Introduction to Numerical Data Science, RUB

Report on Deep Neural Networks

by Peter Moeller

Januar 12, 2024

Introduction

Neural networks have a wide range of applications from object detection, facial and language recognition to detecting tumors in X-ray or MRI scans or suspicious financial activities, and much more. In the following report, we will merely scratch the surface of this vast field, discussing questions such as objective functions, optimization (training) of the network and the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm – one of several, used in order to minimize the costs associated with the problem-solution.

Neural networks are computational models inspired by the structure and function of the human brain. It consists of layers of interconnected nodes (neurons) which process data by applying mathematical transformations (in the form of matrices and vectors) to inputs and passing the results through a series of layers to the final output. To each layer is often applied an activation function, which gives the neural network its non-linear character. This could, for instance, be the sigmoid activation function, whose outputs lie between 0 and 1 and are therefore suited for tasks such as calculating probabilities or mapping gray-scale image recognition (with 0 being black and 1 being a white pixel), or the ReLU activation function, which sets all negative values to 0 (used for example in tasks like image classification and reenforcement learning), and have shown themselves to be highly efficient when compared to, for instance, the sigmoid function. To the values propagating through the network, biases are often added, in effect turning the linear functions of the network into affine ones, giving further flexibility (for instance by subtracting a threshold, which a value has to be larger than, in order to become non-zero, and then applying the ReLU function). The final element, which is our main variable that we seek to *optimize* (not *"solve"*, since we are dealing with an overdetermined problem), are the *weights*, represented by matrices (W_i in our examples below). We will then continue to “train” the network, by feeding it with inputs and outputs, and seek to optimize (i.e. minimize) the following function:

$$f(x) = \frac{1}{m}\sum_{i=1}^{m} f_i(x) = \frac{1}{m}\sum_{i=1}^{m} ||z_i - z(y_i; x)||^2,$$

This function represents the mean squared error between the computed outputs $\hat{z}$ (which has been passed through the network) and the true outputs z provided in the training data. In the tasks following, the determination of the gradient of this function (which will allow us to use quasi-Newton methods to determine its minima) is therefore prerequisite.

In the first task (*Solution of a single-layer neural network problem*), we are given a simple one-layer neural network. We will first identify the various parameters to be followed by defining its derivative. Then we will apply the BFGS algorithm, making use of the Wolfe conditions (with the Armijo condition, on the one hand, ensuring a sufficiently large descending step, while the curvature condition ensures a step size, which isn't too large on the other), and plot the steps in relation to the deviation from the already-given output vector (as this relationship is expressed by the objective function $f(x)$).

The cost of using the Newton method (which needs to compute the inverse Hessian directly) is not great for smaller networks, but it becomes increasingly costly, as the network grows in size, having a cost of $O(n^3)$ in O-notation, while the cost of the BFGS algorithm is $O(n^2)$. We will also see how the adjustment of the paramenters and the starting condition, effects the result of our computation.

In the second task (*Neural network with two hidden layers*), we will derive the general expressions for the derivatives of the objective function with respect to the optimization parameters using "backpropagation". To this effect, we will make good use of the good old chain rule, but on an expanded scale.

Results

What follows below are a discussion of the central computations related to neural networks, as well as the implementation and results in Python code.

1. Solution of a single-layer neural network problem

In the assignment before us, we are first given a one-layer neural network, an input vector $y_i = (y_{i,1}, y_{i,2}, y_{i,3})$, and output vector $z_i = (z_{i,1}, (z_{i,2})^T$ for m data points, and the index $i = (1, ..., m)$ (See Figure 1).

To these, we have a loss function defined as:

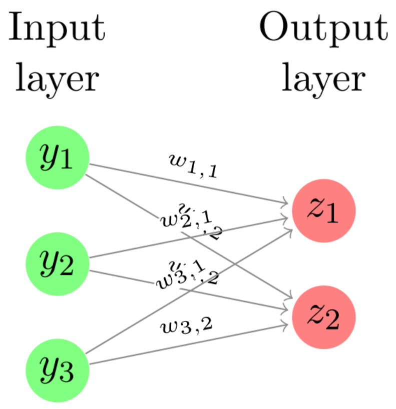

Figure 1: A single-layer neural network

$$f(x) = \frac{1}{m}\sum_{i=1}^{m} f_i(x) = \frac{1}{m}\sum_{i=1}^{m} ||z_i - z(y_i; x)||^2,$$

with the norm being the Euclidean norm. We evaluate the function $\hat{z}$ in two steps. First:

$$\bar{z}(y_i) = W^T y_i,$$

for the weight matrix $W \in R^{3\times 2}$. Next, we insert $\bar{z}$ (entry-by-entry) into the activation function (which we will also call σ later):

$$z_k(y_i) = \frac{1}{1 + e^{-\bar{z}_k(y_i)}}, \quad k = 1, 2,$$

to obtain the final result for z.

