Project for the Computational Mathematics for Learning and Data Analysis Course

Annisa Dininta, Tarasco Pier Paolo

December 9, 2020

Project 19 (M) is so-called extreme learning, i.e., a neural network with one hidden layer, $y = W_2\sigma(W_1x)$, where the weight matrix for the hidden layer W_1 is a fixed random matrix, $\sigma(\dots)$ is an elementwise activation function of your choice, and the output weight matrix W_2 is chosen by solving a linear least-squares problem (with L_2 regularization).

(A1) is a standard momentum descent approach. [12]

(A2) is an algorithm of the class of quasi-Newton methods. [8]

In addition, for (A2), you are expected to vary the dimension of the hidden layer and study how accuracy and computational time vary based on this choice.

Contents

1 Introduction and Overview

Extreme Learning Machines are feedforward neural networks where the weights of the hidden layers needs not to be trained. The power of this neural network comes from their learning algorithms and it was shown that they can be trained thousand of times faster than networks trained using backpropagation. Indeed, in Extreme Learning Machines or ELM, only the weights directly connected to the output units are learned, usually in a single step by solving a linear model. [15]

1.1 Architecture of an Extreme Learning Machine

An ELM consists of one hidden layer with non-linear activation function. Given N distinct samples (x_j, t_j) where $x_j = [x_{j1}, x_{j2}, \ldots, x_{jn}]^T \in R^n$ and $t_j = [t_{j1}, t_{j2}, \ldots, t_{jm}]^T \in R^m$, an ELM with h hidden nodes and activation function $\sigma(x)$ is modelled as:

$$\sum_{i=1}^{h} \beta_i \sigma_i(x_j) = \sum_{i=1}^{h} \beta_i \sigma(w_i \cdot x_j + b_i) = o_j,\ j = 1, \ldots, N \tag{1}$$

where $w_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ is the weight vector connecting the ith hidden node and the input nodes, $\beta_i = [\beta_{i1}, \beta_{i2}, \ldots, \beta_{im}]^T$ is the weight vector connecting the ith hidden node to the output nodes, and b_i is the threshold, or bias, of the ith hidden node. $w_i \cdot x_j$ denotes the inner product of w_i and x_j.

The equation (1) can be written in a compact form as:

$$\begin{bmatrix} t_1 & \ldots & t_N \end{bmatrix} = \begin{bmatrix} \beta_1 & \ldots & \beta_h \end{bmatrix} \begin{bmatrix} \sigma(w_1 \cdot x_1 + b_1) & \ldots & \sigma(w_1 \cdot x_N + b_1) \\ & \vdots & \\ \sigma(w_h \cdot x_1 + b_h) & \ldots & \sigma(w_h \cdot x_N + b_h) \end{bmatrix}$$

$$\begin{bmatrix} t_1^T \\ t_2^T \\ \vdots \\ t_N^T \end{bmatrix} = \begin{bmatrix} \sigma(w_1 \cdot x_1 + b_1) & \ldots & \sigma(w_h \cdot x_1 + b_h) \\ \sigma(w_1 \cdot x_2 + b_1) & & \\ \vdots & & \\ \sigma(w_1 \cdot x_N + b_1) & \ldots & \sigma(w_h \cdot x_N + b_h) \end{bmatrix} \begin{bmatrix} \beta_1^T \\ \beta_2^T \\ \vdots \\ \beta_h^T \end{bmatrix}$$

$$\mathbf{T} = \mathbf{H}\beta$$

where each row j of $\mathbf{H}$ represents the output of the hidden layer on input x_j. $\mathbf{H}$ is called the hidden layer output matrix. $\mathbf{H} \in R^{N \times h}$, $\beta \in R^{h \times m}$ and $\mathbf{T} \in R^{N \times m}$.

An ELM with N hidden nodes is able to approximate N samples with zero error assuming $\mathbf{H}$ is non-singular, formally: $\sum_{i=1}^{N} \|o_i - t_i\| = 0$. [5]

1.2 Training an Extreme Learning Machine

The input weights w_i and the hidden layer biases b_i need not to be updated by the learning algorithm. So, the matrix $\mathbf{H}$ can actually remain unchanged once random values have been assigned in the beginning. Training an ELM then aims to find the value of β.

1.2.1 A First Training Algorithm

If $\mathbf{H}$ is square and invertible, then an ELM as previously defined can approximate the N samples with zero error. The learning algorithm then can be formulated as finding a least-squares solution β^* of the linear system $\mathbf{H}\beta = \mathbf{T}$. The smallest norm least-squares solution of the linear system is $\beta^* = \mathbf{H}^+\mathbf{T}$, where $\mathbf{H}^+$ is the pseudo-inverse of $\mathbf{H}$. [5]

A complete training algorithm for an ELM can be described in 3 steps, given a training set $D = \{(x_i, t_i)\ x_i \in R^n,\ t_i \in R^m, i = 1, \ldots, N\}$, activation function $\sigma(x)$ and h hidden nodes, compute:

Step 1: Randomly assign input weight $\mathbf{w_i}$ and bias b_i, $i = 1, \ldots, h$.

Step 2: Compute hidden layer output matrix $\mathbf{H}$.

Step 3: Compute the output weight $\beta = \mathbf{H}^+\mathbf{T}$, where $\mathbf{T} = [t_1, \ldots, t_N]^T$. [5]

1.2.2 Iterative Approach

As any other neural network, an ELM can be trained by minimizing the model error. Given N training examples, activation function $\sigma(x)$, h hidden nodes, input weight matrix $W \in R^{h \times n}$, and hidden layer bias b, training an ELM leads to a minimization problem: [6]

$$\min_{\beta} \frac{1}{N} \sum_{i=1}^{N} Loss(f(x_i), t_i) \text{ where } f(x_i) = \beta^T \sigma(W x_i + b)$$

It is a common practice to control the model complexity by adding regularization or penalty term to the loss function. Using L2 regularization term with λ as regularization parameter, the problem becomes: [6]

$$\min_{\beta} \frac{1}{N} \sum_{i=1}^{N} Loss(f(x_i), t_i) + \frac{\lambda}{N} \sum_{i=1}^{h} \|\beta_i\|^2$$

In this project, we develop two algorithms in iterative approach using L2 regularization which will be described in chapter 2.

2 Theoretical Background

In this chapter we show and derive some methods used to solve an optimization problem, in our case the minimum of the error of an ELM.

2.1 Gradient Descent Method

The simplest way to solve an optimization problem is by using gradient descent. Using this technique, at each step, we try to move to the minimum error of our model by following small steps in the direction of the steepest descent, this is obtained by the minus of the gradient. Formally, we start at position $\theta^{(0)}$ and move to a new position $\theta^{(1)}$ which decreases the error, this procedure is repeated until reaching a local minima. [6] In general, we can define an equation from the position at iteration t, indicated by $\theta^{(t)}$ to the next position at iteration $(t+1)$, $\theta^{(t+1)}$, as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla f(\theta^{(t)})$$

where $\eta > 0$ is called the learning rate. Intuitively, it regulates the convergence speed of the model; typically a low learning rate will produce a stable but slow convergence to a minima while a high learning rate can be faster but could also never converge.

2.1.1 Gradient Descent Learning of an Extreme Learning Machine

Training an ELM corresponds to tuning the β weights in order to minimize the error of the model. Assuming the error function to be the mean-squared error over the training data, we can formalize the total error of the model as the mean of the errors on each pattern (x_i, t_i). The error on the pattern (x_i, t_i) is defined as $E_i(\beta) = (f(x_i) - t_i)^2$, where $f(x_i) = \beta^T \sigma(W x_i + b)$. Since our aim is to find the minimum of $E(\beta)$, we have to compute the change of the error with respect to the change in the β_j, where β_j are the weight connecting the j hidden unit to the output units, this is equal to compute $\nabla E(\beta)$.

$$\frac{\partial E(\beta)}{\partial \beta_j} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial E_i(\beta)}{\partial \beta_j} \tag{2}$$

Let's focus on computing $\frac{\partial E_i(\beta)}{\partial \beta_j}$.

$$\frac{\partial E_i(\beta)}{\partial \beta_j} = \frac{\partial (f(x_i) - t_i)^2}{\partial \beta_j} = 2(f(x_i) - t_i) \frac{\partial (f(x_i))}{\partial \beta_j} \tag{3}$$

Focusing on the last part $\frac{\partial (f(x_i))}{\partial \beta_j}$, we have:

$$\frac{\partial (f(x_i))}{\partial \beta_j} = \frac{\partial (\beta^T \sigma(W x_i + b))}{\partial \beta_j} = \sigma(w_j \cdot x_i + b_j)$$

Combining together, equation (3) becomes

$$\frac{\partial(E_i(\beta))}{\partial\beta_j} = 2(f(x_i) - t_i)\frac{\partial(f(x_i))}{\partial\beta_j} = 2(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j)$$

Substituting in equation (2), we can compute the total error as:

$$\frac{\partial E(\beta)}{\partial\beta_j} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial E_i(\beta)}{\partial\beta_j} = \frac{1}{N}\sum_{i=1}^{N}2(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j) \tag{4}$$

Given equation (4), we can state the update rule for the weight β_j as follow:

$$\beta_j = \beta_j + \Delta\beta_j$$

$$\Delta\beta_j = -\eta\,\frac{2}{N}\sum_{i=1}^{N}(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j)$$

Adding L2 regularization, we have that the error of the model is equal to:

$$E(\beta) = \frac{1}{N}\sum_{i=1}^{N}E_i(\beta) + \frac{\lambda}{N}\sum_{i=1}^{h}\|\beta_i\|^2$$

Computing the partial derivative of the updated error, we get:

$$\begin{aligned}\frac{\partial E(\beta)}{\partial\beta_j} &= \frac{1}{N}\sum_{i=1}^{N}2(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j) + \frac{\partial(\frac{\lambda}{N}\sum_{i=1}^{h}\|\beta_i\|^2)}{\partial\beta_j} \\ &= \frac{1}{N}\sum_{i=1}^{N}2(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j) + 2\frac{\lambda}{N}\beta_j\end{aligned}$$

The update rule for the β_j weights becomes:

$$\Delta\beta_j = -\eta\,\frac{2}{N}\sum_{i=1}^{N}(f(x_i) - t_i)\sigma(w_j \cdot x_i + b_j) - 2\frac{\lambda}{N}\beta_j$$

2.1.2 Convergence of Gradient Descent

Given the problem

$$min\ E(\beta)$$

$$E(\beta) = \frac{1}{N}\sum_{i=1}^{N}E_i(\beta) + \frac{\lambda}{N}\sum_{i=1}^{h}\|\beta_i\|^2$$

$$E_i(\beta) = (\beta^T\sigma(Wx_i + b) - t_i)^2$$

We want to prove the convergence of the gradient descent method with fixed step size η. We are going to assume that the function $E(\beta)$ is twice differentiable, formally $E(\beta) \in C^2$, also that $E(\beta)$ is strongly convex and L-smooth. We can derive the first and second derivative of our objective function as follow:

$$\nabla E(\beta) = \frac{2}{N} \sum_{i=1}^{N} \sigma(Wx_i + b)(\beta^T \sigma(Wx_i + b) - t_i)^T + \frac{2\lambda}{N} \beta$$

$$\nabla^2 E(\beta) = \frac{2}{N} \sum_{i=1}^{N} \sigma(Wx_i + b)(\sigma(Wx_i + b))^T + \frac{2\lambda}{N} I$$

It can be seen that $\nabla^2 E(\beta)$ is a constant matrix, so we can choose $L = \|\nabla^2 E(\beta)\|$ such that $\nabla^2 E(\beta) \preccurlyeq LI$. This is equivalent to Lipschitz continuity of the gradient, formally: [4, 13]

$$\|\nabla E(v) - \nabla E(w)\| \leq L\|v - w\| \; \forall v, w \in S$$

A differentiable function is said to be L–smooth if its gradient is Lipschitz continuous. Hence, $E(\beta)$ is L-smooth.

