River Ouse Flow Prediction ANN - Report

I am implementing the ANN using Python as it provides a useful set of libraries to aid in neural network programming. Additionally, Python's dynamic typing can be helpful as I will be dealing with different, similar data structures e.g. NumPy arrays and Python lists.

Data pre-processing

I am choosing to do all data processing programmatically so that I can configure how I clean my data whenever I run the program. The program will store each processed data set as a csv file so I can run models and visualization on them.

The user can create datasets using create_model.py. The top of the file contains parameter variables that can be changed before running the program.

Baseline dataset

For my first dataset, I will choose basic operations as this will be a baseline dataset I can compare with when I try more advanced data processing.

First, I anonymize the data. Then, I discard the following values:

1. All non-numerical data
2. All negative values, as nothing we are measuring should ever be below 0
3. All rainfall values >279mm (UK daily rainfall record according to Met office)

I will then lag all predictor columns by 1 day, as otherwise we are training our model to predict known values.

To begin with I will use a 60%/20%/20% split for training/validation/test data. This will be done at random to ensure no bias in terms of the year or season.

I am standardising my data to the range [0.1, 0.9].

Improved dataset

Cleaning

Building on the baseline data, I next culled the values that were a number of standard deviations from the mean in each column. I experimented with the number of deviations to use as a cutoff.

Cutoff Distance from Column Mean (std devs)	Rows Remaining	Percentage of initial rows
No cutoff	1449	99.2%
7	1442	98.7%
6	1439	98.5%
5	1428	97.7%
4	1407	96.3%
3	1384	94.7%
2	1330	91.0%
1	1228	84.1%

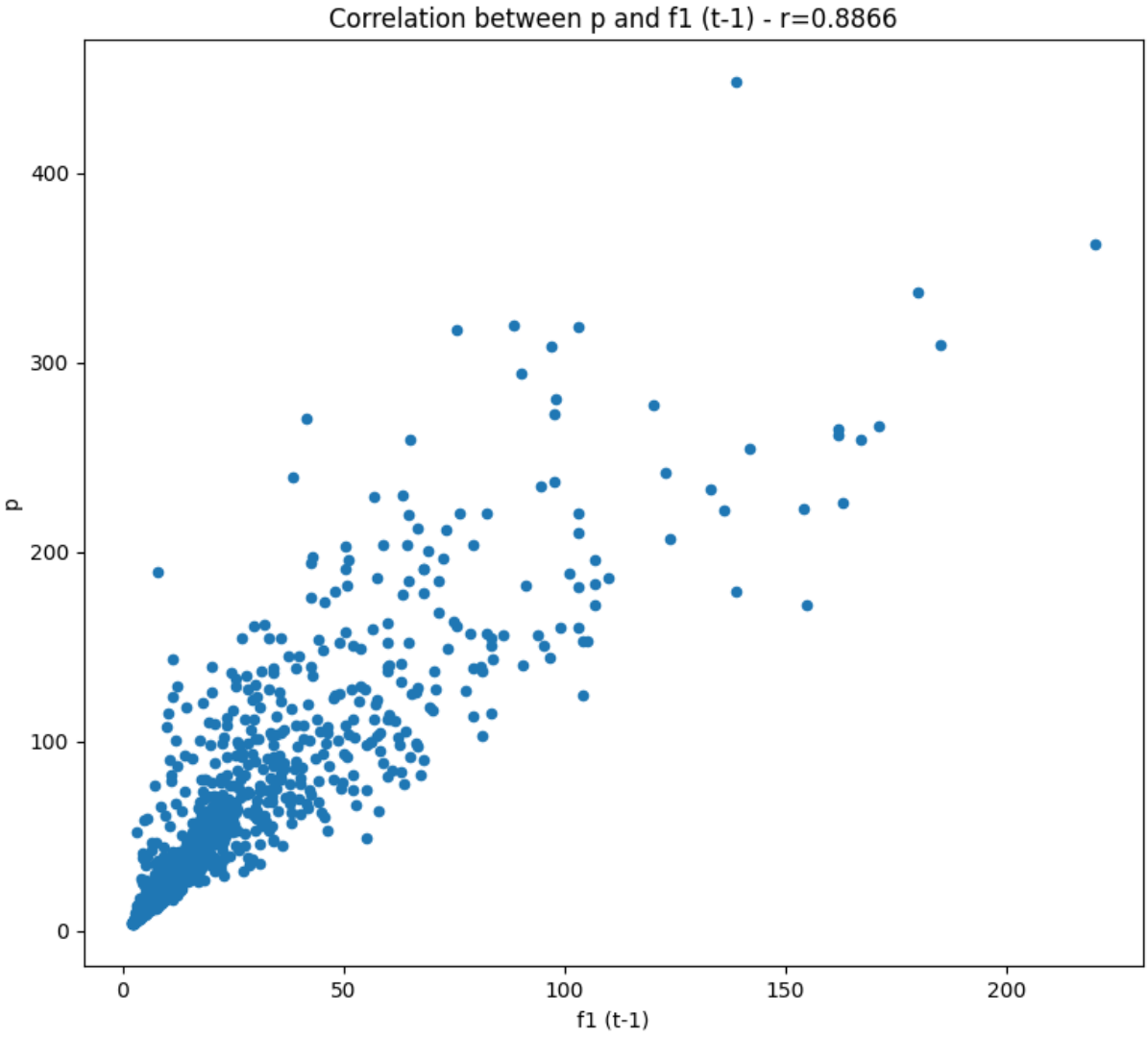
I decided to use 5 standard deviations as this keeps well over 90% of the data.

