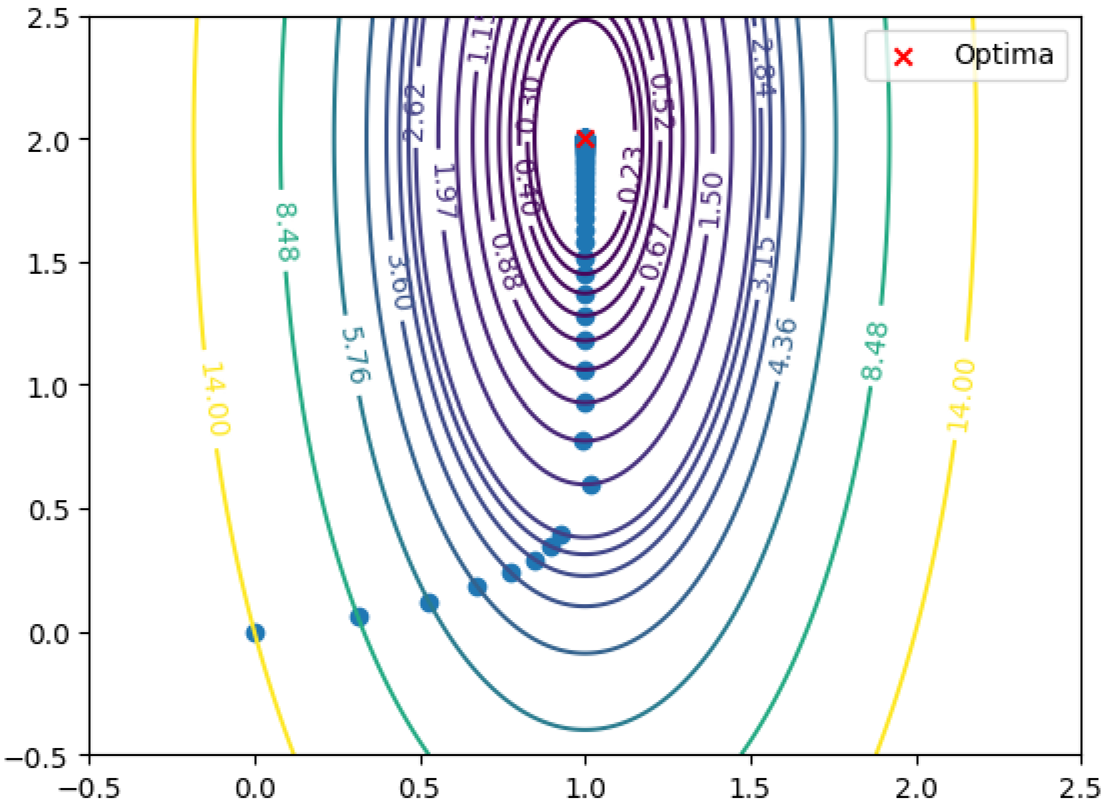


function 2

In [ ]:
```
f, grad_f, x_true, input_size = function2()
testVaryingAlpha(f, grad_f, input_size, x_true, try_alpha=[0.01, 0.05, 0.1, None])
```



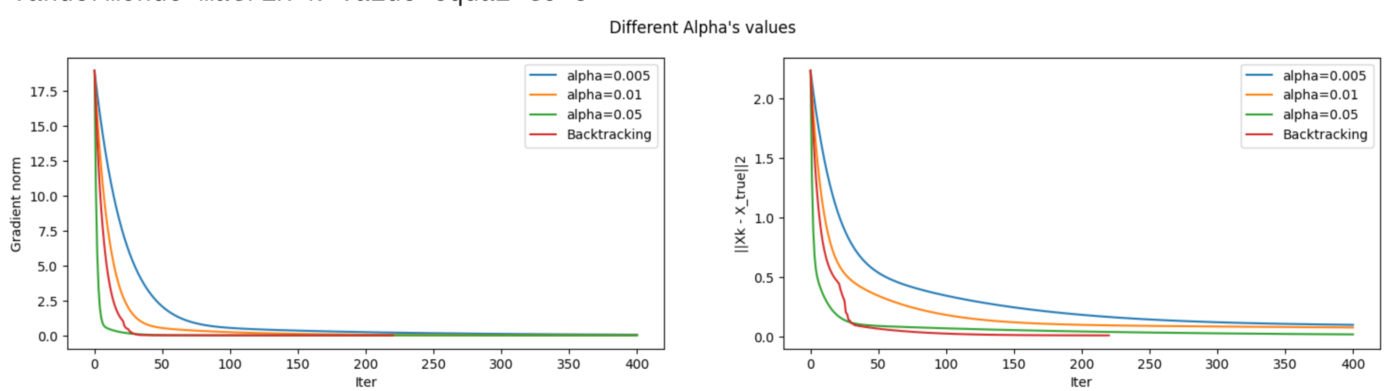
In [ ]:
```
showContour(f, grad_f, x_true,x0 = np.zeros((input_size,)),k_max = 200,tol_f = 1e-5,tol_
```



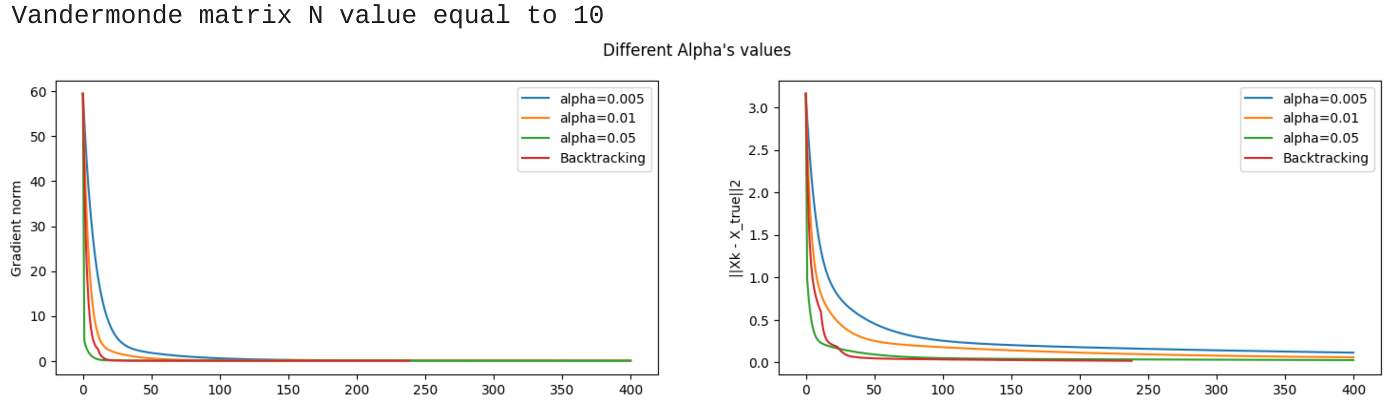
function 3

```
In [ ]: n_values = [5,10,15]
        for n in n_values:
          print(f"Vandermonde matrix N value equal to {n}")
          f, grad_f, x_true, input_size = function3(n)
          testVaryingAlpha(f, grad_f, input_size, x_true,k_max = 400, try_alpha=[0.005, 0.01, 0.
          print('\n')
```

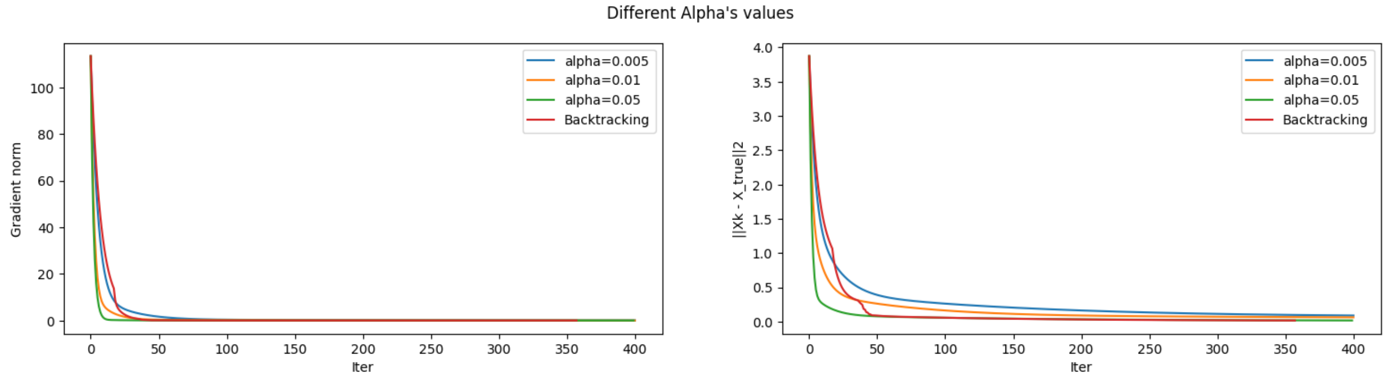
```
Vandermonde matrix N value equal to 5
```




```
Vandermonde matrix N value equal to 10
```