The strong convexity of $E(\beta)$ holds if $\exists m > 0$ such that $g(x) = f(x) - \frac{m}{2}\|x\|^2$ is convex for all x. [7]

We define $g(\beta)$ as:

$$g(\beta) = \frac{1}{N} \sum_{i=1}^{N} (\beta^T \sigma(Wx_i + b) - t_i)^2 + (\frac{\lambda}{N} - \frac{m}{2})\|\beta\|^2$$

We can deduce that $g(\beta)$ is convex if both terms on the right side of the equation are convex, because convexity is preserved under non-negative weighted sum. Mean-Squared Error is convex with respect to the output, in this case, the output is $\beta^T \sigma(Wx_i + b)$. $\beta^T \sigma(Wx_i + b)$ is a linear function with respect to β, which means it is convex. Convexity is preserved under composition with affine function, so the first term is convex. The second term is convex with $0 < m < \frac{2\lambda}{N}$.

Since $E(\beta)$ is differentiable and strongly convex, a necessary and sufficient condition for $\beta^\star$ to be an optimal solution is that $\nabla E(\beta^\star) = 0$. In other words, the algorithm will never find itself stuck in a local minima or a saddle point, because in convex functions, any stationary point is a global minima.

We can now prove the convergence of the gradient descent with fixed step size η. The update rule for the weights β in every iteration is as follows:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \nabla E(\beta^{(t)})$$

If we choose the step size $\eta \leq \frac{1}{L}$, the algorithm will converge according to: [2]

$$\|\beta^{(t+1)} - \beta^\star\|^2 \leq (1 - \eta m)^{t+1} \|\beta^{(0)} - \beta^\star\|^2$$

To prove it, let's compute $\|\beta^{(t+1)} - \beta^\star\|^2$.

$$\|\beta^{(t+1)}-\beta^\star\|^2 = \|\beta^{(t)}-\eta\nabla E(\beta^{(t)})-\beta^\star\|^2 = \|\beta^{(t)}-\beta^\star\|^2-2\eta\langle\nabla E(\beta^{(t)}), \beta^{(t)}-\beta^\star\rangle+\eta^2\|\nabla E(\beta^{(t)})\|^2$$

According to the definition of strong convexity, and since $E(\beta)$ is strongly convex, we have that for $\forall v, w$:

$$E(v) \geq E(w) + \langle E(w), (v-w)\rangle + \frac{m}{2}\|v-w\|^2$$

Using $v = \beta^{(t)}$ and $w = \beta^\star$, we obtain:

$$\|\beta^{(t+1)} - \beta^\star\|^2 \leq (1-\eta m)\|\beta^{(t)} - \beta^\star\|^2 - 2\eta(E(\beta^{(t)}) - E(\beta^\star)) + \eta^2\|\nabla E(\beta^{(t)})\|^2$$

Lemma 1 *If f is L-smooth then* $f(x^\star) - f(x) \leq -\frac{1}{2L}\|\nabla f(x)\|^2$

Using Lemma 1, we obtain

$$\|\beta^{(t+1)} - \beta^\star\|^2 \leq (1-\eta m)\|\beta^{(t)} - \beta^\star\|^2 - 2\eta(E(\beta^{(t)}) - E(\beta^\star)) + 2\eta^2 L(E(\beta^{(t)}) - E(\beta^\star)) =$$

$$(1-\eta m)\|\beta^{(t)} - \beta^\star\|^2 - 2\eta(1-\eta L)(E(\beta^{(t)}) - E(\beta^\star))$$

If $\eta \leq \frac{1}{L}$ then $-2\eta(1-\eta L)$ is negative, so the last term of above inequality can be dropped.

$$\|\beta^{(t+1)} - \beta^\star\|^2 \leq (1-\eta m)\|\beta^{(t)} - \beta^\star\|^2$$
$$\frac{\|\beta^{(t+1)} - \beta^\star\|^2}{\|\beta^{(t)} - \beta^\star\|^2} \leq (1-\eta m)$$
$$\frac{\|\beta^{(t+1)} - \beta^\star\|^2}{\|\beta^{(0)} - \beta^\star\|^2} \leq (1-\eta m)^{t+1}$$

The inequality above means that the algorithm converges with rate $O(c^k)$ with $0 \leq c \leq 1$ (k is number of iteration). This type of rate is called linear convergence. Note that determining L to pick $\eta \leq \frac{1}{L}$ is hard, so one can determine step size by line search. However, doing line search does not produce a significant improvement on the convergence rate [3], and in our problem, L can be determined. So, in this project, we use a fixed step size.

2.1.3 Momentum

Momentum is used to improve the convergence speed of gradient descent while maintaining the same level of accuracy. It works by memorizing the previous $\Delta\beta$ and use it to influence the next value for β. Initially $\Delta\beta^{(0)} \in R^{h\times m}$ is a matrix of zeros and $\alpha \in [0,1]$, classical momentum gives us the following update rule:

$$\Delta\beta^{(t)} = \alpha\Delta\beta^{(t-1)} - \eta\nabla E(\beta^{(t-1)})$$

$$\beta^{(t)} = \beta^{(t-1)} + \Delta\beta^{(t)}$$

where $\eta > 0$ is the learning rate and α controls the decay of the velocity vector. When α is closer to 1, the previous gradient will affect the current direction more, while values closer to 0 will tend to classical gradient descent. Afterwards, we set $\Delta\beta^{(t-1)} = \Delta\beta^{(t)}$.

In our project we will use a variant of Classical Momentum called Nesterov's Accelerated Momentum. It differs from classical momentum by providing a particular formula for the learning rate η and momentum constant α, which are needed to achieve the convergence properties described in the next section. The next difference is for the update mechanism of the velocity vector $\Delta\beta^{(t)}$.

In the following analysis, we will show that if η and α are updated at each iteration t following certain equations, the algorithm will converge with linear convergence rate. [14]

2.1.4 Convergence of Nesterov's Accelerated Momentum

We initialize the algorithm by setting $a^{(0)} = 1$, $\beta^{(0)} = y_0$, where y_0 is an arbitrary parameter, z to an arbitrary value and $\eta^{(0)} = \frac{\|y_0 - z\|}{\|\nabla E(y_0) - \nabla E(z)\|}$. The latter equation for η has the goal of adapting the learning rate to always be smaller than the reciprocal of the "observed" Lipschitz coefficient. However if one knows that the function $E(\beta)$ is L-Lipschitz continuos then we can use a fixed learning rate given by $\eta^{(t)} = \frac{1}{L}, \forall t$. At each iteration the parameters are updated as follows:

$$\eta^{(t)} = 2^{-i}\eta^{(t-1)}$$

where i is the smallest positive integer for which

$$E(y^{(t)}) - E(y^{(t)} - 2^{-i}\eta^{(t-1)}\nabla E(y^{(t)})) \geq 2^{-i}\eta^{(t-1)}\frac{\|\nabla E(y^{(t)})\|^2}{2}$$

$$\beta^{(t)} = y^{(t)} - \eta^{(t)}\nabla E(y^{(t)}) \tag{5}$$

$$a^{(t+1)} = \frac{1 + \sqrt{4(a^{(t)})^2 + 1}}{2}$$

$$y^{(t+1)} = \beta^{(t)} + (a^{(t)} - 1)\frac{\beta^{(t)} - \beta^{(t-1)}}{a^{(t+1)}} \tag{6}$$

To understand the sequence $a^{(t)}$, we note that the function $g(x) = \frac{1+\sqrt{4x^2+1}}{2}$ tends to $x + 0.5$ as $x \to \infty$, so $a^{(t)} \approx \frac{t+4}{2}$ for large t, and thus the $\frac{a^{(t)}-1}{a^{(t+1)}}$ from the last equation (6) behaves like $1 - \frac{3}{t+5}$. The corrections on the velocity vector $\Delta\beta$ and the momentum parameter α are:

$$\Delta\beta^{(t)} = \beta^{(t)} - \beta^{(t-1)} \tag{7}$$

$$\alpha^{(t)} = \frac{a^{(t)} - 1}{a^{(t+1)}}$$

Combining the last equation for the momentum factor with equation (6) we get:

$$y^{(t)} = \beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{(t-1)}$$

which can be used to rewrite equation (5) as follows:

$$\beta^{(t)} = \beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{(t-1)} - \eta^{(t-1)}\nabla E(\beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{(t-1)})$$

From equation (7) we get:

$$\Delta\beta^{(t)} = \alpha^{(t-1)}\Delta\beta^{(t-1)} - \eta^{(t-1)}\nabla E(\beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{(t-1)}) \tag{8}$$

Nesterov showed that if E is convex with an L-Lipschitz continuos derivative, then the method satisfies the following: [14, 1]

$$E(\beta^{(t)}) - E(\beta^{\star}) \leq \frac{4L\|\beta^{(0)} - \beta^{\star}\|^2}{(t+2)^2}$$

It has been proved that, among first-order methods, Nesterov accelerated momentum achieves the fastest linear convergences rate and converges at a rate of $(1 - \sqrt{\frac{\mu}{L}})^{k/2}$, where μ is the strong convexity parameter and k is the number of iterations. [16, 10, 1]
The final update rule is derived by adding the L2 regularization to equation (8), obtaining:

$$\Delta\beta^{(t)} = \alpha^{(t-1)}\Delta\beta^{(t-1)} - \eta^{(t-1)}\nabla E(\beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{(t-1)}) - 2\frac{\lambda}{N}\beta^{(t-1)}$$

$$\beta^{(t)} = \beta^{(t-1)} + \Delta\beta^{(t)}$$

2.1.5 Complexity of Nesterov's Accelerated Gradient

Each iteration of the Nesterov's Accelerated Gradient method is composed of two main computations:

1. Direction. The direction is derived by computing $\nabla E(\beta)$, which costs $O(hm)$.

2. Change in direction. It computes the sum of two matrices in $R^{h\times m}$, both derived from scalar-matrix multiplication, so the cost is $O(hm)$.

It follows that the iteration cost of this method is $O(hm)$.

2.2 Quasi-Newton Method

The problem of gradient descent derives from its slow convergence. To improve the convergence speed of our optimization algorithm, we must search for a better direction to the minima. This is done by the Newton method. The weights β are updated in each iteration as equation (9). Note that in this section, β is the vectorization of our original matrix β, thus $\beta \in R^{hm}$.

$$\Delta\beta = -\eta(\nabla^2 E(\beta))^{-1}\nabla E(\beta) \tag{9}$$

However, computing the inverse of the second derivative is costly. The total cost of implementing Newton's method is proportional to $O(d^3)$, where d is the input dimension. This complexity is almost intractable for large problems, especially compared to the iteration cost of the first-order methods that is $O(d)$ [10]. So in this project, instead of computing the exact inverse Hessian $(\nabla^2 E(\beta))^{-1}$, we will use its approximation $\mathbf{B}^{(t)} \approx (\nabla^2 E(\beta^{(t)}))^{-1}$. This method is called quasi-Newton.

