UNIVERSITY OF VICTORIA
Department of Electrical and Computer Engineering

Investigating different weight updating schemes for Wine quality prediction using Neural Network.

ELEC 503 Optimization for Machine Learning

PROJECT REPORT
Submitted by

Derrell D'Souza (V00901532)

University of Victoria

Date: August 13th, 2018

ABSTRACT

Wine classification is a difficult task as taste is least understood by human senses. Wine quality prediction is essential not only for marketing purposes but also for improving wine production or to support oenologist wine tasting evaluations. This project deals with implementation of Neural Network with one hidden layer for predicting wine quality. The weights of the Neural Network are optimized by a learning process known as backpropagation. Different weight updating schemes are also investigated for obtaining optimized weights. Once the Neural Network is trained using backpropagation, the optimized weights are used to predict the quality of wine using forward propagation.

I. Introduction

In the present era, wine industry is investing into new and efficient techniques that will aid them in production of wine. A major issue involved here is the wine certification which prevents illegal adulteration of wine thereby assuring wine's quality. Wine quality is often assessed by physicochemical and sensory tests although the relationship between them is still not completely understood. [1] Thus there is a need to develop an accurate, computationally efficient and understandable prediction model that can be of great utility for the wine industry. A good wine quality prediction can be valuable in the certification phase, as the sensory analysis performed by human tasters, is clearly a subjective approach. An automatic predictive system can be integrated into a decision support system, helping the speed and quality of the oenologist performance. Such a prediction system can also be beneficial for training oenology students or for marketing purposes. [2]

Today, advances in information technology has made it possible to collect, store and process massive and highly complex datasets. Machine learning techniques can be used to exploit valuable information such as patterns and trends from these datasets to get improved decisions as well as optimize the chances of success. Neural Networks, one of the revolutions in machine learning can be used for wine quality prediction due to their high flexibility and nonlinear learning ability. When applying Neural Networks, variable selection, model selection and selection of hyper-parameters such as number of hidden nodes is very critical to obtain a good prediction accuracy. [1]

In this project, a Neural Network with a single hidden layer is used for prediction of wine quality. The dataset used for training consists of 4898 samples of white variant of the Portuguese "Vinho Verde" wine, each represented by 11 features. The weights of the Neural Network is optimized using a learning process called Back Propagation.[3] Different weight updating schemes such as Gradient Descent, Memoryless-BFGS, Stochastic Gradient Descent, Nesterov's Accelerated Gradient, Adam Optimization, AdaMax Optimization and Nesterov's Accelerated Adam Optimization are investigated for achieving a computationally efficient prediction of wine quality. The optimized weight vectors are used to predict the quality of a new sample using forward propagation.

II. Theoretical Background and Problem Formulation

A. Dataset and its preprocessing

[1] created two datasets, one of them contained red wine samples while the other one contained white wine samples. Each wine sample contained 11 features which included pH values and sensory data. Each expert graded the wine quality between 0(extremely bad) to 10(extremely good). Table 1 lists the feature collected in the datasets.

Both datasets suffer from data imbalance problem i.e. there are much more normal wines than excellent or poor ones. Only white wine dataset is used as it has a much larger number of samples as compared to red wine dataset. Further, it was observed that in this dataset, no wine samples with a quality of 0 to 2 and 10 were present. Table 2 gives a detailed information about the samples in the dataset.

To avoid the data imbalance problem, the dataset was reduced to samples with quality between 4 and 8 i.e. only [4898 – (20+5)] = 4873 samples were used. So the goal is to predict the quality of new and unused sample of white wine. So size of the training data matrix $\boldsymbol{X}$ is 4873 x 11 and size of the label vector $\boldsymbol{y}$ is 4873 x 1.

Sr. No.	Feature
1.	Fixed acidity
2.	Volatile acidity
3.	Citric acid
4.	Residual sugar
5.	Chlorides
6.	Free sulfur dioxide
7.	Total sulfur dioxide
8.	Density
9.	pH
10.	Sulphates
11.	Alcohol

Table 1. Features of each wine sample in the dataset

Quality	Number of samples
0	0
1	0
2	0
3	20
4	163
5	1457
6	2198
7	880
8	175
9	5
10	0

Table 2. Quality and the corresponding number of samples for the white wine dataset

Before defining a useful loss function, preprocessing the original data set becomes necessary as it is acquired from raw measurements. In the real-world, the numerical range of different features can be quite different due to different features representing different physical and artificial characteristics of the data. Hence with different feature scales, the loss function will be difficult to handle as a result of which feature normalization becomes a necessity. If normalization is not used, some weights will update faster than other and hence will oscillate inefficiently down to the optimum due to uneven feature scales. With all the features in the same range by feature normalization, the optimization algorithm can converge efficiently to the optimal value. [4]

Let $\boldsymbol{x}_i$ be the i^{th} feature vector in $\boldsymbol{X}$ where $i \in \{1, 2, 3, \ldots\ldots, 4873\}$. Let $\boldsymbol{\mu}$ be the mean of feature vectors in $\boldsymbol{X}$ and let σ be the corresponding standard deviation. Then every feature $\boldsymbol{x}_i$ in $\boldsymbol{X}$ can be normalized as follows:

$$\bar{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \quad (1)$$

We can then represent the normalized data matrix as $\bar{\boldsymbol{X}}$. Those samples in the normalized data matrix $\bar{\boldsymbol{X}}$ having the same label are put together as one subset. Hence we have five subsets, the samples with level *i* for which label *i* was assigned constitute subset *i* for i = 4 ,5 ,6 ,7 ,8. The five subsets are $\boldsymbol{X}_4, \boldsymbol{X}_5, \boldsymbol{X}_6, \boldsymbol{X}_7$ and $\boldsymbol{X}_8$ .Next, approximately 80% of samples from each subset are selected at random for training and remaining 20% are selected for testing and are grouped together as $\boldsymbol{X}_{\text{train}}$ and $\boldsymbol{X}_{\text{test}}$ respectively. We also generate the corresponding training and testing labels $\boldsymbol{y}_{\text{train}}$ and $\boldsymbol{y}_{\text{test}}$ respectively. [3]

Subset	Total number of samples	Number of samples for training	Number of samples for testing
$\boldsymbol{X}_4$	163	130	33
$\boldsymbol{X}_5$	1457	1165	292
$\boldsymbol{X}_6$	2198	1758	440
$\boldsymbol{X}_7$	880	704	176
$\boldsymbol{X}_8$	175	140	35

Table 3. Subsets and corresponding samples for training and testing

- $\boldsymbol{X}_{\text{train}}$ is the training set of size 11 $\times$ 3897 and $\boldsymbol{y}_{\text{train}}$ is the corresponding training labels of size 1 $\times$ 3897.
- $\boldsymbol{X}_{\text{test}}$ is the testing set of size 11 $\times$ 976 and $\boldsymbol{y}_{\text{test}}$ is the corresponding testing labels of size 1 $\times$ 976.

B. Overview of the Neural Network Architecture

Neural networks are computer systems modelled on human brain and nervous system. In this project, a feed forward neural network is constructed with one hidden layer. The network is shown below in the Figure 1.

