CHAPTER 3
PASS WORD AUTHENTICATION USING BACK PROPAGATION

3.1 Introduction:

This Chapter deals with the introduction of Back propagation Neural Network (BPNN) and how it can be used for overcoming the pitfalls in the traditional Password Authentication Schemes (PAS). Before see how this work can overcome the pitfalls of PAS, let us discuss about basics of back propagation and traditional password authentication scheme.

3.2 Back Propagation:

Back propagation (BP) is the best-known training algorithm for multi-layer neural networks. It defines rules of propagating the network error back from network output to network input units and adjusting network weights along with this back propagation. It requires lower memory resources than most learning algorithms and usually gets acceptable results, although it can be too slow to reach the error minimum and sometimes finds no best solution. Back-propagation can also be considered a generalization of the delta rule for non-linear activation functions and multilayer networks. Back propagation is the generalization of the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Back-propagation can also be considered a generalization of the delta rule for non-linear activation functions and multilayer networks.

Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities. Standard back propagation is a gradient
descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term back propagation refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods.

3.2.1 Training:

Once the network weights and biases are initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior network inputs $p$ and target outputs $t$. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function. The default performance function for feed forward networks is mean square error (MSE)—the average squared error between the networks outputs and the target outputs $t$.

All Back propagation algorithms use the gradient of the performance function to determine how to adjust the weights to minimize performance. The gradient is determined using a technique called back propagation, which involves performing computations backward through the network. The back propagation computation is derived using the chain rule of calculus.

3.2.2 Algorithm:

There are many variations of the back propagation algorithm, several of which are described in next section. The simplest implementation of back propagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as
\[ x_{k+1} = x_k - \alpha_k g_k, \]  \hspace{1cm} \text{(3.1)}

Where \( x_k \) a vector of current weights and biases, \( g_k \) is the current gradient, and \( \alpha_k \) is the learning rate. There are two different ways in which this gradient descent algorithm can be implemented: Incremental mode and Batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs are applied to the network before the weights are updated. The general architecture of the back propagation algorithm is presented in figure 3.1.

![General model of Back propagation Algorithm](image)

**Figure 3.1. General model of Back propagation Algorithm**

### 3.2.3 Activation functions:

We considered three activation functions for hidden layers and four activation functions for the output layer.

For hidden layers can be selected the following functions:
**Linear:** This function produces its input as its output or, in other words, just passes the activation level directly as the output. Its output range is $[-\infty, +\infty]$.

**Logistic:** This function has a sigmoid curve and is calculated using the following formula:

$$F(x) = \frac{1}{1 + e^{-\lambda x}} ,$$

Where $\lambda$ is a learning rate parameter, Its output range is $[0...1]$. This function is used most often and is set by default.

**Hyperbolic tangent:** This function also has a sigmoid curve and is calculated using the following formula:

$$F(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} ,$$

Its output range is $[-1, 1]$. Empirically, it is often found that this function performs better than the Logistic function. We recommend experimenting with it if needs to be improved network performance.

For output layer the same functions can be selected in case of a regression type of problem and Logistic or Softmax function goes with a classification type problem.

**Softmax:** This is exponential function. With this function possible outputs are normalized so that the sum of activations of all output layer units is equal to 1. Its output range is $[0..1]$. With Softmax function network outputs can be interpreted as probabilities of category membership.
\[ p_i = \frac{\exp(a_i)}{\sum_{j=1}^{n} \exp(a_j)} \]

Where \( p_i \) is the value of an output node, \( a_i \) is the net input to an output node, and \( n \) is the number of output nodes.

### 3.2.4 Error Functions:

This work considers two network error functions: Sum-of-Squares and Cross-entropy. Minimization of the error function is the main objective of neural network training. The value of the error function is used to rate the quality of the neural network.

**Sum-of-Squares** is the most common error function. It is best fit for regression problems and can be used for classification problems too. The error is the sum of the squared differences between the actual value (target column value) and neural network output.

**Cross-entropy** is specially designed for classification problems where the network output is interpreted as the probability that the input pattern belongs to a certain class (category). It can be used only with Logistic or Softmax activation functions in the output layer. This error is the sum of the products of the target value and the logarithm of the error value on each output unit.

### Classification Models:

This work considers two classification models: Winner-takes-all and Confidence-limits. The **Winner-takes-all** model performs classification by selecting an output unit with the biggest activation level. The **Confidence limits** model performs classification by checking output unit activation against two levels: the Accept level and the Reject level. If a network has one output unit (Unit_N) with activation above the Accept level and other output units with activation below the Reject level, the
network output would be the class that corresponds to the Unit_N. In all other cases classification is considered to be impossible and classification result is represented with "?" mark.

