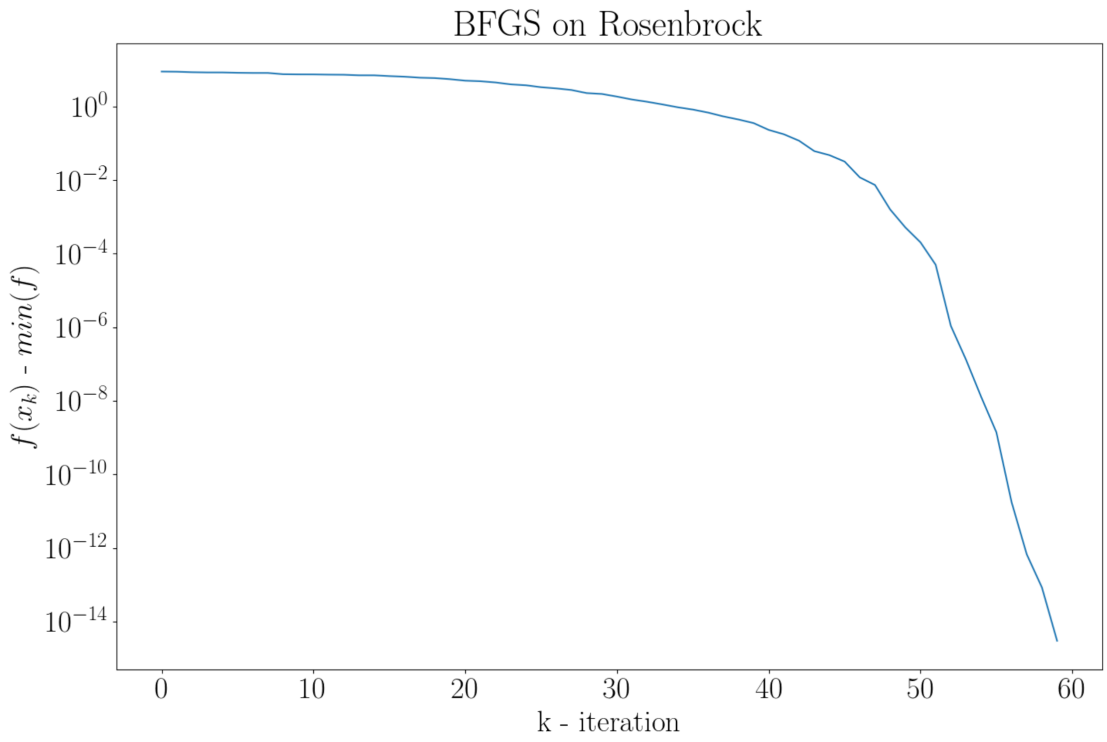
BFGS

Rosenbrock function



Deep Neural Network

Task 3:

Given the following:

$$y = (y_1, y_2) = f(x_1, x_2) = x_1 \cdot e^{-x_1^2 - x_2^2}$$

$$F(x, W) = F(x, W_1, W_2, W_3, b_1, b_2, b_3) = W_3^T \varphi_2 \left(W_2^T \varphi_1 \left(W_1^T x + b_1 \right) + b_2 \right) + b_3$$

$$\varphi_1(x) = \varphi_2(x) = \varphi(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$\psi(r) = r^2, \quad r \triangleq F(x, W) - y$$

As seen in the lecture we can calculate the gradient of the error with respect to in the following way:

$$\frac{d}{dx}\tanh(x) = \frac{d}{dx}\left(\frac{1-e^{-2x}}{1+e^{-2x}}\right) = \frac{2e^{-2x}\left(1+e^{-2x}\right)+2e^{-2x}\left(1-e^{-2x}\right)}{(1+e^{-2x})^2} = \frac{4e^{-2x}}{(1+e^{-2x})^2} = \frac{4}{(e^x+e^{-x})^2}$$

$$\Phi\prime = \begin{pmatrix} \varphi\prime(u_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \varphi\prime(u_n) \end{pmatrix} = \begin{pmatrix} \frac{4}{(e^{u_1}+e^{-u_1})^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{4}{(e^{u_n}+e^{-u_n})^2} \end{pmatrix}$$

$$\nabla_r \psi = 2r$$

$$\nabla_x \psi = J^T_{F(x,W)} \nabla_r \psi = W_1 \Phi\prime_1 W_2 \Phi\prime_2 W_3 \nabla_r \psi$$