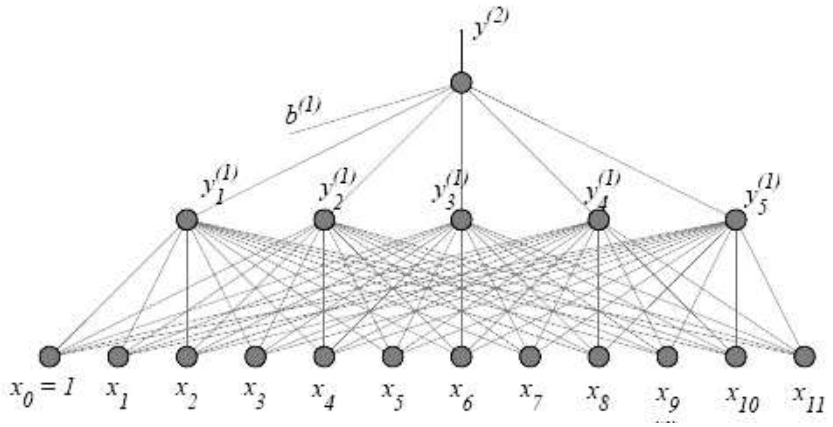

Figure 1. A neural network with one hidden layer and a single output.

The input layer for the neural network consists of 12 units which takes the input data having 11 input features and an additional unit $\boldsymbol{x}_0$ always set to 1. The input is moved forward by connecting input layer to 5 units in the hidden layer and at each path data is weighted multiplicatively. The weights of the path from i^{th} unit of the input layer to the j^{th} unit of the hidden layer is given by $\{w_{i,j}^{(0)}$ for $i = 1,2, \ldots \ldots 11, j = 1,2 \ldots .5\}$ and the multiplicative gain in the path from the input unit $\boldsymbol{x}_0$ to the j^{th} unit of the hidden layer is denoted by $\{b_j^{(0)}$ for $j = 1,2 \ldots .5\}$ .When the weighted signals arrive at the j^{th} unit of the hidden layer they are added up as an input to the activation function (in our case it is sigmoidal function) which produces an output signal $y_j^{(1)}$ for $j = 1,2 \ldots .5$. These signals then move from the hidden layer to the output layer where $\{y_j^{(1)}$ for $j = 1,2 \ldots .5\}$ are multiplied by corresponding weights $\{w_j^{(1)}$ for $j = 1,2 \ldots .5\}$ and then are added up (plus a bias $b^{(1)}$) as an input to sigmoidal activation function that yields output of the network $y^{(2)}$.

The dataflow from the input of the neural network to the output is known as forward propagation. For the given architecture in Figure 1 the forward propagation is as follows.

1. From input layer to the hidden layer:

$$a_j^{(1)} = b_j^{(0)} + \sum_{i=1}^{11} w_{i,j}^{(0)} x_i \text{ for } j = 1,2 \ldots .5 \tag{2}$$

$$y_j^{(1)} = f_1\left(a_j^{(1)}\right) \qquad \text{for } j = 1,2 \ldots .5 \tag{3}$$

where $f_1(a)$ is a sigmoid activation function defined by

$$f_1(a) = \frac{1}{1 + e^{-a}} \tag{4}$$

which maps the entire real axis to the interval (0, 1)

2. From the hidden layer to the output layer:

$$a^{(2)} = b^{(1)} + \sum_{j=1}^{5} w_j^{(1)} y_j^{(1)} \tag{5}$$

$$y^{(2)} = f_2\left(a^{(2)}\right) \tag{6}$$

where $f_2(a)$ is a slightly modified activation function,

$$f_2(a) = \frac{5}{1 + e^{-a}} + 3.5 \tag{7}$$

which maps the real axis to interval (3.5, 8.5) that covers five quality levels from 4 to 8.

A neural network is trained by a process known as backpropagation. Training a neural network means tuning the network parameters i.e. the weights and the biases at different layers. For the problem at hand, there are two sets of parameters to be trained.

a) Weights and the biases involved in connecting the input layer to the hidden layer,
$\{w_{i,j}^{(0)}$ for $i = 1,2, \ldots \ldots .11, j = 1,2 \ldots .5\}$ and $\{b_j^{(0)}$ for $j = 1,2 \ldots .5\}$
b) Weights and the biases involved in connecting the hidden layer to the output layer,
$\{w_j^{(1)}$ for $j = 1,2 \ldots .5\}$ and $b^{(1)}$

In total there are 66 parameters to be trained. The process of parameter tuning is realized by optimization where the parameters are updated iteratively. A key quantity required by most optimization techniques is the gradient of a loss function. For multilayer neural networks, the gradient is evaluated by a technique known as backpropagation (BP) which takes advantages of the neural networks multi-layer structure.

To formulate an optimization problem we have to define a loss function which is to be minimized. The loss function is measure of inaccurate prediction by network and is defined only by the training data in the context of machine learning.

When a training data vector $\boldsymbol{x}_n$ is present at the input of the network, the network is adequately tuned if the output $y^{(2)}(n)$ is a good match with the true label y_n of the data. The L_2 loss function is given by

$$E = \frac{1}{2}\sum_{n=1}^{N_1}[y^{(2)}(n) - y_{\text{train}}(n)]^2 \tag{8}$$

Here $y_{\text{train}}(n)$ is the label of the input data $\boldsymbol{x}_n$, $y^{(2)}(n)$ denotes the neural networks output in response to the input data $\boldsymbol{x}_n$ and N_1 is the number of training samples.

If

$$E_n = \frac{1}{2}[y^{(2)}(n) - y_{\text{train}}(n)]^2 \tag{9}$$

Then the loss function will be,

$$E = \frac{1}{2}\sum_{n=1}^{N_1} E_n \tag{10}$$

and hence gradient ∇E with respect to parameters can be evaluated term by term and then added up as

$$\nabla E = \sum_{n=1}^{N_1} \nabla E_n \tag{11}$$

i) Assuming that the data at the input is $\boldsymbol{x}_n$, from (5),

$$\frac{\partial E_n}{\partial w_j^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial w_j^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}} y_j^{(1)} \text{ for } j = 1,2 \ldots .5 \tag{12}$$

From (6),

$$\frac{\partial E_n}{\partial a^{(2)}} = \frac{\partial E_n}{\partial y^{(2)}(n)} \frac{\partial y^{(2)}(n)}{\partial a^{(2)}} = \frac{\partial E_n}{\partial y^{(2)}(n)} f_2{}'\left(a^{(2)}\right) \tag{13}$$

From (9),

$$\frac{\partial E_n}{\partial y^{(2)}} = y^{(2)}(n) - y_{\text{train}}(n) \tag{14}$$