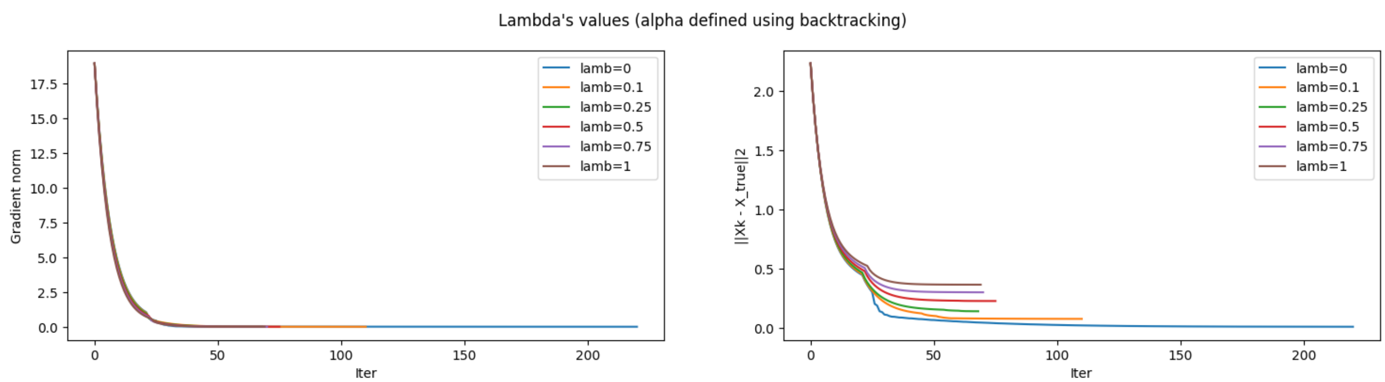
```
Vandermonde matrix N value equal to 15
```

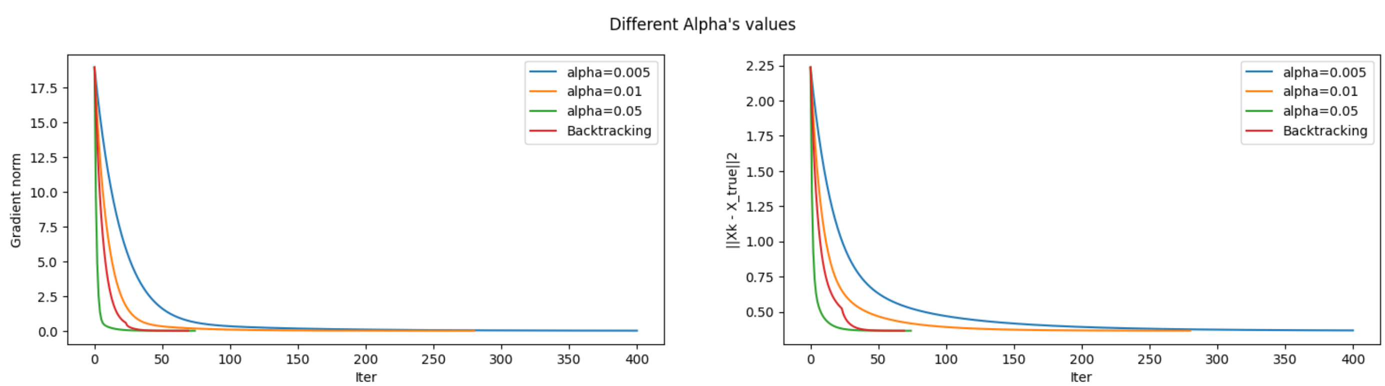


function 4

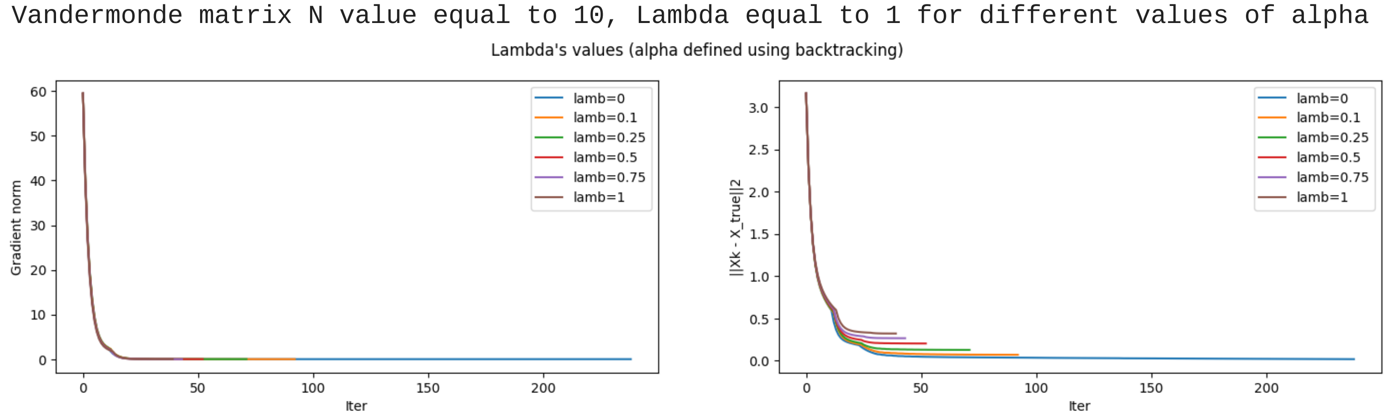
```
In [ ]:
n_values = [5,10,15]
for n in n_values:
  print(f"Vandermonde matrix N value equal to {n}, Lambda equal to 1 for different value
  f, grad_f, x_true, input_size = function4(n, lamb=1)
  testVaryingLambda(function4, input_size, x_true, n=n, k_max = 400)
  testVaryingAlpha(f, grad_f, input_size, x_true, try_alpha=[0.005, 0.01, 0.05, None], k
  print('\n')
```

```
Vandermonde matrix N value equal to 5, Lambda equal to 1 for different values of alpha
```

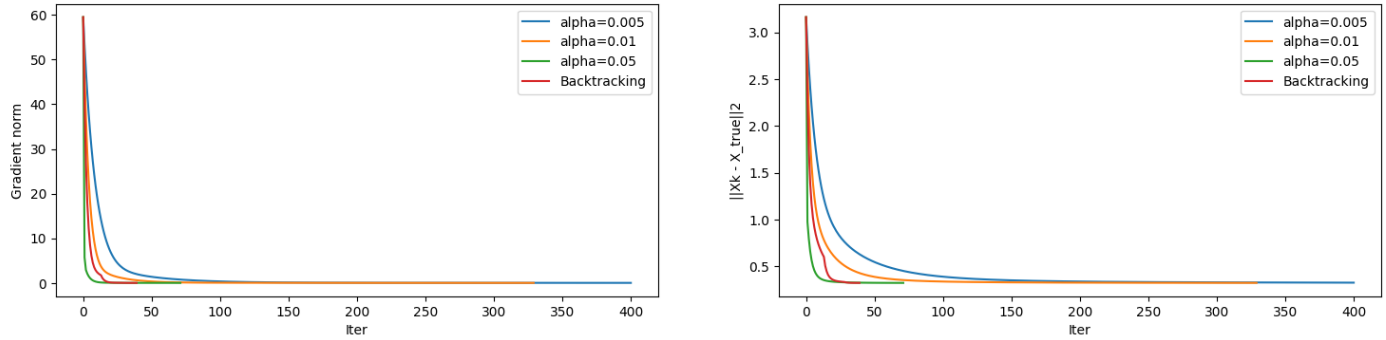





```
Vandermonde matrix N value equal to 10, Lambda equal to 1 for different values of alpha
```

