Backpropagation Neural Network with Genetic Evolutionary Algorithm for Optimisation

Advanced AI Systems - Neural Network Coursework
Tristan Murfitt - B418521
Due - 15 March 2018

A scalable implementation of a **multi-layer perceptron** which can be trained with any numerical data to model a single predictand. This report details the prediction of Pan Evaporation from Fresno, CA using data from a four-year period. The most accurate model was selected by using a **Genetic Evolutionary Algorithm** to search for the optimal hyperparameters of the neural network.
The final model achieved a RMSE of 0.158563 (0.009849 normalised RMSE) on test data.

1. Data Pre-processing

Data analysis was performed in Excel to clean the data and evaluate the suitable predictors for Pan Evaporation.

1.1. Data Cleansing

The conditional formatting tool was used to highlight any cells with non-numerical or null data. Additionally, the columns for each predictor were sorted to reveal the highest and lowest values and reveal any outliers.

Outliers were spotted easily, for example: any temperature reading of 83 Celsius and above was clearly erroneous. For any occurrence of non-clean data, the entire data row was removed. This is done as the data points from all the predictors are required to model the single predictand and the abundance of clean data available meant it was not necessary to interpolate any erroneous values. A total of 18 rows were removed to clean the data.

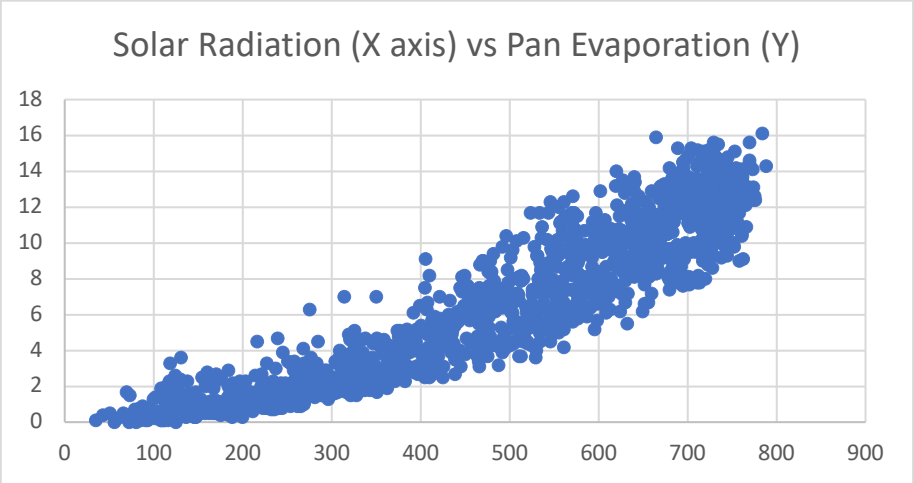
1.2. Selecting Predictors

The first thought of data pre-processing was whether all the predictors were necessary and helpful in predicting the Pan Evaporation. The **correlation** function was used across the data set to evaluate how effective a feature of the neural network should be at modelling the pan evaporation.

Data correlation:

	T	W	SR	DSP	DRH	**PanE**
T	1					
W	0.418529	1				
SR	0.817761	0.543281	1			
DSP	-0.71787	-0.47676	-0.58508	1		
DRH	-0.79887	-0.343	-0.81431	0.560579	1	
PanE	0.913394	0.59376	0.930552	-0.67112	-0.8387	1

This table suggests that all predictors are suitable for predicting the pan evaporation as they have **strong negative or positive correlation**. For example, the solar radiation has the strongest positive correlation with pan evaporation—at 0.93—indicating that as solar radiation increases, pan evaporation increases at a similar rate. This has been demonstrated visually by plotting the data on a scatter graph below.



A strong negative correlation was with percentage humidity, which at -0.84 indicates that as humidity increases, pan evaporation decreases. Analysis of correlation has therefore confirmed that all five of the predictors are suitable for use in the predictive model.

1.3. Data Splitting

The data is **temporal** in nature and is closely linked to the time of year, with winter seasons resulting in a low temperature and summer seasons recording high temperatures and solar radiation. It was therefore important to ensure that data used for training was not entirely from one season, and data for testing to be a different season, as this could lead to inaccurate predictions. Despite this, it was not necessary to evenly disperse the data records by season into the three data sets because analysis revealed that the linear correlation of the predictors to the pan evaporation was strong. As such, it was opted that the data would be randomly distributed before splitting into data sets.

The neural network implementation allows configuration of the data split into respective sizes for training, validating and testing. As a default approach, the 60%/20%/20% was selected due to the importance of plentiful data for training the model and an equal share for validation to prevent overfitting and testing to provide performance metrics on unseen data.

2. Implementation

The neural network was written in **Python** 2.7 with cross compatibility to 3.X to ensure it can be run across the majority of systems. The libraries pandas and numpy were used to streamline the data manipulation process in the backpropagation algorithm, for example by rapidly reading in the cleaned CSV data and using mathematical operations to **normalise** the

data in the program. Additionally, the matplotlib library was used to display results graphically and assist with analysis of network performance.

2.1. Backpropagation Algorithm

A neural network layer is implemented as a Python class and can be fully configured by modifying settings in the configuration file and by changing parameters on initialisation of a layer. When creating a NN layer object, the software creates the desired number of neurons and takes parameters to decide the activation function used. Available **activation functions are sigmoid, tanh, and linear**. The neurons are randomly assigned starting weights for each input, between the values of -2/n and 2/n (where n is the number of neurons in the layer), and an additional bias for each neuron.

```
# NN Model - Neuron Layer
class Layer:
    def __init__(self, num_inputs, num_neurons, activation_method):
        self.inputs = num_inputs
        self.neurons = num_neurons
        self.activation_method = activation_method

        # Stores last adjustments for momentum
        self.last_change_w = np.zeros((self.inputs, self.neurons))
        self.last_change_b = np.zeros((self.neurons))

        # randomly assign neuron starting weights and bias (-2/inputs -> 2/inputs)
        self.weights = np.random.normal(0, float(2)/self.inputs, (self.inputs, self.neurons))
        self.bias = np.random.normal(0, float(2)/self.inputs, (self.neurons))

    def activation(self, S, derivative=False):
        # Sigmoid
        if self.activation_method == 'Sigmoid':
            if derivative:
                return S * (1 - S)
            else:
                return 1 / (1 + np.exp(-S))
```

One epoch is considered as a backpropagation loop for every data row within the training data. The training process continues for the specified number of maximum epochs in the config file, or until **overfitting** is detected by a consecutive increase in the error function against the validation set.