Exploration

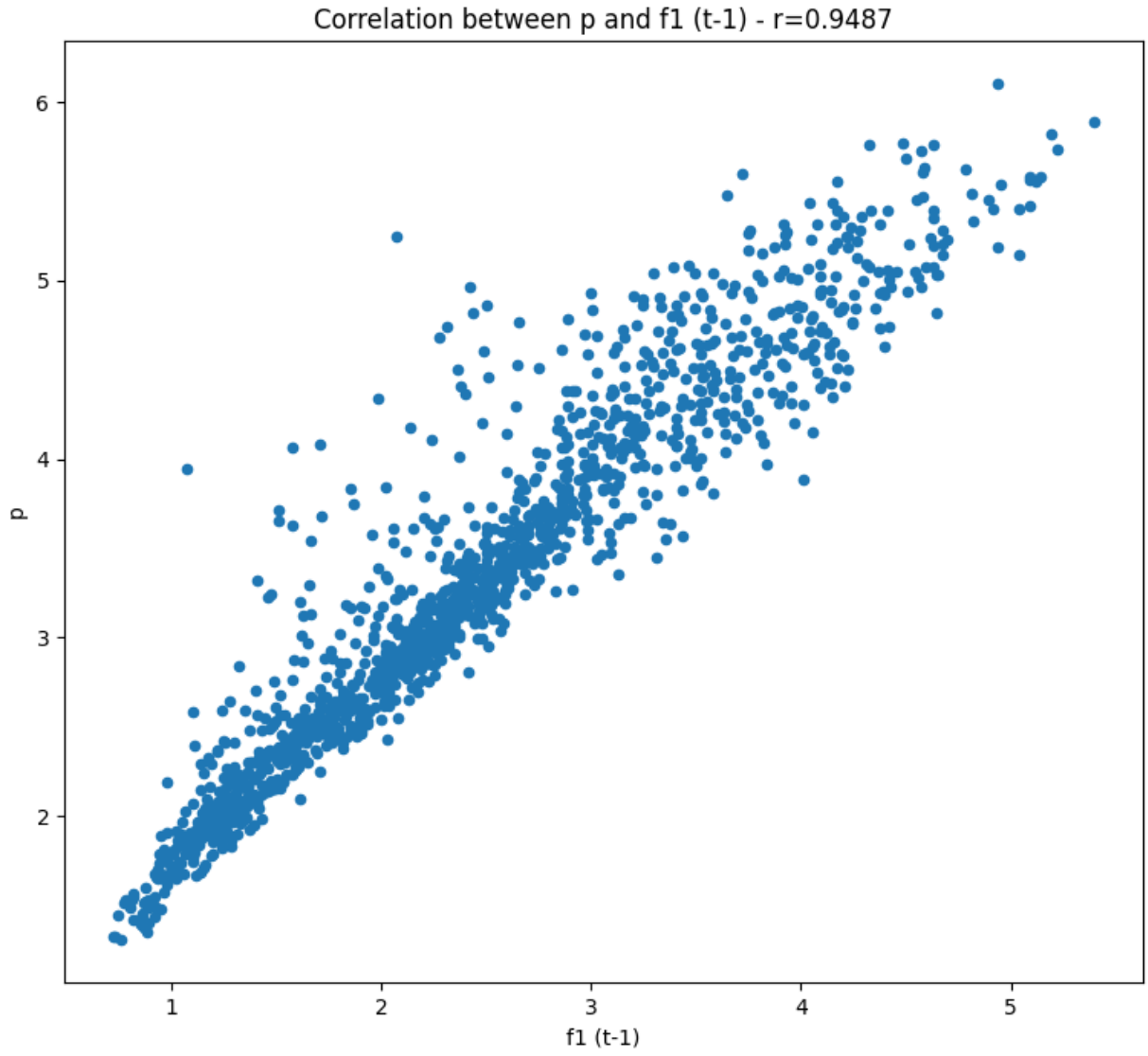
I plotted predictors (x) vs Skelton (y) to identify transformations I could use to improve my dataset based on the calculated Pearson correlation coefficient.