As is clear from the information given, the parameter $x \in R^n$ to be optimized in our case are the elements contained in W^T. With an input vector of dimension 3 and an output vector of dimension 2, we therefore get $W \in R^{2\times 3}$, and hence an optimization parameter of 6 dimensions:

$$\mathbf{x} = (w_{1,1}, w_{2,1}, w_{3,1}, w_{1,2}, w_{2,2}, w_{3,2})^T \in R^6$$

We will now derive an expression for the gradient of $f(x)$, which can be used to optimize this one-layer network:

$$\nabla f(x) = \nabla \frac{1}{m}\sum_{i=1}^{m} f_i(x) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(x).$$

For each of the individual functions f_i, we then get the following:

$$\begin{aligned}
\nabla f_i(x) &= \nabla ||z_i - z(y_i;x)||^2 \\
&= \nabla \left(\sqrt{\sum_{k=1}^{2} (z_{i,k} - z_k(y_i;x))^2} \right)^2 \\
&= \nabla \sum_{k=1}^{2} (z_{i,k} - z_k(y_i;x))^2 \\
&= \sum_{k=1}^{2} \nabla (z_{i,k} - z_k(y_i;x))^2 \\
&= 2\sum_{k=1}^{2} (z_{i,k} - z_k(y_i;x)) \frac{\partial}{\partial x}(-z_k(y_i;x)) \\
&= -2\sum_{k=1}^{2} (z_{i,k} - z_k(y_i;x)) \frac{\partial}{\partial x}\sigma(W^T y_i) \frac{\partial}{\partial x}(W^T y_i)
\end{aligned}$$

Let us look at the expressions $\frac{\partial}{\partial x}\sigma(W^T y_i)$ and $\frac{\partial}{\partial x}(W^T y_i)$ separately. We will first write out the general case choosing some s as our variable, deriving the differential using the quotient rule:

$$\begin{aligned}
\sigma'(s) &= \frac{\partial}{\partial s} \frac{1}{1+e^{-s}} \\
&= \frac{0 - 1 \dot{-} e^{-s}}{(1+e^{-s})^2} \\
&= \frac{-e^{-s}}{(1+e^{-s})^2} \\
&= \left(\frac{1}{1+e^{-s}}\right)\left(\frac{1+e^{-s}}{1+e^{-s}} - \frac{1}{1+e^{-s}}\right) \\
&= \sigma(s)(1-\sigma(s))
\end{aligned}$$

In our particular case, we thus get for each component of $z(y_i)$:

$$\frac{\partial}{\partial x} z_k(y_i;x) = z_k(y_i;x)(1 - z_k(y_i;x))$$

The expression $\frac{\partial}{\partial x}(W^T y_i)$ becomes y_i^T, since we are differentiating with respect to **x**. The final expression for the gradient of $f(x)$ thus becomes:

$$\begin{aligned}
\nabla f(x) &= \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(x) \\
&= -\frac{2}{m} \sum_{i=1}^{m} \sum_{k=1}^{2} (z_{i,k} - z_k(y_i;x)) z_k(y_i;x)(1 - z_k(y_i;x)) y_i^T
\end{aligned}$$

As mentioned in the introduction, when the neural network becomes large, computing the inverse of the Hessian becomes quite costly, and we will therefore apply the BFGS algorithm instead.

We start by initializing the data and choose an x_0 (i.e. a weight matrix W_0) with random values close to, but not equal to, a null matrix; the reason being that either very large or very small values could lead to a vanishing gradient, since $\sigma' = (\sigma)(1-\sigma) \approx 0$ in either case. With the BFGS algorithm relying on the gradient, this would cause a problem, as we will see in a moment. We therefore choose an x_0 near 0.

