Artificial Neural Networks

Implementation

Prakash Waghela

B221256

2017

## Implementation of the MLP Learning Algorithm

By implementing the multi-layer perception learning algorithm, I could construct and train an artificial neural network which can estimate the index flood of an area, given several parameters as input values to the model.

### Network Architecture

To implement the MLP algorithm, I have implemented the following network architecture.

#### Nodes

The lowest level component of the network is a node. A Node is an entity that can take some input parameters and produce some output as a function of the input.

There are three types of nodes used in the network:

* Input Nodes - Input nodes are used to feed data into the network.
* Hidden Nodes - Hidden nodes are used to manipulate how much the input data should affect the output.
* Output Nodes - Output nodes are used to get some data out of the network.

#### Layers

Each type of node is grouped into three layers; the input, hidden and output layers respectively. Each node in each layer is fully connected with the nodes in the layer preceding it.

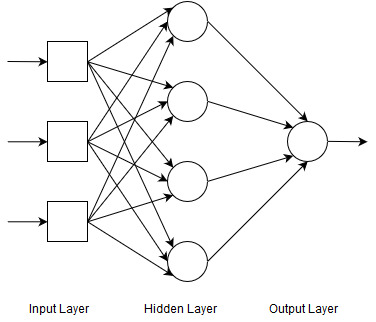


Figure - A blank network configuration

#### Weights and Bias

As shown in figure 1, each node has several connections to another node. These connections represent the output of a node in each layer being used as the input for a node in the succeeding layer. Each of these connections also has a weight value associated with it which is used to adjust how much the output of that node will affect the overall output of the network.

Each node in the hidden and output layers has a numerical bias value which allows the network to tweak the output of the node. Each bias also has a weight associated with it as shown in figure 2. Thus, there are three types of weights in the network:

* Bias weight
* Input node -> Hidden node weight
* Hidden node -> Output node weight

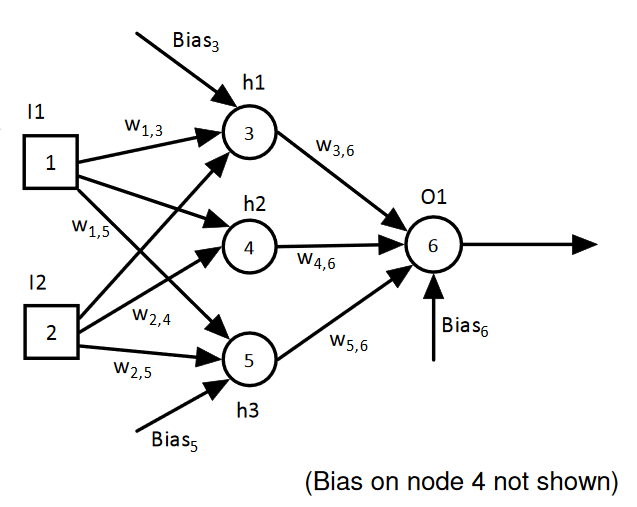


Figure - A network configuration showing biases and weights

### Backpropagation

The MLP Learning algorithm learns by processing a data set, row by row, and adjusting the weights in the network depending on how well the network performs for that row of data (Duffner & Garcia, 2007). The aim is to find the weight value which will produce the lowest network error, thus a gradient decent method is used to calculate weight changes.

First a forward pass is made through the network; the input parameters from the row of data is passed into the network through the input nodes. These nodes apply a weight to each parameter and pass it onto the nodes in the hidden layer, where each parameter is summed and the output of the node is calculated via an activation function such as the sigmoid function. In a similar fashion, the hidden nodes will then apply a weight to its output, as well as apply (by summation) a weighted bias, before calculating and passing the node output onto the output node. The output node repeats this process to produce the network output.

Thus, these calculations can be expressed as:

Equation - Calculating the output of a node

Where is the sum of the inputs to node j, is the weight from node i to node j, is the output of node i and is the output of node j. Biases are also represented in this summation as a special case where is set to 1.

Once the network output has been calculated, the derivative of the output is calculated and used to find the gradient of the output in relation to the network error. Thus, this calculation can be expressed as:

Equation - Calculating the gradient of output for the output node

The product of the error of the network and the derivative of the activation (sigmoid) function, where is the gradient of the output in relation to the network error, C is the correct output for that row, and is the predicted output for that row.