After some experimentation with flow data, the relationship sqrt(log(y)) = sqrt(log(x)) gave a very strong correlation value between Skelton and each of the predictor flow columns.

BEFORE

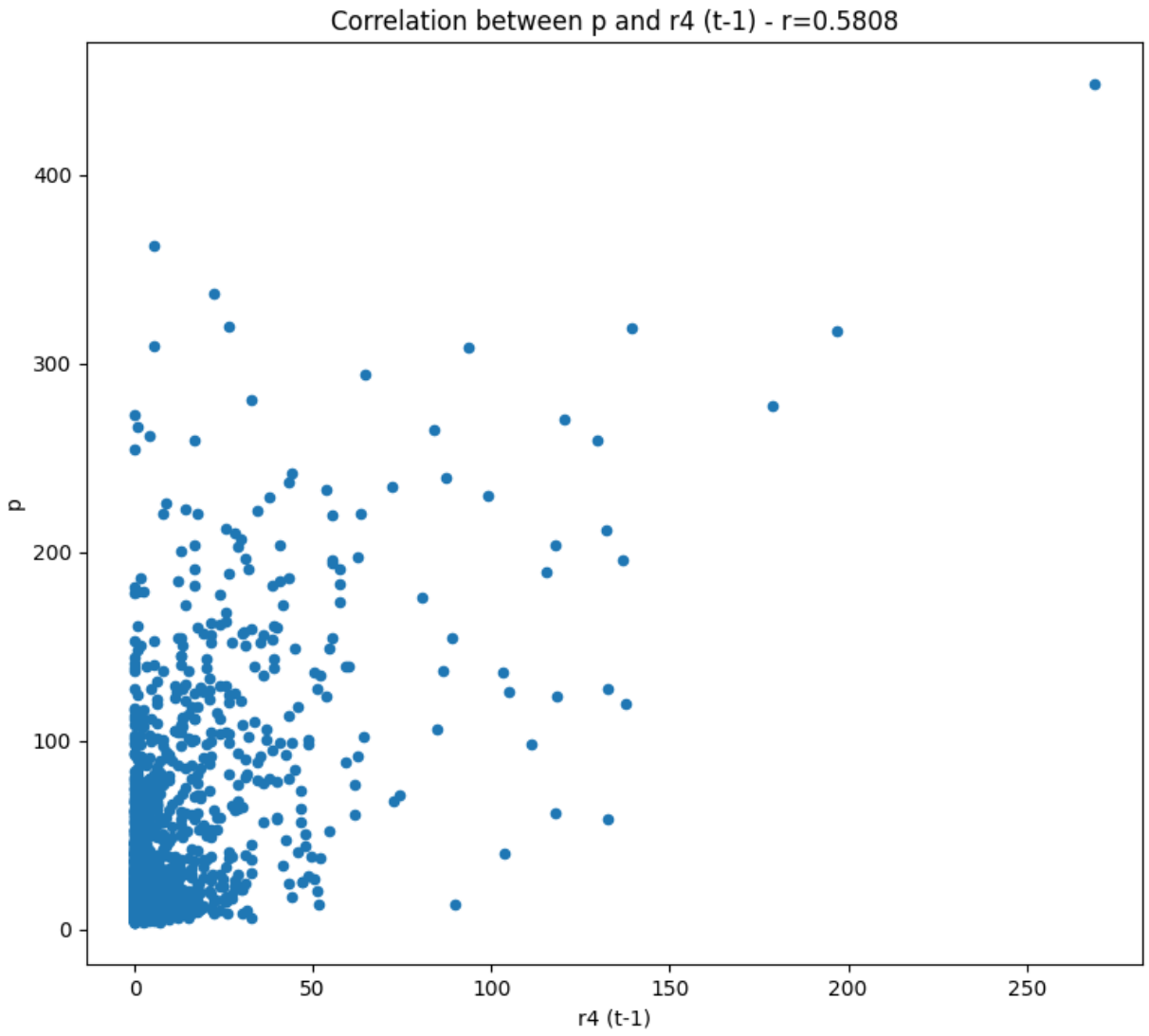

Correlation between p and f1 (t-1) - r=0.8866
p
f1 (t-1)
0
100
200
300
400
0
50
100
150
200