The backpropagation algorithm consists of a feed forward phase to evaluate the model's prediction of the training X data (values of predictors), followed by a backpropagation phase to adjust the neuron weights depending on the error compared to the Y data (expected value of predictand). This online method of training searches the weight space by **gradient descent** to find the set of weights which yields the lowest error. This means that each row of training data, in every epoch, updates the weights of the neurons by directing the weight changes down the steepest gradient in an attempt to find the global minima of the error function. Each neuron's weights and bias are updated by multiplying the delta (calculated from the error of the feed forward phase) with the configurable learning rate and the neuron inputs.

```python
    def update_weights(self, delta, inputs):
        # Reshape inputs to form: u(i), transposed for input to each neuron
        inputs = inputs.reshape(len(inputs),1)

        # Update weights by w(i,j) = w(i,j) + LR * delta(j) * u(i)
        w_change = const.LEARNING_RATE * delta
        w_change = w_change * inputs
        self.weights += w_change

        # Update bias
        b_change = const.LEARNING_RATE * delta
        self.bias += b_change

        # Momentum
        if const.MOMENTUM:
            self.weights += self.last_change_w * const.MOMENTUM_ALPHA
            self.bias += self.last_change_b * const.MOMENTUM_ALPHA
            # Record changes for next iteration
            self.last_change_w = w_change
            self.last_change_b = b_change
```

2.2. Algorithm Improvements

Momentum was added into the backpropagation algorithm to accelerate the learning process by multiplying the last iteration's weight changes with a momentum term and adding it into the current iteration's changes. This speeds weight changes up by continually increasing weight changes in a certain favourable direction. This had a direct effect on the number of epochs required to reach a minima of the error function and sped up the model training, with the final model demonstrating that 2000 epochs results in the same error with momentum as 10000 does without. Additionally, momentum prevented the algorithm from reaching a **local minima** in the majority of cases and therefore trained the network more effectively and improved results. This can be attributed to the larger weight changes preventing traps in small gradient minima and surpassing these points.

```
# Momentum = True
Epoch 00000: 0.011720 MSE (0.108260 RMSE) on validation set
Epoch 01000: 0.000125 MSE (0.011186 RMSE) on validation set
Epoch 02000: 0.000105 MSE (0.010266 RMSE) on validation set
...
Epoch 10000: 0.000092 MSE (0.009601 RMSE) on validation set
...
Epoch 13500: 0.000092 MSE (0.009595 RMSE) on validation set
Epoch 13500: 0.000131 MSE (0.011451 RMSE) on test set
Epoch 13500: 0.033991 MSE (0.184367 RMSE) on denormalised test set

# Momentum = False
Epoch 00000: 0.005601 MSE (0.074840 RMSE) on validation set
Epoch 01000: 0.000152 MSE (0.012317 RMSE) on validation set
Epoch 02000: 0.000135 MSE (0.011624 RMSE) on validation set
...
Epoch 10000: 0.000105 MSE (0.010241 RMSE) on validation set # Momentum's error at 2000 epochs
...
Epoch 14999: 0.000102 MSE (0.010096 RMSE) on validation set
Epoch 14999: 0.000138 MSE (0.011740 RMSE) on test set
Epoch 14999: 0.035726 MSE (0.189014 RMSE) on denormalised test set
```

The **bold driver** improvement was also implemented into the algorithm to assess the effects on performance. This method modifies the learning rate, and therefore the scale of weight adjustment of the network, which affects performance as a too small learning rate leads to slow convergence and a too high learning rate to divergence (and overfitting).

Implementing the bold driver every weight update (every iteration of every epoch) lead to some problems: the weight changes are very minor and the error rate does not change enough to have an effect on the error function – therefore the learning rate can be reduced to zero too quickly. This also has a huge effect on performance because the error function on a test set would require running the entirety of the large data set through the network on every iteration of every epoch. Another implementation problem of bold driver is that running the bold driver improvement every epoch instead of every weight update would lead to a conflict with the overfitting check on the validation set. Both would seek an increase in the error function, with the bold driver looking to modify the learning rate but the overfitting check looking to stop the network training.

To solve these, the bold driver was implemented every epoch, with the weight adjustment from all iterations of the training set being assessed on the performance metric. If the validation error increased, a sign of the learning rate being too large and not precise enough, the bold driver would throw away the recent epoch and reduce the learning rate of the algorithm. If the error decreased, the new weights were saved but the learning rate was increased a minor amount to **increase convergence** for the next epoch. After performing the bold driver, the validation set is checked at an interval for model performance against the previous check to ensure that overfitting is not occurring. Bold driver did not have a significant effect on the results because it tended to oscillate the learning parameter frequently and just result in the algorithm taking much longer to train fully. For this reason, it was used as a one-time comparison with the standard backpropagation algorithm and was not used to evaluate the neural models.

2.3. Designing the Genetic Evolutionary Algorithm

Network selection and evaluation, as discussed later in this report, are essential in order to find the optimal machine learning model which produces the most accurate predictions. Commonly, experts employ trial-and-error approaches with experienced insight to **modify the hyperparameters** of a neural network to reduce the error. Instead, this report details a more novel approach to exploring the weight space – a **computationally demanding** problem due to the compute power required for evaluating each model and the sheer number of parameter combinations in the search space.

A **genetic algorithm** reflects the biological process of natural selection where the fittest individuals of a population, the models with the lowest prediction error in this case, are selected for reproduction in order to produce offspring for the next evolutionary generation. A starting population, with each individual having randomised genes, allows the search space to be explored by favouring those individuals with the best chromosomes (set of genes) and by introducing mutations and crossovers to further explore options in the parameter space.

It was decided that the main hyperparameters affecting result outcome was the combination of **number of neurons in the hidden layer with the model's learning rate**. This could be improved, given enough time and compute power, to include the number of training epochs and the momentum alpha.

Each individual was assigned a random **chromosome with 12 genes**. This allowed the first 8 genes (bits) to represent the number of neurons, and the final 4 genes to represent the learning rate of the model.

Chromosome:

1	0	1	0	0	0	0	0	1	1	0	1

Number of hidden neurons Learning rate

These numbers were chosen specifically to satisfy the size of the expected search space, allowing an individual model to have up to 256 hidden neurons, and a learning rate with 16 possible options. A formula was applied to each chromosome to retrieve the individual's neurons and learning rate.