To make sure quasi-Newton works, the direction $p^{(t)} = -\mathbf{B}^{(t)}\nabla E(\beta^{(t)})$ must be a descent direction. This can be achieved if $\mathbf{B}^{(t)}$ is symmetric positive definite, because

$$\langle p^{(t)}, \nabla E(\beta^{(t)})\rangle = -\nabla E(\beta^{(t)})^T \mathbf{B}^{(t)} \nabla E(\beta^{(t)}) < 0.$$

We use the BFGS algorithm which computes the inverse Hessian approximation for each iteration as

$$\mathbf{B}^{(t)} = \mathbf{B}^{(t-1)} + \Delta\mathbf{B}^{(t)} \tag{10a}$$

$$\Delta\mathbf{B}^{(t)} = \frac{1}{y^T s}[(1 + \frac{1}{y^T s} y^T \mathbf{B}^{(t-1)} y) s s^T - (\mathbf{B}^{(t-1)} y s^T + s y^T \mathbf{B}^{(t-1)})] \tag{10b}$$

where $y = \nabla E(\beta^{(t)}) - \nabla E(\beta^{(t-1)})$ and $s = \beta^{(t)} - \beta^{(t-1)}$ [8]. $\mathbf{B}^{(t)}$ will be symmetric positive definite if $\mathbf{B}^{(t-1)}$ is symmetric positive definite and y and s satisfy the curvature condition

$$s^T y < 0.$$

Curvature condition is guaranteed to hold if the step size η satisfies the Wolfe or strong Wolfe conditions [11]. Hence, inexact line search is commonly done for BFGS to work. This process tries to find η that satisfies

$$E(\beta^{(t)} + \eta p^{(t)}) \leq E(\beta^{(t)}) + c_1 \eta \nabla E(\beta^{(t)})^T p^{(t)} \tag{11a}$$

$$\nabla E(\beta^{(t)} + \eta p^{(t)})^T p^{(t)} \geq c_2 \nabla E(\beta^{(t)})^T p^{(t)} \tag{11b}$$

with $0 < c_1 < c_2 < 1$. The value of c_1 is typically very small, for example 10^{-4}, while usually $c_2 = 0.9$. When the objective function is strongly convex, the curvature condition is already guaranteed to hold, but a step size satisfying Wolfe condition is important to attain a fast rate of convergence.

An issue in the BFGS algorithm is determining the initial inverse Hessian approximation $\mathbf{B}^{(0)}$. It is often set as a multiple of identity matrix, but there is no good general strategy for choosing the multiplier. One may use $\mathbf{B}^{(0)} = \delta \left\|\nabla E(\beta^{(0)})\right\|^{-1} I$, where δ is the norm of the first step we want. Other choice is by setting $\mathbf{B}^{(0)} = I$, then before equation (10a) is applied to obtain $\mathbf{B}^{(1)}$, $\mathbf{B}^{(0)}$ is updated with

$$\mathbf{B}^{(0)} = \frac{y^T s}{y^T y} I.$$

2.2.1 Convergence of BFGS

First, we will prove that the sequence $\{\beta^{(t)}\}$ generated by the BFGS algorithm converges to $\beta^\star$. Note that the theorems used in this proof hold when the step size η satisfies the Wolfe or strong Wolfe conditions, and the initial step size $\eta^{(0)}$, which is the parameter of the line search algorithm, is 1.

Theorem 1 *Suppose the objective function f is twice continuously differentiable. Let B_0 be any symmetric positive definite initial matrix, and let x_0 be a starting point for which the level set $L = \{x \in R^n \mid f(x) \leq f(x_0)\}$ is convex, and there exists positive constants m and M such that*

$$m\|z\|^2 \leq z^T H(x) z \leq M\|z\|^2 \tag{12}$$

for all $z \in R^n$ and $x \in L$ (H is the Hessian of f). Then, the sequence $\{x_k\}$ generated by the BFGS algorithm converges to the minimizer $x^\star$ of f [11].

We already established that the objective function $E(\beta)$ is twice continuously differentiable. Since $E(\beta)$ is convex, then for an arbitrary starting point $\beta^{(0)}$, the level set $L = \{\beta \mid E(\beta) \leq E(\beta^{(0)})\}$ is convex. The Hessian $\nabla^2 E(\beta)$ is a constant function, so (12) is satisfied. Hence, we can conclude that the BFGS algorithm converges for $E(\beta)$.

Theorem 2 *Suppose f is twice continuously differentiable and that the iterates generated by the BFGS algorithm converge to an optimal solution $x^\star$. Suppose also the Hessian matrix of f is Lipschitz continuous at $x^\star$, then the iterates converge to $x^\star$ at a superlinear rate* [11].

$E(\beta)$ is twice continuously differentiable and we have proved that the BFGS algorithm converges for $E(\beta)$. We need to know whether $\nabla^2 E(\beta)$ is Lipschitz continuous at $\beta^\star$.

A Hessian matrix H is Lipschitz continuous at $x^\star$ if for all x near $x^\star$, there exists $M > 0$ such that $\|H(x) - H(x^\star)\| \leq M\|x - x^\star\|$. Since $\nabla^2 E(\beta)$ is a constant function, then for any β and $M > 0$,

$$\|\nabla^2 E(\beta) - \nabla^2 E(\beta^\star)\| = 0 \leq M\|\beta - \beta^\star\|.$$

Since $E(\beta)$ satisfies all preconditions in Theorem 2, then the BFGS algorithm converges to $\beta^\star$ at a superlinear rate, that is

$$\lim_{t\to\infty} \frac{E(\beta^{(t+1)}) - E(\beta^\star)}{E(\beta^{(t)}) - E(\beta^\star)} = 0.$$

This rate is achieved from iteration t_o, where for all $t \geq t_o$, $\eta^{(t)} = 1$ always satisfies the Wolfe condition. This is why it is important to set $\eta^{(0)} = 1$, so the line search algorithm always outputs 1 when it satisfies the Wolfe condition.

2.2.2 Complexity of BFGS

Each iteration of the quasi-Newton method computes three main things: direction, step size, and inverse Hessian approximation.

1. Direction. Direction is computed as a matrix-vector multiplication, which is $p = -\mathbf{B}\nabla E(\beta)$, where $\mathbf{B} \in R^{hm \times hm}$ and $\nabla E(\beta) \in R^{hm}$. The cost for this step is $O(h^2m^2)$.

2. Step size. We use two line search algorithms: backtracking line search (BLS) and Armijo-Wolfe line search (AWLS) with quadratic interpolation. Both algorithms evaluate the objective function in every iteration until it finds the optimal step size, but AWLS also computes the first derivative. So, AWLS is more costly, but should produce a more optimal step size, because it satisfies condition (11a) and (11b), unlike BLS which only search for a step satisfying (11a).

3. Inverse Hessian approximation. The main computation of (10b) is a matrix-vector multiplication, where $\mathbf{B} \in R^{hm \times hm}$ and $y, s \in R^{hm}$. The cost for this step is $O(h^2m^2)$.

We can conclude that the complexity of BFGS is $O(h^2m^2)$, with an additional cost of function evaluation and its first derivative, i.e. it is quadratic to the number of hidden units and the output dimension.

3 Algorithm Implementation and Development

The algorithms were implemented using the MATLAB language, the project is divided into the following files:

- ObjectiveFunc.m Implementation of the Mean-Squared-Error with L_2 regularization; it returns, given a point, the error and the gradient at that point.
- NAG.m Optimizes a given objective function using Nesterov's Accerated Gradient.
- grid_search.m Interface for the NAG optimizer, it tries different values for α and λ.
- BFGS.m Optimizes a given objective function using a Quasi-Newton method.

Algorithm 1: NAG

```
Require: starting point $\beta^{(-1)}$
Require: learning rate $\eta = \frac{1}{L}$
Require: convergence tolerance $\varepsilon > 0$
  $t \leftarrow 0$
  $\Delta\beta^{(-1)} \leftarrow \mathbf{0}$
  $a^{(-1)} \leftarrow 1$
  while $\|\nabla E(\beta^{(t)})\|/\|\nabla E(\beta^{(0)})\| > \varepsilon$ do
    $a^{(t)} \leftarrow \frac{1+\sqrt{4a^{(t-1)^2}+1}}{2}$
    $\alpha^{(t-1)} \leftarrow \frac{a^{(t-1)}-1}{a^{(t)}}$
    $\Delta\beta^{(t)} \leftarrow \alpha^{(t-1)}\Delta\beta^{(t-1)} - \eta\nabla E(\beta^{(t-1)} + \alpha^{(t-1)}\Delta\beta^{t-1)})$
    $\beta^{(t)} \leftarrow \beta^{(t-1)} + \Delta\beta^{(t)}$
    $a^{(t-1)} \leftarrow a^{(t)}$
    $\Delta\beta^{(t-1)} \leftarrow \Delta\beta^{(t)}$
    $t \leftarrow t + 1$
  end while
```

Algorithm 2: BFGS

```
Require: starting point $\beta^{(0)}$
Require: convergence tolerance $\varepsilon > 0$
Require: inverse Hessian approximation $\mathbf{B}^{(0)}$
  $t \leftarrow 0$
  while $\|\nabla E(\beta^{(t)})\| / \|\nabla E(\beta^{(0)})\| > \varepsilon$ do
    $p^{(t)} \leftarrow -\mathbf{B}^{(t)} \nabla E(\beta^{(t)})$
    $\eta^{(t)} \leftarrow$ line search procedure to satisfy Wolfe condition
    $\beta^{(t+1)} \leftarrow \beta^{(t)} + \eta^{(t)} p^{(t)}$
    $s^{(t)} \leftarrow \beta^{(t+1)} - \beta^{(t)}$ and $y^{(t)} \leftarrow \nabla E(\beta^{(t+1)}) - \nabla E(\beta^{(t)})$
    Compute $\mathbf{B}^{(t+1)}$ by means of (10a) and (10b)
    $t \leftarrow t + 1$
  end while
```

4 Experiments Results

The algorithms are tested on 3 different datasets:

1. Sine function
2. MONK [9]
3. Random matrix

We analyze for each dataset the following properties: the computation time per iteration, the number of iterations to reach a certain value, the convergence rate, how close the solution to the true solution is, and the scalability of Nesterov's Accelerated Gradient and BFGS. Before going into detail of the three datasets, we will describe how to obtain the true solution in the following section.

4.1 True Solution

For each dataset, the true solution is given by a closed formula derived by setting $\nabla E(\beta) = 0$. We can derive it as follows:

$$\nabla E(\beta) = 0$$

$$\frac{2}{N}\sum_{i=1}^{N}(\sigma(Wx_i+b)(\beta^T\sigma(Wx_i+b)-t_i)^T)+\frac{2\lambda}{N}\beta = 0$$

$$\sum_{i=1}^{N}(\sigma(Wx_i+b)(\beta^T\sigma(Wx_i+b)-t_i)^T)+\lambda\beta = 0$$

By setting $a_i = \sigma(Wx_i+b)$ we get:

$$\sum_{i=1}^{N}(a_i(\beta^T a_i - t_i)^T)+\lambda\beta = 0$$

$$\sum_{i=1}^{N}(a_i(a_i^T\beta - t_i^T))+\lambda\beta = 0$$

$$\sum_{i=1}^{N}(a_ia_i^T\beta - a_it_i^T)+\lambda\beta = 0$$

$$\sum_{i=1}^{N}(a_ia_i^T\beta) - \sum_{i=1}^{N}(a_it_i^T)+\lambda\beta = 0$$

$$(\sum_{i=1}^{N}(a_ia_i^T)+\lambda I)\beta = \sum_{i=1}^{N}(a_it_i^T)$$

$$\beta = (\sum_{i=1}^{N}(a_i a_i^T) + \lambda I)^{-1} \sum_{i=1}^{N}(a_i t_i^T) \tag{13}$$

4.2 Sine Function

The dataset contains 100 points of the form $(x, sin(x))$, where $x \in \mathbb{R}$ is the input and $sin(x) \in \mathbb{R}$ is the output. We can visualize the dataset in figure 1:

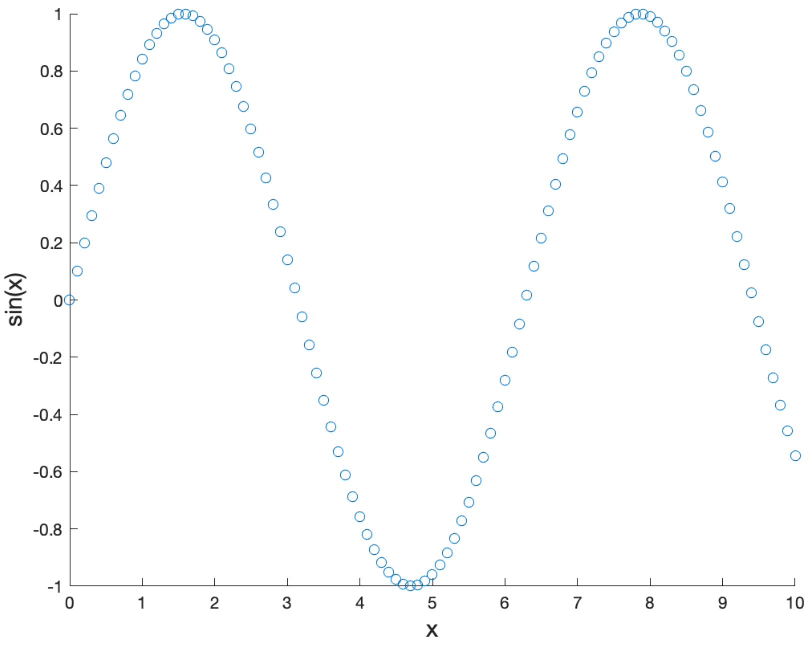


Figure 1: The dataset for the $sin(x)$ function

4.2.1 Training

We set the number of hidden units h equal to the number of samples in the training set and collect the optimization results in the table 1. We first derive the true solution $E^\star$ and then test the optimization algorithms with three different thresholds for the stopping condition. The threshold is given by the relative error $(E(\beta^\star) - E^\star)/E^\star$, which basically indicates the precision of the best solution found by the algorithms $E(\beta^\star)$ with respect to the true solution. On a technical note, when the true solution $E^\star$ is close to zero the computation of the relative error is changed into $E(\beta^\star) - E^\star$.

For every configuration, we analyze the time per iteration by repeating the same optimization algorithm 100 times and compute its total execution time. This ensures that the elapsed seconds are $\geq 10^{-1}$, which is the precision time for the **tic** function in MATLAB.

The total time is then divided by 100 to get the time taken by the optimization algorithm, and then divided by the number of iterations taken in order to get the time per iteration shown in the table. Also, since the training data has no noise, we set $\lambda = 0$.

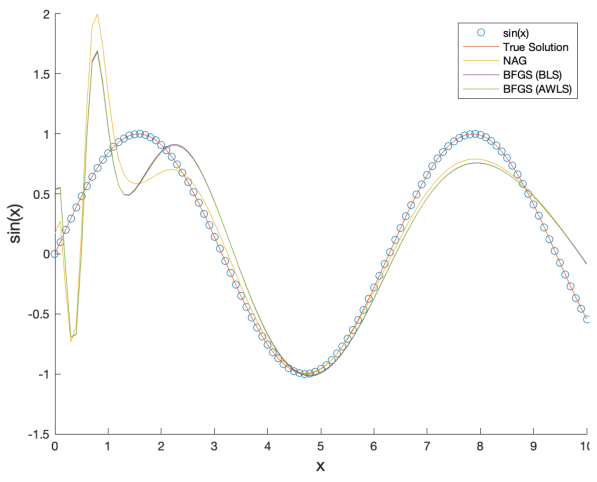
Optimizer	**h**	**Step size**	**Time/iter[s]**	**#Iterations**	$(E(\beta^\star) - E^\star)/E^\star$
NAG	100	$1/L$	1.112623×10^{-3}	94	9.942142×10^{-2}
BFGS	100	BLS	1.550070×10^{-3}	31	9.457204×10^{-2}
BFGS	100	AWLS	1.390907×10^{-3}	31	9.552601×10^{-2}
NAG	100	$1/L$	8.548285×10^{-4}	950	9.994368×10^{-4}
BFGS	100	BLS	7.621043×10^{-4}	88	8.848556×10^{-4}
BFGS	100	AWLS	6.611822×10^{-4}	88	8.848640×10^{-4}
NAG	100	$1/L$	7.665144×10^{-4}	79172	9.999873×10^{-7}
BFGS	100	BLS	6.279796×10^{-4}	285	9.966062×10^{-7}
BFGS	100	AWLS	5.671104×10^{-4}	285	9.965865×10^{-7}

Table 1: Measurements of the computational time per iteration, number of iterations and optimal value for the sin(x) dataset when the relative error is $\leq \{10^{-1}, 10^{-3}, 10^{-6}\}$

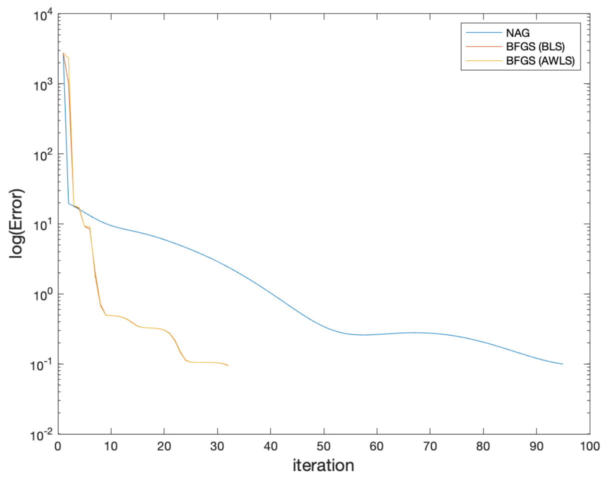
We can see that BFGS consistently does fewer iterations to reach the same relative error of NAG.

4.2.2 Convergence Rate and Predictions

In this section we graphically show the results found in the previous table 1, for each threshold of the relative error we plot the predictions of the trained model and also the rate of convergence. The latter is computed by taking the difference from the error at an iteration t with the optimal error given by the true solution. Formally, this mean computing $E(\beta^{(t)}) - E(\beta^\star)$ for each iteration t, the results are then shown on a logarithmic scale. NAG shows an almost linear convergence at the tail, while BFGS shows a superlinear convergence at the tail.

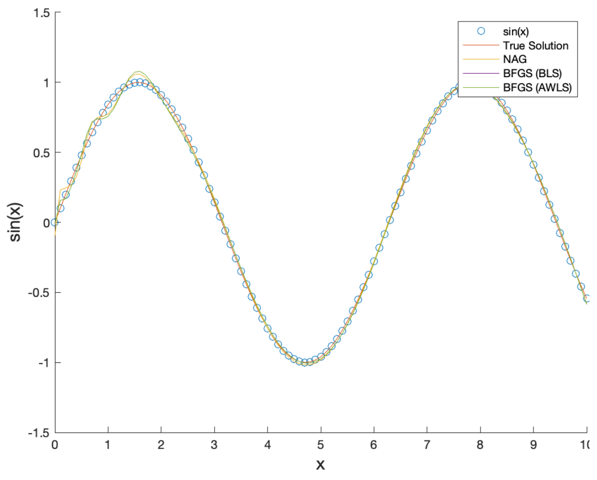


(a) Plot of predictions

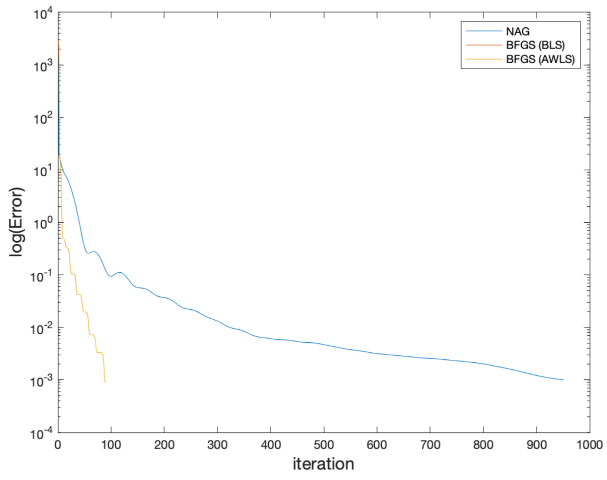


(b) Convergence Rate

Figure 2: relative error $\leq 10^{-1}$, h = 100



(a) Plot of predictions



(b) Convergence Rate

Figure 3: relative error $\leq 10^{-3}$, h = 100

Focusing on the convergence rate of BFGS only:

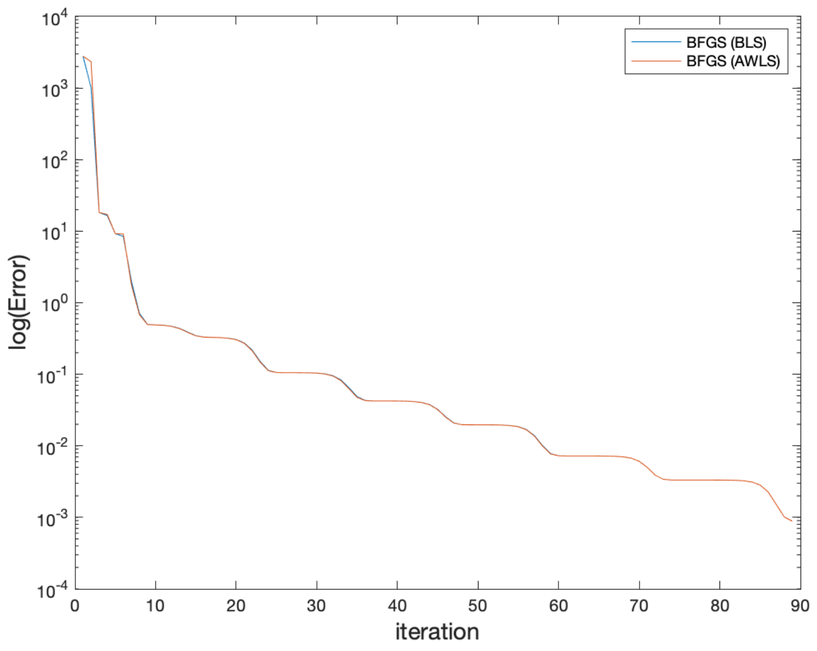
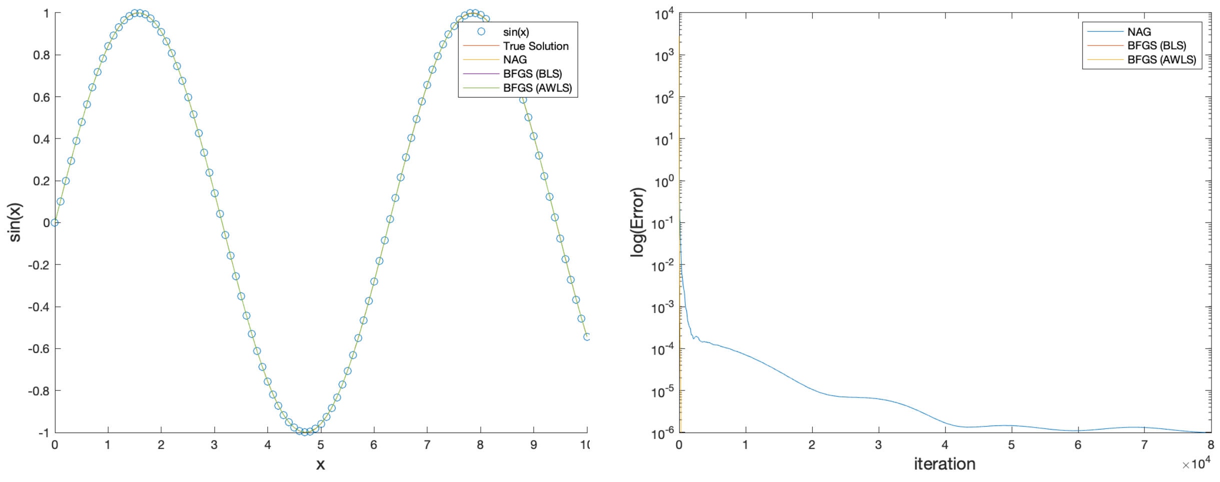