Our optimization process focuses on minimizing $f(x)$, named `obj_f` in the code (see the Appendix at the end). We first check, if the direction of p is a decent. If this condition is met, we apply a loop, which keep iterating until the Armijo and curvature conditions are satisfied. The Armijo condition makes sure that the step in the descent direction is large enough (the factor η is initially set to 0.1), while the curvature condition makes sure the step is not *too* large (the factor μ is initially set at 10^{-4}). If the conditions are met, the factor α is returned.

The BFGS algorithm begins by creating a list to store the objective function values into at each iteration, which we use for the plot later. Starting with an initial approximation `B_inv` of the inverse Hessian (set to the identity matrix of the same dimension as W), we let the algorithm iterate 20 times. Each iteration calculates the search direction $p = -B_{\text{inv}}\nabla f(W)$ and uses the line search function to determine the step size α. The weights are then updated as $W_{\text{new}} = W + \alpha p$, and the gradient $\nabla f(W)$ is recomputed.

In order to refine our inverse Hessian approximation, a step vector $s = W_{\text{new}} - W$ and gradient difference $y_k = \nabla f(W_{\text{new}}) - \nabla f(W)$ are used in the formula:

$$B_{\text{new}}^{-1} = \left(I - \rho s y_k^T\right) B_{\text{inv}} \left(I - \rho y_k s^T\right) + \rho s s^T,$$

with $\rho = \frac{1}{y_k^T s}$.

We save each step in the process in our list `hist`.

The initial plot (Figure 2) is based on the conditions for μ and η provided in the task. In the plot following (Figure 3), we have tightened the conditions μ and η. With the logarithmic y-scale, we clearly observe that the iterations follow a negative exponential trajectory on all cases, with the rate of approximation being slower in the case where the Armijo and curvature conditions are stronger. This highlights the general balance between precision/stability on the one hand and speed on the other. Since the task did not call for an ϵ-threshold, but rather for a plotting of the first 20 iterations, we have left this out of the code. Yet, one clearly sees from the plot how the BFGS algorithm would be able to attain a precission better than an arbitrarily small ϵ.

In Figure 4, we have chosen to illustrate what could happen, when x_0 is chosen with random numbers further away from 0, which we argued for being the optimal choice. All three plots are based on values for x_0 based on a normal distribution (with mean $\mu = 0$ and standard deviation $\sigma = 1$), but with the blue plot, the values has been multiplied by 0.01, the orange plot not multiplied by any factor, and the green plot multiplied by 100, in which last case there is no convergence at all; it is even common for it be overloaded. So, the chosen start value for x_0 (i.e. W_0) is indeed crucial for the result.

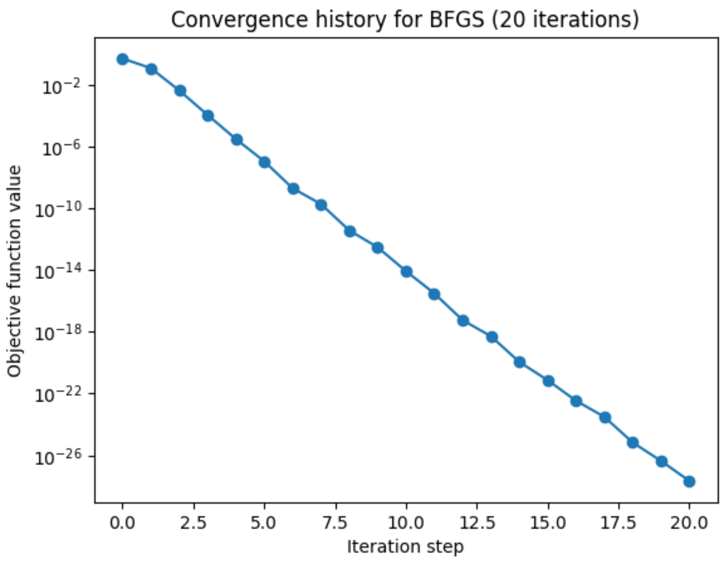

Figure 2: Convergence history for BFGS

3. Neural network with two hidden layers

Next, we are given the following neural network (see Figure 4). The input vector y has 6 components, and the first hidden layer h also has 6 components, including one bias component h_0, which be computed based on the sigmoid function. This is followed by the second hidden layer H, which has 5 components, one of them H_0 being the bias component computed via the TanH activation function. Lastly the output layer z has 3 components. The weights (the matrices mapping one layer unto the following layer), represented by W_i, will therefore have the following form:

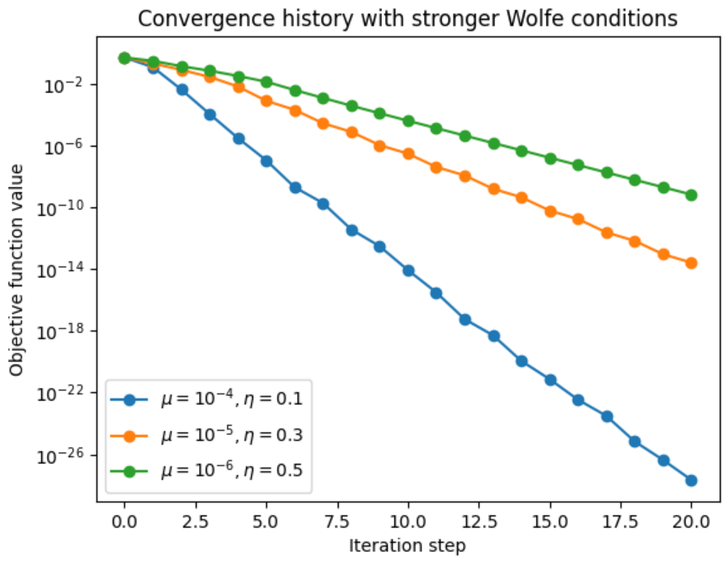

Figure 3: Convergence history for stronger conditions

$$
\begin{aligned}
W_1 &\in R^{5\times 6} \\
W_2 &\in R^{4\times 5} \\
W_3 &\in R^{3\times 4}.
\end{aligned}
$$

The first computed bias component h_0 is defined in the following terms:

$$
\begin{aligned}
h_0 &= \sigma(\bar{h}) \\
&= \sigma\left(\sum_{j=1}^{6} W_{1j} y_j\right) \quad \text{with } \sigma = \frac{1}{1+e^{-\bar{h}}}
\end{aligned}
$$

And the second component H_0:

$$
\begin{aligned}
H_0 &= \tanh(\bar{H}_0) \\
&= \tanh\left(\sum_{j=1}^{5} W_{2j} h_j + h_0\right) \quad \text{with } tanh = \frac{e^{\bar{H}_0} - e^{-\bar{H}_0}}{e^{\bar{H}_0} + e^{-\bar{H}_0}}
\end{aligned}
$$

We will define h^* and H^* as the h and H components without h_0 and H_0 respectively. No activation function is to be applied to h^*. Our neural network thus contains the following steps:

$$h^* = W_1 y,$$

and thus $h = (h_0, h^{*T})^T$. Next, we compute H^*:

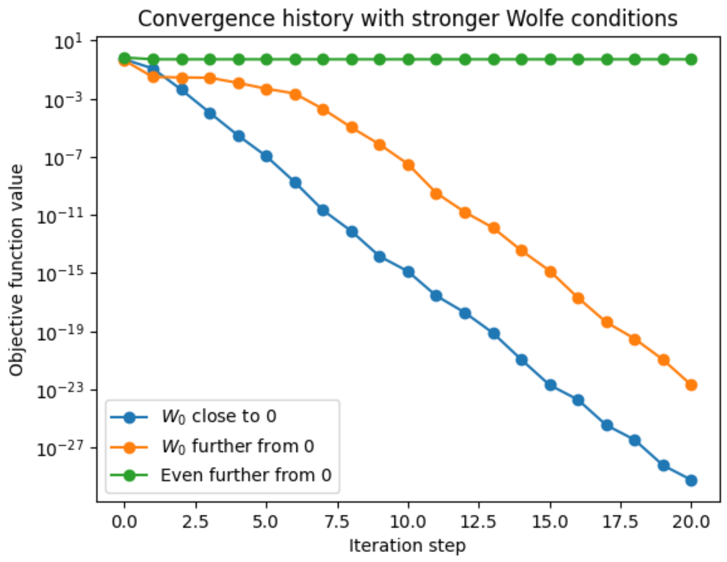

Figure 4: Convergence history for different starting values

$$H^* = tanh(W_2 h^* + h_0 \mathbf{1}),$$

with $\mathbf{1}$ being a vector of ones of dimension 4 and $H = (H_0, H^{*T})^T$.