Hidden neurons = decimal value + 1 (to prevent a case with 0 hidden neurons)
Learning rate = 1 / (decimal value + 2) (returning a rate between 0.059 and 0.5)

The genetic algorithm then evaluates each model in the starting population and performs several basic operations at each generation: **selection** of the best network configurations to retain, a configurable probability for chromosome **crossover** to create new solutions from existing ones, and a probability of gene **mutation** in a chromosome to add diversity into the solution pool.

The process is repeated for a given number of generations, each one evaluating individual model performance by training it on the backpropagation algorithm. At the end of **evolution**, the final population reveals which individuals remain (and are likely to be the fittest) along with a list of the best individuals recorded from all generations.

Using the DEAP library, the genetic algorithm was implemented with a population size of 28 for a total of 20 generations. This is fully configurable but was tested to be an effective way to cover a large amount of the parameter space without taking too long to compute.

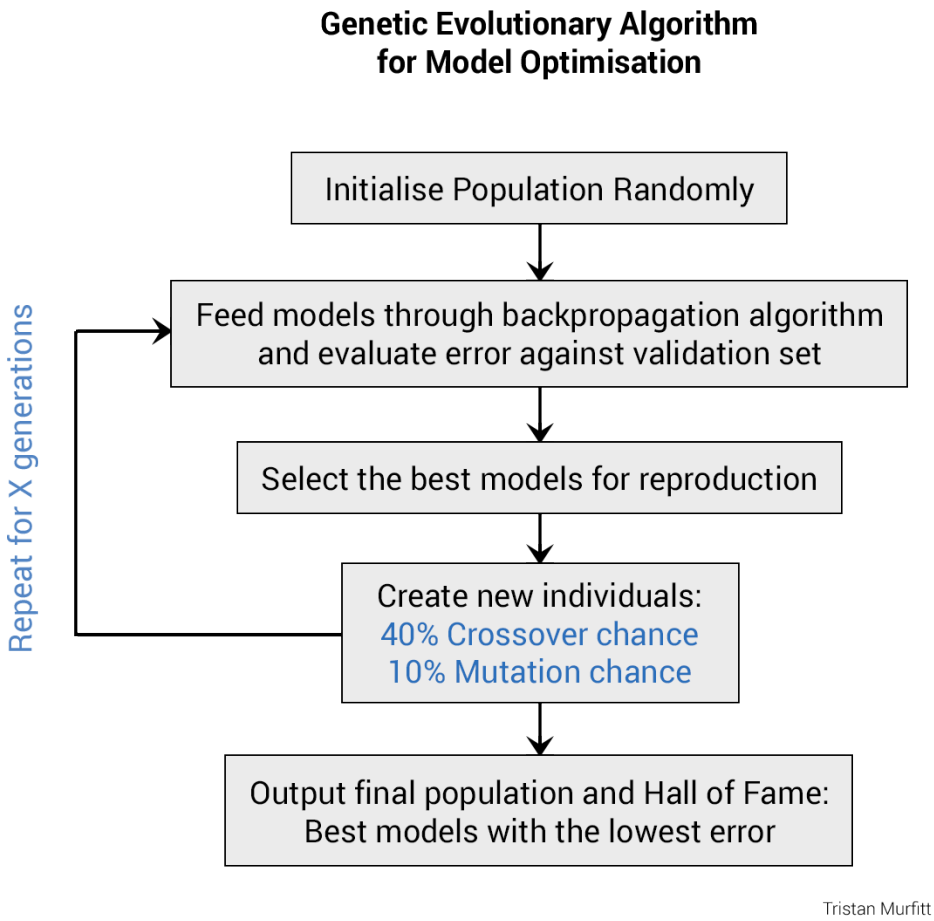
```
# declare individual fitness
creator.create('MaxFitness', base.Fitness, weights = (-1.0,))   # error minimisation problem
creator.create('Individual', list, fitness = creator.MaxFitness)

toolbox = base.Toolbox()
# create chromosomes for individuals in the population
toolbox.register('binary', bernoulli.rvs, 0.5)
toolbox.register('individual', tools.initRepeat, creator.Individual, toolbox.binary,
    n = const.GA_GENE_LEN)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)

# register genetic operations
toolbox.register('mate', tools.cxOrdered)
toolbox.register('mutate', tools.mutShuffleIndexes, indpb = 0.6)
toolbox.register('select',  tools.selTournament, tournsize=4)
toolbox.register('evaluate', train_nn)

pop = toolbox.population(n = const.GA_POP_SIZE)
hof = tools.HallOfFame(const.GA_BEST_INDIVIDUALS)

# run evolutionary algorithm
r = algorithms.eaSimple(pop, toolbox, cxpb = const.GA_CROSSOVER_PB,
    mutpb = const.GA_MUTATION_PB,
    ngen = const.GA_GENERATIONS, halloffame = hof, verbose = True)
```



2.4. Scalability

The code has been built with scalability in mind, allowing multiple data sets to be trained as efficiently as possible. One method of achieving this was by writing custom data normalisation and denormalisation functions into the software to prevent manually performing this in excel on any new data set.

The network is fully customisable, with a separate configuration file allowing the user to specify hyperparameters of the network and data processing options – like the number of hidden neurons, learning rate, and algorithm improvements like momentum for the neural network, and the training, validation and testing data split. A model can also be saved after training and loaded into the system for predictive purposes.

3. Training and Network Selection

3.1. Error Function

To calculate model performance on the validation data set during training, the difference between the predicted and expected values was squared and averaged out, resulting in the **Mean Squared Error** and **Root Mean Squared Error** statistical metric. This was chosen as the squared value is always positive, avoiding errors of predictions too high and low

cancelling each other out, and also because it emphasises error values further from the target value where the estimation is poor. Square rooting the MSE scales the value back to the original units, which is useful when comparing the error of models on the denormalised data.

3.2. Top Networks from Genetic Algorithm

The **genetic algorithm** is used to explore the parameter space to find the most optimal neural network model. This makes the process of network selection far easier than a trial and error approach as it automatically explores hyperparameter combinations and outputs the most efficient model after multiple generations of testing. This approach increases the likelihood of finding a **close-to-optimal network architecture** and allows further detailed network selection on the fittest individuals in the population.
A large population for each generation was chosen to ensure that a wide range of parameter values would be tested in the entire population. The genetic algorithm was configured to evaluate each model for 500 epochs to find a compromise between compute time and completeness of evaluation. This was found to be a level which many models began to converge at, but given more time and compute power, it would have been more representative to test each individual with at least 5000 epochs (or until overfitting was detected). This is primarily because networks with many hidden neurons took longer to converge on a minimum error value.