3.2.5 Heuristics:

There are some heuristics methods for improvement of back propagation algorithms to achieve the best correct classification rate (CCR).

Weight Randomization Method:

Weights randomization methods are used to initialize network weights before training. The main purpose of weights randomization techniques is to avoid sigmoid saturation problems that cause slow training. We considered four kinds of weight randomization methods that perform correct classification on data sets using the chosen back propagation algorithms. The following types are:

- Manual randomization range (MRR).
- Automatic randomization range (ARR).
- Optimized for Uniform distribution of network inputs (OUDR).
- Optimized for Gaussian distribution of network inputs (OGDR).

Hidden Layers:

A hidden layer is any layer of a neural network between the input and output layers. Hidden layers provide the neural network's non-linear modeling capabilities. A network with too few hidden units only roughly discovers hidden dependencies in our data whereby the network produces a significant number of errors. A network with too many hidden units will tend to memorize all our data instead of finding relations that also lead to bigger network errors. This work found that the best solution for many problems and their dataset. In this study, the majority of problems are
solved best with 1 hidden layer, another part is solved best with 2 layers and only some problems require 3 layers or more.

The maximum number of hidden units is rarely required to exceed the number of inputs by more than 4 times. This work strictly recommend retraining each architecture at least 3 times with different initial weight randomizations and saving only the best one for comparison with other architectures. The study recommends experimenting with different data partition and saving only the best result for future comparison with other architectures.

3.2.6 Learning:

Whenever new users are creating accounts network has to adjust weights so that it can recognize all the users who are registered. This process of changing weights is called learning. Here in order to do learning this scheme used Back Propagation Neural Network (BPNN) algorithm. The Learning rate is an important part of the more sophisticated error correction learning schemes. The maximal number of learning steps depends on the learning rate and the initial weight vector and on the general random sequence of training patterns submitted. There are two alternatives here, either to keep the learning rate small and fixed or to change it during the iterative adaptation procedure. The smaller is the smoother, but slower will be the approach to the optimal. The learning rate controls the stability and rate of adaptation.

A typical rule of thumb is to start with some larger learning rate and reduce it during optimization. Many improved algorithms have been proposed in order to find a reliable and fast strategy for optimizing the learning rate in a reasonable amount of computing time. The learning rate parameter used by several learning algorithms, which affects the changing of weights. The bigger learning rates cause bigger weight changes during each iteration.
3.3 Password Authentication Procedure:

This work on password authentication uses two types of passwords: Textual password – alphanumerical character set and Graphical password – predefined images or stored pictures. This scheme may use any one of these two passwords and can train the network, so that it can authenticate users.

3.3.1 Textual password:

Textual password will be converted into its corresponding probabilistic values and those values can be used as input to the neural network.

<table>
<thead>
<tr>
<th>Password</th>
<th>Normalized Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>LETMEIN</td>
<td>0.44#0.16#0.76#0.48#0.16#0.32#0.52</td>
</tr>
<tr>
<td>APPLE</td>
<td>0#0.6#0.6#0.44#0.16#</td>
</tr>
<tr>
<td>GETIN</td>
<td>0.24#0.16#0.76#0.32#0.52</td>
</tr>
</tbody>
</table>

Here we used ‘#’ as delimiter to separate each probabilistic value

*Table 3.1. Normalized values for text password*

3.3.2 Graphical password:

Before giving the image as password the image should be converted in to its Read, Green and Blue (RGB) values and these values will be normalized using our normalization function. As we cannot give the image directly as input to the neural network, here we converted image into matrix (or text).

**Conversion of image to matrix (or text):**

This method reads the color of each pixel of the image. Then converts the color into red, green and blue (RGB) parts as each color can be produced using these colors.
By using the above procedure we can convert any image into a matrix consisting of set of numbers representing all the pixels of the image. After converting image into a matrix which consists of set of numbers we can give it to the neural network as input and we can train it using username and image matrix as a training sample. So here user has the choice of selecting either text or an image depending on the requirement and security he expects.

This scheme normalizes numbers in matrix given in figure 3.3 to increase the security. As Red, Blue, Green can have maximum of 255 and minimum of 0, it uses the following formula to normalize the numbers of matrix.
Figure 3.3. Matrix representation of an image.

\[
\begin{bmatrix}
135 & 206 & 235 & 154 & 85 & 25 & 69 & 158 & \cdots & 196 \\
148 & 58 & 157 & 35 & 154 & 129 & 35 & 78 & \cdots & 254 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
148 & 58 & 157 & 35 & 154 & 129 & 35 & 78 & \cdots & 148
\end{bmatrix}
\]

\[C_n = \frac{C_t}{255} \quad (3.5)\]

Where \(C_t\) is the value of Red or Green or Blue components of the graphical password and 255 is the maximum value for any color.