Now we can calculate the gradient with respect to intermediate values:

$$u_1 = W_1^T x + b_1 \;\Rightarrow\; \nabla_{u_1}\psi = \Phi\prime_1 W_2 \Phi\prime_2 W_3 \nabla_r \psi$$

$$u_2 = W_2^T \varphi(u_1) + b_2 \;\Rightarrow\; \nabla_{u_2}\psi = \Phi\prime_2 W_3 \nabla_r \psi$$

$$u_3 = W_3^T \varphi(u_2) + b_3 \;\Rightarrow\; \nabla_{u_3}\psi = \nabla_r \psi$$

Now we calculate the gradient with respect to the weights:

$$d\psi = \nabla_{u_1}\psi^T du_1 = \nabla_{u_1}\psi^T dW_1{}^T x == Tr\left(\nabla_{u_1}\psi^T dW_1{}^T x\right) = Tr\left(x\nabla_{u_1}\psi^T dW_1{}^T\right) = \left\langle \nabla_{u_1}\psi x^T, dW_1 \right\rangle \Rightarrow \underline{\nabla_{W_1}\psi = \nabla_{u_1}\psi x^T = \Phi\prime_1 W_2 \Phi\prime_2 W_3 \nabla_r \psi x^T}$$

$$d\psi = \nabla_{u_1}\psi^T du_1 = \nabla_{u_1}\psi^T db_1 = \langle \nabla_{u_1}\psi, db_1 \rangle \Rightarrow \underline{\nabla_{b_1}\psi = \nabla_{u_1}\psi = \Phi\prime_1 W_2 \Phi\prime_2 W_3 \nabla_r \psi}$$

$$d\psi = \nabla_{u_2}\psi^T du_2 = \nabla_{u_2}\psi^T dW_2{}^T \varphi(u_1) == Tr\left(\nabla_{u_2}\psi^T dW_2{}^T \varphi(u_1)\right) = Tr\left(\varphi(u_1)\nabla_{u_2}\psi^T dW_2{}^T\right) = \left\langle \nabla_{u_2}\psi\varphi(u_1)^T, dW_2 \right\rangle \Rightarrow \underline{\nabla_{W_2}\psi = \nabla_{u_2}\psi\varphi(u_1)^T = \Phi\prime_2 W_3 \nabla_r \psi\varphi(u_1)^T}$$

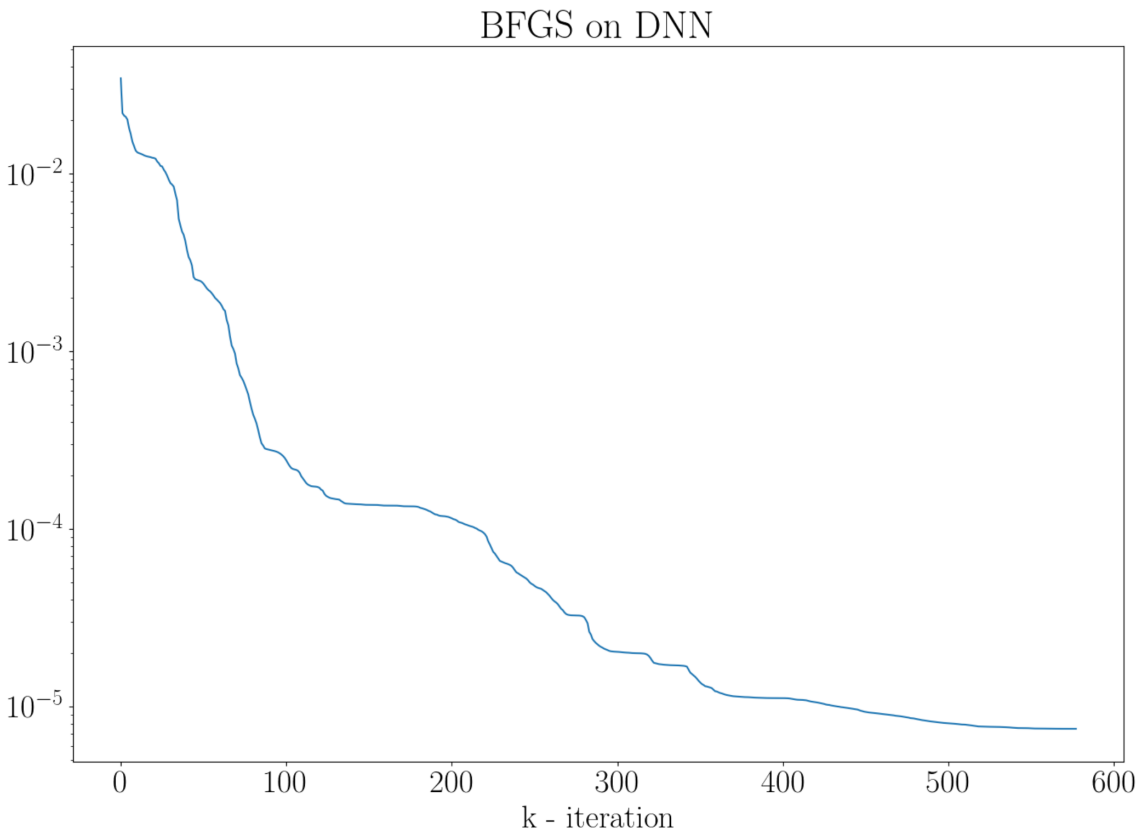
$$d\psi = \nabla_{u_2}\psi^T du_2 = \nabla_{u_2}\psi^T db_2 = \langle \nabla_{u_2}\psi, db_1 \rangle \Rightarrow \underline{\nabla_{b_2}\psi = \nabla_{u_2}\psi = \Phi\prime_2 W_3 \nabla_r \psi}$$