This gradient is in turn used to calculate the gradient in relation to the error of each node in the hidden layer. As there is a single output node for the network, this calculation can be simplified as:

Equation - Calculating the gradient of the output for a hidden node

Where is the gradient of the output of node j in relation to the error, is the weight from node j to the output node o, is the gradient of the output node o in relation to the error and is the output for the node j.

Once this is known for each hidden layer node, the weights can be updated accordingly. The gradient enforces whether the weight should be increased or decreased, thus the calculation can be expressed as:

Equation - Calculating the new weight

Where is the new weight from node i to node j, is the current weight from node i to node j, is the step parameter (typically 0.1) and is the output for node i.

This process is completed sequentially for all nodes in the network, for all rows of available data, until some halting condition is satisfied.

### MLP Implementation

To implement the MLP algorithm, I have constructed the network architecture in C++.

To begin, the network is constructed. It is made up of several nodes, with the exact number depending on how many individual input parameters I feed into the network and how many hidden nodes I decide to use. I have stored the nodes in a network in a flat, one dimensional vector. This can be seen in figure 3.



Figure - Code snipper show the initialising the network

Then, I populate a matrix to store the weights for each connection. The matrix is structured such that if there is a node x that is connected to a node y, the weight for this connection is stored in weightsMatrix[x][y]. Also, the bias weight for a node z is stored in the matrix at position weightsMatrix[0][z].



Figure - Code snippet showing the initialisation of the weights matrix

The weights are initialised to a random value in the range:

Figure - The range of initial weights

Where n in the number of input nodes and w is the randomised weight.

At this point, the network has been fully initialised and is ready to begin training. A forward pass through the network is made to calculate the output of the network with a row of data.



Figure - Code snipper showing data being fed into the network

This data is then passed to the hidden nodes; two loops are used to pass data from each input node to each hidden node, taking the relevant weights and biases into account.



Figure - Code snippet showing the output of the hidden nodes being calculated

The function node.setNodeOutput(value) is used to calculate the output of the node via Equation 1, with the implementation of the function depending on the type of node.



Figure - Code snippet showing the implementation of Equation 1

Once the hidden layer has completed computing its outputs, they are passed onto the output node, taking the relevant weights and biases into account.



Figure - Code snippet showing the network output being calculated

This concludes the forward pass. The next step is the backwards pass; first the slope gradient is calculated for the output node and each hidden node via Equation 2.



Figure - Code snippet showing the network output being calculated

Following this, the weights are updated using Equation 4. This uses loops to access the relevant entries of the matrix.



Figure - Code snippet showing the weights being updated

This concludes the backward pass.

### Modifications and Improvements

There are several modifications to the standard backpropagation algorithm that I have implemented to fine-tune the performance of the networks produced by the algorithm.

#### Momentum

Momentum is a method used to improve the quality of weight changes by considering the change in weight that was made in the last pass. Thus, the calculation to calculate a weight using momentum is:

Equation - Calculating the weight change with momentum

Where is the new weight from node i to node j, is the current weight from node i to node j, is the step parameter (typically 0.1), is the output for node i, is the percentage you want the previous change to influence the new weight change (typically 0.9) and is the change in weight during the last pass.

This was implemented by storing the weight changes from the previous pass in a matrix, and including this when the weights are updated.



Figure - Code snippet showing the weight changes with momentum

#### Bold Driver

Bold Driver is a method used to refine the step parameter used while updating the weights. It works by checking whether a pass through the data has improved the network. If the network has improved, then the step parameter is increased so larger changes can be made to the weights. Otherwise, the step parameter is reduced and the changes made during the pass are reverted. This is repeated until the network error is no longer improved at the minimum step parameter.

A small increase is made to the step parameter (10% increase) so the step parameter will steadily increase if the network is improving. However, if the network evaluates to perform worse, then a large change is made to the step parameter (50% decrease) before continuing training. This is to allow the network to recover the lost ground whilst it continues training. A minimum step parameter is required to be set to prevent the step parameter from decreasing to zero and the network getting stuck in a training loop.

Thus, the general change in the step parameter over time with the bold driver approach is show in Figure 13.

Figure - Graph showing how the step parameter changes over time with bold driver

This was implemented by creating a backup of the weights matrix and node list before training and reverting the network to use those backups if the network error increased.