3.4 Implementation details:

The network layers in the figure above are implemented as arrays of structures. The nodes of the layers are implemented as follows:

```
[Serializable]
struct PreInput
{
    public double Value;
    public double[] Weights;
};
```

```
[Serializable]
struct Input
{
    public double InputSum;
};
```
public double Output;
public double Error;
public double[] Weights;
};

[Serializable]
struct Hidden
{
    public double InputSum;
    public double Output;
    public double Error;
    public double[] Weights;
};

[Serializable]
struct Output<T> where T : IComparable<T>
{
    public double InputSum;
    public double Output;
    public double Error;
    public double Target;
    public T Value;
};

The layers are implemented as follows

private PreInput[] PreInputLayer;
private Input[] InputLayer;
private Hidden[] HiddenLayer;
private Output<string>[] OutputLayer;
3.4.1 Training the network:
The training of the network happens step by step as follows:

- Apply normalized input to the neural network.
- Calculate the output in the form of normalized values in the range [0...1].
- Compare the resulting output with the desired output for the given input. This is called the error.
- Modify the weights for all neurons using the error.
- Repeat the process until the error reaches an acceptable value (e.g. error < 1%), which means that the neural network was trained successfully, or if we reach a maximum count of iterations, which results that the neural network training was not successful.

**It is represented as shown below:**

```c
void TrainNetwork(TrainingSet, MaxError)
{
    while(CurrentError > MaxError)
    {
        foreach(Pattern in TrainingSet)
        {
            ForwardPropagate(Pattern); // calculate output
            BackPropagate(); // fix errors, update weights
        }
    }
}
```
3.4.2 Implementing Training Using BP:

```csharp
public bool Train()
{
    double currentError = 0;
    int currentIteration = 0;
    NeuralEventArgs Args = new NeuralEventArgs();

    do
    {
        currentError = 0;
        foreach (KeyValuePair<T, double[]> p in TrainingSet)
        {
            NeuralNet.ForwardPropagate(p.Value, p.Key);
            NeuralNet.BackPropagate();
            currentError += NeuralNet.GetError();
        }

        currentIteration++;

        if (IterationChanged != null && currentIteration % 5 == 0)
        {
            Args.CurrentError = currentError;
            Args.CurrentIteration = currentIteration;
            IterationChanged(this, Args);
        }
    }
    while (currentError > maximumError && currentIteration < maximumIteration && !Args.Stop);
```
if (IterationChanged != null)
{
    Args.CurrentError = currentError;
    Args.CurrentIteration = currentIteration;
    IterationChanged(this, Args);
}

if (currentIteration >= maximumIteration || Args.Stop)
    return false;//Training Not Successful

return true;

3.4.3 Forward Propagation:
private void ForwardPropagate(double[] pattern, T output)
{
    int i, j;
    double total;
    //Apply input to the network
    for (i = 0; i < PreInputNum; i++)
    {
        PreInputLayer[i].Value = pattern[i];
    }
    //Calculate The First(Input) Layer's Inputs and Outputs
    for (i = 0; i < InputNum; i++)
    {
        total = 0.0;
        for (j = 0; j < PreInputNum; j++)
\{
    total += PreInputLayer[j].Value * PreInputLayer[j].Weights[i];
\}
InputLayer[i].InputSum = total;
InputLayer[i].Output = F(total);
\}
//Calculate The Second(Hidden) Layer's Inputs and Outputs
for (i = 0; i < HiddenNum; i++)
{
    total = 0.0;
    for (j = 0; j < InputNum; j++)
    {
        total += InputLayer[j].Output * InputLayer[j].Weights[i];
    }
    HiddenLayer[i].InputSum = total;
    HiddenLayer[i].Output = F(total);
}
//Calculate The Third(Output) Layer's Inputs, Outputs, Targets and Errors
for (i = 0; i < OutputNum; i++)
{
    total = 0.0;
    for (j = 0; j < HiddenNum; j++)
    {
        total += HiddenLayer[j].Output * HiddenLayer[j].Weights[i];
    }
    OutputLayer[i].InputSum = total;
OutputLayer[i].output = F(total);
OutputLayer[i].Target = OutputLayer[i].Value.CompareTo(output) == 0 ? 1.0 : 0.0;
OutputLayer[i].Error = (OutputLayer[i].Target - OutputLayer[i].output) * (OutputLayer[i].output) * (1 - OutputLayer[i].output);