$$d\psi = \nabla_{u_3}\psi^T du_3 = \nabla_{u_3}\psi^T dW_3{}^T \varphi(u_2) == Tr\left(\nabla_{u_3}\psi^T dW_3{}^T \varphi(u_2)\right) = Tr\left(\varphi(u_2)\nabla_{u_3}\psi^T dW_3{}^T\right) = \left\langle \nabla_{u_3}\psi\varphi(u_2)^T, dW_3 \right\rangle \Rightarrow \underline{\nabla_{W_3}\psi = \nabla_{u_3}\psi\varphi(u_2)^T = \nabla_r \psi\varphi(u_2)^T}$$

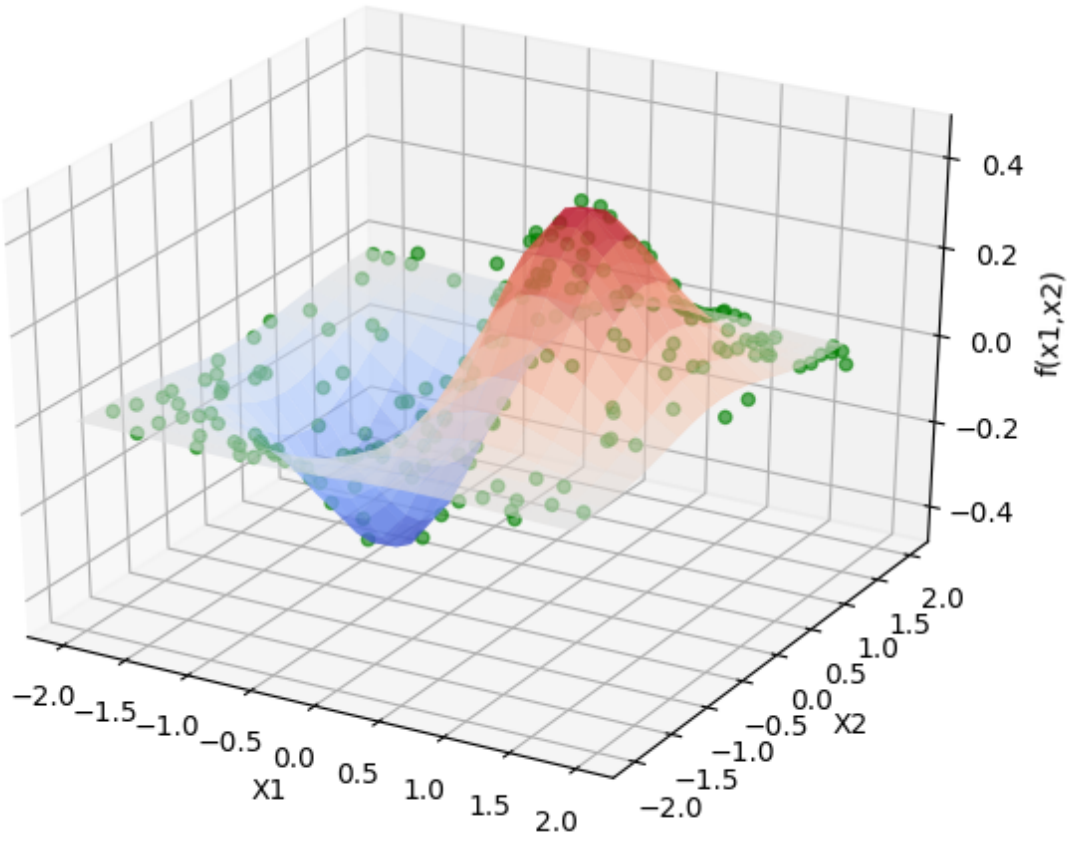
$$d\psi = \nabla_{u_3}\psi^T du_3 = \nabla_{u_3}\psi^T db_3 = \langle \nabla_{u_2}\psi, db_1 \rangle \Rightarrow \underline{\nabla_{b_3}\psi = \nabla_{u_3}\psi = \nabla_r \psi}$$