Figure - Code snippet showing bold driver

#### Annealing

Annealing is another method used to refine the step parameter. It takes advantage of the fact that the closer to the minimum error value the network gets, the smaller the steps required to find the minimum. However, it also considers that the smaller steps are most likely to be most effective if the network is close to the minimum. Thus, the method will reduce the step parameter as shown in Figure 15.

Figure - Graph showing annealing the step parameter

Once the minimum step parameter has been reached, the network will continue training with the minimum step parameter until the minimum has been found.

I have implemented annealing as a follow up procedure to the bold driver approach. When using bold driver to train a network, I noticed that the step parameter would oscillate between the minimum and one step above the minimum (min \* 1.1). This was causing a high number of epochs to be thrown away as the network would degrade after training with the higher step parameter. I decided to implement annealing to slowly reduce the step parameter from the point at which the step parameter began to oscillate to improve the fine-tuning training phase of the network.

Figure - Graph showing bold driver and annealing hybrid



Figure - Code snippet showing annealing

An improvement I have made to the standard annealing method is to adjust the rate at which the step parameter will be annealed depending on the number of hidden nodes in the network. I found that networks tended to exit the training process prematurely; that is, before the step parameter is minimised.



Figure - Code snippet showing the modifications to annealing

I found that I could maximise the amount of time a network would train for, and minimise the step parameter at termination by calculating the rate at which the step parameter should be annealed using the number of hidden nodes in the network.

## Data Pre-processing

The supplied data has the following columns of data available:

* Input parameters
  + AREA - Catchment area in km2
  + BFIHOST - Base flow index
  + FARL - Flood attenuation due to reservoirs and lakes
  + FPEXT - Flood plain extent
  + LDP - Longest drainage path
  + PROPWET - Proportion of wet days
  + RMED-1D - Median annual maximum 1-day rainfall
  + SAAR - Standard annual average rainfall
* Predictand
  + Index Flood - Median of the annual maximum series of catchment flow in m3/s

### Data Cleaning

Before parsing the input data to the network, I developed several stages in MS Excel to check the data for anomalous or erroneous values and select which data should be used in the model.

1. Create a correlation table of each input parameter against the flood index.

This step is designed to highlight which columns should be considered as useful input parameters. It utilises the CORREL function in MS Excel and shows if and how much each column correlates with the index flood.

The (absolute) output from this step is shown in Table 1.

Table - Shows the correlation of each input parameter to the flood index

|  |  |
| --- | --- |
| **Input Parameter** | **Correlation to Index Flood (%)** |
| AREA | 72.1% |
| BFIHOST | 29.4% |
| FARL | 2.1% |
| FPEXT | 2.1% |
| LDP | 66.4% |
| PROPWET | 40.4% |
| RMED-1D | 18.0% |
| SAAR | 23.9% |

This step clearly highlights two columns which have a low correlation rate; FARL and FPEXT. This would suggest that these two columns could be removed from the data set to improve the accuracy that can be achieved with the training algorithm, thus both columns were removed from the data set.

1. Create a validation matrix for each column of data.

This step is designed to highlight any statistical errors that exist in the data set. The matrix has been implemented by utilising the following measures:

**Measure**: Is value numerical?  
**Formula**: =SUMPRODUCT(ISNUMBER(*[Column]*) + 0) < (COUNTA(*[Column]*)-1)**Description**: This formula will check if each cell in the column of data is numeric, returning a Boolean array. The array is then summed and checked against the number of elements in the column, returning TRUE if there exist any cells which are not numeric or FALSE if all cells are numeric.

**Measure**: Is value textual?  
**Formula**: =SUMPRODUCT(ISTEXT([*Column]*) + 0) > 0  
**Description**: This formula will check if each cell in the column of data is textual, returning a Boolean array. The array is then summed, returning TRUE if there exist any cells which contain text or FALSE if no textual cells are found.

**Measure**: Is value blank?  
**Formula**: =SUMPRODUCT(ISBLANK([*Column]*) + 0) > 0  
**Description**: This formula will check if each cell in the column of data is empty, returning a Boolean array. The array is then summed, returning TRUE if there exist any cells which are empty or FALSE if no empty cells are found.

**Measure**: Is value negative?  
**Formula**: =IF(COUNTIF([*Column]*,"<0") > 0, TRUE, FALSE)  
**Description**: This formula will check if each cell in the column of data contains a negative value, returning a Boolean array. The array is then summed, returning TRUE if there exist any cells which are negative or FALSE if no negative cells are found.