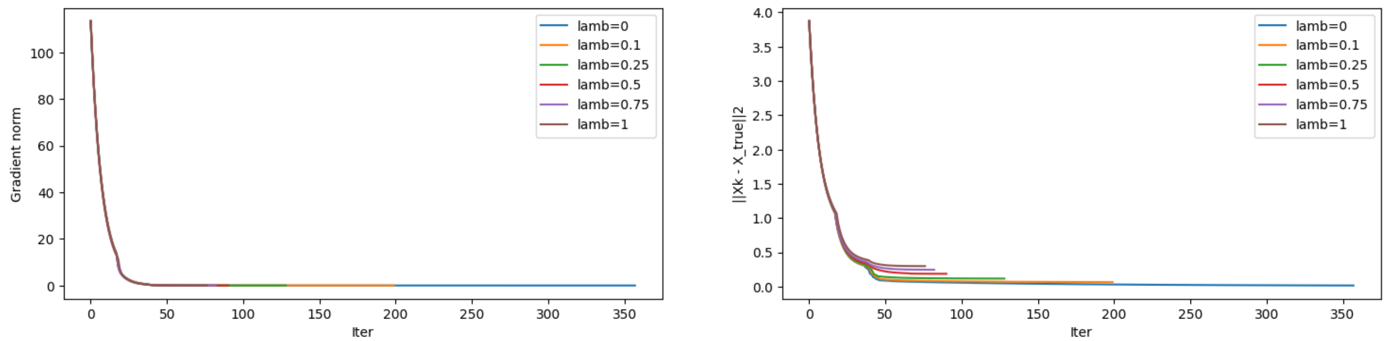


Different Alpha's values

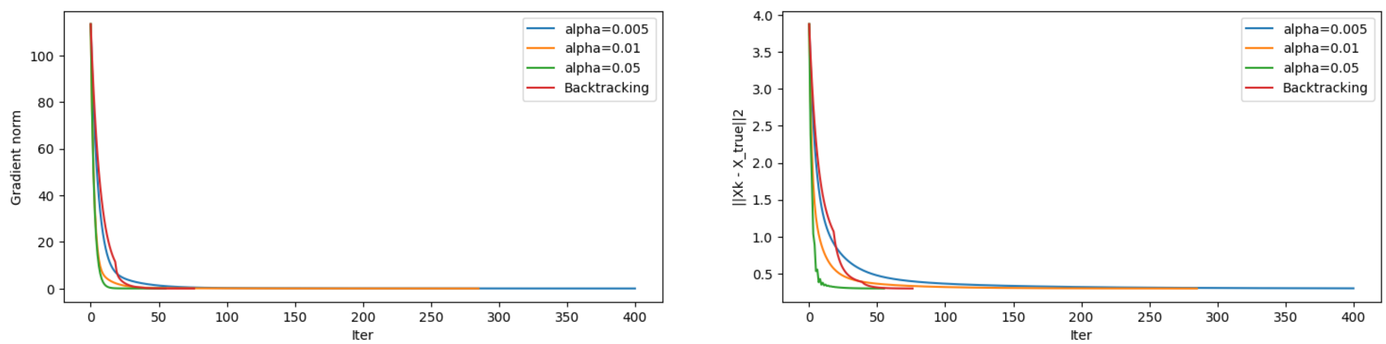



```
Vandermonde matrix N value equal to 15, Lambda equal to 1 for different values of alpha
```

Lambda's values (alpha defined using backtracking)



Different Alpha's values



Out[ ]:

```
'\nwhen lamda is equal to 1, we obtain that the error is equal approximately to 0.5, it
means that we are in the middle between the minimization of lambda*x (=0) and the reachi
ng of the optimal solution (=1)\n'
```

when lamda is equal to 1, we obtain that the error is equal approximately to 0.5, it means that we are in the middle between the minimization of lambda*x (=0) and the reaching of the optimal solution (=1)

function 5

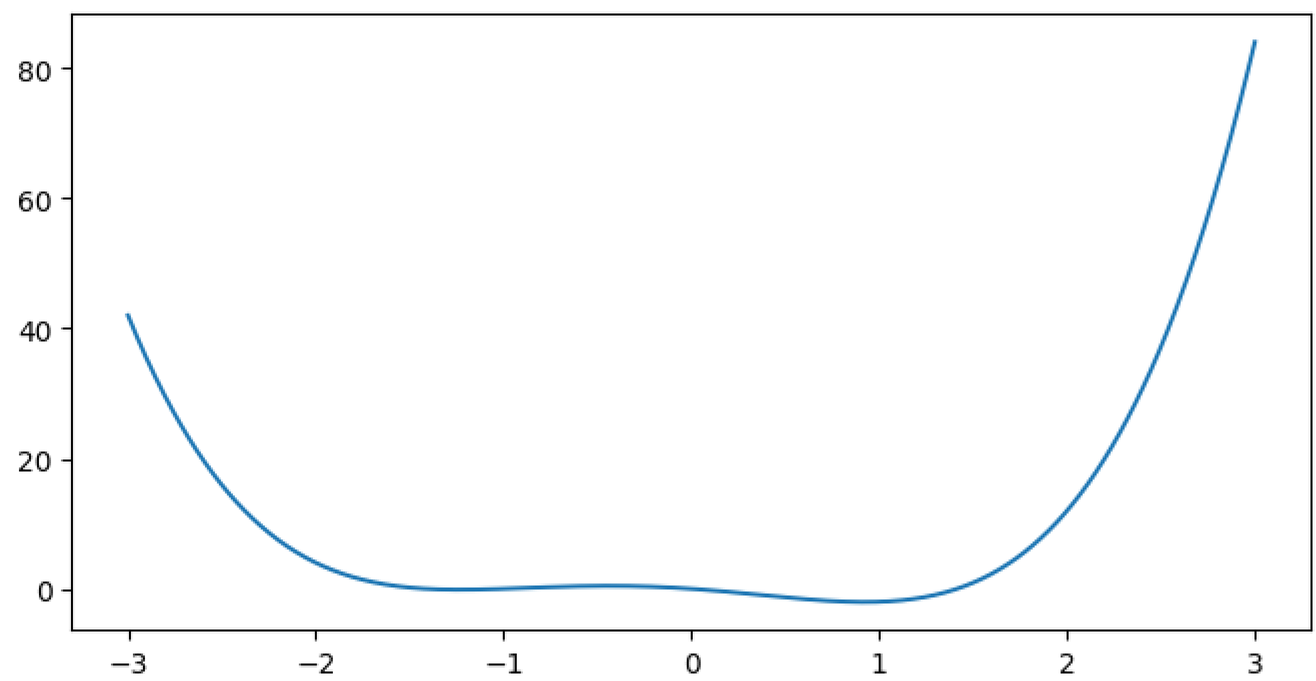
In [ ]:

```python
def plotSteps(fn, grad_fn, x0, try_alpha=[0.01, 0.05, 0.1, None]):
    plt.figure(figsize=(20, 4))
    plt.suptitle(f"x0 = {x0}")
    for i, alpha in enumerate(try_alpha):
        history_x, curr_k, history_f, history_grad, history_err = gd(fn, grad_fn, x0, 50
        plt.subplot(1, len(try_alpha), i+1)
        plt.title(f"alpha={alpha}")
        plt.plot(np.linspace(-3, 3, 1000), fn(np.linspace(-3, 3, 1000)))
        plt.plot(history_x[0], fn(np.array(history_x[0])), "s", color="green", label="x0
        plt.plot(history_x[1:-1], fn(np.array(history_x[1:-1])), ".")
        plt.plot(history_x[-1], fn(history_x[-1]), "X", color="darkred", label="x_pred")
        plt.legend()
    plt.show()
```

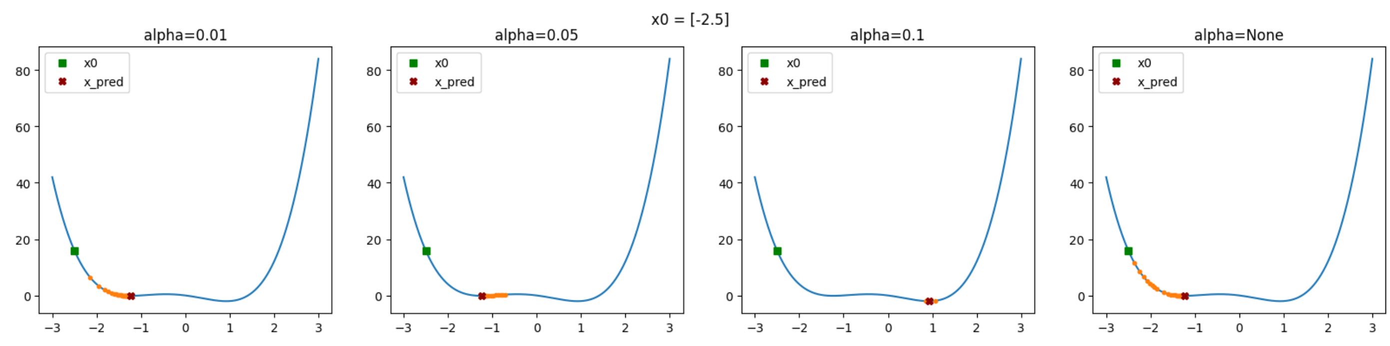
In [ ]:

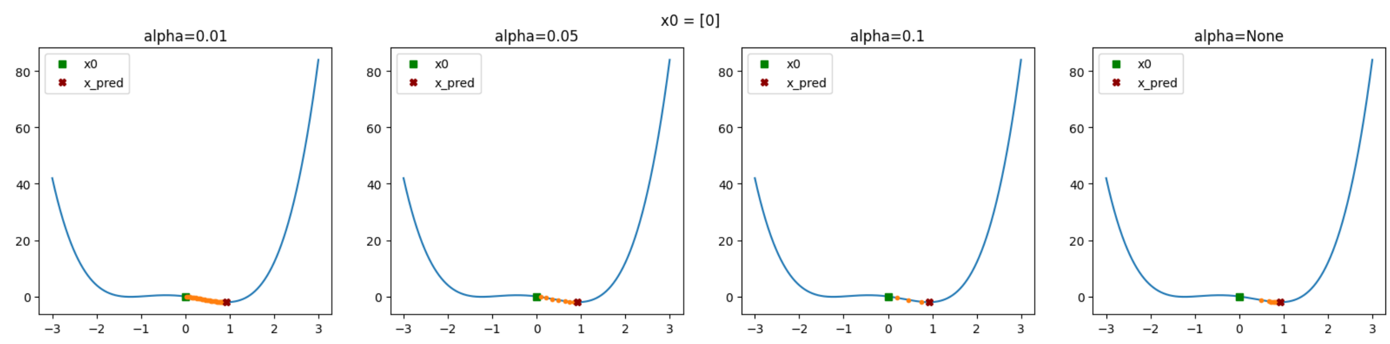
```python
f, grad_f, x_true, in_size = function5()
```