Training Error of Loss

Figure 1: Training Error of Loss



Function Reconstruction for the Test Set


0.4
0.2
0.0
−0.2
−0.4
f(x1,x2)
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
X1
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
X2

HW2.py - For BFGS on Rosenbrock

```
import numpy as np
import matplotlib.pyplot as plt
from HW2.mcholmz import modified_chol
from scipy.io import loadmat
from functools import partial


def call_foreach(funcs, x):
    try:
        return tuple(map(lambda f: f(x), funcs))
    except TypeError:
        pass
    return funcs(x, 2)


class FunctionAt:
    def __init__(self, x, derivative_sequence):
        self.x = x
        self.der_i_at_x = call_foreach(derivative_sequence, x)


def normalize(v):
    norm = np.linalg.norm(v)
    if norm == 0:
        norm = np.finfo(v.dtype).eps
    return v/norm


def armijo(f,
           g_x,
           x,
           d,
           alpha=1.,
           beta=0.5,
           sigma=0.25):
    try:
        f0 = f[0]
    except TypeError:
        f0 = lambda v: f(v, 1)
    f_at_x = f0(x)
    df = np.dot(g_x.T, d)

    while f0(x + alpha * d) - f_at_x > sigma * df * alpha:
        # print(f0(x + alpha * d) - f_at_x, ":", sigma * df * alpha)
        alpha *= beta

    return x + alpha * d
```

```
def rosenbrock_f(x):
    return np.sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0, axis
        =0)


def rosenbrock_g(x):
    x_mid = x[1:-1]
    x_prev = x[:-2]
    x_next = x[2:]

    g = np.zeros_like(x)

    # vectorize all but the edges
    g[1:-1] = (200.0 * (x_mid - x_prev**2.0) -
               400.0 * (x_next - x_mid**2.0) * x_mid -
               2.0 * (1 - x_mid))

    # set the edges manually
    g[0] = (-400.0 * x[0] * (x[1] - x[0]**2.0) -
            2.0 * (1 - x[0]))
    g[-1] = 200.0 * (x[-1] - x[-2]**2.0)
    return g


def rosenbrock_h(x):
    x = np.atleast_1d(x.reshape(-1))
    H = np.diag(-400.0 * x[:-1], 1) - np.diag(400.0 * x[:-1], -1)
    diagonal = np.zeros(x.shape, dtype=x.dtype)
    diagonal[0] = 1200.0 * x[0]**2.0 - 400.0 * x[1] + 2.0
    diagonal[-1] = 200.0
    diagonal[1:-1] = 202.0 + 1200.0 * x[1:-1]**2.0 - 400.0 * x[2:]
    H = H + np.diag(diagonal)
    return H


def rosenbrock(nargout=1):
    assert 1 <= nargout <= 3

    f = rosenbrock_f
    if nargout == 1:
        return f

    g = rosenbrock_g
    if nargout == 2:
        return f, g

    H = rosenbrock_h
```

```
    return f, g, H


def rosenbrock_at(x, par, nargout=1):
    return call_foreach(rosenbrock(nargout), x)


def find_newton_direction(f_der_x, _):
    g, H = f_der_x[1:3]
    L, d, e = modified_chol(H)
    y = substitution(L, -g, direction=FORWARD_SUBSTITUTION)
    z = y.reshape(-1, 1) / d
    return substitution(L.T, z, direction=BACKWARD_SUBSTITUTION).reshape(-1,
        1)


FORWARD_SUBSTITUTION = 1
BACKWARD_SUBSTITUTION = -1


def substitution(L, b, direction=FORWARD_SUBSTITUTION):
    rows = len(L)
    x = np.zeros(rows, dtype=L.dtype)
    row_sequence = reversed(range(rows)) if direction == BACKWARD_SUBSTITUTION
        else range(rows)
    for row in row_sequence:
        delta = b[row] - np.dot(L[row], x)
        cur_x = delta / L[row, row]
        x[row] = cur_x
    return x


def bfgs_direction(f_der_x, b):
    # return step direction
    return -b.dot(f_der_x[1])


def sqr(v):
    return v.dot(v.T)


def bfgs_update(b_prev, f_der_x, f_der_x_prev):
    if not f_der_x_prev:
        return b_prev

    p = f_der_x.x - f_der_x_prev.x
    q = f_der_x.der_i_at_x[1] - f_der_x_prev.der_i_at_x[1]
    s = b_prev.dot(q)
    t = s.T.dot(q)
```

```
    m = p.T.dot(q)
    v = p/m - s/t

    b = b_prev + sqr(p)/m - sqr(s)/t + t * sqr(v)

    return b


def iterative_minimization(get_direction,
                           f_derivatives_sequence,
                           initial_guess=np.zeros((10, 1)),
                           alg_data=None,
                           update_data=None,