From (7),

$$f_2{}'(a) = \frac{5e^{-a}}{(1 + e^{-a})^2} \tag{15}$$

$$\frac{\partial E_n}{\partial b^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial b^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}} \tag{16}$$

ii) Now we go backward to evaluate first order derivative of E_n w.r.t. $w_{i,j}^{(0)}$. This is done by

$$\frac{\partial E_n}{\partial w_{i,j}^{(0)}} = \frac{\partial E_n}{\partial a_j^{(1)}} \boldsymbol{x}_i \tag{17}$$

$$\frac{\partial E_n}{\partial a_j^{(1)}} = \frac{\partial E_n}{\partial y_j^{(1)}} \frac{\partial y_j^{(1)}}{\partial a_j^{(1)}} = \frac{\partial E_n}{\partial y_j^{(1)}} f_1{}'\left(a_j^{(1)}\right) \tag{18}$$

$$\frac{\partial E_n}{\partial y_j^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial y_j^{(1)}} = \frac{\partial E_n}{\partial a^{(2)}} w_j^{(1)} \tag{19}$$

where $\frac{\partial E_n}{\partial a^{(2)}}$ is given by (13) and (14) as

$$f_1{}'(a) = \frac{e^{-a}}{(1 + e^{-a})^2} \tag{20}$$

In addition the first order derivative of E_n w.r.t. $b_j^{(0)}$ is calculated using

$$\frac{\partial E_n}{\partial b_j^{(0)}} = \frac{\partial E_n}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial b_j^{(0)}} = \frac{\partial E_n}{\partial a_j^{(1)}} \tag{21}$$

Here $\frac{\partial E_n}{\partial a_j^{(1)}}$ is given by (18) and (19)

C. Optimization problem formulation

Let $\widehat{\boldsymbol{W}}_0=\begin{bmatrix} b_j^{(0)} \\ w_{i,j}^{(0)} \end{bmatrix}$ be the set of parameters for connection of input layer to hidden layer and let $\widehat{\boldsymbol{W}}_1 = \begin{bmatrix} b^{(1)} \\ w_j^{(1)} \end{bmatrix}$ be the set of parameters for connection of hidden layer to output layer . Here i = 1, 2, 3…….11 and j = 1, 2...5
Hence $\widehat{\boldsymbol{W}}_0$ is a 12 x 5 matrix and $\widehat{\boldsymbol{W}}_1$ is a 6 x 1 matrix.

Our optimization problem can be formulated as:

Find optimized weights $\{\widehat{\boldsymbol{W}}_0{}^*, \widehat{\boldsymbol{W}}_1{}^*\}$ such that the L_2 loss function given by $E = \frac{1}{2}\sum_{n=1}^{N_1}[y^{(2)}(n) - y_{\text{train}}(n)]^2$ is minimized.

$$\widehat{\boldsymbol{W}}^* \leftarrow \underset{\widehat{\boldsymbol{W}}^*}{\operatorname{argmin}} E$$

where $\widehat{\boldsymbol{W}}^* = \{\widehat{\boldsymbol{W}}_0{}^*, \widehat{\boldsymbol{W}}_1{}^*\}$

D. Optimization Algorithms for weight updating

The weight updating scheme effects the performance of the prediction system. In this project seven different types of different weight updating schemes are used.

1. Gradient Descent(GD)Algorithm

The weight updating scheme using Gradient Descent Approach is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}}_0{}^{(0)}, \widehat{\boldsymbol{W}}_1{}^{(0)}\}$, a learning rate α and the number of iterations K. Also set $k = 0$.
2. Calculate the gradient $\nabla\widehat{\boldsymbol{W}}^{(k)}$ using the backpropagation algorithm.
3. Set $\widehat{\boldsymbol{W}}^{(k+1)} = \widehat{\boldsymbol{W}}^{(k)} - \alpha\nabla\widehat{\boldsymbol{W}}^{(k)}$.
4. If $k = K$ output $\widehat{\boldsymbol{W}}^* = \{\widehat{\boldsymbol{W}}_0{}^*, \widehat{\boldsymbol{W}}_1{}^*\}$ and stop.
5. Otherwise set $k \coloneqq k + 1$ and continue from 2.

2. Memoryless-BFGS(ML-BFGS) Algorithm

The weight updating scheme using Memoryless-BFGS algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}}_0{}^{(0)}, \widehat{\boldsymbol{W}}_1{}^{(0)}\}$ a learning rate α , and the number of iterations K. Also set $k = 0$.
2. Calculate the gradient $\nabla\widehat{\boldsymbol{W}}^{(k)}$ using backpropagation algorithm and set $\boldsymbol{d}^{(k)} = -\nabla\widehat{\boldsymbol{W}}^{(k)}$.
3. If $k = K$ output $\widehat{\boldsymbol{W}}^* = \{\widehat{\boldsymbol{W}}_0{}^*, \widehat{\boldsymbol{W}}_1{}^*\}$ and stop. Otherwise
   a) Compute $\boldsymbol{\delta}^{(k)} = \widehat{\boldsymbol{W}}^{(k+1)} - \widehat{\boldsymbol{W}}^{(k)}$, $\boldsymbol{\gamma}^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k+1)} - \nabla\widehat{\boldsymbol{W}}^{(k)}$, $\boldsymbol{\rho}^{(k)} = \frac{1}{\boldsymbol{\gamma}^{(k)T}\boldsymbol{\delta}^{(k)}}$, $t^{(k)} = \boldsymbol{\delta}^{(k)T}\nabla\widehat{\boldsymbol{W}}^{(k+1)}$, $q^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k+1)} - \boldsymbol{\rho}^{(k)}t^{(k)}\boldsymbol{\gamma}^{(k)}$. Set $k \coloneqq k + 1$.
   b) Compute search direction $\boldsymbol{d}^{(k+1)} = \boldsymbol{\rho}^{(k)}\left(\boldsymbol{\gamma}^{(k)}q^{(k)} - t^{(k)}\right)\boldsymbol{\delta}^{(k)} - q^{(k)}$ and continue from step 3.

3. Stochastic Gradient Descent(SGD) Algorithm

The weight updating scheme using Stochastic Gradient Descent algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}}_{\boldsymbol{0}}{}^{(0)}, \widehat{\boldsymbol{W}}_{\boldsymbol{1}}{}^{(0)}\}$, a learning rate , a decreased learning rate α_{K_1}, an integer m and the number of iterations K and iterations K_1 after which the learning rate decreases to α_{K_1}. Also set $k = 0$.
2. If k < K, select m indices $\{ i_1^{(k)}, i_2^{(k)}, i_3^{(k)} \ldots\ldots i_m^{(k)}\}$ uniformly from a random set $\{1,2, \ldots \ldots N\}$ and compute search direction as follows:

$$\boldsymbol{d}^{(k)} = -\frac{1}{m}\sum_{i=1}^{m} \nabla\widehat{\boldsymbol{W}}_{i_t}{}^{(k)}$$

The individual gradients are calculated using backpropagation algorithm. Go to step 3 otherwise output $\widehat{\boldsymbol{W}}^{(k)}$ as the solution and stop.