Applying these conditions on each column created the validation matrix shown in Table 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **AREA** | **BFIHOST** | **LDP** | **PROPWET** | **RMED-1D** | **SAAR** | **Index flood** |
| **Numeric** | FALSE | FALSE | FALSE | TRUE | FALSE | TRUE | FALSE |
| **Textual** | FALSE | FALSE | FALSE | TRUE \*2 | FALSE | FALSE | FALSE |
| **Blank** | FALSE | FALSE | FALSE | TRUE \*5 | FALSE | TRUE \*3 | FALSE |
| **Negative** | TRUE \*1 | FALSE | FALSE | FALSE | FALSE | FALSE | TRUE \*4 |
| **Valid?** | **Failed** | **Passed** | **Passed** | **Failed** | **Passed** | **Failed** | **Failed** |

Table - Validation Matrix used to check the integrity of the data

After making this matrix, I manually searched out the erroneous values until each cell in the valid row reported that it had passed the validation. My manual method involved sorting suspect columns that reported an error so the error would float to either the top or the bottom of the column.

Each row that contained an erroneous value was removed from the data set as it was not possible to extrapolate any useful information from the row unless the entire row is complete and valid. This method highlighted the following rows:

* \*1 - Row 79: ‘-999’ value for AREA
* \*2 - Row 114: ‘a’ value for PROPWET
* \*3 - Row 538: Missing value for SAAR
* \*4 - Row 548: ‘-999’ value for Index flood
* \*5 - Row 587: Missing value for PROPWET

1. Plot a scatter graph for each column of data.

This step is designed to highlight any outliers that exist in the data. One possible outliers was identified through this method.