AFTER – log(y), log(x) applied

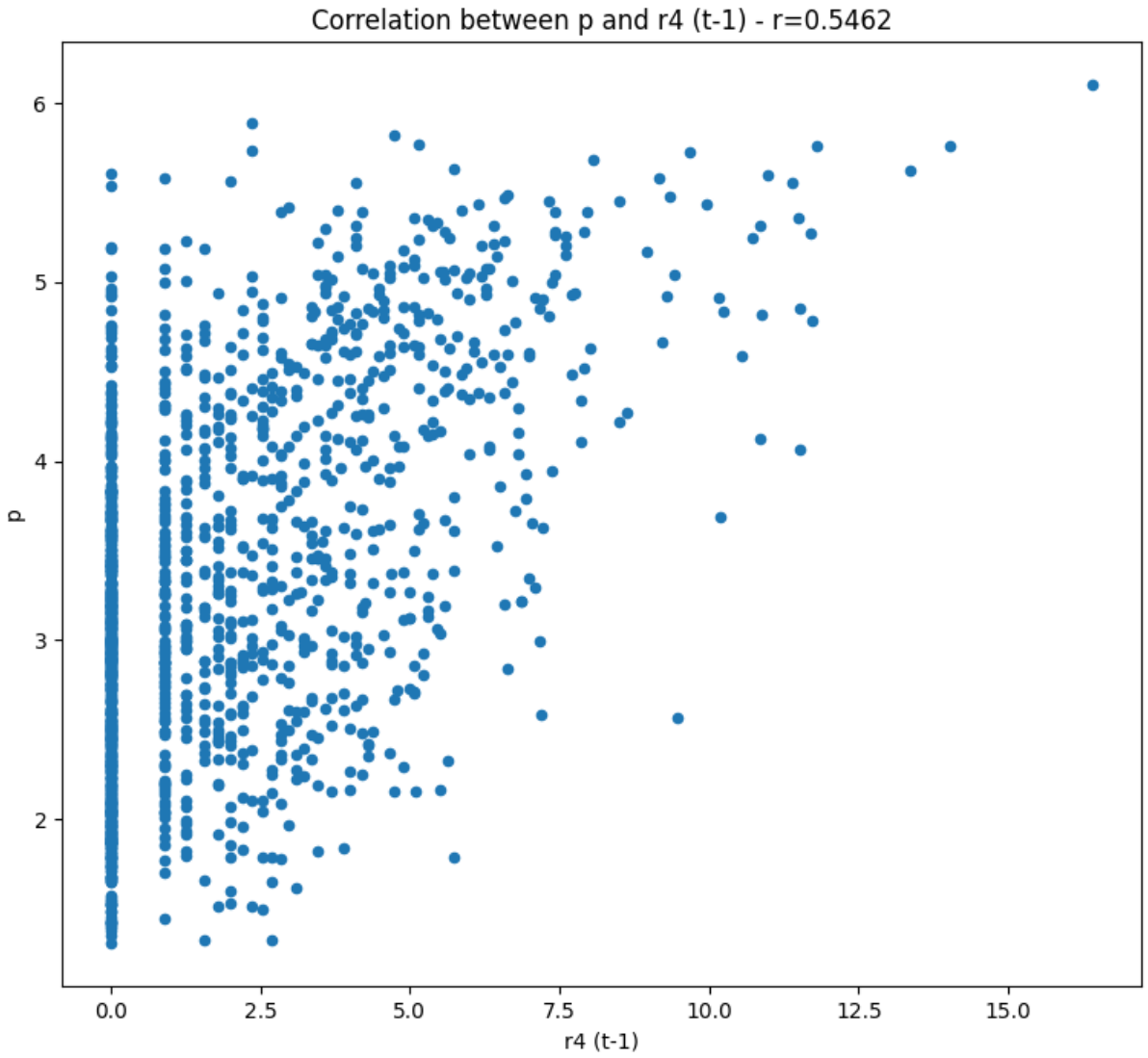


Rainfall data is much less strongly correlated. However, I seemed to get the strongest result from log(y) = sqrt(x).

BEFORE



AFTER – log(y), sqrt(x) applied



Interestingly, there are clear vertical bands of data, indicating that the rainfall values are clustered around intervals.

This means my final operations are: log(x) for all flow data, sqrt(x) for all rainfall data.

My final exploration step before training my first models was to identify cross-correlation between predictors.

source	f1 (t-1)	f2 (t-1)	f3 (t-1)	r1 (t-1)	r2 (t-1)	r3 (t-1)	r4 (t-1)	p
source								
f1 (t-1)	1.000000	0.939842	0.959894	0.330202	0.197409	0.308754	0.402375	0.947749
f2 (t-1)	0.939842	1.000000	0.911627	0.344747	0.222517	0.330219	0.393070	0.924823
f3 (t-1)	0.959894	0.911627	1.000000	0.412077	0.237763	0.411009	0.477570	0.958557
r1 (t-1)	0.330202	0.344747	0.412077	1.000000	0.683938	0.664609	0.669750	0.462087
r2 (t-1)	0.197409	0.222517	0.237763	0.683938	1.000000	0.493932	0.473485	0.306900
r3 (t-1)	0.308754	0.330219	0.411009	0.664609	0.493932	1.000000	0.700876	0.433664
r4 (t-1)	0.402375	0.393070	0.477570	0.669750	0.473485	0.700876	1.000000	0.527122
p	0.947749	0.924823	0.958557	0.462087	0.306900	0.433664	0.527122	1.000000

After applying my log and sqrt functions, we can see that all 3 of the predictor flow values correlate with each other with a coefficient over 0.9. Therefore, I will only use the value with the highest correlation to the predictand (f3).

Splitting

I am not going to alter my proportions of data as I believe that 60/20/20 is good enough to produce a useful network.

Standardisation

I am not changing my standardisation.

Implementation of the MLP algorithm

My MLP algorithm functions including backpropagation are found in training.py.

Network representation

I am representing a neural network as a 3D array, with the index hierarchy layer->node->weight. Each node contains a weight for each input on that node (i.e. a weight for each node on the previous layer) and an extra weight w0 which represents bias.

For example, a neural network with 2 inputs, a hidden layer with 2 nodes, and 1 output node would be stored as:

```
[
  [                          Hidden layer 1
    [w0, w1, w2],                 node 1
    [w0, w1, w2]                  node 2
  ],
  [                          Output layer
    [w0, w1, w2]                  node 1
  ]
]
```

This does not store input nodes directly, as input values are fed through functions on the network.