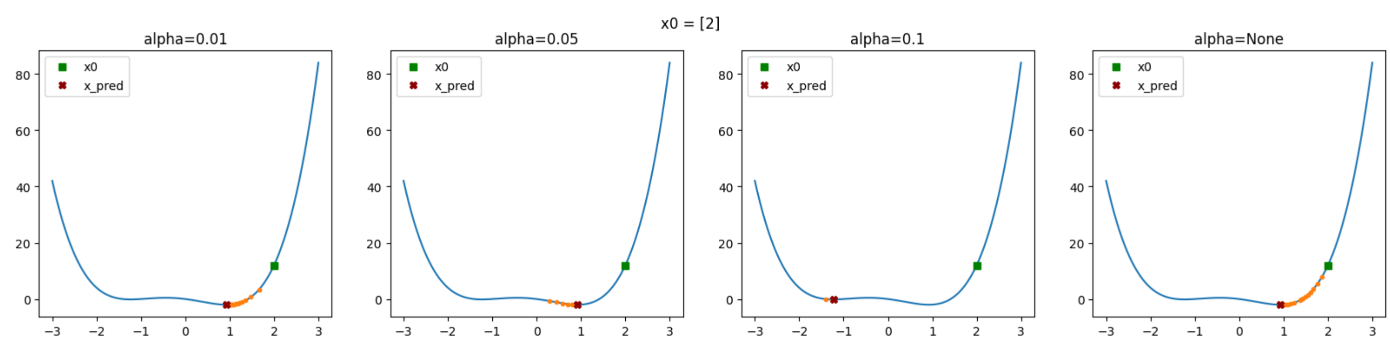
```
x_axis = np.linspace(-3, 3, 1000)
y_axis = f(x_axis)
plt.figure(figsize=(8, 4))
plt.plot(x_axis, y_axis)
plt.show()
```



In [ ]:
```
plotSteps(f, grad_f, np.array([-2.5]))
plotSteps(f, grad_f, np.array([0]))
plotSteps(f, grad_f, np.array([2]))
```







Optimization via Stochastic Gradient Descent

To test the script above, consider the MNIST dataset we used in the previous laboratories, and do the following:

1. From the dataset, select only two digits. It would be great to let the user input the two digits to select.
2. Do the same operation of the previous homework to obtain the training and test set from (X, Y), selecting the Ntrain you prefer.
3. Implement a logistic regression classificator as described in the corresponding post on my website.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/smm/homeworks/')
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import math
from tqdm import tqdm
from itertools import combinations
from utils.optimization import gd
from utils.PCAClassifier import PCAClassifier
from utils.SVDClassifier import SVDClassifier

np.random.seed(42)
```

```
Mounted at /content/drive
```

In [ ]:
```
def train_test_split(X, Y, train_size, random_seed=42):
    idxs = np.arange(0, X.shape[1])
    np.random.default_rng(random_seed).shuffle(idxs)
    X_train = X[:, idxs[:train_size]]
    Y_train = Y[idxs[:train_size]]
    X_test = X[:, idxs[train_size:]]
    Y_test = Y[idxs[train_size:]]
    return X_train, Y_train, X_test, Y_test

def filterDigits(X, Y, digits):
    select_mask = np.isin(Y, digits)
    return X[:, select_mask], Y[select_mask]

# bias
def addCoefficient(A):
    if A.ndim == 1:
        out = np.ones((A.shape[0]+1,))
        out[1:] = A #i have just added one row equal to 1
    else:
        out = np.ones((A.shape[0]+1, A.shape[1]))
        out[1:, :] = A
    return out

# all the label became 0 or 1 in according to the digit
def createDataset(X, Y, digits, train_ratio = 0.75):
    X, Y = filterDigits(X, Y, digits)
    Y[Y == digits[0]] = 0
    Y[Y == digits[1]] = 1
    return train_test_split(X, Y, int(train_ratio*X.shape[1]))
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
file_path = '/content/drive/MyDrive/smm/homeworks/homework2/data/data.csv'
```

```
data = pd.read_csv(file_path)
data = data.to_numpy()

full_X = data[:, 1:].T
full_Y = data[:, 0].T
print(full_X.shape, full_Y.shape)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
(784, 42000) (42000,)
```

In [ ]:

```
#logistic regressor
def sigmoid(x): #activation function
    return 1 / (1 + np.exp(-x))

def f(w, x):
    return sigmoid(x.T @ w) # classifier

def mse(fn, w, X, y):
    return (1/2) * np.linalg.norm(fn(w, X) - y, 2)**2

def mse_grad(fn, w, X, y):
    N = X.shape[1]
    fn_X = fn(w, X)
    return ( X @ (fn_X * (1-fn_X) * (fn_X - y)) )

def loss(w, X, Y):
    N = X.shape[1]
    return (1/N) * mse(f, w, X, Y)

def grad_loss(w, X, Y):
    N = X.shape[1]
    return (1/N) * mse_grad(f, w, X, Y)
```

In [ ]:

```
def sgd(loss, grad_loss, w0, data, batch_size, n_epochs, lr, random_seed=42):
    X, y = data
    data_size = X.shape[1]
    curr_w = w0
    history_w = [w0]
    history_loss = [loss(w0, X, y)]
    history_grad = [grad_loss(w0, X, y)]
    history_err = [np.linalg.norm(history_grad[-1], 2)]
    rng = np.random.default_rng(random_seed)

    for _ in range(n_epochs): #for each epoch
        idxs = np.arange(0, data_size)
        rng.shuffle(idxs) #i shuffle the data index to generate different batches

        for i in range(math.ceil(data_size / batch_size)): # for each batch (n batch = n
            batch_idxs = idxs[i*batch_size : (i+1)*batch_size] #current batch ( 0*batch_
            batch_X = X[:, batch_idxs]
            batch_y = y[batch_idxs]

            curr_w = curr_w - lr*grad_loss(curr_w, batch_X, batch_y)

        history_w.append(curr_w)
        history_loss.append(loss(curr_w, X, y))
        history_grad.append(grad_loss(curr_w, X, y))
        history_err.append(np.linalg.norm(history_grad[-1], 2) )