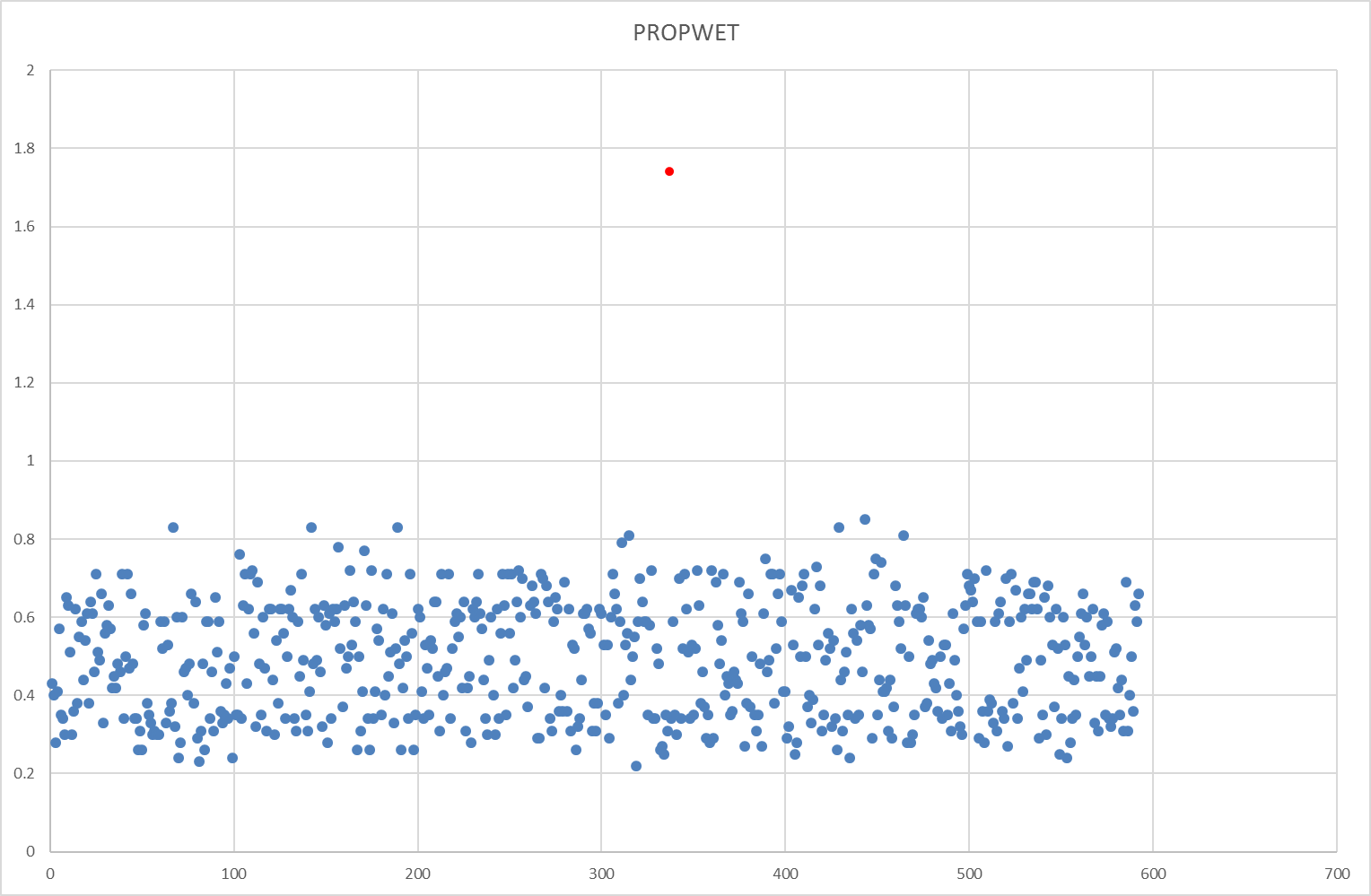


Figure - Graph showing an outlier

The point highlighted in red were identified as a potential outlier in the PROPWET column. Research into the column (Centre for ecology and Hydrology, 2006) revealed that values should fall between 0.2 and 0.8. The rest of the data set seems to fit this range with a few extremes existing, but none out of the range as far as this point. Therefore, I decided that the point was anomalous and removed the row of data from the data set.

### Data Standardisation

As the input parameters represent different physical quantities and the numeric values have different scales of magnitude, I standardised the data before training the neural network.

The data was standardised to between 0.1 and 0.9 using the following formula:

Equation - Standardising the data

Where *S* is the standardised value, *R* is the raw value and Min and Max are the minimum and maximum values over the column of data.

This formula can be re-arranged to de-standardise a value. This was useful as it allowed de-standardising the output allowed me to visualise more clearly what the error function calculated.

Equation - De-standardising the data

Where *S* is the de-standardised value, *R* is the standardised value and Min and Max are the minimum and maximum values over the column of data.

## Training and Network Selection

Two methods of training the network were explored while trying find a network configuration that yielded the lowest error.

### Split Set Training Method (Static)

The Split Set training method is to split the data set into three sets; a training set, a validation set and a test set. The three sets are mutually exclusive in that the rows of data they contain are randomised and do not overlap.

The data has been split into the sets programmatically, with each network trained using a different composition for each set. This randomisation allows for more variation in the networks created, thus creating a larger chance to find a network with an optimum performance.

The training set is 60% of the data set and is fed into the network to train the network; thus, in this regard the training set is said to be “seen” by the network. This set is the largest of the three as it’s important to train using a variety of data so generally the more data that can be used in training the better.

The validation set is a further 20% of the data set and is used to measure the performance of the network; at some interval, I measure the performance of the network against the validation set by running a forward pass on the network using the validation set. This allows me to calculate a predicted output for each row of data, in turn allowing me to evaluate the performance of the network by comparing it with the expected output. By tracking the performance of the network over time, I can implement a halting function. Once the performance of the network reaches an optimum level, that is, it begins to get worse instead of improving, I can stop training the network.

This method of using the validation set to measure performance allows me to prevent the network from over-training on the training set. If this happened, the network would over-fit on the training set and the performance of the network on any valid but “unseen” row of data would not be optimal. Thus, by measuring performance using a validation set that has not been used in training allows me to prevent the network from over-fitting to the training set.

The test set is the remaining 20% of the data set. This is used to evaluate the performance of the final, trained network. As the validation set is “seen” by the network in evaluating the halting function, the test set allows for a completely unseen data set to be used to evaluate the final performance of the trained network.



Figure - Code snippet showing the data being split into three sets



Figure - Code snippet showing the split sets training method

### K-fold Cross Validation Method

The k-fold cross validation method is like the split set method in that it splits the input data set into sets to be used for training and for evaluation the performance of the network. However, it can utilise the entire data set for both training and validation.

It splits the set into k number of evenly distributed sets, or folds. Then, it utilises a loop to train the network, using nine out of ten folds as the training set and the remaining fold as a validation set. The validation set is then shifted to the next fold and the process is repeated until each fold has been used as a validation set.