To get around this limitation, the top six individuals from the genetic algorithm would be evaluated independently on 5000 epochs to determine which is the optimal solution to the problem.

Genetic Algorithm Results (sorted):

```
Final population:
['10100000|1101'x22,'10100000|1110','10101000|1001','00000111|0011','01010000|1101','01010001|0011','00011111|0000']
['161 | 0.06667'x22,'161 | 0.06250','169 | 0.09091','008 | 0.20000','081 | 0.06667','082 | 0.20000','032 | 0.50000']
['RMSE 0.011716'x22,'RMSE 0.011850','RMSE 0.013177','RMSE 0.013458','RMSE 0.017565','RMSE 0.017840','RMSE 0.046888']

Hall of Fame:
['10100000|1101','10111001|1101','10100000|1110','00111001|0100','00000111|1001','00000111|1000']
['161 | 0.06667','186 | 0.06667','161 | 0.06250','058 | 0.16667','008 | 0.09091','008 | 0.10000']
['RMSE 0.011716','RMSE 0.011825','RMSE 0.011850','RMSE 0.012010','RMSE 0.012183','RMSE 0.012190']
```

This reveals that after 20 generations:

- The majority of the final population is a model with hyperparameters of 161 hidden neurons and 0.06667 for the learning rate, achieving a score of 0.011716 normalised RMSE.
- One other model in the final population scored to be in the top six performing models throughout all generations.
- The remaining population individuals have a below satisfactory error scores, which could be due to mutations or undesired crossover traits from reproduction, and will likely not survive for future generations.
- The hall of fame recorded other models which had very low RMSE scores on the validation data set. These are from other generations and did not survive due to

dwindling population or mutations. This could have been prevented with a larger population size per generation but would significantly increase computation time.

3.3. Finding the Best Network

Next, the six best models from the genetic algorithm are independently evaluated on 5000 epochs to determine which model has the best results according to the error function.

Name	Epochs	RMSE (val)	RMSE (test)	Neurons	LR	Rank (val)
R01	5000	0.01029	0.01199	161	0.06667	5
R02	5000	0.010498	0.012230	186	0.06667	6
R03	5000	0.010205	0.011952	161	0.06250	4
R04	5000	0.010037	0.011983	058	0.16667	3
R05	5000	0.009718	0.011771	008	0.09091	2
R06	5000	0.009716	0.011699	008	0.1	1

Benchmarking with randomly selected first generation models:

Name	Epochs	RMSE (val)	RMSE (test)	Neurons	LR
Ctrl1	500	0.01882	0.02014	032	0.076923
Ctrl2	500	0.29494	0.32685	141	0.333333
Ctrl3	5000	0.013519	0.015093	005	0.09091

It is clear that the sixth best model from the GA process had the best choice of hyperparameters for the neural network. Thorough testing on each individual model revealed that a simpler network topology, one with fewer hidden neurons, performed better on the validation and test data sets.

4. Evaluation of Final Model

The successful model hyperparameters are 8 hidden neurons with a learning rate of 0.1. The final model could then be tweaked further in an attempt at improving the model's predictive performance.

One simple modification to improve results on the Fresno pan evaporation data set is by changing the **random seed** value of all randomised computations within the software. This seed affects the initiated starting weights of the model and affects the order and distribution of data rows into their respective sets for training, testing and validation. Changing the seed value has a hugely significant effect on the error statistic for a model. A thorough evaluation should test as many seeds as possible to find the most optimal performance. Due to limited compute power, a random selection of 50 seeds were evaluated on the model with a maximum of 20,000 epochs. It is now important to find the minimal error on the test set, as this will be submitted as the prediction of pan evaporation.

Seed modification results:

The best result was attained by a seed of 10. This model converged to a minimum error very quickly and began to detect overfitting at only 4500 epochs. The model achieved a score of 0.010633 RMSE on the validation set and 0.010040 RMSE on the test data set.

A random seed of 50 detected overfitting on the 13,000th epoch and had an impressive RMSE of 0.009595 on the validation set. However, it only achieved 0.011453 RMSE on the test set. In comparison, a seed of 60 performed the worst, only scoring 0.017496 RMSE on the testing data set after detecting the start of overfitting at epoch 4750.

Algorithm performance results:

After selecting the best random seed, the model was re-trained with the **momentum and bold driver improvements** added as a comparison to the default algorithm. In addition to this, the **momentum alpha** value was tweaked to assess the performance on the outputted results.

Standard algorithm	Epoch 07000: 0.000117 MSE (0.010837 RMSE) on validation set Epoch 07000: 0.000101 MSE (0.010026 RMSE) on test set Epoch 07000: 0.026058 MSE (0.161425 RMSE) on denormalised test set
Momentum Alpha = 0.9	Epoch 04500: 0.000113 MSE (0.010633 RMSE) on validation set Epoch 04500: 0.000101 MSE (0.010040 RMSE) on test set Epoch 04500: 0.026128 MSE (0.161642 RMSE) on denormalised test set
Bold driver	Epoch 90250: 0.000141 MSE (0.011856 RMSE) on validation set Epoch 90250: 0.000136 MSE (0.011643 RMSE) on test set Epoch 90250: 0.035138 MSE (0.187452 RMSE) on denormalised test set
Momentum & Bold driver	Epoch 36750: 0.000176 MSE (0.013256 RMSE) on validation set Epoch 36750: 0.000180 MSE (0.013416 RMSE) on test set Epoch 36750: 0.046658 MSE (0.216004 RMSE) on denormalised test set
Momentum Alpha = 0.7	Epoch 04500: 0.000114 MSE (0.010662 RMSE) on validation set Epoch 04500: 0.000102 MSE (0.010081 RMSE) on test set Epoch 04500: 0.026341 MSE (0.162298 RMSE) on denormalised test set
Momentum Alpha = 1.1	Epoch 50500: 0.000106 MSE (0.010292 RMSE) on validation set Epoch 50500: 0.000097 MSE (0.009849 RMSE) on test set Epoch 50500: 0.025142 MSE (0.158563 RMSE) on denormalised test set
Momentum Alpha = 1.3	Epoch 66250: 0.000105 MSE (0.010268 RMSE) on validation set Epoch 66250: 0.000097 MSE (0.009849 RMSE) on test set Epoch 66250: 0.025145 MSE (0.158573 RMSE) on denormalised test set