Main functions

Backpropagation is split into two functions: forward_pass() and backpropagate(). This is done so that I can get the output of a pass over some data using another function predict() without updating weights.

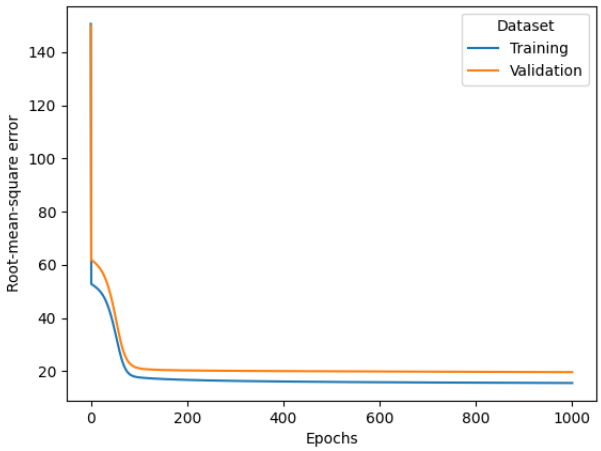
To train models, I have implemented a train() function which can be configured with different hyperparameters and extensions to the backpropagation algorithm. This algorithm saves a table of all predicted and observed values for the training and validation datasets every 100 epochs for use in error calculations and visualisation after training.

Activation functions

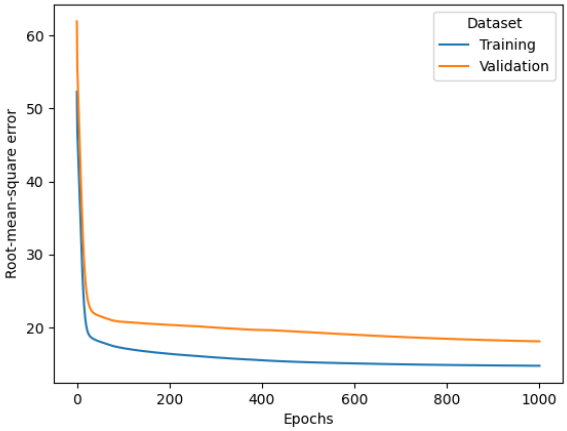
Initially, I used the sigmoid function for all values. However, testing with multiple hidden layers showed that networks would often suffer from vanishing gradient as a result of large weighted sums at nodes. For this reason, I decided to switch to leaky ReLU, which is much

less likely to cause a vanishing gradient. This also caused my networks with one hidden layer to converge more quickly to a lower RMSE.

Sigmoid



Leaky ReLU



Initial weights

Initially, I randomised weights in the range [-2/n, 2/n], where n is the number of inputs. I compared this with Xavier initialization on XOR data to see which would converge the most often.

Networks trained for 10000 epochs (3 hidden layers, 4 nodes per hidden layer)
Epochs to reach 0.0 RMSE

	Epochs to converge by initialization method (nearest 100)		
Training cycle	[-2/n, 2/n]	Uniform Xavier	Normal Xavier
1	6500	2500	200
2	300	600	3300
3	N/A	200	200
4	300	1400	1800
5	300	200	300
Average epochs	1480 (4/5 success)	**980**	1160

I selected uniform Xavier initialization as it provided the fastest convergence on average.

Extensions

I have implemented momentum, weight decay, bold driver and annealing as extensions for my algorithm. I tested different combinations of these on my datasets when training.

Training and network selection

I use a training function to backpropagate over the training dataset and then predict values in the validation dataset, repeating this for the number of epochs.

To begin with, I am training using basic backpropagation with no improvements. I will run this over my baseline dataset to create a model which I will compare other models against. I then implemented momentum, weight decay, bold driver and annealing, and I will then run training over both datasets and compare results.

Fixed network properties:

Activation function – Leaky ReLU

Weight initialization – uniform Xavier initialization

Hyperparameters to train with:

Epochs – 1000

Learning parameter – 0.1

Hidden layers – 1

Nodes per hidden layer – inputs/2 to 2 * inputs inclusive

Out of a group of trained networks, I am selecting the “best” against the validation set as the network with the lowest RMSE on the validation set after training.