Figure 4: BFGS Convergence Rate



(a) Plot of predictions

(b) Convergence Rate

Figure 5: relative error $\leq 10^{-6}$, h = 100

Focusing on the convergence rate of BFGS only:

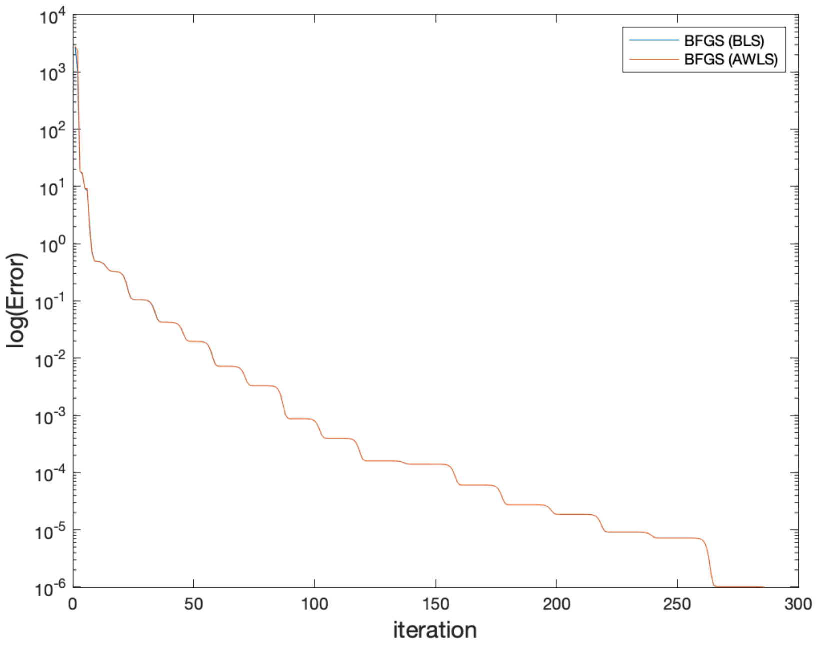


Figure 6: BFGS Convergence Rate

4.2.3 Scalability

In order to see what happens to the computational time as the number of hidden units increases we constructed an experiment where we sampled 10 values of h uniformly distributed between $[100; 10000]$, we then compute the time taken per iteration in the same way we did for the results in the table 1 and plot the results in figure 7. As expected, the execution time increases linearly for NAG and quadratically for BFGS. There also isn't a practical difference in the line search chosen for BFGS, both BLS and AWLS performs similarly in our test.

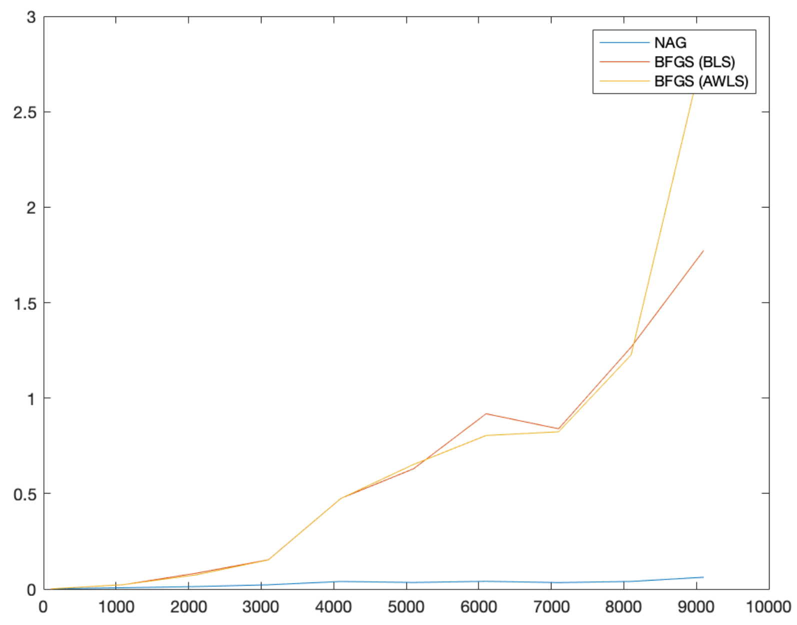


Figure 7: Computational time per iteration versus the number of hidden units

In the last result, by taking into consideration only the time required to compute one iteration, it seems that NAG is faster than BFGS. In practise, this is generally not true, because we are not interested in the time required to do one iteration but rather in the time it takes to reach a certain accuracy. To take into account this last property, we plot in Figure 8 and 9 the total time in seconds to reach a relative error of 10^{-3} and 10^{-7} versus the number of hidden units h. We can see that in Figure 8, when $h < 4000$ BFGS takes less time to converge to the predefined accuracy, even if the time per iteration is larger. As the relative error is decreased, NAG takes far more iteration to converge, thus become much slower than BFGS.

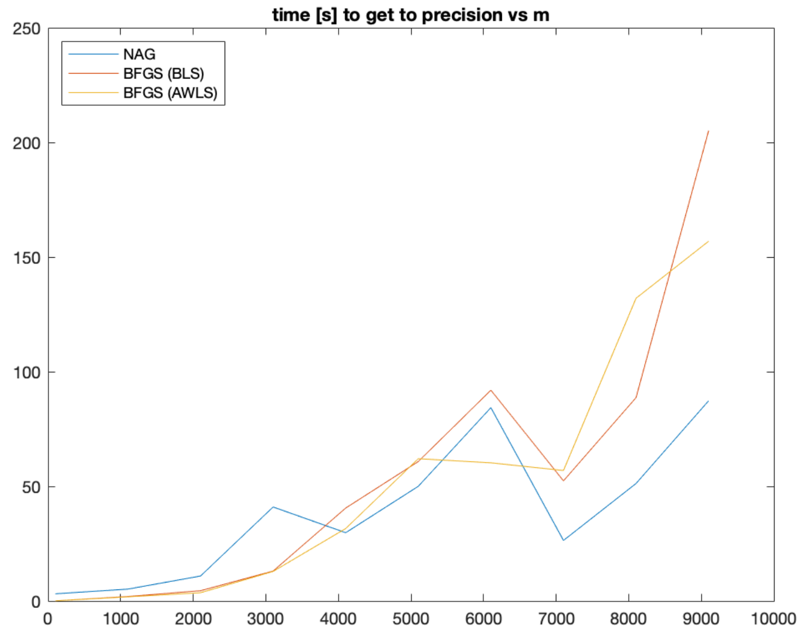
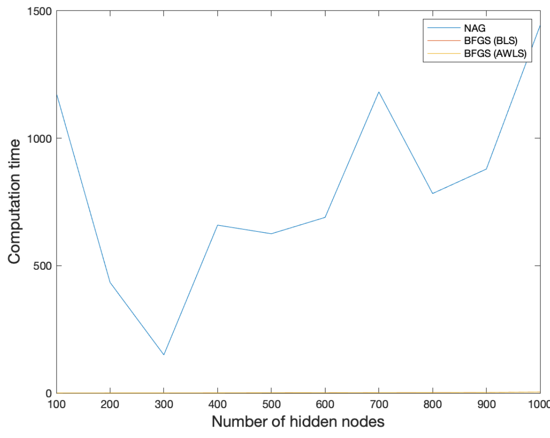
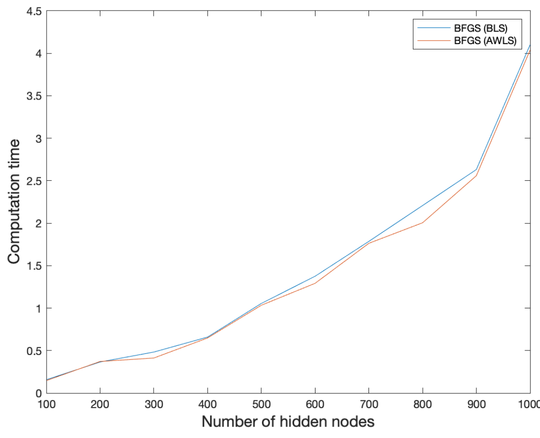


Figure 8: Computational time to reach a relative error of 10^{-3} versus the number of hidden units h



(a) NAG and BFGS



(b) BFGS

Figure 9: Computational time to reach a relative error of 10^{-7} versus the number of hidden units h

4.2.4 Noise

Measurements in the real world are never perfect and some noise is the data is always present. In this section we analyze a controlled experiment where the introduced error on the $sin(x)$ is known and we study the effect of the L2 regularizer λ on the model. The error is introduced in every sample point of the dataset and varies from $[-0.5; +0.5]$. In figure 12 we plot the original sin(x) dataset versus the new dataset containing noise.

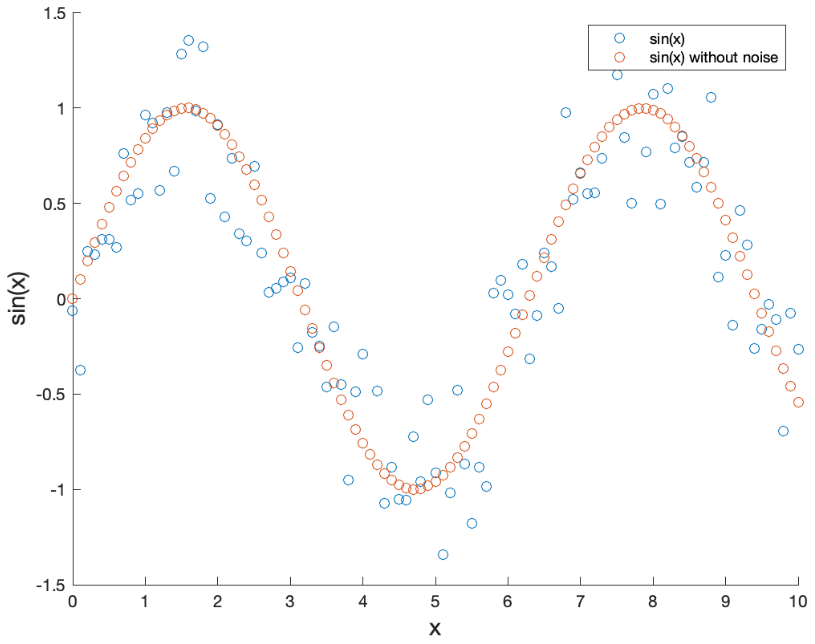


Figure 10: sin(x) dataset with noise

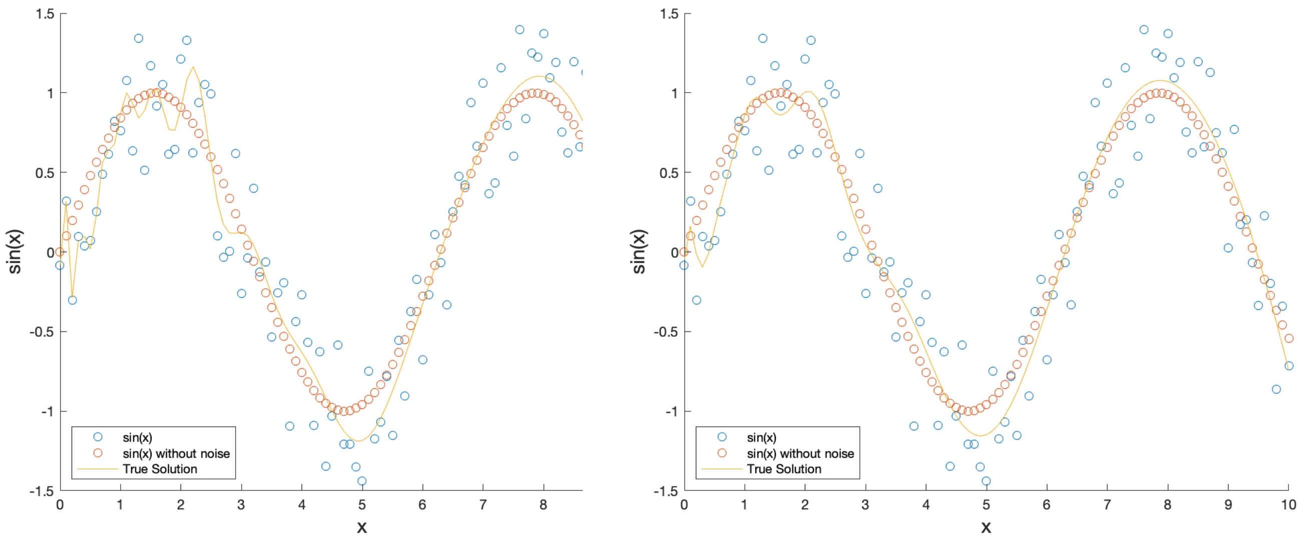
The model is now trained on the noisy data and we report the results for different choices of λ in table 2. The true solution $E^{\star}$ is computed both on the noisy data, shown as **E*** **noise**, and on the real $sin(x)$, shown as **E*** **real**.