    return history_w, history_loss, history_grad, history_err
```

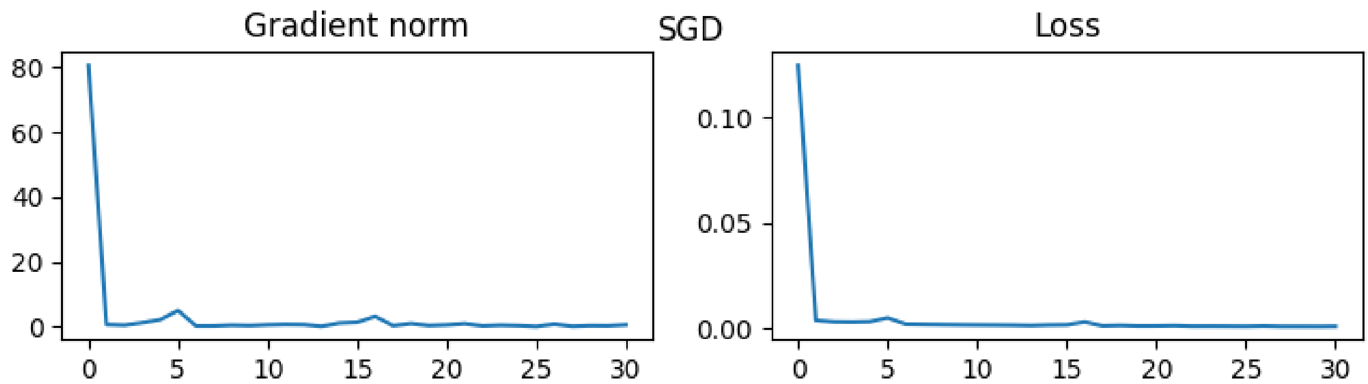
. Test the logistic regression classificator for different digits and different training set dimensions. The training procedure will end up with a set of optimal parameters w $*$

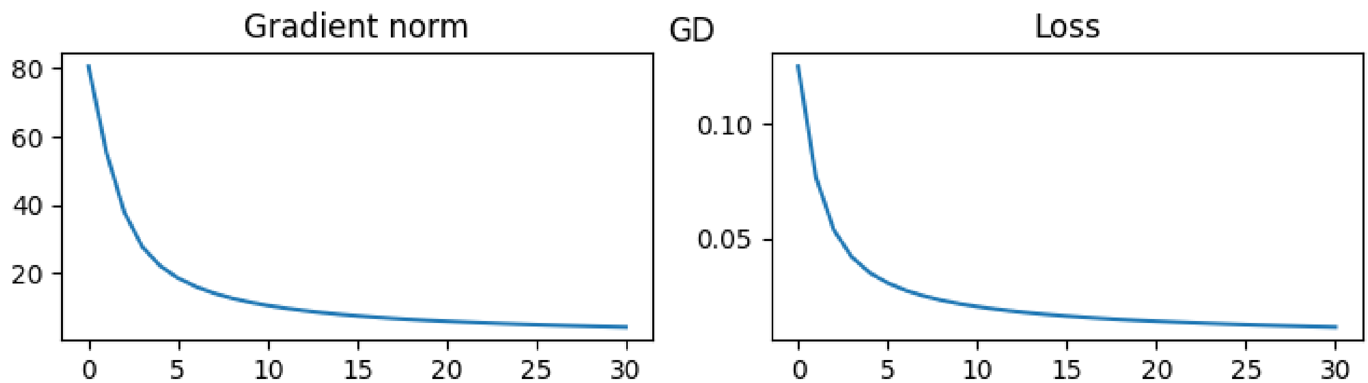
. Compare w∗ when computed with Gradient Descent and Stochastic Gradient Descent, for different digits and different training set dimensions.

In [ ]:
```
def binary_logistic_regression(X_train, Y_train, algorithm, batch_size=128, epochs=30, l
    X_train = addCoefficient(X_train) #adding the bias
    w0 = np.zeros(X_train.shape[0])
    if algorithm == "sgd":
        history_w, history_loss, history_grad, history_err = sgd(loss=loss,grad_loss=gra
            #batch_size=batch_size, n_epochs=epochs,lr=lr)
            batch_size=64, n_epochs=epochs,lr=lr)
        plt.figure(figsize=(9, 2))
        plt.suptitle("SGD")
        plt.subplot(1, 2, 1)
        plt.plot(range(len(history_err)), history_err)
        plt.title("Gradient norm")
        plt.subplot(1, 2, 2)
        plt.plot(range(len(history_loss)), history_loss)
        plt.title("Loss")
        plt.show()
    else:
        history_w, curr_k, history_loss, history_grad, history_err = gd(loss=loss,grad_l
            w0=w0,data=(X_train, Y_train),k_max=epochs,tol_loss=tol_loss,tol_w=tol_w, al
        )
        plt.figure(figsize=(9, 2))
        plt.suptitle("GD")
        plt.subplot(1, 2, 1)
        plt.plot(range(len(history_err)), history_err)
        plt.title("Gradient norm")
        plt.subplot(1, 2, 2)
        plt.plot(range(len(history_loss)), history_loss)
        plt.title("Loss")
        plt.show()
    return history_w[-1]

# we assign 1 or 0 in order allow the evaluation
def predict_binary_logistic(X, w, threshold=0.5):
    X = addCoefficient(X) # we add the coeff. 1 for the bias
    if X.ndim == 1:
        return 1 if f(w, X) >= threshold else 0
    else:
        return np.array([1 if f(w, X[:, i]) >= threshold else 0 for i in range(X.shape[1

def evaluate_binary_logistic(model_w, X_test, Y_test):
    correct = 0
    for i in range(X_test.shape[1]):
        if predict_binary_logistic(X_test[:, i], model_w) == Y_test[i]:
            correct += 1
    return correct / X_test.shape[1]
```

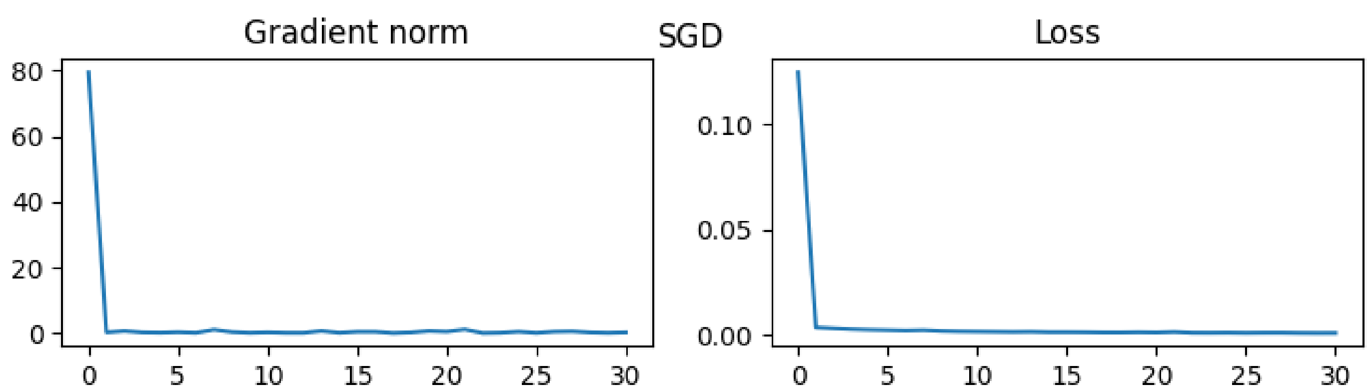
In [ ]:
```
def binary_digit_evaluation(digit1, digit2, train_ratio):
    X_train, Y_train, X_test, Y_test = createDataset(full_X, full_Y, [digit1, digit2], t
    model_w_sgd = binary_logistic_regression(X_train, Y_train, algorithm="sgd")
    model_w_gd = binary_logistic_regression(X_train, Y_train, algorithm="gd")
    accuracy_sgd = evaluate_binary_logistic(model_w_sgd, X_test, Y_test)
    accuracy_gd = evaluate_binary_logistic(model_w_gd, X_test, Y_test)
    accuracy_sgd_train= evaluate_binary_logistic(model_w_sgd, X_train, Y_train)
    accuracy_gd_train= evaluate_binary_logistic(model_w_gd, X_train, Y_train)
    norm_diff = np.linalg.norm(model_w_sgd - model_w_gd)
    print(f'Xtrain = {X_train.shape[1]}, Xtest = {X_test.shape[1]}')
    print(f"Digits: ({digit1}, {digit2}), Train Ratio: {train_ratio}, ||w_sgd - w_gd||_2
          f"SGD Acc Test: {accuracy_sgd:.4f}, GD Acc Test: {accuracy_gd:.4f}")