Best model for each set of conditions

Selected model information			Error functions on test data			
Dataset	**Extensions**	**Hdn. Nodes**	**RMSE**	**MRSE**	**CE**	**R^2**
Baseline	None	9	19.16	0.084	**0.886**	**0.897**
Baseline	Momentum	9	19.58	0.070	0.882	0.894
Baseline	Momentum, Weight decay, Bold driver	4	21.05	0.228	0.863	0.891
Baseline	Momentum, Weight decay, Annealing	12	21.65	0.712	0.855	**0.897**
Improved	None	8	**17.42**	**0.065**	0.854	0.863
Improved	Momentum	9	18.22	0.069	0.840	0.851
Improved	Momentum, Weight decay	9	17.55	0.071	0.851	0.852
Improved	Momentum, Weight decay, Bold driver	2	17.47	0.077	0.853	0.854
Improved	Momentum, Weight decay, Annealing	6	19.51	0.118	0.816	0.844

Comparing datasets

The results show that the improved dataset resulted in lower RMSE and MRSE values on average. This implies that models trained on the improved dataset predict values closer to the observed values on average.

However, this also resulted in lower CE and R^2 values. This implies the overall shape of the graph of predicted values matches less closely with the graph of observed values than for models trained on the baseline dataset, and the models trained on the improved data perform slightly worse compared to a no-knowledge prediction.

I theorize that the improved models perform slightly worse as applying log and sqrt functions to the data will make any linear error in the standardized values exponentially larger on the real values, meaning small discrepancies in the shape of the graph will become larger.

Comparing backpropagation extensions

None of the extensions to the algorithm produced improved results on the error functions. I believe this is because I did not consider multiple hidden layers, and I believe using batch processing would allow me to better take advantage of these features when training as I could more easily train for more epochs.

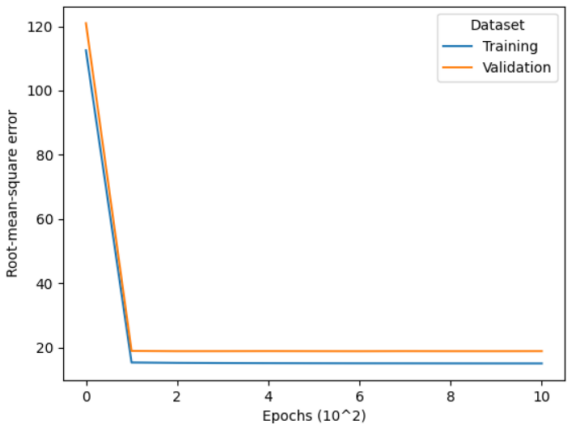
However, the use of bold driver showed a strong reduction in the complexity of models with only a small reduction in performance – this was especially true on the improved dataset, which resulted in a network with only 2 hidden nodes and low RMSE and MRSE values at the cost of only slightly worse CE and R^2 values compared to the baseline model.

Evaluation of final model

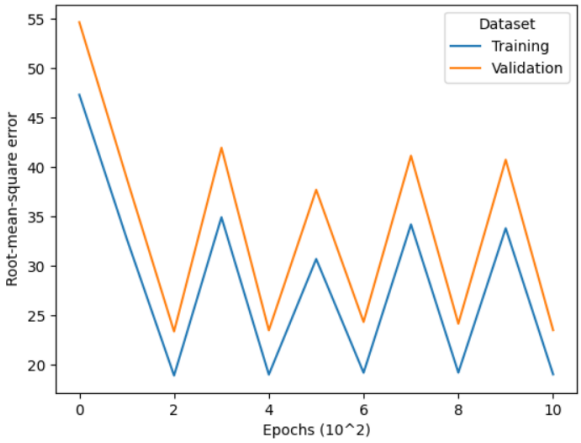
I selected the model trained with bold driver on the improved dataset as it provided strong results with just 2 hidden nodes.