3. Update $\widehat{\boldsymbol{W}}^{(k)}$ to $\widehat{\boldsymbol{W}}^{(k+1)} = \widehat{\boldsymbol{W}}^{(k)} + \alpha^{(k)}\boldsymbol{d}^{(k)}$. Here the learning rate is taken to be $\alpha^{(k)} = \left(1 - \frac{k}{K_1}\right)\alpha^{(0)} + \frac{k}{K_1}\alpha_{K_1}$ for first K_1 iterations and α_{K_1} thereafter.
4. Set $k \coloneqq k + 1$ and continue from 2.

4. Nesterov's Accelerated Gradient(NAG) Algorithm

The weight updating scheme using Nesterov's Accelerated Gradient algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}}_{\boldsymbol{0}}{}^{(0)}, \widehat{\boldsymbol{W}}_{\boldsymbol{1}}{}^{(0)}\}$, Lipschitz constant L , number of iterations K and initialize tolerance ε. Set $\boldsymbol{y}^{(1)} = \widehat{\boldsymbol{W}}^{(0)}, \alpha = \frac{1}{L}$, $t^{(1)} = 1$ and $k = 1$.
2. Compute $\widehat{\boldsymbol{W}}^{(k)} = \boldsymbol{y}^{(k)} - \alpha\nabla f(\boldsymbol{y}^{(k)})$ where $\nabla f(\boldsymbol{y}^{(k)})$ is calculated using backward propagation algorithm is.
3. Compute $t^{(k+1)} = \frac{1+\sqrt{1+4(t^{(k)})^2}}{2}$.
4. Update $\boldsymbol{y}^{(k+1)} = \widehat{\boldsymbol{W}}^{(k)} + (\frac{t^{(k)}-1}{t^{(k+1)}})(\widehat{\boldsymbol{W}}^{(k)} - \widehat{\boldsymbol{W}}^{(k-1)})$.
5. If $k = K$ output $\widehat{\boldsymbol{W}}^{*}$and stop.
   Otherwise set $k \coloneqq k + 1$ and continue from 2.

5. Adaptive Moment Estimation(ADAM) Algorithm

The weight updating scheme using ADAM algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}}_{\boldsymbol{0}}{}^{(0)}, \widehat{\boldsymbol{W}}_{\boldsymbol{1}}{}^{(0)}\}$, a learning rate α, β_1 and $\beta_2 \in [0,1]$(exponential decay rates for moment estimates), $\in$ (convergence parameter) and number of iterations K. Also set $m^{(k)} = 0(1^{\text{st}}$ moment vector), $v^{(k)} = 0\left(2^{\text{nd}}\right.$ moment vector$\left.\right)$ and $k = 0$.
2. If $k = K$ output $\widehat{\boldsymbol{W}}^{*}$and stop. Otherwise set k: = k + 1.
3. Calculate gradient $g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)}$by selecting m indices $\{ i_1^{(k)}, i_2^{(k)}, i_3^{(k)} \ldots\ldots i_m^{(k)}\}$ uniformly from a random set $\{1,2, \ldots \ldots N\}$ and compute search direction as follows:

$$g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)} = \frac{1}{m}\sum_{i=1}^{m} \nabla\widehat{\boldsymbol{W}}_{i_t}{}^{(k-1)}$$

The individual gradients are calculated using backpropagation algorithm.

4. Calculate $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)\, g^{(k)}$, $v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)\,(g^{(k)})^2$.

5. Calculate $\hat{m}^{(k)} = m^{(k)}/(1-(\beta_1)^k)$, $\hat{v}^{(k)} = v^{(k)}/(1-(\beta_2)^k)$.
6. Update $\widehat{\boldsymbol{W}}^{(k)} = \widehat{\boldsymbol{W}}^{(k-1)} - \alpha\hat{m}^{(k)}/(\sqrt{\hat{v}^{(k)}} + \in)$.

6. Adam based on infinity norm(AdaMax) Algorithm

The weight updating scheme using AdaMax Algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}_0}^{(0)}, \widehat{\boldsymbol{W}_1}^{(0)}\}$, a learning rate α, β_1 and $\beta_2 \in [0,1]$(exponential decay rates for moment estimates), $\in$ (convergence parameter) and number of iterations K. Also set $m^{(k)} = 0$(1^{st} moment vector), $v^{(k)} = 0$(exponentially weighted infinity norm) and $k = 0$.
2. If $k = K$ output $\widehat{\boldsymbol{W}}^*$and stop. Otherwise set k: = k + 1.
3. Calculate gradient $g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)}$by selecting m indices $\{i_1^{(k)}, i_2^{(k)}, i_3^{(k)} \ldots\ldots i_m^{(k)}\}$ uniformly from a random set $\{1,2, \ldots \ldots N\}$ and compute search direction as follows:

$$g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)} = \frac{1}{m}\sum_{i=1}^{m} \nabla\widehat{\boldsymbol{W}}_{i_t}^{(k-1)}$$

The individual gradients are calculated using backpropagation algorithm.

4. Calculate $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)\, g^{(k)}$, $v^{(k)} = \max(\beta_2 . v^{(k-1)}, |\, g^{(k)}|)$.
5. Update $\widehat{\boldsymbol{W}}^{(k)} = \widehat{\boldsymbol{W}}^{(k-1)} - (\alpha/(1 - (\beta_1)^k)) . m^{(k)}/v^{(k)}$.

7. Nesterov's accelerated Adaptive Moment Estimation(NAadam) Algorithm

The weight updating scheme using NAadam algorithm is as follows:

1. Input initial weight $\widehat{\boldsymbol{W}}^{(0)} = \{\widehat{\boldsymbol{W}_0}^{(0)}, \widehat{\boldsymbol{W}_1}^{(0)}\}$, a learning rate α, β_1 and $\beta_2 \in [0,1]$(exponential decay rates for moment estimates), $\in$ (convergence parameter) and number of iterations K. Also set $m^{(k)} = 0$(1^{st} moment vector), $v^{(k)} = 0\big(2^{\text{nd}}$ moment vector$\big)$ and $k = 0$.
2. If $k = K$ output $\widehat{\boldsymbol{W}}^*$and stop. Otherwise set k: = k + 1.
3. Calculate gradient $g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)}$by selecting m indices $\{i_1^{(k)}, i_2^{(k)}, i_3^{(k)} \ldots\ldots i_m^{(k)}\}$ uniformly from a random set $\{1,2, \ldots \ldots N\}$ and compute search direction as follows:

$$g^{(k)} = \nabla\widehat{\boldsymbol{W}}^{(k-1)} = \frac{1}{m}\sum_{i=1}^{m} \nabla\widehat{\boldsymbol{W}}_{i_t}^{(k-1)}$$

The individual gradients are calculated using backpropagation algorithm.

4. Calculate $\hat{g}^{(k)} = \frac{g^{(k)}}{1-(\beta_1)^k}$.
5. Calculate $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)\, g^{(k)}$, $v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)\, (g^{(k)})^2$.
6. Calculate $\hat{m}^{(k)} = m^{(k)}/(1-(\beta_1)^{(k+1)})$, $\hat{v}^{(k)} = v^{(k)}/(1-(\beta_2)^k)$.
7. Calculate $\bar{m}^{(k)} = (1 - \beta_1)\, \hat{g}^{(k)} + \beta_1 \hat{m}^{(k)}$.
8. Update $\widehat{\boldsymbol{W}}^{(k)} = \widehat{\boldsymbol{W}}^{(k-1)} - \alpha\bar{m}^{(k)}/(\sqrt{\hat{v}^{(k)}} + \in)$.