import random
num_pairs = 3
```

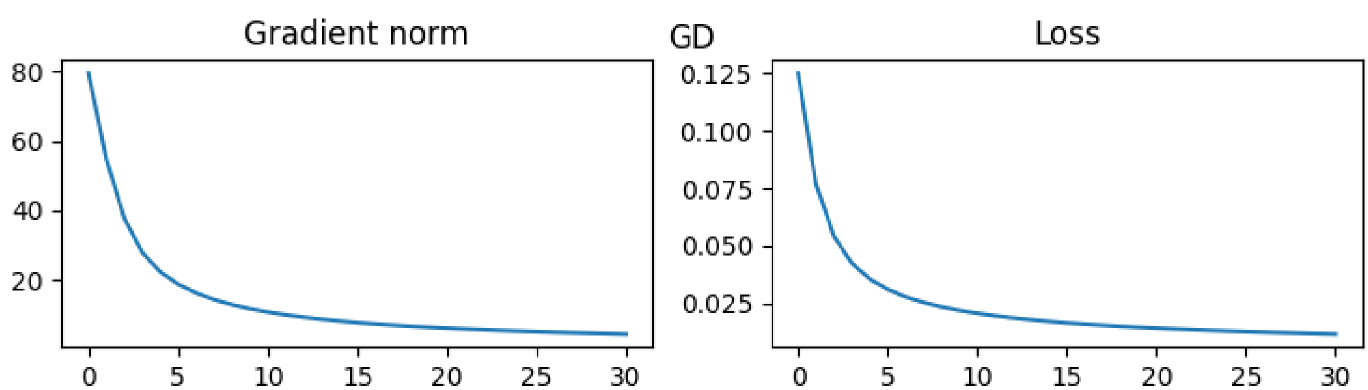
```python
train_ratios = [0.5, 0.7, 0.8]
for _ in range(num_pairs):
    print('\n')
    digit1, digit2 = random.sample(range(10), 2)
    for train_ratio in train_ratios:
        binary_digit_evaluation(digit1, digit2, train_ratio)
```



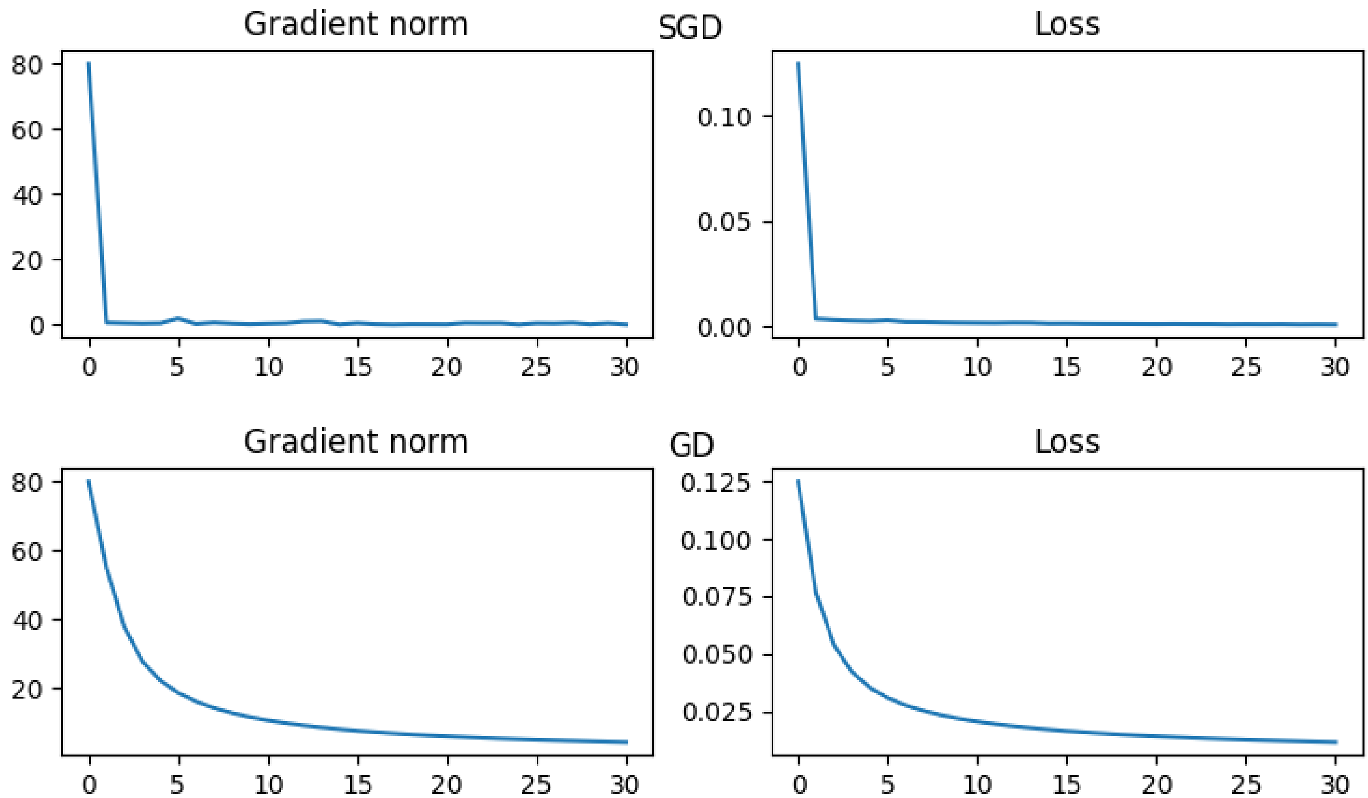



```
Xtrain = 4517, Xtest = 4518
Digits: (3, 1), Train Ratio: 0.5, ||w_sgd - w_gd||_2 = 0.0245, SGD Acc Train: 0.9991, GD
Acc Train: 0.9845, SGD Acc Test: 0.9923, GD Acc Test: 0.9823
```

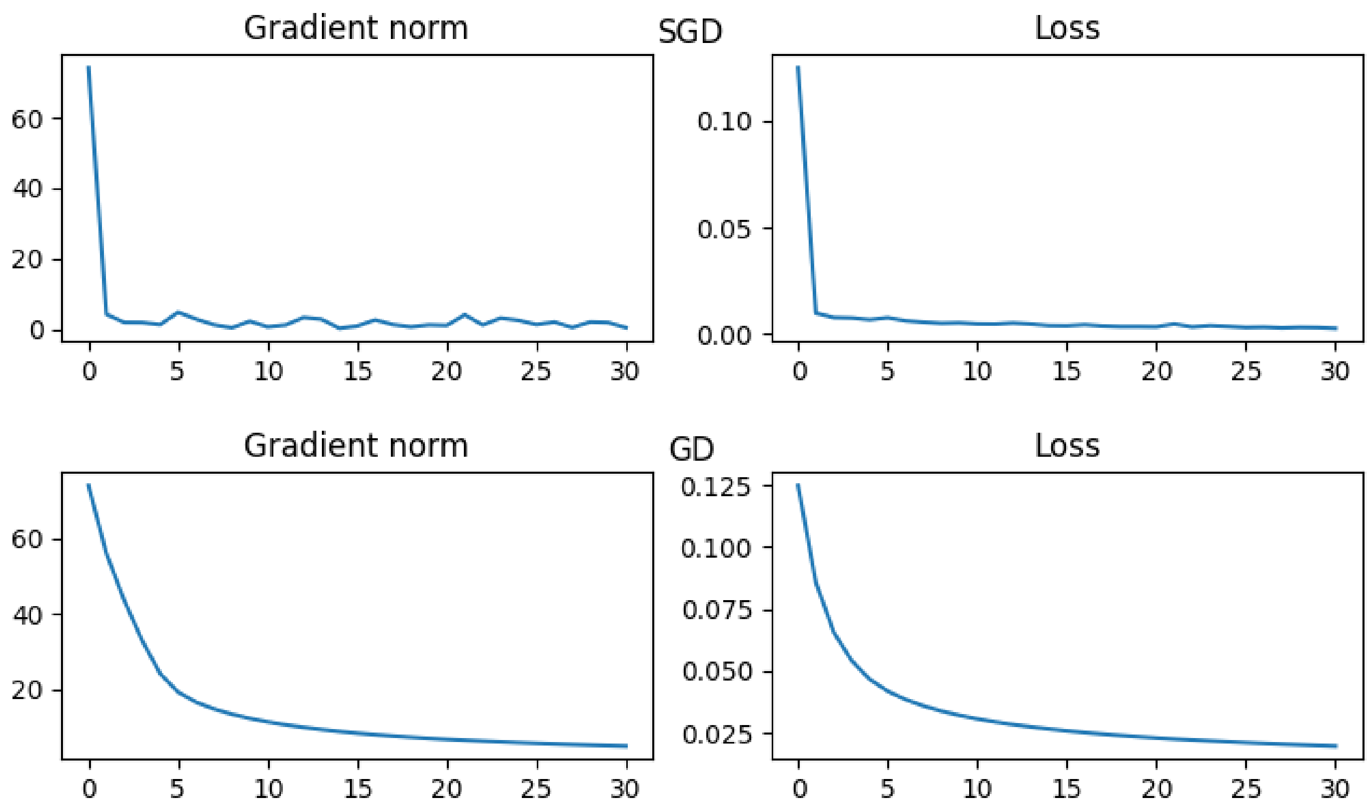




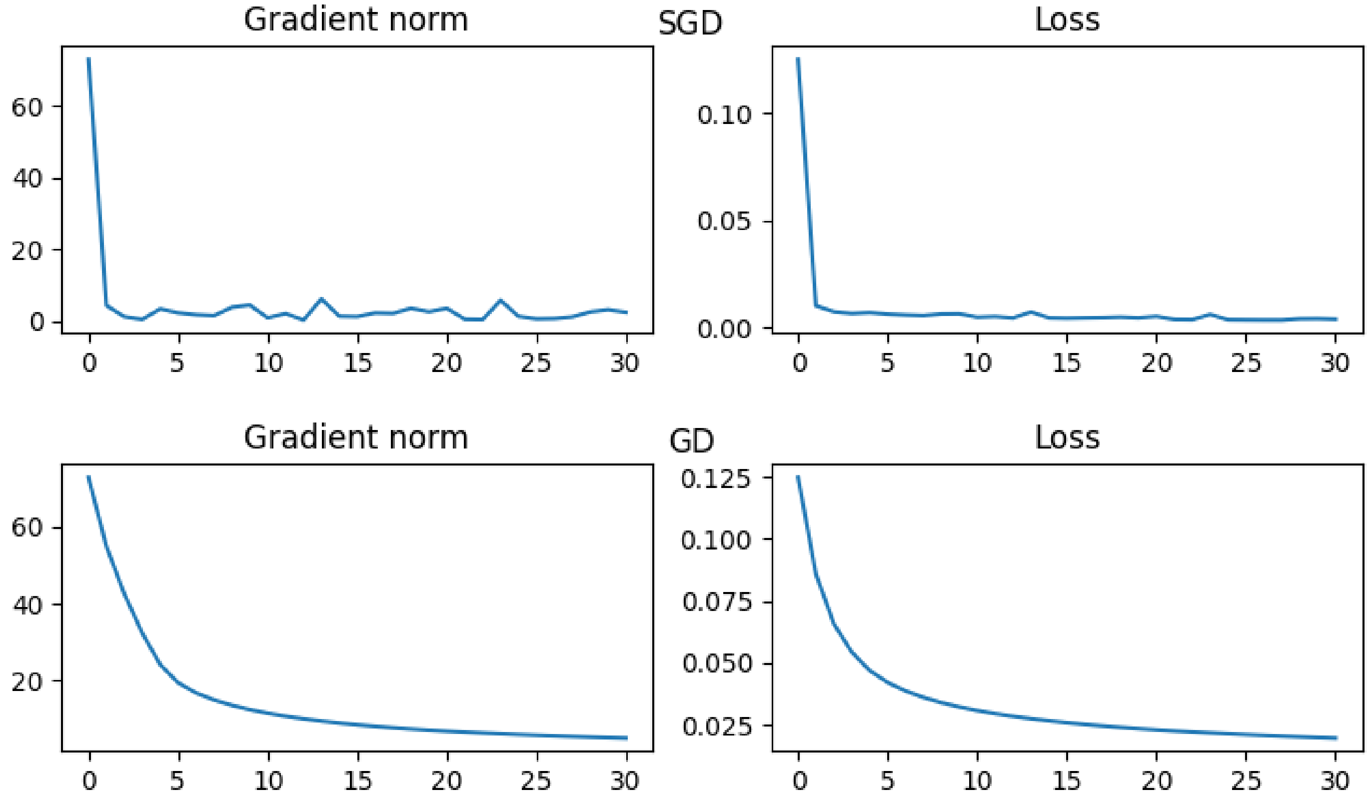