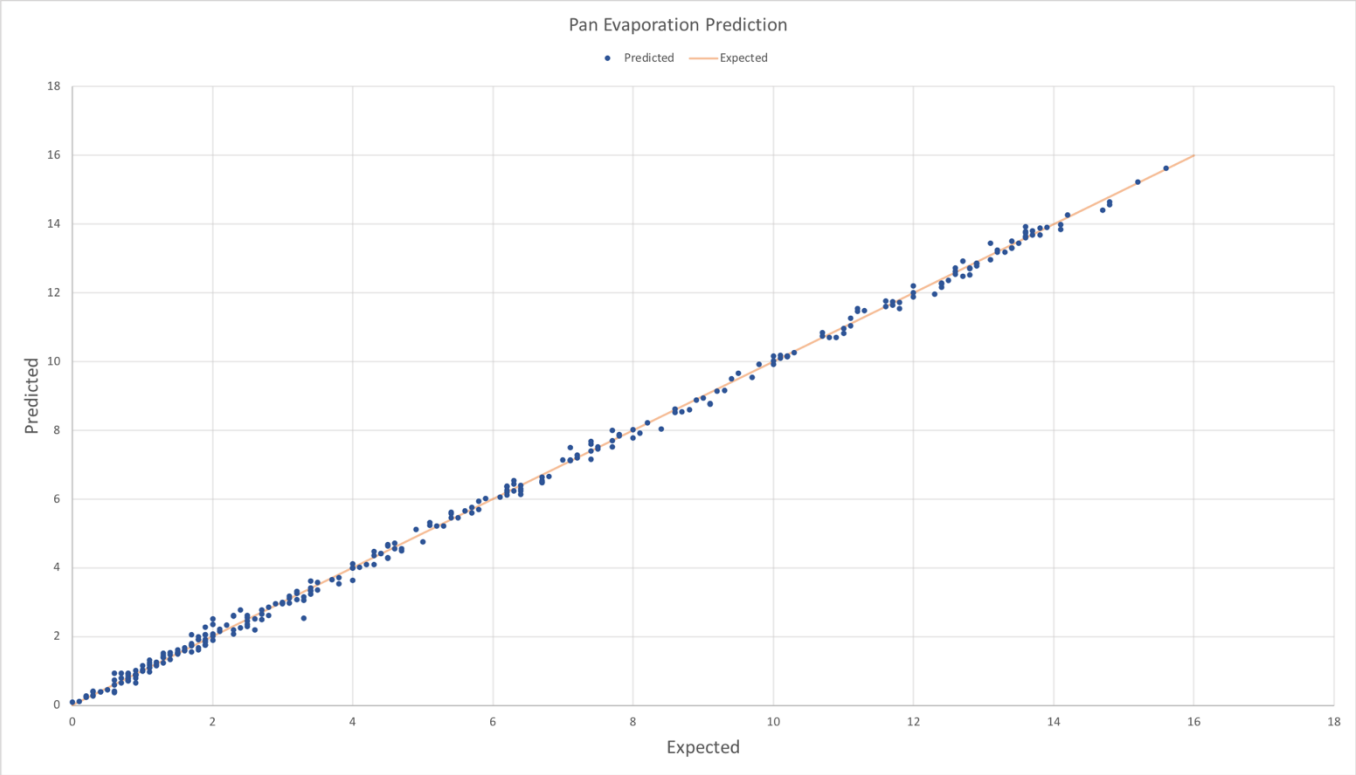
A further improvement of this final model evaluation method would be to implement a genetic algorithm to search the parameter space to find the best combination of random seed, momentum alpha and data splitting.

Best overall model:

The final model achieving best performance on the test data set:

Hidden neurons	8
Learning rate	0.1
Data split	60/20/20
Momentum	True (alpha = 1.1)
Bold Driver	False
Random seed	10
Validation RMSE	0.010292
Test RMSE	0.009849
Denormalised RMSE	0.158563

Results:



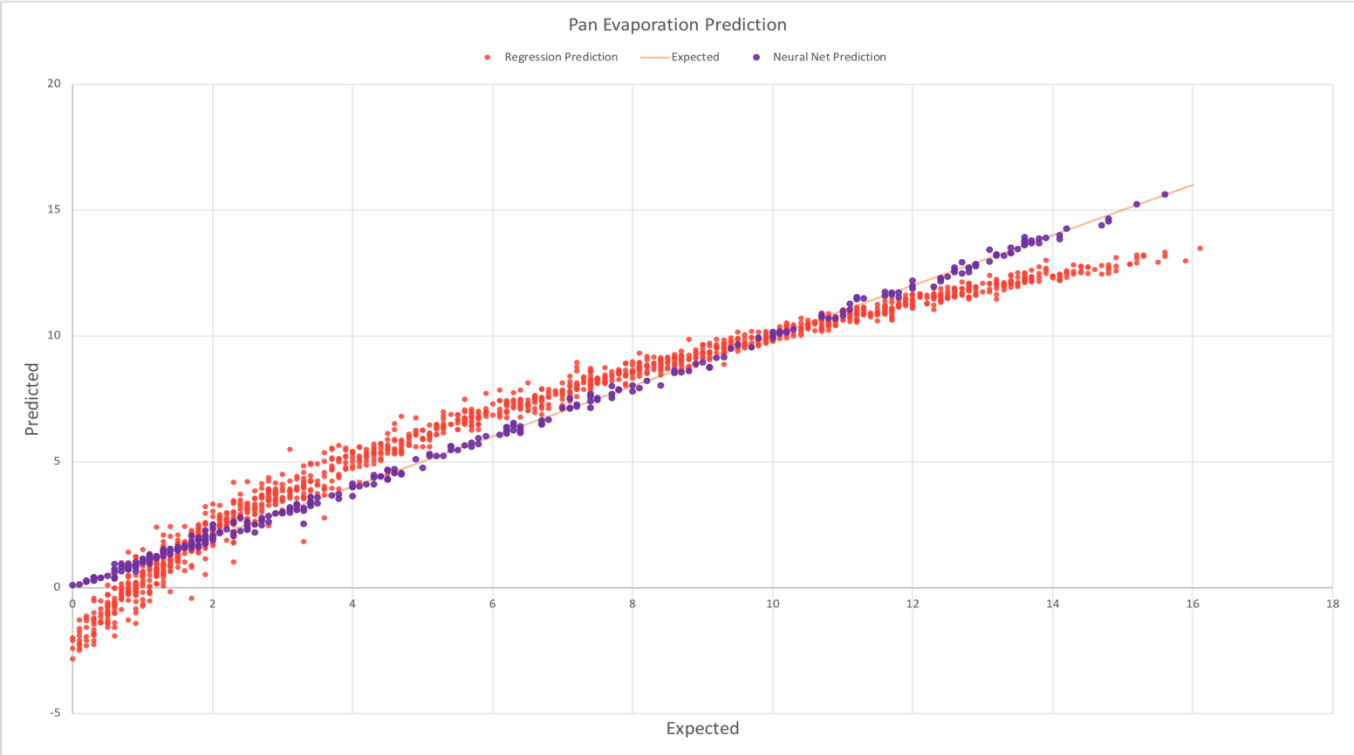
5. Comparison with Regression Model

The chosen comparative model was a **multiple linear regression**, performed in excel. This approach uses a multi factor polynomial to model the 5 predictors (X values) into the single predictand (Y value).