λ	$E^{\star}$ **noise**	$(E(\beta) - E^{\star})/E^{\star}$ **noise**	$E^{\star}$ **real**	$(E(\beta) - E^{\star})/E^{\star}$ **real**
0	6.491617×10^{-2}	2.185719×10^{-2}	2.491112×10^{-2}	2.158690×10^{-2}
10^{-5}	5.923412×10^{-2}	8.299871×10^{-3}	1.425164×10^{-2}	3.307295×10^{-3}
10^{-4}	6.054717×10^{-2}	7.524480×10^{-3}	1.440434×10^{-2}	3.689380×10^{-3}
10^{-3}	6.454987×10^{-2}	8.759276×10^{-3}	1.579581×10^{-2}	7.511719×10^{-3}
10^{-2}	7.796378×10^{-2}	3.565953×10^{-2}	2.668131×10^{-2}	3.707162×10^{-2}
10^{-1}	1.296719×10^{-1}	5.961345×10^{-3}	7.911807×10^{-2}	7.482534×10^{-3}
1	1.960611×10^{-1}	2.346872×10^{-5}	1.597265×10^{-1}	1.572784×10^{-4}

Table 2: True solution on noise sin(x) with h = 100

We can observe that **E*** **noise** generally increases as $\lambda > 0$. But, the interesting observation is that the real error **E*** **real** decreases for some $\lambda > 0$ obtaining its smallest value when $\lambda = 10^{-5}$. We also report the relative error on the noisy data and on the real function obtained by BFGS, using the AWLS. While the results from NAG are not reported due to the matrix a_i being close to singular and so it would take a large number of iterations to obtain a solution worth reporting.

Intuitively we can describe the effect of λ as a sort of smoothing, best described by looking at the figure 11. We can see that when $\lambda = 0$ the solution fits the noisy data diverging from the true values. While the solution found with $\lambda = 10^{-6}$ is closer to the original data, thus obtaining a smaller Mean-Squared-Error on the data without noise. Note that the true solution doesn't perfectly fit the data due to the matrix a_i (defined in 13) being close to singular.



(a) $\lambda = 0$ (b) $\lambda = 10^{-6}$

Figure 11: h = 100

In figure 12 we can also see the predictions from the model trained with BFGS with $\lambda = 0$ and $\lambda = 10^{-5}$.

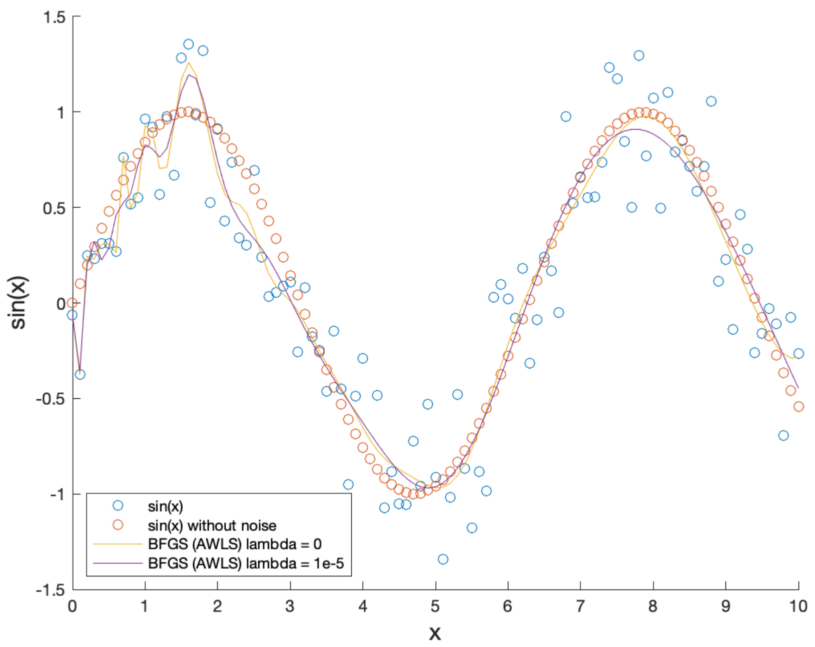


Figure 12: BFGS model predictions trained until the relative error $> 10^{-1}$ and $\lambda \in \{10^{-5}, 0\}$

It is clear that these experiments are not realistic since in reality the error isn't known, the best one can do is to use heuristics to estimate the possible error in the data. In ML this is usually estimated by partitioning the data into a training set and a test set, the latter is only used for hyperparameter selection.

4.3 MONK

The MONK dataset contains three binary classification tasks, each having six categorical attributes. Before training, we applied one-hot encoding to all three tasks, transforming all attributes into 17 numeric attributes. It is known that only the third task has data that contains noise, so the value of λ (regularization parameter) for the first and second task is 0. For the third task, we performed grid search and obtained 0.01 as the best value of λ. We set the number of hidden nodes to be larger than the number of samples to obtain 100% accuracy. This configuration, along with the true solution, can be seen in Table 3.

Table 4 shows the number of iteration performed by NAG and BFGS to reach a relative error under 10^{-8}, or $((E(\beta^\star) - E^\star)/E^\star) < 10^{-8}$, along with the average computational time per iteration. We can observe that BFGS requires less iterations compared to NAG to achieve the same accuracy. Between the two inexact line search methods in BFGS, AWLS requires slightly less iterations than BLS. In terms of computational time per iteration, BFGS with AWLS is consistently faster, followed by BFGS with BLS then NAG.

Task	**#Hidden nodes**	λ	$E^\star$
MONK1	150	0	2.638862×10^{-25}
MONK2	200	0	4.132223×10^{-24}
MONK3	150	0.01	1.087361×10^{-3}

Table 3: Configuration and true solution of MONK datasets.

Task	**Optimizer**	**Step size**	**#Iterations**	**Time per iter [s]**	$(E(\beta^\star) - E^\star)/E^\star$
MONK1	NAG	$1/L$	18726	1.001152×10^{-3}	9.948071×10^{-9}
	BFGS	BLS	930	6.652207×10^{-4}	9.841591×10^{-9}
	BFGS	AWLS	927	6.501318×10^{-4}	9.821793×10^{-9}
MONK2	NAG	$1/L$	50847	1.761510×10^{-3}	9.973003×10^{-9}
	BFGS	BLS	1068	1.149862×10^{-3}	9.676016×10^{-9}
	BFGS	AWLS	1066	1.052728×10^{-3}	9.685004×10^{-9}
MONK3	NAG	$1/L$	18266	9.848436×10^{-4}	9.995019×10^{-9}
	BFGS	BLS	820	6.517204×10^{-4}	9.876909×10^{-9}
	BFGS	AWLS	817	6.464575×10^{-4}	9.248982×10^{-9}

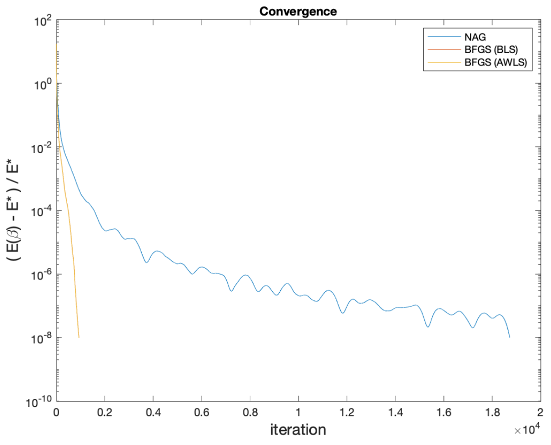
Table 4: Number of iterations, computation time, and relative error for MONK datasets.

4.3.1 Convergence Rate

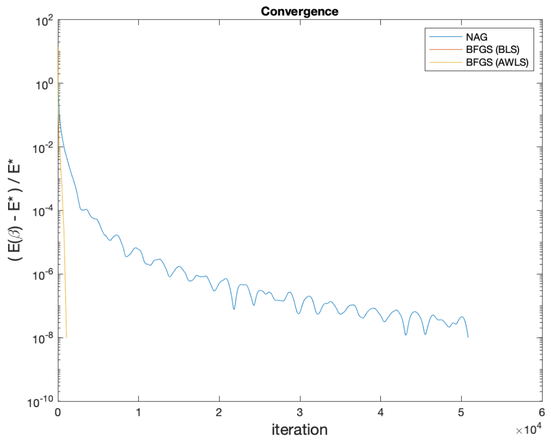
To verify the convergence rate of NAG and BFGS, we plot the relative error per iteration in logarithmic scale as shown in Figure 13. NAG shows a slower convergence rate and the value per iteration does not decrease monotonically. Zooming in on the BFGS convergence plot as in Figure 14, we can see that it converges superlinearly at the tail and both line search methods do not show significant differences.

4.3.2 Scalability

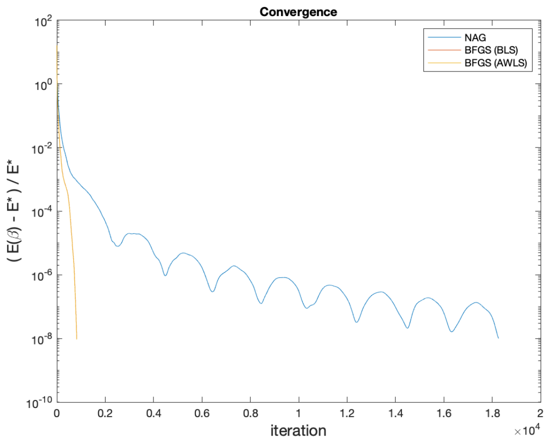
We have observed that NAG requires longer computation time per iteration as shown in Table 4. This happens when the number of hidden nodes is relatively small (less than 200). When it is increased up to 10,000 nodes, BFGS shows a quadratic increase as can be seen in Figure 15.