```
Xtrain = 6324, Xtest = 2711
Digits: (3, 1), Train Ratio: 0.7, ||w_sgd - w_gd||_2 = 0.0277, SGD Acc Train: 0.9989, GD
Acc Train: 0.9847, SGD Acc Test: 0.9919, GD Acc Test: 0.9808
```



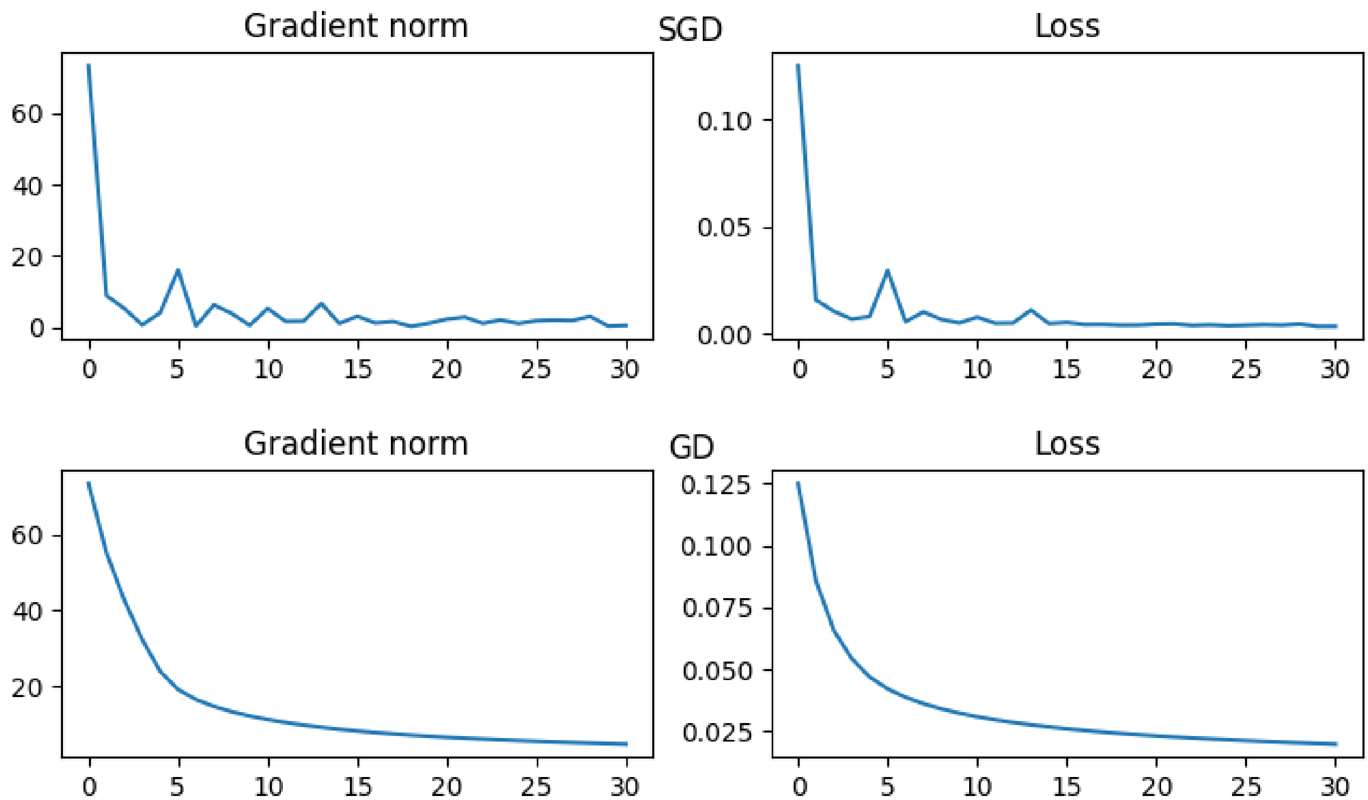

```
Xtrain = 7228, Xtest = 1807
Digits: (3, 1), Train Ratio: 0.8, ||w_sgd - w_gd||_2 = 0.0296, SGD Acc Train: 0.9989, GD
Acc Train: 0.9844, SGD Acc Test: 0.9923, GD Acc Test: 0.9806
```



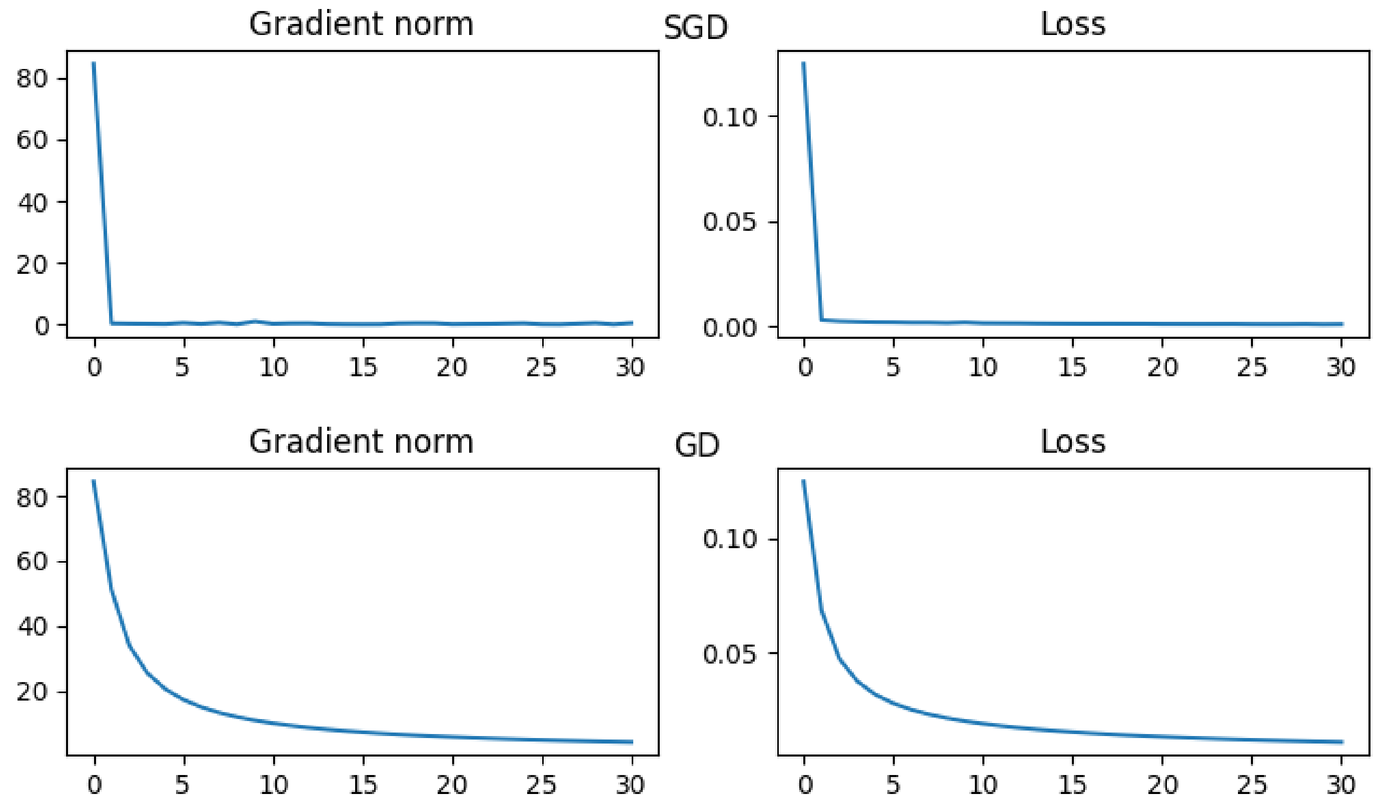