                           epsilon=1e-5):
    x = initial_guess
    f_history = []
    f_der_x_prev = None

    while True:
        f_der_x = FunctionAt(x, f_derivatives_sequence)
        f_history.append(f_der_x.der_i_at_x[0])

        # print("#", len(f_history), " f:", f_der_x.der_i_at_x[0], " g:", np.
            linalg.norm(f_der_x.der_i_at_x[1]))

        if update_data:
            alg_data = update_data(alg_data, f_der_x, f_der_x_prev)

        if np.linalg.norm(f_der_x.der_i_at_x[1]) < epsilon:  # ||g(x)|| < e
            break

        f_der_x_prev = f_der_x

        # Armijo line search
        x = armijo(f=f_derivatives_sequence,
                   g_x=f_der_x.der_i_at_x[1],
                   x=x,
                   d=get_direction(f_der_x.der_i_at_x, alg_data))

    return x, np.array(f_history)


def gradient_descent(f_derivatives_sequence, initial_guess=np.zeros((10, 1))):
    return iterative_minimization(get_direction=lambda f_der_x, _: -f_der_x
        [1],  # return -g(x)
                                  f_derivatives_sequence=
                                      f_derivatives_sequence[0:2],
                                  initial_guess=initial_guess)
```

```
def newton_method(f_derivatives_sequence, initial_guess=np.zeros((10, 1))):
    return iterative_minimization(get_direction=find_newton_direction,
                                  f_derivatives_sequence=
                                      f_derivatives_sequence,
                                  initial_guess=initial_guess)


def bfgs(f_derivatives_sequence, initial_guess=np.zeros((10, 1))):
    return iterative_minimization(get_direction=bfgs_direction,
                                  update_data=bfgs_update,
                                  alg_data=np.eye(initial_guess.size),
                                  f_derivatives_sequence=
                                      f_derivatives_sequence,
                                  initial_guess=initial_guess)


def plot_convergence(f_values, f_min, title, plot_num):
    if plot_num != 0:
        plt.subplot(3, 1, plot_num)
    plt.title(title)
    plt.xlabel(r'k - iteration')
    plt.ylabel(r'$f(x_k$) - $min(f)$')
    plt.semilogy(f_values - f_min)


def quad_f(x, h):
    return (0.5 * x.T.dot(h).dot(x)).reshape(-1)


def quad_g(x, h):
    return h.dot(x)


def quad_h(x, h):
    return h


def quad(h):
    return (partial(quad_f, h=h),
            partial(quad_g, h=h),
            partial(quad_h, h=h))


def main():
    mat_file = loadmat("h.mat")
    well, ill = mat_file['H1'], mat_file['H2']
    f_well = quad(well)
    f_ill = quad(ill)
```

```
    dim = (10, 1)

    method_name = {gradient_descent: r'Gradient Descent', newton_method: r'
        Newton Method', bfgs: r'BFGS'}
    for minimization_method in (gradient_descent, newton_method, bfgs):
        i = 1
        plt.figure(figsize=(15, 30))
        plt.rc('text', usetex=True)
        plt.rc('font', family='serif', size=28)
        for fname, f_der_sequence, initial_guess in ((r'Rosenbrock',
            rosenbrock(3), np.zeros(dim)),
                                                     (r'Well Conditioned
                                                         Quadratic Problem',
                                                         f_well, np.ones(dim))
                                                         ,
                                                     (r'Ill Conditioned
                                                         Quadratic Problem',
                                                         f_ill, np.ones(dim))
                                                     ):

            title = method_name[minimization_method] + r' on ' + fname
            _, f_values = minimization_method(f_der_sequence, initial_guess)
            plot_convergence(f_values, 0, title, i)
            i += 1
        plt.show()


if __name__ == '__main__':
    main()
```