Finally for z:

$$\hat{z}(y; x) = tanh(W_3 H^* + H_0 \mathbf{1}),$$

with $\mathbf{1}$ having a dimension of 3 in this case and x representing all the elements to be optimized:

$$x = \{W_1, W_2, W_3\},$$

based on the same function we made use of in the first task, namely:

$$f(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x) = \frac{1}{m} \sum_{i=1}^{m} ||z_i - z(y_i; x)||^2.$$

Overall number of components associated with the optimization problem will be:

$$(6 \times 5) + (5 \times 4) + (4 \times 3) = 62,$$

since the biases in this case are all calculated directed, and are not independent variables here, as is often the case. All these aspects taken together, we can thus state our function as follows:

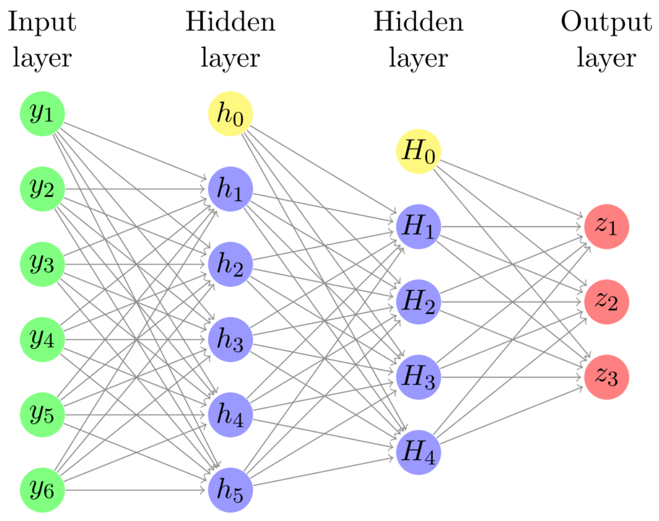

Figure 5: Neural network with two hidden layers

$$\hat{z}(y;x) = tanh\left(W_3 \cdot tanh\left(W_2 \cdot (W_1 y) + h_0\mathbf{1}\right) + H_0\mathbf{1}\right)$$

To derive the general expressions for the derivatives of the objective function $f(x)$ with respect to the optimization parameters, we will use **backpropagation**, which entails applying the chain rule to determine the gradient, layer by layer, with respect to the different W_i.

We proceed as follows:

$$\frac{\partial f_i}{\partial W_3} = \frac{\partial f_i}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial W_3}$$

The first part becomes:

$$\begin{aligned}\frac{\partial f_i}{\partial \hat{z}} &= \frac{\partial ||z_i - \hat{z}_i||^2}{\partial \hat{z}_i} \\ &= 2(z_i - \hat{z}_i)\end{aligned}$$

Next, we take the derivative of $\hat{z}$ with respect to W_3:

$$\begin{aligned}\frac{\partial \hat{z}}{\partial W_3} &= \frac{\partial tanh(W_3 H^* + H_0\mathbf{1})}{\partial W_3} \\ &= 1 - tanh(W_3 H^* + H_0\mathbf{1})^2 \cdot H^T \\ &= (1 - \hat{z}^2) \cdot H^T.\end{aligned}$$

Combining the two we thus get:

$$\frac{\partial f_i}{\partial W_3} = 2(z_i - \hat{z}_i) \odot (1 - \hat{z}_i^2) \cdot H_i^T,$$

with $\odot$ representing component-wise multiplication.

We continue with the derivative with respect to W_2 again using the chain rule:

$$\begin{aligned} \frac{\partial f_i}{\partial W_2} &= \frac{\partial f_i}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial H} \cdot \frac{\partial H}{\partial W_2} \\ &= \big(2(z_i - \hat{z}_i) \odot (1 - \hat{z}_i^2)\big) \cdot W_3^T \cdot \big((1 - H_i^2) h_i^T\big) \end{aligned}$$

And finally, for f_i with respect to W_1 we get:

$$\begin{aligned} \frac{\partial f_i}{\partial W_1} &= \frac{\partial f_i}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial H} \cdot \frac{\partial H}{\partial h} \cdot \frac{\partial h}{\partial W_1} \\ &= \big(2(z_i - \hat{z}_i) \odot (1 - \hat{z}_i^2)\big) \cdot W_3^T \cdot \big((1 - H_i^2) \cdot W_2^T \cdot h_i\big) \cdot y_i^T \end{aligned}$$

We thus see that the gradient in general depends on the error from the objective function $f(x)$ as it propagates backwards through the different layers, the activation derivatives $(1 - tanh(x)^2)$ and the respective input layers for each mapping.