III. Results and Discussion

A. Numerical Results

WUS	K	α	β_1, β_2	m	TrAcc (%)	TeAcc (%)	ISE	OSE	τ(sec)
Gradient Descent	10000	3.25e-5	-	-	55.07	53.07	0.4842	0.5102	63.94
ML-BFGS	700	3.25e-5	-	-	55.17	53.59	0.4850	0.5010	14
Stochastic Gradient Descent*	700/ 400	0.95/0.143	-	-	53.91	53.28	0.5183	0.5195	0.8125
Nesterov's Accelerated Gradient	200	6.5e-5	-	-	54.61	53.59	0.4924	0.5113	6.8594
Adam	1000	0.0125	0.9,0.75	3	53.58	54.51	0.5160	0.5133	0.7031
Adamax	1000	0.025	0.45,0.75	3	52.78	55.02	0.5355	0.5154	0.6875
Nesterov's Accelerated Adam	1000	0.039	0.7,0.75	3	54.48	55.23	0.5012	0.5010	0.6717

Table 4. Numerical results from simulations

WUS- Weight Updating Scheme
K – Number of iterations
α – Learning Rate
β_1, β_2- Exponential decay rates for moment estimates
m- Number of indices for gradient calculation
TrAcc – Training Accuracy
TeAcc – Testing Accuracy
ISE- In-Sample Average Prediction Error
OSE- Out-of Sample Average Prediction Error
τ – Average CPU time calculated by running the algorithm for 10 iterations
*-SGD has two learning rates: α (0.95) decreases linearly to $\alpha_{K1}(0.143)$ for k = 1 up to K1 (400) and for K1 up to K (700) learning rate is $\alpha_{K1}(0.143)$. K is denoted as K/K1 and α is denoted as α/α_{K1} in table 4.

B. Analysis

The Average CPU time has been calculated on an Intel® Core ™ i3-6006U CPU @ 2.0 GHz. We can see from Table 4 that the Gradient Descent Algorithm takes a very long time as well as a considerably higher number of iterations to reach the desired performance. For Data Analysis and Machine Learning huge amount of data is needed as a result of which the speed of the Weight Updating scheme plays a major role. SGD, Adam, AdaMax and NAadam are the fastest, reaching a performance near to that of GD within a second. They are all stochastic optimization schemes and converge faster as compared to other traditional weight updating schemes due to their reliance on performing a very large number of relatively-inexpensive updates that often outperform one which performs a smaller number of much "smarter" but computationally expensive updates.

We tried to circumvent the data imbalance problem by using only quality levels 4,5,6,7,8 for training but from low accuracy rate in the results it is possible that the data imbalance problem still exists as different subsets are not represented equally as the sample size is uneven.

C. Discussion

The functions for forward propagation and backward propagation were vectorized to take the whole matrix of training samples and labels at once instead of looping for every other sample. Vectorization increased the execution speed of the algorithm. Line search was not used for Gradient Descent and ML-BFGS as it is computationally expensive and results in a significantly larger computation time.

IV. Conclusion

Based on the results, it can be concluded that it is possible to update weights faster for the backpropagation algorithm in Neural Networks by designing and using stochastic algorithms. The results imply that stochastic schemes of SGD, Adam, AdaMax and NAadam perform faster by a huge margin as compared to inherently batch approaches. The predictor doesn't perform well due to the imbalanced dataset problem which is as a result of uneven distribution of data amongst different subsets of quality level. If data is properly balanced, the accuracy of the predictor can be improved.

V. References

[1] Cortez P., Cerdeira A., Almeida F., Matos T., Reis J. , "Modelling wine preferences by data mining from physicochemical properties", *Decision Support Systems*, Elsevier, 47(4):547-553. ISSN: 0167-9236, 2009.
[2] Nebot A., Mugica F., Escobet A., "Modelling wine preferences from physicochemical properties using fuzzy techniques", *5th International Conference on Simulation and Modelling Methodologies, Technologies and Applications*, UPC, 2015.
[3] W.S.Lu, "LabManual-ECE403-503-2018", University of Victoria, 2018.
[4] Andrew Ng. "Machine Learning: Yearning", http://www.mlyearning.org/, 2018
[5] W.S.Lu, "Notes-ECE403-503-2018", University of Victoria, 2018.
[6] Kingma D., Ba J.,"ADAM: A method for stochastic optimization",*ICLR*,2015
[7] Dozat T., "Incorporating Nesterov's momentum into Adam", Stanford University, 2015

Appendix A

Investigating different weight updating schemes for Wine quality prediction using Neural Network.

By Derrell D'Souza(V00901532)

I:Load the data matrix

```
load D_white.mat %12 x 4898 : 11x4898 = data ; 1 x 4898 = label
```

II:Generate training and testing data sets and their labels

```
X = D_white(1:11,:); %first 11 rows of the data matrix is training data matrix
y = D_white(12,:);   %last row is the label vector
%data normalization
[d,N] = size(X);
Xh = zeros(11,4898);
for i = 1:d %over d rows of data
    xi = X(i,:);%select each row
    mi = mean(xi);%calculate the mean
    vi = sqrt(var(xi));%calculate the deviation
    Xh(i,:) = (xi - mi)/vi;%normalize
end
%data for training and testing
ind4 = find(y == 4);%labels for level 4
ind5 = find(y == 5);%labels for level 5
ind6 = find(y == 6);%labels for level 6
ind7 = find(y == 7);%labels for level 7
ind8 = find(y == 8);%labels for level 8
X4 = Xh(:,ind4);%subset for level 4
X5 = Xh(:,ind5);%subset for level 5
X6 = Xh(:,ind6);%subset for level 6
X7 = Xh(:,ind7);%subset for level 7
X8 = Xh(:,ind8);%subset for level 8
%80% of data for training and 20% for testing
rand('state',7)
p4 = randperm(163);
X4te = X4(:,p4(1:33));%testing samples for level 4 subset
X4tr = X4(:,p4(34:163));%training samples for level 4 subset
rand('state',8)
p5 = randperm(1457);
X5te = X5(:,p5(1:292));%testing samples for level 5 subset
X5tr = X5(:,p5(293:1457));%training samples for level 5 subset
```

```
rand('state',9)
p6 = randperm(2198);
X6te = X6(:,p6(1:440));%testing samples for level 6 subset
X6tr = X6(:,p6(441:2198));%training samples for level 6 subset
rand('state',10)
p7 = randperm(880);
X7te = X7(:,p7(1:176));%testing samples for level 7 subset
X7tr = X7(:,p7(177:880));%training samples for level 7 subset
rand('state',11)
p8 = randperm(175);
X8te = X8(:,p8(1:35));%testing samples for level 8 subset
X8tr = X8(:,p8(36:175));%training samples for level 8 subset
Xtest = [X4te X5te X6te X7te X8te];%11 x 976
ytest = [4*ones(33,1);5*ones(292,1);6*ones(440,1);7*ones(176,1);8*ones(35,1)];
Xtr = [X4tr X5tr X6tr X7tr X8tr]; %11 x 3897
ytr = [4*ones(130,1);5*ones(1165,1);6*ones(1758,1);7*ones(704,1);8*ones(140,1)];
```