Figure - Code snippet showing the data being split into k folds

Thus, as each fold is used as the validation set at some point during the training, it is possible to track the performance of the network against each fold, which can then be combined to evaluate the performance of the network against the entire data set.



Figure - Code snippet showing the k-fold cross validation method

### Network Selection

In selecting the best network, the performance evaluator that was utilised in the halting function was the root mean squared error of the validation set used. This gave a reliable performance measure across the entire validation set.

This could be calculated by:

Equation - Equation to calculate the room mean squared error

Where RMSE is the room mean squared error, is the correct output for row i of the data set, is the predicted output for row i of the data set and n is the number of rows in the data set.

The root mean squared error can measure the overall distance between the observed and modelled data sets.

Others errors functions such as the R2 error and mean square relative error were implemented but not used in the halting function. However, they are used in the network selection process, after training is complete.

Equation - Calculation of the mean squared relative error

Where MSRE is the mean squared relative error, is the correct output for row i of the data set, is the predicted output for row i of the data set and n is the number of rows in the data set.

Equation - Calculation of the R squared error

Where is the difference in predicted value and correct value for row i of the data set, is the predicted value for row i of the data set and is the mean predicted value.

The R2 error measures how accurate the predicted data is to the overall shape the observed model would draw on a graph.

These error values can be used to compare different methods of training a network.

## Evaluation of final model

In this section, I will compare the methods discussed in the sections above and deduce the best configuration to find an optimal network.

### Training Method Comparison

Table 3 shows the difference made when using a split set training method versus the k-fold cross validation method when training twelve networks, each with ten hidden nodes.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Split Sets | k-fold X Validation | % change (impr/decl) |
| Average epochs required | 5430 | 41583 | +665% - decline |
| Average RMSE (test set) | 0.049 | 0.025 | -50% - improve |
| Average R2 | 0.756 | 0.941 | +24 - improve |
| Average MSRE (test set) | 0.089 | 0.014 | -84% - improve |

Table - Table showing the difference between training methods

Despite the large increase in epochs required for the network to train to the minimum error, the large decrease in the error functions leads me to conclude that the k-fold cross validation allows for training better networks than the split sets method.

However, despite the split set training method performing worse on average, the method will sometimes produce networks that perform far better than the others in a similar range. I believe this fluctuation in performance is caused by the random nature at which the networks are trained; the initial spread of the data will have a high impact on the quality of the network, thus by randomising this, you are able increase the potential of producing a high performing network.

### Momentum

Table 4 shows the difference made by implementing momentum when training twelve networks using a k-cross validation method, each with ten hidden nodes.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Without Momentum | With Momentum | % change (impr/decl) |
| Average epochs required | 600 | 3290 | +448% - decline |
| Average RMSE (test set) | 0.054 | 0.042 | 22% - improve |
| Average R2 | 0.645 | 0.81 | +25% - improve |
| Average MSRE (test set) | 0.051 | 0.036 | -29% - improve |

Table - Table showing the difference momentum makes in training

By trading off number of epochs required to train the network, the other performance measures are improved with the addition of momentum, so it can be concluded that implementing momentum will improve the quality of networks produced while training. Similar results were found with the k-cross validation training method.

### Bold Driver

Table 5 shows the difference made by implementing bold driver when training twelve networks using a k-fold cross validation training method, each with ten hidden nodes.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Without Bold Driver | With Bold Driver | % change (impr/decl) |
| Average epochs required | 86083 | 41583 | -52% - improve |
| Average RMSE (test set) | 0.026 | 0.024 | -5% - improve |
| Average R2 | 0.933 | 0.941 | +1% - improve |
| Average MSRE (test set) | 0.016 | 0.014 | -16% - improve |

Table - Table showing the difference bold driver makes in training

The largest improvement made was in the number of epochs required which is due to larger weight changes being made if the network is constantly improving. However, the other performance measures also improved, albeit marginally, thus concluding that bold driver will improve the quality of networks produced while training.