For the derivatives of $f(x)$ we therefore end up with the following expressions:

$$\frac{\partial f(x)}{\partial W_3} = \frac{1}{m} \sum_{i=1}^{m} \big(2(\hat{z}_i - z_i) \odot (1 - \hat{z}_i^2)\big) \cdot H_i^T$$

$$\frac{\partial f(x)}{\partial W_2} = \frac{1}{m} \sum_{i=1}^{m} \big[\big(\big(2(\hat{z}_i - z_i) \odot (1 - \hat{z}_i^2)\big) \cdot W_3^T\big) \odot (1 - H_i^2)\big] \cdot h_i^T$$

$$\frac{\partial f(x)}{\partial W_1} = \frac{1}{m} \sum_{i=1}^{m} \big[\big(\big(\big(2(\hat{z}_i - z_i) \odot (1 - \hat{z}_i^2)\big) \cdot W_3^T \cdot (1 - H_i^2) \cdot W_2^T\big)\big) \odot h_i\big] \cdot y_i^T$$

(For reasons of time, the author was not able to implement these derivations in Python code.)

Conclusion

With the growing importance of machine learning and neural networks more efficient algorithms are increasingly becoming a necessity. In this small report, we primarily looked at the BFGS algorithm, which showed itself to converge exponentially with each iteration, as shown in the "linear" graph on the logarithmic scale. We saw how the algorithm was sensitive to the hyperparameters

μ and η, which pointed towards the common theme of how to strike the right balance between convergence speed on the one hand, and stability and precision on the other. We further showed, and argued theoretically, why it is important to choose the right starting value for x_0, and showed a case, where the convergence not only seemed slightly more unstable at the beginning, but even a case, where no convergence took place at all.

The other key investigation here was the efficient computation of the derivatives of $f(x)$ with respect to the variables $W_i, i = (1, 2, 3)$. To do this, we used **backpropagation**, which relies heavily on the famous chain rule. One way to increase efficiency here was to reuse the intermediate results, like for instance $\frac{\partial f_i}{\partial \hat{z}}$, which was used several times in our derivations. This approach shows (although not demonstrated conclusively here, at least *potentially*), that this approach can be taken, even when the network is significantly larger than our examples, making it an indispensable tool for training/optimizing deep neural networks.

This short report was merely a proverbial "dipping of the toe" into the vast ocean of neural networks, but it is clear that a clear and structured approach exists, and we hope that these short examples helped give a sense of this.