The coefficients table allows us to calculate the linear regression algorithm.

$Y = W_1X_1 + W_2X_2 + W_3X_3 + W_4X_4 + W_5X_5 + \text{Intercept}$
$Y = 0.234493*T + 0.601302*W + 0.008455*SR + 0.190915*DSP – 0.02954*DRH – 21.2007$

In regression analysis, the difference between the observed and predicted value is the **residual**. A residual output for the data set is returned by the model, allowing the points to be plotted on a graph and compared to the neural network prediction.



The **regression model achieves a RMSE of 0.93375**, calculated with the sum of squares of the residuals divided by the residual degrees of freedom. The regression RMSE is considerably higher than the neural network prediction (0.158163), indicating that the error is greater, and the **prediction is worse**.

This is clearly shown by the plotted predictions in the graph above. The regression model achieves moderate performance at predicting pan evaporation values in the range of 8-12mm/day but performs poorly at other ranges.

This regression model could be improved slightly by capping the lowest prediction to 0, as there cannot be a negative pan evaporation reading. However, this would only improve a number of predictions and the overall performance would still be far worse than the neural network model.

6. Conclusion

Neural network models using the backpropagation algorithm achieve an impressive predictive performance of pan evaporation. The best model was successfully found through the implementation of a genetic evolutionary algorithm and achieved a root mean squared error of 0.158563. Adding momentum to the backpropagation algorithm further improved results.

Source Code

For a better experience, please visit https://github.com/TristanJM/neural-network

NN.py (59 lines)

```
import numpy as np
import modules.const as const

np.random.seed(const.RANDOM_SEED)

# NN Model - Neuron Layer
class Layer:
    def __init__(self, num_inputs, num_neurons, activation_method):
        self.inputs = num_inputs
        self.neurons = num_neurons
        self.activation_method = activation_method

        # Stores last adjustments for momentum
        self.last_change_w = np.zeros((self.inputs, self.neurons))
        self.last_change_b = np.zeros((self.neurons))

        # randomly assign neuron starting weights and bias (-2/inputs -> 2/inputs)
        self.weights = np.random.normal(0, float(2)/self.inputs, (self.inputs, self.neurons))
        self.bias = np.random.normal(0, float(2)/self.inputs, (self.neurons))

    def activation(self, S, derivative=False):
        # Sigmoid
        if self.activation_method == 'Sigmoid':
            if derivative:
                return S * (1 - S)
            else:
                return 1 / (1 + np.exp(-S))
        # Linear
        elif self.activation_method == 'Linear':
            if derivative:
                return 1
            else:
                return S

    def output(self, values):
        S = np.dot(values, self.weights)
        S = np.add(S, self.bias)
        return self.activation(S)

    def update_weights(self, delta, inputs):
        # Reshape inputs to form: u(i), transposed for input to each neuron
        inputs = inputs.reshape(len(inputs),1)

        # Update weights by w(i,j) = w(i,j) + LR * delta(j) * u(i)
        w_change = const.LEARNING_RATE * delta
        w_change = w_change * inputs
        self.weights += w_change

        # Update bias
        b_change = const.LEARNING_RATE * delta
        self.bias += b_change

        # Momentum
        if const.MOMENTUM:
            self.weights += self.last_change_w * const.MOMENTUM_ALPHA
            self.bias += self.last_change_b * const.MOMENTUM_ALPHA
            # Record changes for next iteration
            self.last_change_w = w_change
            self.last_change_b = b_change
```

const.py (34 lines)

```
# Data
FILE = './data/data_clean.csv'
FEATURES = 5    # predictors

TRAIN_SPLIT = 0.6
VALIDATION_SPLIT = 0.2

# Model
NEURONS = [FEATURES, 8, 1] # Input, Hidden neurons, Output neurons
LEARNING_RATE = 0.1
```

```
MOMENTUM_ALPHA = 1.1
MOMENTUM = True
BOLD_DRIVER = False
BOLD_DRIVER_INCREASE = 1.1  # Multiply LR by this amount if error is lower
BOLD_DRIVER_DECREASE = 0.5
MAX_EPOCHS = 50000
VALIDATION_EPOCHS = 250 # check validation set error every X epochs

# Genetic Evol Algorithm
GA = False
GA_POP_SIZE = 28
GA_GENERATIONS = 20
GA_GENE_LEN = 12
GA_BEST_INDIVIDUALS = 6
GA_CROSSOVER_PB = 0.4
GA_MUTATION_PB = 0.1

# Options
TRAIN = False
MODEL_DIR = './model/'
MODEL_NAME = ''   # save model
LOAD_MODEL = 'best'   # '' or model name

RANDOM_SEED = 10
```