III: Training neural network for wine quality prediction

a)Using Gradient Descent as the Weight Updating Scheme

```
a = 3.25e-5; %learning rate
K = 1e4; %number of iterations
randn('state',7)
W0 = 0.25*randn(12,5);
randn('state',19)
W1 = 0.25*randn(6,1);
%Gradient Descent Algorithm
t1 = cputime;%cputime starts here
[W0_s, W1_s] = grad_desc_NN(Xtr,ytr,W0,W1,a,K);%Run the algorithm
t2 = cputime;%cputime ends here
cpu_time = (t2-t1);%calculate the total CPU time
```

b)Using Memoryless BFGS as the Weight Updating Scheme

```
K = 700;%number of iterations
a = 3.25e-5; %learning rate
randn('state',7)
W0 = 0.25*randn(66,1);%combine weights into a single matrix
t1 = cputime;%cputime starts here
% %ML BFGS algorithm
[Ws,fs] = bfgs_ML_NN('f_NN','g_NN',a, W0,K);
t2 = cputime;%cputime ends here
cpu_time = (t2-t1);%calculate the total CPU time
W_0=Ws(1:60);%Weights for input layer to hidden layer
W_1=Ws(61:66);%Weights for hidden layer to output layer
W0_s =reshape(W_0,12,5);%Weight Matrix for input layer to hidden layer
W1_s =reshape(W_1,6,1); %%Weight matrix for hidden layer to output layer
```

c)Using Stochastic Gradient Descent as the Weight Updating Scheme

```
a = 0.95;%learning rate
K = 700;%number of iterations
K1 =400;%iterations at which learning rate will decrease
```

```
a_k1 = 0.15*a;%decread learning rate
randn('state',7)
W0 = 0.25*randn(12,5);
randn('state',19)
W1 = 0.25*randn(6,1);
m = 3;%number of indices
%Stochastic Gradient Descent Algorithm
t1 = cputime;%cputime starts here
[W0_s, W1_s] = SGD_NN(Xtr,ytr,W0,W1,a,K,K1,a_k1,m);
t2 = cputime;%cputime ends here
cpu_time = t2-t1;%calculate the total cpu time
```

d)Using Nesterov's Accelerated Gradient as the Weight Updating Scheme

```
a = 6.5e-5;%learning rate
K = 200;%number of iterations
randn('state',7)
W0 = 0.25*randn(66,1);%combine weights into a single matrix
% Nesterov's Accelerated Gradient Algorithm
t1 = cputime;%cputime starts here
[Ws,fs] = NAG_NN('f_NN','g_NN',W0,a,K);
t2 = cputime;%cputime ends here
cpu_time = t2-t1;%calculate the total cpu time
W_0=Ws(1:60);%Weights for input layer to hidden layer
W_1=Ws(61:66);%Weights for hidden layer to output layer
W0_s =reshape(W_0,12,5);%Weight Matrix for input layer to hidden layer
W1_s =reshape(W_1,6,1); %%Weight matrix for hidden layer to output layer
```

e)Using Adam as the Weight Updating Scheme

```
a = 0.0125;%learning rate
K = 1000;%number of iterations
b1 = 0.9;%exponential decay rate for 1st moment
b2 = 0.75;%exponential decay rate for second moment
m_ind  = 3; %number of indices
e = 1e-8;%convergence parameter
randn('state',7)
W0 = 0.25*randn(12,5);
randn('state',19)
W1 = 0.25*randn(6,1);
%Adams Algorithm
t1 = cputime;%cputime starts here
[W0_s,W1_s] = adam_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e);
t2 = cputime;%cputime ends here
cpu_time = (t2-t1);%calculate the total CPU time
```

e)Using AdaMax as the Weight Updating Scheme

```
a = 0.025;%learning rate
K = 1000;%number of iterations
b1 = 0.45;%exponential decay rate for 1st moment
b2 = 0.75;%exponential decay rate for second moment
m_ind  = 3; %number of indices
e = 1e-8;%convergence parameter
```

```
randn('state',7)
W0 = 0.25*randn(12,5);
randn('state',19)
W1 = 0.25*randn(6,1);
%AdaMax Algorithm
t1 = cputime;%cputime starts here
[W0_s,W1_s] = adamax_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e);
t2 = cputime;%cputime ends here
cpu_time = (t2-t1);%calculate the total CPU time
```

g)Using NAadam as the Weight Updating Scheme

```
a = 0.039;%learning rate
K = 1000;%number of iterations
b1 = 0.7;%exponential decay rate for 1st moment
b2 = 0.75;%exponential decay rate for second moment
m_ind  = 3; %number of indices
e = 0;%convergence parameter
randn('state',7)
W0 = 0.25*randn(12,5);
randn('state',19)
W1 = 0.25*randn(6,1);
%Nadam Algorithm
t1 = cputime;%cputime starts here
[W0_s,W1_s] = nadam_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e);
t2 = cputime;%cputime ends here
cpu_time = (t2-t1);%calculate the total CPU time
```

IV: Prediction for testing data samples and training data samples

```
%Use forward propagation
[~,~,~,y2_train] = fwp(Xtr,W0_s,W1_s);%run the forward propagation to get neural networks output
pin = round(y2_train);%round the output
cmp_train = pin'-ytr;%compare by subtraction
[~,~,~,y2_test] = fwp(Xtest,W0_s,W1_s);%run the forward propagation to get neural networks output
pout = round(y2_test);%round the output
cmp_test = pout'-ytest;%compare by subtraction
fprintf('The number of training class misclassifications is %d out of %d training samples\n',
nnz(cmp_train),size(ytr,1))
fprintf('The number of testing class misclassifications is %d out of %d testing samples\n',
nnz(cmp_test),size(ytest,1))
train_err_rate = nnz(cmp_train)/size(ytr,1)*100;%training error rate
test_err_rate = nnz(cmp_test)/size(ytest,1)*100;%testing error rate
fprintf('The training error rate is %2.2f%%\n',train_err_rate)
fprintf('The testing error rate is %2.2f%%\n',test_err_rate)
in_sample_error = 1/size(ytr,1)*norm((cmp_train),1);%in sample average prediction error
out_of_sample_error = 1/size(ytest,1)*norm((cmp_test),1);%out of sample average prediction error
fprintf('The in sample average prediction error is %.4f\n',in_sample_error)
fprintf('The out of sample average prediction error is %.4f\n',out_of_sample_error)
fprintf('The total CPU time for the algorithm is %2.4f seconds\n',cpu_time)
```