3.4.4 Back Propagation Implementation:
private void BackPropagate()
{
    int i, j;
    double total;
    //Fix Hidden Layer's Error
    for (i = 0; i < HiddenNum; i++)
    {
        total = 0.0;
        for (j = 0; j < OutputNum; j++)
        {
            total += HiddenLayer[i].Weights[j] * OutputLayer[j].Error;
        }
        HiddenLayer[i].Error = total;
    }
    //Fix Input Layer's Error
    for (i = 0; i < InputNum; i++)
    {
        total = 0.0;
        for (j = 0; j < OutputNum; j++)
        {
            total += HiddenLayer[i].Weights[j] * OutputLayer[j].Error;
        }
        HiddenLayer[i].Error = total;
    }
}
for (j = 0; j < HiddenNum; j++)
{
    total += InputLayer[i].Weights[j] * HiddenLayer[j].Error;
}
InputLayer[i].Error = total;
}

#pragma region Update The First Layer's Weights
for (i = 0; i < InputNum; i++)
{
    for(j = 0; j < PreInputNum; j++)
    {
        PreInputLayer[j].Weights[i] +=
            LearningRate * InputLayer[i].Error * PreInputLayer[j].Value;
    }
}
#pragma endregion

#pragma region Update The Second Layer's Weights
for (i = 0; i < HiddenNum; i++)
{
    for (j = 0; j < InputNum; j++)
    {
        InputLayer[j].Weights[i] +=
            LearningRate * HiddenLayer[i].Error * InputLayer[j].Output;
    }
}
#pragma endregion

#pragma region Update The Third Layer's Weights
for (i = 0; i < OutputNum; i++)
{
    for (j = 0; j < HiddenNum; j++)
    {

```
HiddenLayer[j].Weights[i] +=
    LearningRate * OutputLayer[i].Error * HiddenLayer[j].Output;
}

3.5 Results For PAS using BPNN:
BPNN For Textual Passwords:

![Figure 3.4. Screen showing how to setup network and backpropagation algorithm](image)

This screen indicates the Maximum Error and maximum error that we may ignore while training the network. Error can be decreased either by changing learning rate parameter or by changing number of input and hidden units. Number of patterns are restricted by number of output units, if we want to recognize more patterns we have to use more output units.
Training the Network Using BPNN:

This scheme used back propagation algorithm to train the network. Since backpropagation method is very slow, it may take more time to train the network. Once the training is completed it will displays the following message.

Figure 3.5. Screen showing training the network using BP.

Figure 3.6. Training completed message
User Authentication:

Once the network is trained the values will be stored in the weight table and whenever user enters his password, the network fetches the values from the weight table and compares with the ciphertext. If both are same user will be authenticated otherwise invalidate the user.

This application provides fields for entering the input data, settings and to see unique code for entered data. It also provides buttons for training the given input and recognizing the given password as shown in figure 3.7
BPNN For Graphical Passwords:
This method provides options for specifying the number of layers, maximum error and number of output units. It also specifies the directory which contains the required pattern.

![Backpropagation Neural Network for Image Recognition](image)

Figure 3.8. Screen showing how to setup network and backpropagation algorithm

In figure 3.8, the Maximum Error indicates maximum error that we may ignore while validating the user. In order to decrease error, change the learning rate parameter and change the number of input and hidden units. Number of patterns are restricted by value in the Number of Output Unit. In order to recognize more patterns, use more output units.
Figure 3.9 Screen shot showing Patterns

Training the Network Using BP:

Figure 3.9 shows the screen with full of patterns. After selecting a particular pattern if click the “Train Network” button it will load all the images located in the specified directory and it will adjust the weights.

Figure 3.10 Training process
Validating the User:

Once the training is completed inorder to select an image that is to be tested, click the “Browse“ button. After clicking “Recognize“ button it loads image as input and try to recognize it , if it can recognize the user is authorized. the n it will consider the pattern that matched with less error. In the figure 3.11 the matched pattern E is having 93% similarity where as 6 is having 6% similarity . it is having E is having minimum error our algorithm recognizes E as matched pattern.

Figure 3.11. User validation
3.6 Conclusion:

In this chapter password authentication using back propagation is implemented for both textual and graphical passwords. In the training process normalized input values were used for enhancing the password authentication.

3.6.1 Advantages:

- This method is very difficult to attack (Provides More Security). To decrypt cipher text attacker has to identify weight matrix (Even one element of matrix changes attacker can’t decrypt), number of hidden layers, output function, character set (Including order of characters in character set), minimum and maximum values used in character set.

- Even though we use existing character set we can improve security by changing order of characters or minimum value to change unique number and probabilistic values associated to each character so that attacker may confuse in guessing the unique numbers or probabilistic values.

- We can increase security by increasing no. of Hidden layers.

- The users (Organization) of this algorithm can define their own character set, by doing so users can add new characters into their character set.

3.6.2 Disadvantages:

Since the output of this BPNN method is in the form of probabilistic values, the system can introduces noise, due which we may not do the efficient authentication.

Training time for BPNN is extremely large. Easy to remember passwords are vulnerable to password guessing attacks [43].