main.py (248 lines)

```
import numpy as np
import pandas as pd
import copy
import modules.const as const
import modules.data
import modules.NN as NN

# GA
from deap import base, creator, tools, algorithms
from scipy.stats import bernoulli
from bitstring import BitArray
gen_counter = 0
computed_errs = {}

np.random.seed(const.RANDOM_SEED)

def main():
    # If Genetic Algorithm
    if const.GA:
        # declare individual fitness
        creator.create('MaxFitness', base.Fitness, weights = (-1.0,))   # error minimisation problem
        creator.create('Individual', list, fitness = creator.MaxFitness)

        toolbox = base.Toolbox()
        # create chromosomes for individuals in the population
        toolbox.register('binary', bernoulli.rvs, 0.5)
        toolbox.register('individual', tools.initRepeat, creator.Individual, toolbox.binary, n = const.GA_GENE_LEN)
        toolbox.register('population', tools.initRepeat, list, toolbox.individual)

        # register genetic operations
        toolbox.register('mate', tools.cxOrdered)
        toolbox.register('mutate', tools.mutShuffleIndexes, indpb = 0.6)
        toolbox.register('select',  tools.selTournament, tournsize=4)
        toolbox.register('evaluate', train_nn)

        pop = toolbox.population(n = const.GA_POP_SIZE)
        hof = tools.HallOfFame(const.GA_BEST_INDIVIDUALS)

        # run evolutionary algorithm
        r = algorithms.eaSimple(pop, toolbox, cxpb = const.GA_CROSSOVER_PB, mutpb = const.GA_MUTATION_PB,
            ngen = const.GA_GENERATIONS, halloffame = hof, verbose = True)

        # Final population (sorted by best first)
        final_pop = sorted(pop, key=lambda ind: ind.fitness, reverse=True)
        print 'Final population:\n', \
            map(lambda x: '%s|%s' % (''.join(str(y) for y in x[0:8]), ''.join(str(y) for y in x[8:])), final_pop),'\n',\
            map(lambda x: '%03d | %.5f' % (BitArray(x[0:8]).uint+1, float(1)/(BitArray(x[8:]).uint+2)),final_pop),'\n',\
            map(lambda x: 'RMSE %.6f' % (x.fitness.values), final_pop)

        # Hall of Fame (best individuals from all generations)
        print 'Hall of Fame:\n', \
            map(lambda x: '%s|%s' % (''.join(str(y) for y in x[0:8]), ''.join(str(y) for y in x[8:])), hof), '\n', \
            map(lambda x: '%03d | %.5f' % (BitArray(x[0:8]).uint+1, float(1)/(BitArray(x[8:]).uint+2)), hof), '\n', \
            map(lambda x: 'RMSE %.6f' % (x.fitness.values), hof)
```

```
    else:
        mse = train_nn()

def train_nn(ga_individual=None):
    # Re-seed as GAs repeat this process
    np.random.seed(const.RANDOM_SEED)

    # Read in cleaned data CSV
    train_x, train_y, val_x, val_y, test_x, test_y = modules.data.read_data()

    # Set params from GA Individual
    hidden_neurons = const.NEURONS[1]
    if ga_individual:
        hidden_neurons = BitArray(ga_individual[0:8]).uint + 1
        const.LEARNING_RATE = float(1) / (BitArray(ga_individual[8:]).uint + 2)

        # Stats
        global gen_counter
        gen_counter += 1
        print '(Idx: %04d) | Neurons: %03d, LR: %05f' % (gen_counter, hidden_neurons, const.LEARNING_RATE)

        # Prevent recalculating the same model errors
        if (hidden_neurons,const.LEARNING_RATE) in computed_errs:
            return (computed_errs[(hidden_neurons,const.LEARNING_RATE)],)

    # Generate NN with random starting weights
    layer_hidden = NN.Layer(const.NEURONS[0], hidden_neurons, 'Sigmoid')
    layer_output = NN.Layer(hidden_neurons, const.NEURONS[2], 'Linear')

    load_model(layer_hidden, layer_output)

    # Train model
    j = 0
    if const.TRAIN:
        overfit = False
        last_val_err = None
        while j < const.MAX_EPOCHS and not overfit:

            # Bold driver - automatic Learning Rate adjustment
            if const.BOLD_DRIVER:
                adjustment_needed = True
                adjustment_count = 0
                while adjustment_needed and adjustment_count < 10:
                    layer_hidden_BD = copy.deepcopy(layer_hidden)
                    layer_output_BD = copy.deepcopy(layer_output)

                    err_before = eval_model(j, train_x, train_y, layer_hidden_BD, layer_output_BD)
                    # Train on train data
                    for idx, train_x_row in enumerate(train_x):
                        # Backpropagate - feed forward/back and calculate weights
                        vals, out_delta, hid_delta = backprop(train_x_row, train_y[idx], layer_hidden, layer_output)
                        # Update weights
                        layer_hidden_BD.update_weights(np.array(hid_delta), train_x_row)
                        layer_output_BD.update_weights(np.array(out_delta), vals)

                    err_after = eval_model(j, train_x, train_y, layer_hidden_BD, layer_output_BD)

                    if err_after > err_before:
                        # If error goes up, reduce the Learning Rate
                        if const.LEARNING_RATE * const.BOLD_DRIVER_DECREASE != 0.0:
                            const.LEARNING_RATE *= const.BOLD_DRIVER_DECREASE
                        adjustment_needed = True
                        adjustment_count += 1
                    else:
                        # If the error goes down, Learning Rate may be too small
                        adjustment_needed = False
                        const.LEARNING_RATE *= const.BOLD_DRIVER_INCREASE
                        clone_layer(layer_hidden_BD, layer_hidden)
                        clone_layer(layer_output_BD, layer_output)
            else:
                # Train on train data
                for idx, train_x_row in enumerate(train_x):
                    # Backpropagate - feed forward/back and calculate weights
                    vals, out_delta, hid_delta = backprop(train_x_row, train_y[idx], layer_hidden, layer_output)

                    # Update weights
                    layer_hidden.update_weights(np.array(hid_delta), train_x_row)
                    layer_output.update_weights(np.array(out_delta), vals)

            if j % const.VALIDATION_EPOCHS == 0:
                err = eval_model(j, val_x, val_y, layer_hidden, layer_output, 'validation')
                if last_val_err is not None and err >= last_val_err:
                    print "Overfitting detected"
                    overfit = True
```

```
                last_val_err = err
            j += 1

        save_model(layer_hidden, layer_output)

    val_error = eval_model(j-1, val_x, val_y, layer_hidden, layer_output, 'validation')
    tst_error = eval_model(j-1, test_x, test_y, layer_hidden, layer_output, 'test')

    computed_errs[(hidden_neurons,const.LEARNING_RATE)] = np.sqrt(val_error)

    if not const.GA:
        prediction = predict(test_x, layer_hidden, layer_output)

        # denormalise
        dn_prediction = modules.data.denormalise_data(prediction)
        dn_test_y = modules.data.denormalise_data(test_y)
        dn_text_x = modules.data.denormalise_data(test_x, True)

        # denormalised RMSE
        dn_sq_err = (dn_test_y - dn_prediction)**2
        dn_mse = np.mean(dn_sq_err)
        print 'Epoch %04d: %.6f MSE (%.6f RMSE) on %s set' % (j-1, dn_mse, np.sqrt(dn_mse), 'denormalised test')

        # Plot on graph
        modules.data.plot(dn_prediction, dn_test_y, 'scatter')

    return (np.sqrt(val_error),)

def backprop(train_x_row, expected, layer_hidden, layer_output):
    # Feed forward
    vals = layer_hidden.output(train_x_row)
    prediction = layer_output.output(vals)

    # Backward pass - output neuron
    output_derivative = layer_output.activation(prediction, True)
    output_delta = (expected - prediction) * output_derivative    # output neuron delta

    # Backward pass - hidden neurons
    hidden_delta = []
    for i, val in enumerate(vals):
        hidden_derivative = layer_hidden.activation(val, True)
        hidden_delta.append(((output_delta * layer_output.weights[i]) * hidden_derivative)[0])

    return [vals, output_delta, hidden_delta]

# Calculate model error statistic
def eval_model(epoch_num, x_data, y_data, layer_hid, layer_out, eval_type=''):
    p = predict(x_data, layer_hid, layer_out)

    sq_err = (y_data - p)**2
    mse = np.mean(sq_err)

    if len(eval_type) > 0:
        print 'Epoch %04d: %.6f MSE (%.6f RMSE) on %s set' % (epoch_num, mse, np.sqrt(mse), eval_type)
    return mse

# Use model to predict on given X data
def predict(data, layer_hid, layer_out):
    prediction = []
    for row in data:
        vals = layer_hid.output(row)
        prediction.append(layer_out.output(vals))
    return np.array(prediction)

# Save model weights to text file
def save_model(layer_hidden, layer_output):
    if len(const.MODEL_NAME) > 0:
        np.savetxt(const.MODEL_DIR + 'model_' + const.MODEL_NAME + '_hidden.txt', layer_hidden.weights)
        np.savetxt(const.MODEL_DIR + 'model_' + const.MODEL_NAME + '_output.txt', layer_output.weights)
        np.savetxt(const.MODEL_DIR + 'model_' + const.MODEL_NAME + '_hidden_bias.txt', layer_hidden.bias)
        np.savetxt(const.MODEL_DIR + 'model_' + const.MODEL_NAME + '_output_bias.txt', layer_output.bias)

# Load model from previous train
def load_model(layer_hidden, layer_output):
    if len(const.LOAD_MODEL) > 0:
        try:
            print "Loading model..."
            # Weights
            loaded_hidden = np.loadtxt(const.MODEL_DIR + 'model_' + const.LOAD_MODEL + '_hidden.txt')
            loaded_output = np.loadtxt(const.MODEL_DIR + 'model_' + const.LOAD_MODEL + '_output.txt')
            loaded_output = loaded_output.reshape(len(loaded_output),1)

            layer_hidden.weights = loaded_hidden
            layer_output.weights = loaded_output
```

```
            # Biases
            hidden_bias = np.loadtxt(const.MODEL_DIR + 'model_' + const.LOAD_MODEL + '_hidden_bias.txt')
            output_bias = np.loadtxt(const.MODEL_DIR + 'model_' + const.LOAD_MODEL + '_output_bias.txt')
            output_bias = output_bias.reshape(1)

            layer_hidden.bias = hidden_bias
            layer_output.bias = output_bias
        except IOError as e:
            print e

# Print model weights
def show_weights(layer_hidden, layer_output, info_str):
    print ">> Weights after %s:" % (info_str)
    print "Hidden weights:\n{}\nOutput weights:\n{}".format(layer_hidden.weights, layer_output.weights)

# Clone layer weights from layer 1 into layer 2
def clone_layer(layer1, layer2):
    layer2.weights = layer1.weights
    layer2.bias = layer1.bias
    layer2.last_change_w = layer1.last_change_w
    layer2.last_change_b = layer1.last_change_b

if __name__ == '__main__':
    main()
```