(a) MONK1

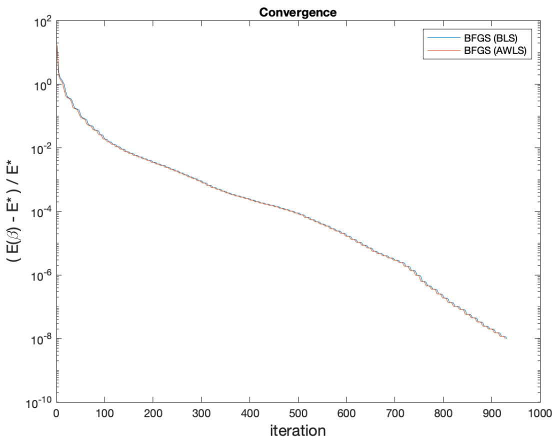


(b) MONK2

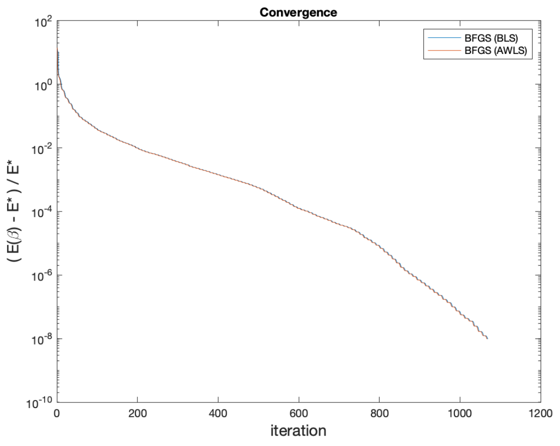


(c) MONK3

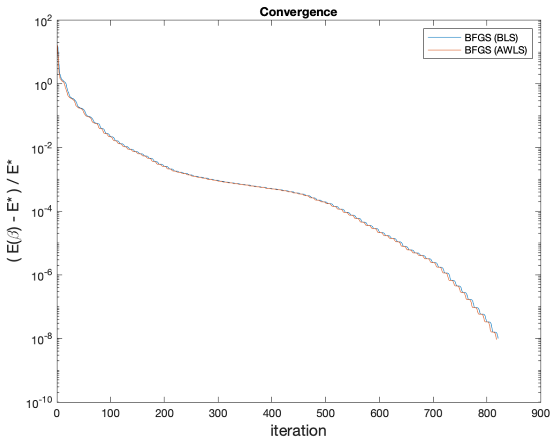
Figure 13: Convergence plot of MONK dataset.



(a) MONK1

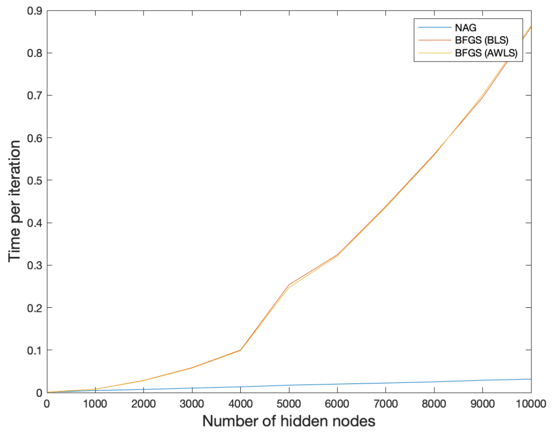


(b) MONK2

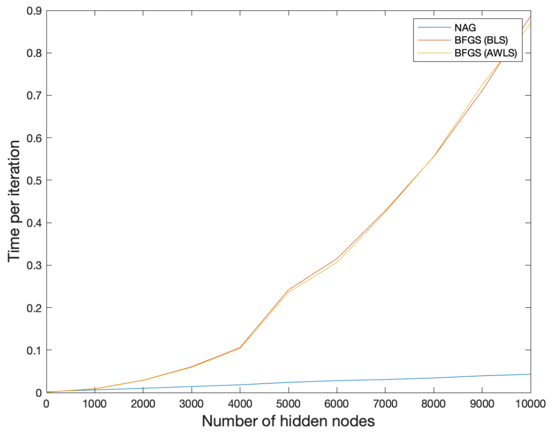


(c) MONK3

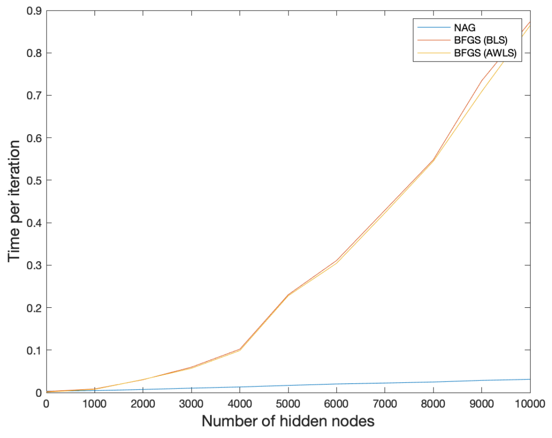
Figure 14: BFGS convergence plot of MONK dataset.



(a) MONK1



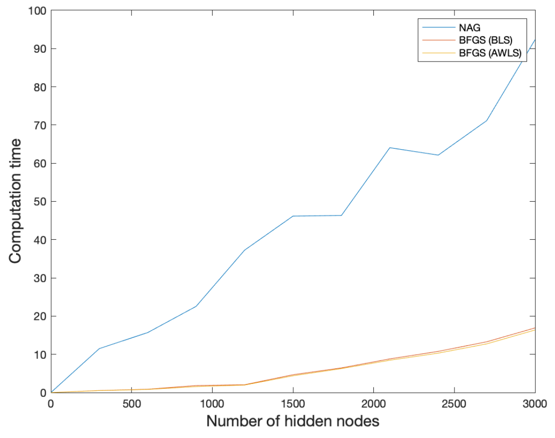
(b) MONK2



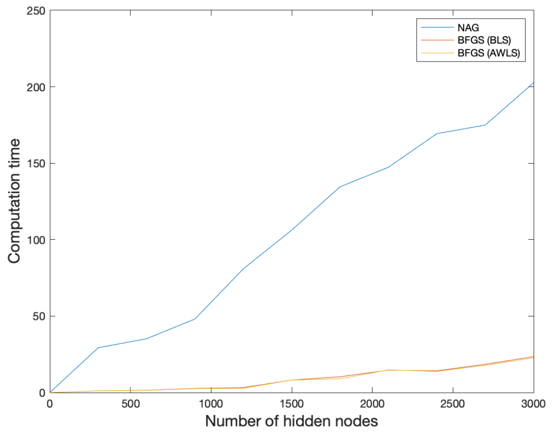
(c) MONK3

Figure 15: Computation time per iteration versus hidden nodes for MONK dataset.

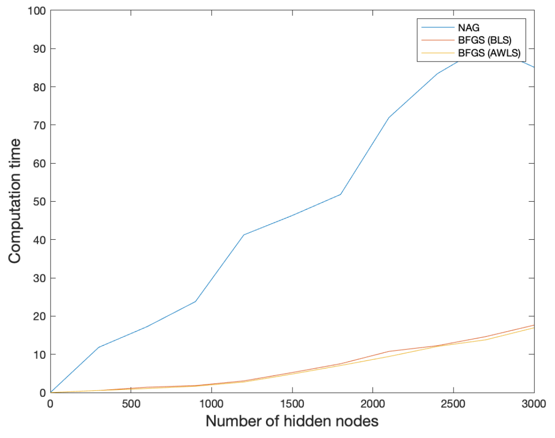
Since BFGS requires less iteration to reach the same accuracy as NAG, we also measure the total computation time of both algorithm to reach a relative error of 10^{-7}. The result, which is plotted in Figure 16, shows that NAG still takes longer computation time than BFGS due to higher number of iterations.



(a) MONK1



(b) MONK2



(c) MONK3

Figure 16: Computation time to reach a relative error of 10^{-7} versus hidden nodes for MONK dataset.

4.4 Random Matrix

In the last experiment, we test the algorithms on a randomly generated dataset and report the results of the optimizations in table 5. The input and output dimensions are equal to 5 and $N = h = 250$. The true solution $E^\star$ has a Mean-Squared-Error of 5.982501×10^{-18}.

The convergence rate is shown in figure 17.

Optimizer	**Step size**	**#Iterations**	**Time per iter [s]**	$(E(\beta^\star) - E^\star)/E^\star$
NAG	$1/L$	20635	4.912102×10^{-3}	9.999947×10^{-2}
BFGS	BLS	4403	1.633321×10^{-2}	9.998011×10^{-2}
BFGS	AWLS	4412	1.610975×10^{-2}	9.995762×10^{-2}
NAG	$1/L$	85833	5.258030×10^{-3}	9.999598×10^{-3}
BFGS	BLS	8383	1.601413×10^{-2}	9.997161×10^{-3}
BFGS	AWLS	8385	1.609678×10^{-2}	9.846367×10^{-3}
NAG	$1/L$	118234	5.766629×10^{-3}	9.999811×10^{-4}
BFGS	BLS	9741	1.932943×10^{-2}	9.971241×10^{-4}
BFGS	AWLS	9736	2.271561×10^{-2}	9.997550×10^{-4}
NAG	$1/L$	370087	6.251286×10^{-3}	9.999968×10^{-6}
BFGS	BLS	10837	1.560822×10^{-2}	9.396543×10^{-6}
BFGS	AWLS	10850	1.573680×10^{-2}	9.766289×10^{-6}

Table 5: Measurements of the computational time per iteration, number of iterations and optimal value for the random dataset when the relative error is $\leq \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-5}\}$

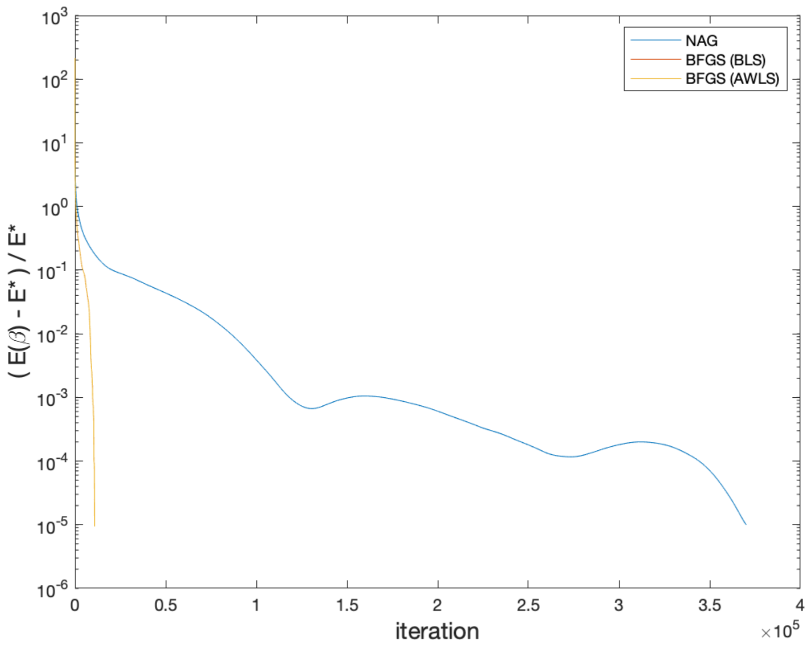


Figure 17: Convergence rate for N = h = 250, n = 5, m = 5, relative error $\leq 10^{-5}$, $\lambda = 0$

4.4.1 Scalability

With the same methodology performed for the sin(x) and MONK dataset, we show the relation between the computational time per iterations versus the output dimension m by making it vary from $m = 1$ to $m = 10$. We show the results in figure 18 confirming the linear increment in time for NAG and the quadratically increase for BFGS.

The next experiments compares the total computational time to reach a certain relative error with respect to different output dimensions m. We can see, in figure 19, that if the relative error we want is on the order of 10^{-3} both algorithms can reach this solution relatively fast. This implies that then when $m > 5$ the dominant term becomes the computational time per iteration and this is the reason why NAG seems to outperform BFGS. Although, if we repeat the test in order to reach a relative error of 10^{-6}, shown in figure 20, we can see that BFGS is always faster. Also, we had to limit the number of iterations of NAG to 500,000 in order to achieve the results in a reasonable amount of time, indeed for some choices of m NAG fails to reach the required accuracy and so the time reported is lower than the real time it would have taken.