### Bold Driver / Annealing Hybrid

Table 6 shows the difference made by implementing annealing as a follow up procedure to a bold driver approach when training twelve hidden networks.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pure Bold Driver | Hybrid Bold Driver/Annealing | % change (impr/decl) |
| Average epochs required | 41583 | 42916 | +3% - decline |
| Average RMSE (test set) | 0.0246 | 0.0247 | +0.7% - decline |
| Average R2 | 0.9409 | 0.9404 | -0.05% - decline |
| Average MSRE (test set) | 0.0136 | 0.0137 | +0.8% - decline |

Table - Table showing the difference the bold driver / annealing hybrid method makes in training

Results show that the combination of bold driver with annealing have a very slight negative effect on training a network.

### Hidden Nodes

Figure 24 shows the effect the number of hidden nodes in a network has on the RMSE of a network. The graph contained data from training twelve networks per hidden node configuration, from 2 to 17 hidden nodes.

Figure - Graph showing how node configuration affect the performance of the network

The data suggests that there exists a minimum number of nodes that would produce an optimal network. Solving the equation for the trend line to find the global minimum would suggest that the optimal node configuration is 14 nodes.

Following a similar method for R2 and MSRE suggests that a minimum configuration could also exist at 13 nodes (MSRE) or 18 nodes (R2). As both MSRE and RMSE directly measure the error in the graph, I can conclude that the best configuration would either be with 13 or 14 nodes.

### Optimal Solution

Combining the factors discussed above, I concluded that the optimal solution could be found using Momentum and Bold Driver, with the k-fold cross validation method, with 13 or 14 hidden nodes. Thus, by training multiple networks with this configuration, I could create the following optimal network:

|  |  |
| --- | --- |
| Hidden Nodes | 14 |
| Epochs trained | 133,000 |
| Validation RMSE (average over 10 folds) | 0.0208 (25.84 de-normalised) |
| Test RMSE (over entire data set) | 0.0201 (25.82) |
| R2 | 0.958 |
| MRSE | 0.011 |

Table - Showing the outcome of the best trained network

Figure - Graph showing the actual value vs predicted value

Table - Graph showing the fit of the predicted outputs against the correct outputs

Comparing the predicted output to the correct output shows the model is fairly accurate. The model tends to fit low (between 0.1 and 0.13) and high values (more than 0.4) in the data very well, however there is some more noise in the output, particularly towards the 0.2-0.3 range.

## Comparison with other data driven models

To further validate the networks trained through the backpropagation algorithm, I have created a simple model in excel which attempts to map the input parameters to the output. I have done this using the LINEST function in excel which uses a least squares method to calculate a model to fit the data. The steps I took to create the model are detailed below:

1. Clean the data set as described in section 1.2.1.
2. Normalise the data to between 0.1 and 0.9.
3. Randomly split the data into two sets, 80% of the data for training and 20% of the data for evaluating the model.
   1. This was repeated 3 times to allow for multiple models to be created and the error averaged.
4. Create the model using the LINEST function on the training set. This returned a gradient value for each input parameter and an intercept value.
5. Apply the model to the training set, calculating the predicted flood index.
6. Calculate the error (RMSE, R2, MSRE) in the model for the training set.
7. Apply the model to the validation set, calculating the predicted flood index.
8. Calculate the error (RMSE, R2, MSRE) in the model for the validation set.

The results of the model can be seen and compared against a network trained with the backpropagation algorithm in Table 9.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | LINEST Model (Training Set) | LINEST Model (Validation Set) | Network trained model (Validation set) | % change (impr/decl) |
| Average RMSE | 0.055 | 0.059 | 0.021 | -64% - improve |
| Average R2 | 0.912 | 0.569 | 0.956 | +68% - improve |
| Average MSRE | 0.059 | 0.060 | 0.012 | -80% - improve |

Table - Table showing the results from the LINEST data model

In all measures used to evaluate error in the model, the network-trained model outperformed the data-driven model by a wide margin, showing that the performance of the network trained model is far better than the LINEST model.

# References

Battiti, R. (1989). *Accelerated Backpropagation Learning: Two Optimization Methods.* California: Caltech Concurrent Computation Program.

Centre for ecology and Hydrology. (2006). *FEH Catchment Descriptors*. Retrieved from National River Flow Archive: http://nrfa.ceh.ac.uk/feh-catchment-descriptors

Duffner, S., & Garcia, C. (2007). *An Online Backpropagation Algorithm with Validation Error-Based Adaptive Learning Rate.* Cesson-Śevigne: Orange Labs.

Wikipedia. (2017, 02). *Coefficient of determination*. Retrieved March 2017, from Wikipedia: https://en.wikipedia.org/wiki/Coefficient\_of\_determination