HW3.py - DNN

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from HW1 import hw1
from HW2 import hw2
import math


def pack_params(params):
    # save the packing dimensions
    pack_params.shapes = [param.shape for param in params]

    return np.hstack(np.ravel(param) for param in params).reshape(-1, 1)
```

```
def unpack_params(packed: np.ndarray):
    shapes = pack_params.shapes
    sizes = map(lambda x: x[0] * x[1], shapes)
    idxs = list(sizes)
    for i in range(len(idxs)-1):
        idxs[i+1] += idxs[i]

    arrays = np.split(packed, idxs)
    return (arr.reshape(shape) for arr, shape in zip(arrays, shapes))


def the_function_f(x, y):
    return x * math.exp(-x**2 - y**2)


the_function_vectorized = np.vectorize(the_function_f)


# activation function
def phi_f(x: np.ndarray):
    # x_exp = np.exp(-2*x)
    # return (1-x_exp)/(1+x_exp)
    return np.tanh(x)


def phi_g(x: np.ndarray):
    # it should be equivalent somehow to the derivative I calculated by hand
    g_i_i = 4. / (np.square(np.exp(-x) + np.exp(x)))
    return np.diag(np.ravel(g_i_i))


def phi_at(x: np.ndarray, nargout=1):
    assert 1 <= nargout <= 2

    f = phi_f(x)
    if nargout == 1:
        return (f,)

    g = phi_g(x)
    return f, g


def dnn_forward(x: np.ndarray, parameters, nargout=1):
    assert 1 <= nargout <= 2

    b1, b2, b3, W1, W2, W3 = unpack_params(parameters)

    u1 = W1.T @ x + b1
```

```
    act_u1_der_x = phi_at(u1, nargout=nargout)
    u2 = W2.T @ act_u1_der_x[0] + b2

    act_u2_der_x = phi_at(u2, nargout=nargout)
    u3 = W3.T @ act_u2_der_x[0] + b3

    if nargout == 1:
        return u3

    return u3, (act_u1_der_x, act_u2_der_x)


def error_f(out, y):
    return (out - y) ** 2


def dnn_error(x: np.ndarray, y, parameters, nargout=1):
    assert 1 <= nargout <= 2

    if nargout == 1:
        out = dnn_forward(x, parameters, nargout=nargout)
        return error_f(out, y)

    out, layer_der = dnn_forward(x, parameters, nargout=nargout)
    error = error_f(out, y)