RMSE against epochs during training

Baseline model



Selected model

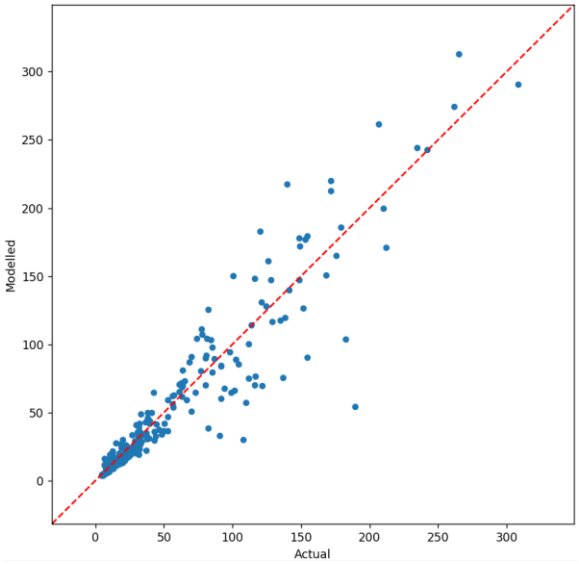


The RMSE graph for the final model is very extreme as I am applying bold driver every 200 epochs, which is very often, and I am only training for a relatively short period of 1000 epochs. The adjusted learning parameter consistently increases the error for 100 epochs before the network adjusts to the new learning rate.

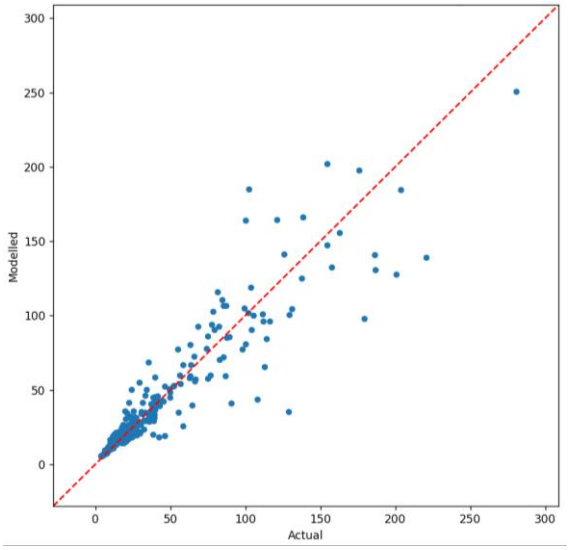
Interestingly, the final model has a higher RMSE for both the training and validation data at the end of training, as well as a higher discrepancy between the two. This would imply that the model is not predicting as accurately, but its comparable performance on the test data may imply that this simply means the model is not overfitting the training data.

Actual vs Modelled Values with error

Baseline model



Selected model



Baseline - The cluster of values on the lower end of the scale are somewhat spread around the line, implying that the baseline model predicts these values with a consistent small error. We can see that the baseline model tends to have much greater error for predictand values above around 50, which can be explained by the majority of points lying below this value. The model also seems to underpredict to a greater extreme than when it overpredicts.

Selected - The predicted values cluster more tightly at lower values, forming a cone shape along the ideal line – this shows that the final model is more accurate at lower predicted values, especially compared to the baseline model. We can see that the selected model's errors at high values are somewhat more spread out, however they appear more consistent than the baseline model. Additionally, there are fewer extreme values due to previous culling.

Conclusion

Overall, the final model provides improvement over the baseline model in its reduced complexity (9 hidden nodes vs 2) with only a small decrease in performance. I believe that I could further improve this model by simply training for more epochs.

There are further improvements to my program and models that could have potentially improved performance:

- Implementation using an object-oriented approach for improved readability

This would have saved me time when extending the algorithm and resulted in a better-performing training program.

- Training with multiple hidden layers

I believe training with more layers would allow my networks to better leverage the extensions to the backpropagation algorithm.

- Implement batch processing to better allow training complex networks over many epochs

This would have allowed me to more quickly test different combinations of dataset and algorithm extensions for very complex networks, which could lead to an improved final model.