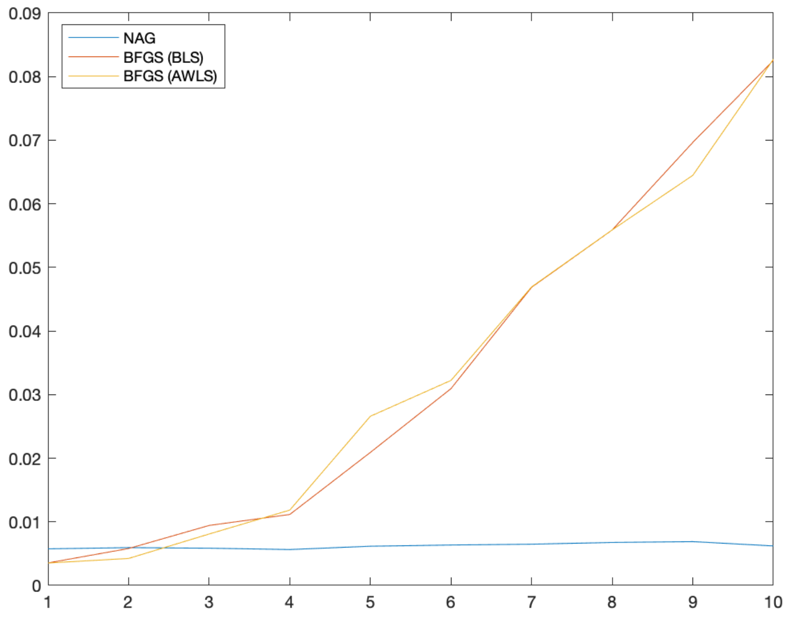


Figure 18: Computational time per iteration versus the output dimension m

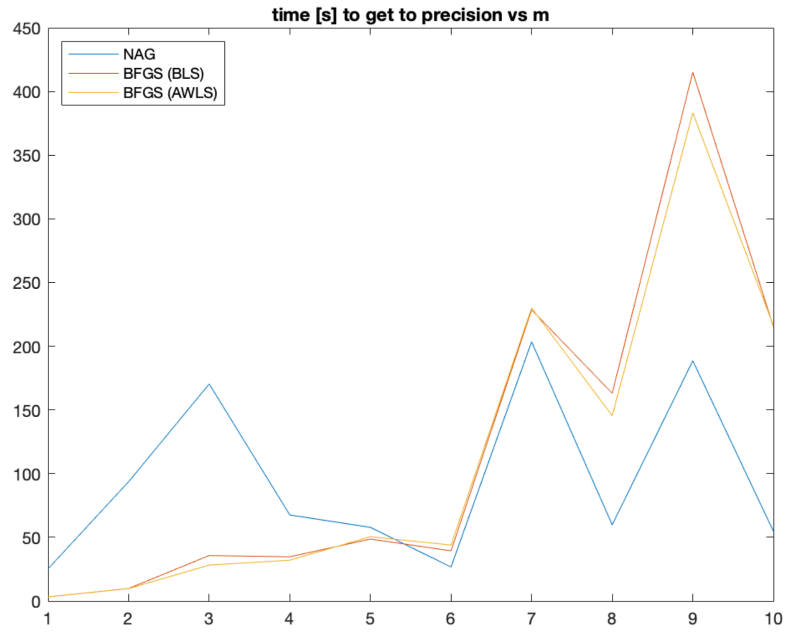


Figure 19: Computational time to reach a relative error of 10^{-3} versus the output dimension m

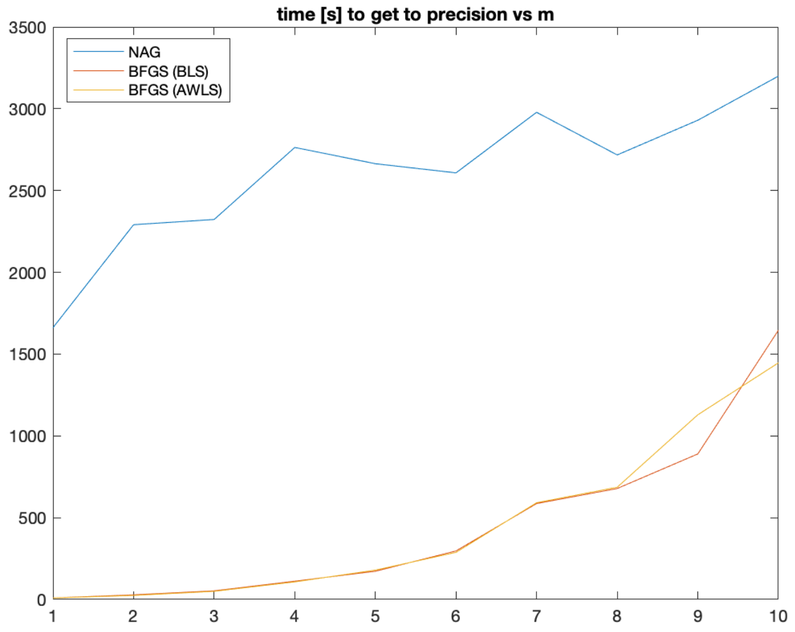


Figure 20: Computational time to reach a relative error of 10^{-6} versus the output dimension m

4.4.2 Effects of Singular Matrix

In this experiment, we test what happens when the matrix a_i, which is the output of the hidden layer (defined in 13), is close to singular. In order to make it close to singular we compute the singular value decomposition of the matrix W and set the last two singular values > 0 equal to zero, we then recompute the matrix W; also we set the last two values for b equal to zero. Then we optimize with the same parameters used for the result in table 5 comparing the results with the true solution $E^\star = 2.086132 \times 10^{-2}$. The results can be seen in table 6.

Optimizer	**Step size**	**#Iterations**	**Time per iter [s]**	$(E(\beta^\star) - E^\star)/E^\star$
NAG	$1/L$	6700	6.660564×10^{-3}	2.020861
BFGS	BLS	5310	2.179565×10^{-2}	1.020810
BFGS	AWLS	5311	2.245879×10^{-2}	1.020618

Table 6: N = h = 250, n = 5, m = 5, $\lambda = 0$

As expected the relative error increases with respect to the test performed on a well-conditioned matrix.

As for the convergence rate, shown in figure 21, we got the same results obtained on the previous datasets, obtaining for NAG a linear convergence at the tail, while for BFGS a superlinear convergence at the tail.

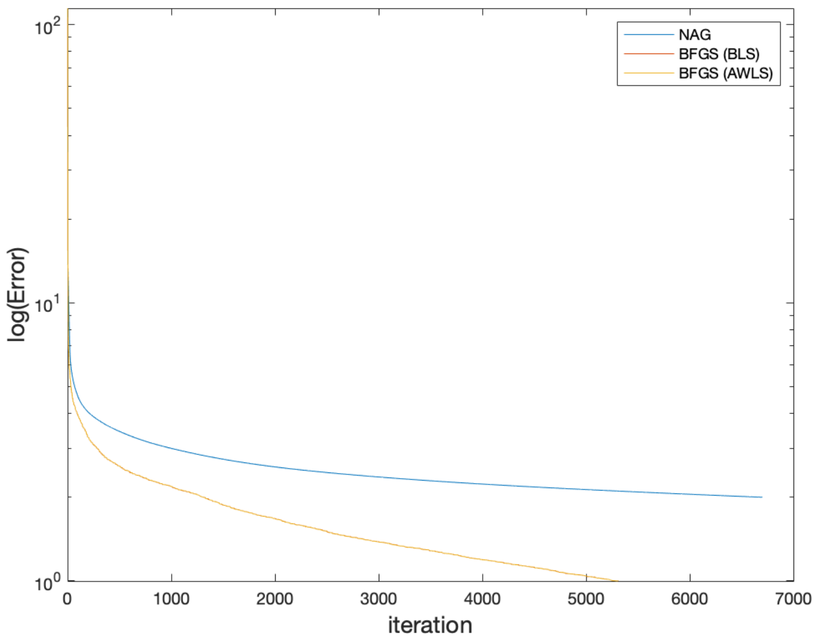


Figure 21: Convergence rate for N = h = 250, n = 5, m = 5, $\lambda = 0$ where a_i is close to singular

5 Summary and Conclusions

The experiment results show that NAG and BFGS follow the properties as described in the theory.

1. BFGS is able to reach an optimal solution closer to the true solution in less iterations compared to NAG.

2. In early iterations, there is a significant drop of the error and the convergence plot looks sublinear. However, later, the convergence becomes linear in NAG, and BFGS looks almost superlinear at the tail.

3. The computation time of NAG and BFGS is affected by the number of hidden units and the output dimension. In random matrix dataset and some scenarios of sine function, NAG shows a faster time per iteration, while in MONK dataset, BFGS is faster. This is due to the different number of hidden units and output dimension. If we change these values, then NAG shows a linear growth of computation time, while BFGS shows a quadratic growth.

4. The two line search methods (BLS and AWLS) do not give significant difference in the obtained solution, the number of iterations, and computation time.

5. Regularization helps to generalize the solution in noisy data. Even though the mean-squared error increase as the regularization parameter increased, if we plot the solution, it is closer to the true function.

6. As the output matrix from the hidden layer becomes close to singular, both NAG and BFGS gives worse solution with the same number of iteration compared to the case when the matrix is well-conditioned.

References

[1] *A method of solving a convex programming problem with convergence rate $O(1/k^2)$.* http://mpawankumar.info/teaching/cdt-big-data/nesterov83.pdf.

[2] *Convergence Theorems for Gradient Descent.* https://gowerrobert.github.io/pdf/M2_statistique_optimisation/grad_conv.pdf.

[3] *Convex Optimization – Boyd and Vandenberghe.* https://web.stanford.edu/~boyd/cvxbook/.

[4] *CPSC 540: Machine Learning, Convergence of Gradient Descent.* https://www.cs.ubc.ca/~schmidtm/Courses/540-W18/L4.pdf.

[5] *Extreme learning machine: Theory and applications.* https://www.sciencedirect.com/science/article/pii/S0925231206000385.

[6] *Extreme Learning Machines with Regularization for the Classification of Gene Expression Data.* http://ceur-ws.org/Vol-2473/paper11.pdf.

[7] *Gradient Descent.* https://www.seas.upenn.edu/~aarpit/opt-notes/cvx-opt4.pdf.

[8] *Modified quasi-Newton methods for training neural networks.* http://www.sciencedirect.com/science/article/pii/0098135495002286.

[9] *MONK's Problems Data Set.* https://archive.ics.uci.edu/ml/datasets/MONK%27s+Problems.

[10] *Non-asymptotic Superlinear Convergence of Standard Quasi-Newton Methods.* https://arxiv.org/pdf/2003.13607.pdf.

[11] *Numerical Optimization – Jorge Nocedal and Stephen J. Wright.* https://www.csie.ntu.edu.tw/~r97002/temp/num_optimization.pdf.

[12] *On the importance of initialization and momentum in deep learning.* http://www.cs.toronto.edu/~fritz/absps/momentum.pdf.

[13] *Optimization.* https://www.stat.purdue.edu/~vishy/introml/notes/Optimization.pdf.

[14] *Training Recurrent Neural Networks.* http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

[15] Wikipedia contributors. *Extreme learning machine — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Extreme_learning_machine&oldid=983919323. [Online; accessed 3-December-2020]. 2020.

[16] *Y. Nesterov.Introductory lectures on convex optimization: A basic course, volume 87.Springer Science & Business Media, 2013.*