V: Results

a)Using Gradient Descent as the Weight Updating Scheme

```
The number of training class misclassifications is 1751 out of 3897 training samples
The number of testing class misclassifications is 458 out of 976 testing samples
The training error rate is 44.93%
The testing error rate is 46.93%
The in sample average prediction error is 0.4842
The out of sample average prediction error is 0.5102
The total CPU time for the algorithm is 65.2188 seconds
```

b)Using Memoryless BFGS as the Weight Updating Scheme

```
The number of training class misclassifications is 1747 out of 3897 training samples
The number of testing class misclassifications is 453 out of 976 testing samples
The training error rate is 44.83%
The testing error rate is 46.41%
The in sample average prediction error is 0.4850
The out of sample average prediction error is 0.5010
The total CPU time for memoryless BFGS algorithm is 14.0000 seconds
```

c)Using Stochastic Gradient Descent as the Weight Updating Scheme

```
The number of training class misclassifications is 1796 out of 3897 training samples
The number of testing class misclassifications is 456 out of 976 testing samples
The training error rate is 46.09%
The testing error rate is 46.72%
The in sample average prediction error is 0.5183
The out of sample average prediction error is 0.5195
The total CPU time for the algorithm is 0.8125 seconds
```

d)Using Nesterov's Accelerated Gradient as the Weight Updating Scheme

```
The number of training class misclassifications is 1769 out of 3897 training samples
The number of testing class misclassifications is 453 out of 976 testing samples
The training error rate is 45.39%
The testing error rate is 46.41%
The in sample average prediction error is 0.4924
The out of sample average prediction error is 0.5113
The total CPU time for running the algorithm is 6.8594 seconds
```

e)Using Adam as the Weight Updating Scheme

```
The number of training class misclassifications is 1809 out of 3897 training samples
The number of testing class misclassifications is 444 out of 976 testing samples
The training error rate is 46.42%
The testing error rate is 45.49%
The in sample average prediction error is 0.5163
The out of sample average prediction error is 0.5133
The total CPU time for the algorithm is 0.7031 seconds
```

f)Using AdaMax as the Weight Updating Scheme

```
The number of training class misclassifications is 1840 out of 3897 training samples
The number of testing class misclassifications is 439 out of 976 testing samples
```

```
The training error rate is 47.22%
The testing error rate is 44.98%
The in sample average prediction error is 0.5355
The out of sample average prediction error is 0.5154
The total CPU time for the algorithm is 0.8594 seconds
```

g)Using NAadam as the Weight Updating Scheme

```
The number of training class misclassifications is 1774 out of 3897 training samples
The number of testing class misclassifications is 437 out of 976 testing samples
The training error rate is 45.52%
The testing error rate is 44.77%
The in sample average prediction error is 0.5012
The out of sample average prediction error is 0.5010
The total CPU time for the algorithm is 0.6719 seconds
```

Appendix B

FUNCTIONS a) Forward Propagation

```
fwp_1 computes the signals along the forward path.
% Input:
% xn: input training data of size 11 x 3897.
% W0: weights and biases used for connecting input and hidden units.
% W0 is a matrix of size 12 by 5 organized as in Eq. (4.3) of the course notes.
% W1: weights and bias used for connecting hidden and output units.
% W1 is a matrix of size 6 by 1 organized as in Eq. (4.3) of the course notes.
% Output:
% a1: a matrix of size 5 by 3897 obtained by (L4.1).
% a2: a vector of size 1 by 3897 obtained by (L4.4)
% y1: a matrix of size 5 by 3897 obtained by (L4.2).
% y2: a vector of size 1 by 3897 which is the output of the network obtained by (L4.5)
function [a1,a2,y1,y2] = fwp(xn,W0,W1)
y0 = xn; yh0 = [ones(1,size(xn,2)); y0];
a1 = W0'*yh0;
y1 = 1./(1+exp(-a1));
yh1 = [ones(1,size(xn,2)); y1];
a2 = W1'*yh1;
y2 = 3.5 + 5./(1+exp(-a2));
end
```

b) Back Propagation

```
% bwp_1 computes the gradient of the L2 loss function using BP.
% Input:
% xn: xn: input training data of size 11 x 3897.
% a1: a matrix of size 5 by 3897 obtained by (L4.1).
% a2: a vector of size 1 by 3897 obtained by (L4.4)
% y1: a matrix of size 5 by 3897 obtained by (L4.2).
% y2: a vector of size 1 by 3897 which is the output of the network obtained by (L4.5)
% W0: weights and biases used for connecting input and hidden units.
% W0 is a matrix of size 12 by 5 organized as in Eq. (4.3) of the
% course notes.
% W1: weights and bias used for connecting hidden and output units.
% W1 is a matrix of size 6 by 1 organized as in Eq. (4.3) of the
% course notes.
% ytn: label vector of input data xn of size 3897 x 1.
% Output:
% dw0: a matrix of size 12 by 5, which is the 1st-order derivative of
% W0.
% dw1: a matrix of size 6 by 1, which is the 1st-order derivative of
% W1.
```

```
function [dw0,dw1] = bwp(xn,a1,a2,y1,y2,W0,W1,ytn)
ytn = ytn';
y0 = xn;
d1 = size(W0,2);
eyL = y2 - ytn;
df2 = 5*(exp(-a2)./((1+exp(-a2)).^2));
ea2 = eyL.*df2;
ey1 = W1(2:d1+1,:)*ea2;
df1 = exp(-a1)./((1+exp(-a1)).^2);
ea1 = ey1.*df1;
dw0 = [ones(1,size(y0,2)); y0]*ea1';
dw1 = [ones(1,size(y0,2)); y1]*ea2';
end
```

c)Gradient Descent for Neural Network

```
function [W0_s, W1_s] = grad_desc_NN(Xtr,ytr,W0,W1,alpha,K)
format compact
format long
k = 1;
[a1,a2,y1,y2] = fwp(Xtr,W0,W1);
[dw0,dw1] = bwp(Xtr,a1,a2,y1,y2,W0,W1,ytr);
while 1
    W0 = W0 - alpha*dw0;
    W1 = W1 - alpha*dw1;
    if k == K
        break
    end
    [a1,a2,y1,y2] = fwp(Xtr,W0,W1);
    [dw0,dw1] = bwp(Xtr,a1,a2,y1,y2,W0,W1,ytr);
    k = k + 1;
end
W0_s = W0;
W1_s = W1;
end
```

d)Memoryless BFGS for Neural Network

```
function [ws,fs] = bfgs_ML_NN(fname,gname,a, w0,K)
format compact
format long
wk = w0;
fk = feval(fname,wk);
gk = feval(gname,wk);
dk = -gk;
dtk = -a*gk;
wk_new = wk + dtk;
fk_new = feval(fname,wk_new);

for n=1:K
      gk_new = feval(gname,wk_new);
      gmk = gk_new - gk;
      gk = gk_new;
```

```
      rk = 1/(dtk'*gmk);
      if rk <= 0
         dk = -gk;
      else
         tk = dtk'*gk;
         qk = gk - (rk*tk)*gmk;
         bk = rk*(gmk'*qk - tk);
         dk = bk*dtk - qk;
      end
      fk = fk_new;
      wk = wk_new;
      dtk = a*dk;
      wk_new = wk + dtk;
end
ws = wk_new;
fs = feval(fname,ws);
```