    _, _, _, W1, W2, W3 = unpack_params(parameters)

    grad_b3 = 2 * (out - y)
    grad_W3 = (grad_b3 @ layer_der[1][0].T).T

    grad_b2 = layer_der[1][1] @ W3 @ grad_b3
    grad_W2 = (grad_b2 @ layer_der[0][0].T).T

    grad_b1 = layer_der[0][1] @ W2 @ grad_b2
    grad_W1 = (grad_b1 @ x.T).T

    return np.array((error, pack_params((grad_b1, grad_b2, grad_b3,
                                         grad_W1, grad_W2, grad_W3))))


def target(X, Y, parameters, nargout=1):
    assert 1 <= nargout <= 2

    error_sum = sum(dnn_error(x=x.reshape(-1, 1),
                              y=y.reshape(-1, 1),
                              parameters=parameters,
                              nargout=nargout)
```

```
                    for x, y in zip(X.T, Y.T))
    return error_sum / X.shape[1]


def get_target_f_of_xy(X, Y):
    return lambda x, n: target(X=X, Y=Y, parameters=x, nargout=n)


def generate_W(m, n):
    return np.random.rand(m, n) / math.sqrt(n)


def generate_b(n):
    return np.zeros((n, 1))


def generate_W_b():
    return pack_params((generate_b(4),
                        generate_b(3),
                        generate_b(1),
                        generate_W(2, 4),
                        generate_W(4, 3),
                        generate_W(3, 1)))


def main():
    # plot the target function
    line = np.arange(-2, 2, .2)

    X1, X2 = np.meshgrid(line, line)

    Y = the_function_vectorized(X1, X2)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    ax.plot_surface(X1, X2, Y, cmap=plt.cm.coolwarm, alpha=.6)

    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('f(x1,x2)')

    # generate train and test data
    n_train = 500
    n_test = 200
    n_data = (n_train, n_test)
    data_X = []
    data_Y = []
```

```
    for n in n_data:  # train & test
        X = 4 * np.random.rand(2, n) - 2
        Y = the_function_vectorized(X[0], X[1])
        data_X.append(X)
        data_Y.append(Y)

    # initial weights
    parameters = generate_W_b()

    learned_params, f_history = hw2.bfgs(get_target_f_of_xy(data_X[0],
                                                            data_Y[0]),
                                         parameters)

    data_Y[1] = np.array(list(dnn_forward(x=x.reshape(-1, 1), parameters=
        learned_params, nargout=1) for x in data_X[1].T))

    ax.scatter(data_X[1][0], data_X[1][1], data_Y[1], c='g', alpha=.61)
    plt.show(block=False)

    plt.figure(figsize=(15, 30))
    plt.rc('text', usetex=True)
    plt.rc('font', family='serif', size=28)
    hw2.plot_convergence(f_history.reshape((-1, 1)), 0, "BFGS on DNN", 1)

    plt.show(block=False)


if __name__ == '__main__':
    main()
```

tests.py - Gradient correctness using numdiff

```
import unittest
import numpy.testing as npt
import numpy as np
from HW1 import hw1
from HW3 import hw3


class Test_target_function(unittest.TestCase):
    def test_target_function_f(self):
        npt.assert_almost_equal(0, hw3.the_function_f(0, 0))
        npt.assert_almost_equal(0, hw3.the_function_f(20, 0))
        npt.assert_almost_equal(0, hw3.the_function_f(0, 40))
        npt.assert_almost_equal(np.exp(-1), hw3.the_function_f(1, 0))

    def test_packing(self):
        a1 = np.array([[4, 5, 6], [41, 51, 63], [1, 2, 1]])
```

```
        a2 = np.array([[100]])
        a3 = np.array([[411, 225, 446, 55], [-411, -225, -446, -55]])

        p = hw3.pack_params((a1, a2, a3))

        b1, b2, b3 = hw3.unpack_params(p)

        npt.assert_equal(a1, b1)
        npt.assert_equal(a2, b2)
        npt.assert_equal(a3, b3)

    def test_grad_numdiff(self):
        params = hw3.generate_W_b()
        x = 4 * np.random.rand(2, 1) - 2
        y = hw3.the_function_f(x[0], x[1])

        par = {'epsilon': 2e-15}
        ana = hw3.dnn_error(x, y, parameters=params, nargout=2)[1]
        nmr = hw1.numdiff(lambda x_, par: hw3.dnn_error(x, y, x_), params, par
            )
        npt.assert_almost_equal(ana, nmr, 9)

    def test_grad_numdiff_many(self):
        for _ in range(100):
            self.test_grad_numdiff()
```