data.py (83 lines)

```
import warnings
import modules.const as const
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.random.seed(const.RANDOM_SEED)
plt.rcParams['figure.figsize'] = [16.0, 10.0]
warnings.simplefilter(action='ignore', category=FutureWarning)  # suppress np/mpl FutureWarning

col_max = None
col_min = None

# Read clean data from CSV, normalise, and split
def read_data():
    df = pd.read_csv(const.FILE, index_col='Date')
    df = df.drop('Month', 1)

    # normalise
    global col_max
    global col_min
    col_max = df.max()
    col_min = df.min()
    def normalise_data(row):
        for i, col in enumerate(df):
            row[col] = np.interp(row[col], [col_min[i], col_max[i]], [0, 1])
        return row

    df = df.apply(normalise_data, axis=1)
    data = np.array(df)

    # shuffle data
    np.random.shuffle(data)

    train_size = int(const.TRAIN_SPLIT * len(data))
    val_size = int(const.VALIDATION_SPLIT * len(data))

    train_x = data[:train_size][:, :-1]
    train_y = data[:train_size][:, -1:]
    val_x = data[train_size:train_size+val_size][:, :-1]
    val_y = data[train_size:train_size+val_size][:, -1:]
    test_x = data[train_size+val_size:][:, :-1]
    test_y = data[train_size+val_size:][:, -1:]

    return train_x, train_y, val_x, val_y, test_x, test_y

# Denormalise
def denormalise_data(val, dn_all=False):
    if dn_all:
        for i, row in enumerate(val):
            for j, col in enumerate(row):
                max_val = col_max[j]
                min_val = col_min[j]
                row[j] = (col * (max_val - min_val)) + min_val
        return val
    else:
```

```
        max_val = col_max['PanE']
        min_val = col_min['PanE']
        return (val * (max_val - min_val)) + min_val

# Plot prediction data
def plot(pred, expected, plot_type='scatter'):
    fig = plt.figure()
    ax1 = plt.subplot2grid((1,1), (0,0))
    ax1.grid(True)

    if plot_type == 'line':
        ax1.plot(pred, 'x', markersize=7, label='predicted', color='r')
        ax1.plot(expected, 'o', markersize=7, label='expected', color='b')
        plt.xlabel('Data point')
        plt.ylabel('PanE')
    else:
        max_val = col_max['PanE']
        min_val = col_min['PanE']
        ax1.plot([min_val,max_val],[min_val,max_val], color='black', alpha=0.5, label='Expected')   # reference line
        ax1.scatter(expected, pred, label='PanE')
        plt.xlabel('Expected')
        plt.ylabel('Predicted')

    plt.title('PanE Prediction')
    plt.legend()
    plt.subplots_adjust(left=0.06, bottom=0.18, right=0.95, top=0.95, wspace=0.2, hspace=0)
    plt.show()
```