e)Stochastic Gradient Descent for Neural Network

```
function [W0_s, W1_s] = SGD_NN(Xtr,ytr,W0,W1,a,K,K1,a_k1,m)
format compact
format long
k = 0;
while k<K
    rand('state',25+k)
    p = randperm(3897);
    dw0_sum = zeros(12,5);
    dw1_sum = zeros(6,1);
    for i = p(1:m)
        [a1,a2,y1,y2] = fwp(Xtr(:,i),W0,W1);
        [dw0,dw1] = bwp(Xtr(:,i),a1,a2,y1,y2,W0,W1,ytr(i));
        dw0_sum = dw0_sum + dw0;
        dw1_sum = dw1_sum + dw1;
    end
    dk0 = -1/m*dw0_sum;
    dk1 = -1/m*dw1_sum;
    if k<=K1
        a_k  = (1-k/K1)*a + (k/K1)*a_k1;
    else
       a_k = a_k1;
    end
    W0 = W0 + a_k*dk0;
    W1 = W1 + a_k*dk1;
    k = k+1;
end
W0_s = W0;
W1_s = W1;
end
```

f)Nesterov's Accelerated Gradient for Neural Network

```
function [ws,fs] = NAG_NN(fname,gname,W0,a,K)
k = 1;
```

```
tk = 1;
yk = W0;
wk_minus_one = W0;
f = feval(fname,W0);
format long
format compact
for k = 1:K
    gk = feval(gname,yk);
    wk = yk - a*gk;
    tk_plus_one = 0.5*(1 + sqrt(1+4*tk^2));
    yk_plus_one = wk + ((tk - 1)*(wk-wk_minus_one))/tk_plus_one;
    fnew = feval(fname,wk);
    f = [f;fnew];
    tk = tk_plus_one;
    yk = yk_plus_one;
    wk_minus_one = wk;
    k = k+1;
end
format short
ws = wk;
fs = feval(fname,wk);
end
```

g)Adam for Neural Network

```
function [W0_s,W1_s] = adam_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e)
%b1,b2 are exponential decay rates for the moment estimates
m = zeros(66,1); %initialize first moment vector
v = zeros(66,1); %initialize second moment vector
k = 0; %initialize time step
while k<=K
    k = k + 1;
    W = [W0(:);W1(:)];
    p = randperm(3897);
    dw0_sum = zeros(12,5);
    dw1_sum = zeros(6,1);
    for i = p(1:m_ind)
        [a1,a2,y1,y2] = fwp(Xtr(:,i),W0,W1);
        [dw0,dw1] = bwp(Xtr(:,i),a1,a2,y1,y2,W0,W1,ytr(i));
        dw0_sum = dw0_sum + dw0;
        dw1_sum = dw1_sum + dw1;
    end
    dw0_stoc = 1/m_ind*dw0_sum;
    dw1_stoc = 1/m_ind*dw1_sum;
    g =[dw0_stoc(:);dw1_stoc(:)];%get gradients
    m = b1*m + (1-b1)*g;%update biased first moment estimate
    v = b2*v + (1-b2)*g.^2;%update biased second moment estimate
    m_cap = m/(1-(b1)^k);%compute bias corrected first moment estimate
    v_cap = v/(1-(b2)^k);%compute bias corrected second moment estimate
    W = W - a*(m_cap./(sqrt(v_cap) + e));%update parameters
    W_0=W(1:60);
    W_1=W(61:66);
    W0 =reshape(W_0,12,5);
```

```
    W1 =reshape(W_1,6,1);
end
    W0_s = W0;%solution weights
    W1_s = W1;%solution weights
end
```

h)AdaMax for Neural Network

```
function [W0_s,W1_s] = adamax_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e)
%b1,b2 are exponential decay rates for the moment estimates
m = zeros(66,1); %initialize first moment vector
v = zeros(66,1); %initialize second moment vector
k = 0; %initialize time step
while k<=K
    k = k + 1;
    W = [W0(:);W1(:)];
    p = randperm(3897);
    dw0_sum = zeros(12,5);
    dw1_sum = zeros(6,1);
    for i = p(1:m_ind)
        [a1,a2,y1,y2] = fwp(Xtr(:,i),W0,W1);
        [dw0,dw1] = bwp(Xtr(:,i),a1,a2,y1,y2,W0,W1,ytr(i));
        dw0_sum = dw0_sum + dw0;
        dw1_sum = dw1_sum + dw1;
    end
    dw0_stoc = 1/m_ind*dw0_sum;
    dw1_stoc = 1/m_ind*dw1_sum;
    g =[dw0_stoc(:);dw1_stoc(:)];%get gradients
    m = b1*m + (1-b1)*g;%update biased first moment estimate
    v = max(b2*v,norm(g));%update the exponentially weighted infinity norm
    W = W - (a/(1-(b1)^k))*(m/v);%update parameters
    W_0=W(1:60);
    W_1=W(61:66);
    W0 =reshape(W_0,12,5);
    W1 =reshape(W_1,6,1);
end
    W0_s = W0;%solution weights
    W1_s = W1;%solution weights
end
```

i)Nesterov's Accelerated Adam for Neural Network

```
function [W0_s,W1_s] = adamax_NN(Xtr,ytr,W0,W1,a,m_ind,b1,b2,K,e)
%b1,b2 are exponential decay rates for the moment estimates
m = zeros(66,1); %initialize first moment vector
v = zeros(66,1); %initialize second moment vector
k = 0; %initialize time step
while k<=K
    k = k + 1;
    W = [W0(:);W1(:)];
    p = randperm(3897);
    dw0_sum = zeros(12,5);
```

```
    dw1_sum = zeros(6,1);
    for i = p(1:m_ind)
        [a1,a2,y1,y2] = fwp(Xtr(:,i),W0,W1);
        [dw0,dw1] = bwp(Xtr(:,i),a1,a2,y1,y2,W0,W1,ytr(i));
        dw0_sum = dw0_sum + dw0;
        dw1_sum = dw1_sum + dw1;
    end
    dw0_stoc = 1/m_ind*dw0_sum;
    dw1_stoc = 1/m_ind*dw1_sum;
    g =[dw0_stoc(:);dw1_stoc(:)];%get gradients
    g_cap = g/(1-b1^k);
    m = b1*m + (1-b1)*g;%update biased first moment estimate
    v = b2*v + (1-b2)*g.^2;%update biased second moment estimate
    m_cap = m/(1-b1^(k+1));%compute bias corrected first moment estimate
    v_cap = v/(1-b2^k);%compute bias corrected second moment estimate
    m_bar = (1-b1)*g_cap + b1*m_cap;
    W = W - a*(m_bar./(sqrt(v_cap) + e));%update parameters
    W_0=W(1:60);
    W_1=W(61:66);
    W0 =reshape(W_0,12,5);
    W1 =reshape(W_1,6,1);
end
    W0_s = W0;%solution weights
    W1_s = W1;%solution weights
end
```