The Python code

```python

import numpy as np
import matplotlib.pyplot as plt

y = np.array([[1.2, 2.4, 1.7], [-1.3, 1.0, 0.0], [0.0, 1.3,
    0.4], [0.2, 0.5, 0.9], [-0.3, 1.1, -0.6], [-1.2, -0.7,
    -0.1]])
z = np.array([[1, 0], [0, 1], [1, 0], [1, 0], [0, 1], [0,
    1]])
x_0 = np.random.randn(6)*0.01
x_1 = np.random.randn(6)
x_2 = np.random.randn(6)*100
W_0 = x_0.reshape(3,2)
W_1 = x_1.reshape(3,2)
W_2 = x_2.reshape(3,2)
n = len(y[0])
m = len(y)

def z_bar(y, W=W_0):
    return y@W

def z_hat(y, W=W_0):
    z_ = z_bar(y,W)
    return 1 / (1 + np.exp(-z_))
```

```

# objective function computing the mean squared error
def obj_f(y,z, W=W_0):
    z_ = z_hat(y,W)  # Predicted values
    return np.mean(np.sum((z-z_)**2, axis=1))

# function computing the gradient
def grad_f(y, z, W=W_0):
    z_ = z_hat(y, W)  # computed z values
    diff = z - z_     # difference between computed and
        given z values
    grad = -(y.T @ (diff*z_*(1-z_)))/m
    return grad

# function for the line search using the Wolfe conditions
def line_search(y,z,W,p, alpha_0=1, mu=1e-4, eta=0.1):
    alpha = alpha_0
    f_current = obj_f(y,z,W)
    grad_current = grad_f(y,z,W)
    grad_dot_p = np.sum(grad_current*p)

    if grad_dot_p >= 0:
        print("p is not a descend.")
        return

    while True:
        W_new = W + alpha*p
        f_new = obj_f(y,z,W_new)
        grad_new = grad_f(y,z,W_new)

        armijo = f_new <= f_current + mu*alpha*grad_dot_p
        curvature = np.sum(grad_new*p) >= eta*grad_dot_p

        # if conditions are fulfilled, break out of the loop
             and return alpha
        if armijo and curvature:
            break

        # set up the conditions as specified in the task
        if np.sum(grad_new*p)*grad_dot_p < 0:
            alpha = alpha/1.5
        else:
            alpha = alpha*2

    return alpha

def bfgs(y, z, W_0, max_iter=20, alpha_0=1, mu=1e-4, eta
    =0.1):
    W = W_0.copy()
    B_inv = np.eye(W.size)
    hist = []
    hist.append(obj_f(y,z,W))
    grad = grad_f(y,z,W).flatten()

    for _ in range(max_iter):
        p = -B_inv @ grad
        p = p.reshape(W.shape)

        # line search
        alpha = line_search(y,z,W,p,alpha_0,mu,eta)

        # the new weights
        W_new = W + alpha*p
        grad_new = grad_f(y,z,W_new).flatten()

        # differences between old and new weights
        s = (W_new - W).flatten()
        y_k = grad_new - grad

        # new inv. hessian approx.
        rho = 1/np.dot(y_k,s)
        E = np.eye(len(s))
        B_inv = (E-rho*np.outer(s,y_k)) @ B_inv @ (E-rho*np.
            outer(y_k,s)) + rho*np.outer(s,s)

        # new weight and gradient
        W = W_new
        grad = grad_new

        # save obj. function value in hist list
        hist.append(obj_f(y,z,W))

    return hist

# convergence history for 20 iterations
conv_hist = bfgs(y,z,W_0,20)

plt.plot(range(len(conv_hist)), conv_hist, 'o-', label=r'$\
    mu=10^{-4}, \eta=0.1$')
plt.xlabel("Iteration step")
plt.ylabel("Objective function value")
plt.yscale('log')
plt.title("Convergence history for BFGS (20 iterations)")
plt.show()

conv_hist_2 = bfgs(y,z,W_0,20,1,mu=1e-5,eta=0.3)
conv_hist_3 = bfgs(y,z,W_0,20,1,mu=1e-6,eta=0.5)

plt.plot(range(len(conv_hist)), conv_hist, 'o-', label=r'$\
    mu=10^{-4}, \eta=0.1$')
```

```python
plt.plot(range(len(conv_hist_2)), conv_hist_2, 'o-', label=r'$\mu=10^{-5}, \eta=0.3$')
plt.plot(range(len(conv_hist_3)), conv_hist_3, 'o-', label=r'$\mu=10^{-6}, \eta=0.5$')
plt.xlabel("Iteration step")
plt.ylabel("Objective function value")
plt.yscale('log')
plt.legend(loc='lower left')
plt.title("Convergence history with stronger Wolfe conditions")
plt.show()

conv_hist_4 = bfgs(y,z,W_1,20,1,mu=1e-4,eta=0.1)
conv_hist_5 = bfgs(y,z,W_2,20,1,mu=1e-4,eta=0.1)

plt.plot(range(len(conv_hist)), conv_hist, 'o-', label=r"$W_0$ close to 0")
plt.plot(range(len(conv_hist_4)), conv_hist_4, 'o-', label=r"$W_0$ further from 0")
plt.plot(range(len(conv_hist_5)), conv_hist_5, 'o-', label=r"Even further from 0")
plt.xlabel("Iteration step")
plt.ylabel("Objective function value")
plt.yscale('log')
plt.legend(loc='lower left')
plt.title("Convergence history with stronger Wolfe conditions")
plt.show()
```