```
Xtrain = 3966, Xtest = 3966
Digits: (6, 5), Train Ratio: 0.5, ||w_sgd - w_gd||_2 = 0.0339, SGD Acc Train: 0.9962, GD
Acc Train: 0.9670, SGD Acc Test: 0.9768, GD Acc Test: 0.9637
```



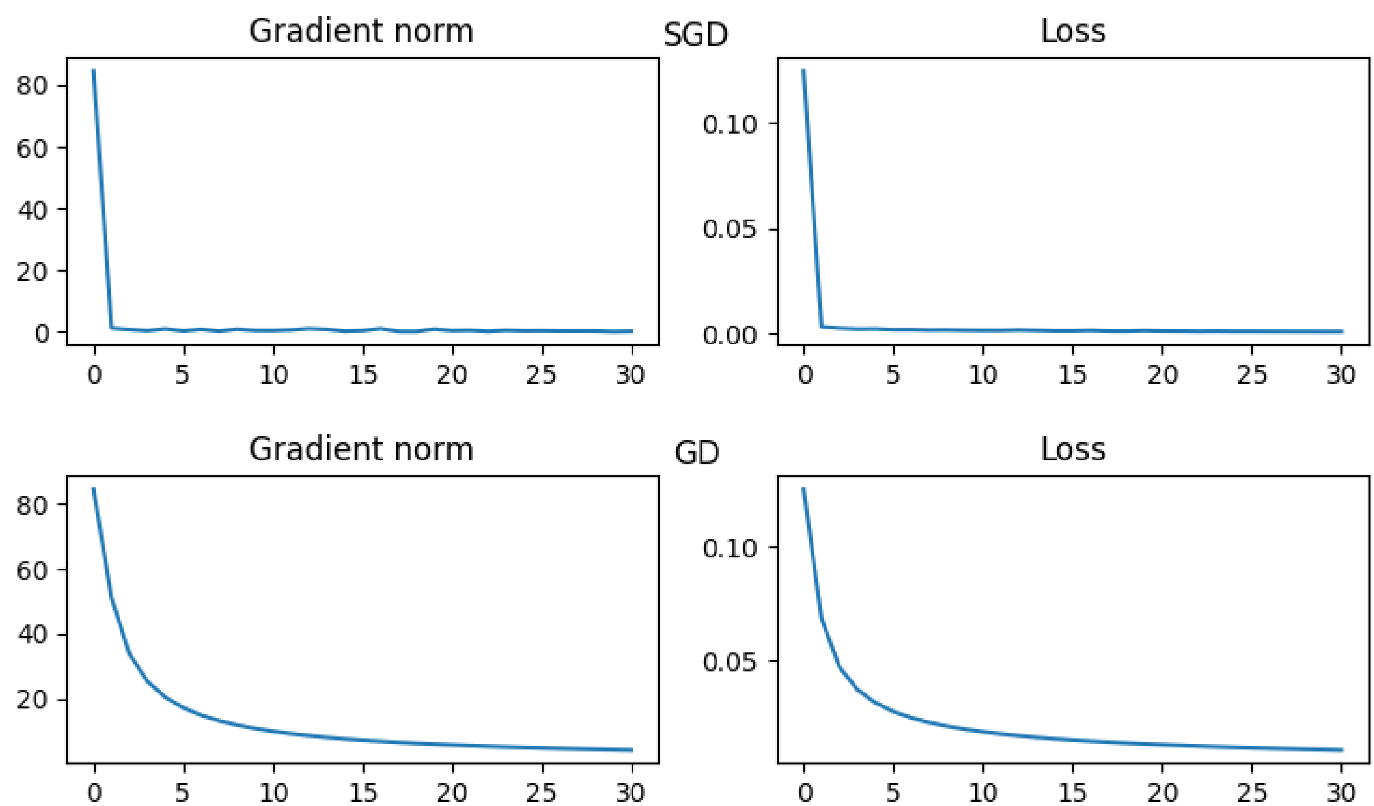

```
Xtrain = 5552, Xtest = 2380
Digits: (6, 5), Train Ratio: 0.7, ||w_sgd - w_gd||_2 = 0.0376, SGD Acc Train: 0.9908, GD
Acc Train: 0.9687, SGD Acc Test: 0.9727, GD Acc Test: 0.9626
```



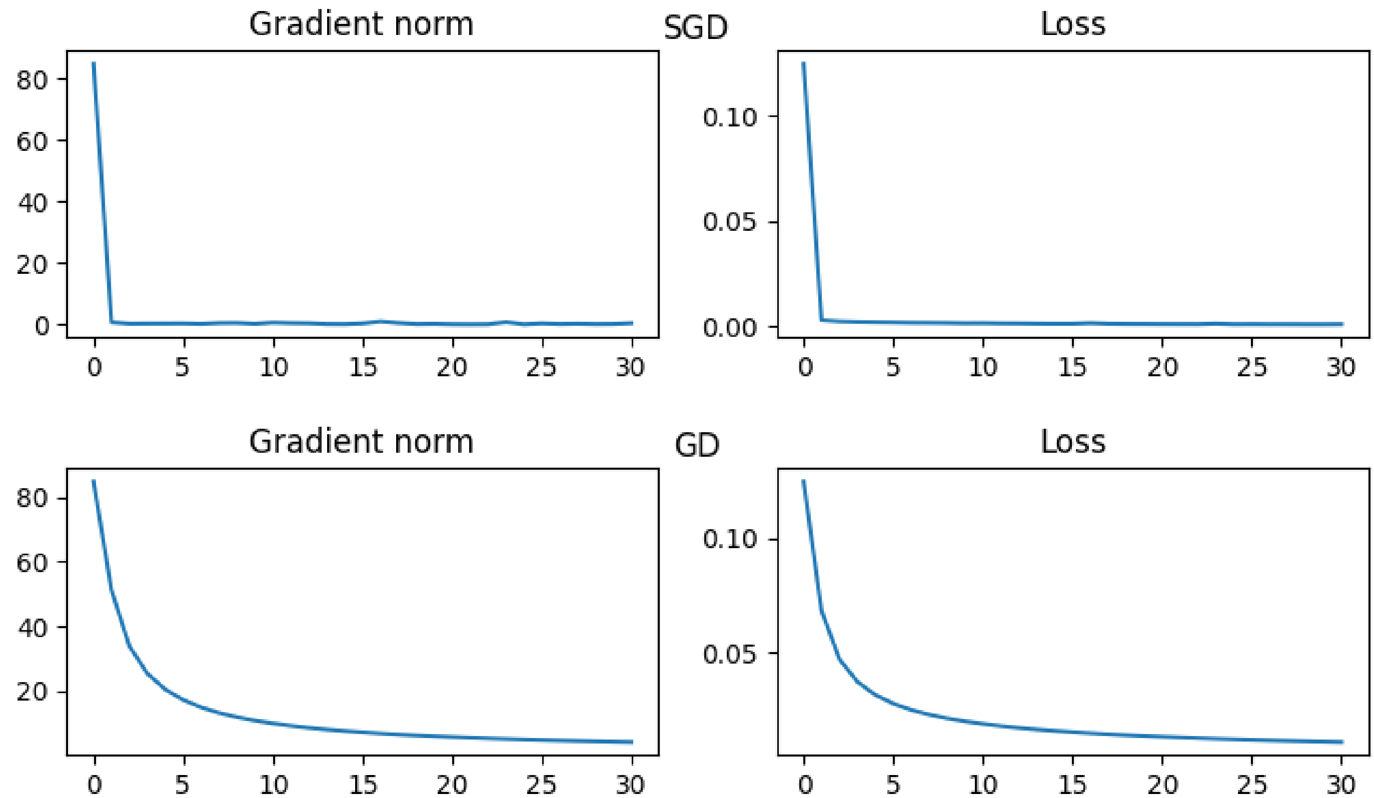

```
Xtrain = 6345, Xtest = 1587
Digits: (6, 5), Train Ratio: 0.8, ||w_sgd - w_gd||_2 = 0.0399, SGD Acc Train: 0.9939, GD
Acc Train: 0.9683, SGD Acc Test: 0.9754, GD Acc Test: 0.9590
```



Xtrain = 4542, Xtest = 4543
Digits: (7, 1), Train Ratio: 0.5, ||w_sgd - w_gd||_2 = 0.0219, SGD Acc Train: 0.9989, GD Acc Train: 0.9846, SGD Acc Test: 0.9949, GD Acc Test: 0.9844



Xtrain = 6359, Xtest = 2726
Digits: (7, 1), Train Ratio: 0.7, ||w_sgd - w_gd||_2 = 0.0259, SGD Acc Train: 0.9989, GD Acc Train: 0.9833, SGD Acc Test: 0.9967, GD Acc Test: 0.9886




```
Xtrain = 7268, Xtest = 1817
Digits: (7, 1), Train Ratio: 0.8, ||w_sgd - w_gd||_2 = 0.0267, SGD Acc Train: 0.9986, GD
Acc Train: 0.9839, SGD Acc Test: 0.9972, GD Acc Test: 0.9884
```