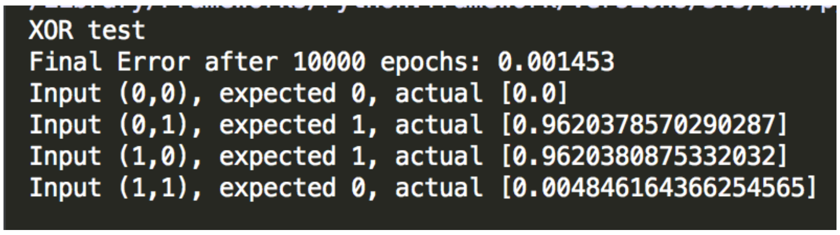
## To run the assignment:

1. Navigate to the directory within your terminal
2. run ‘python main.py’ to run the program in its default mode. This is the XOR example
3. Alternatively, you may use flags to run it in other modes. The flags can be combined to run multiple modes consecutively. Accepted flags are:
   - -x : XOR mode
   - -v : Vector mode
   - -w : Letter recognition mode
4. The error at each epoch is output to a csv file in the included outputs folder during execution.
5. The results of the test data are printed in the terminal, along with the final error after training

## XOR

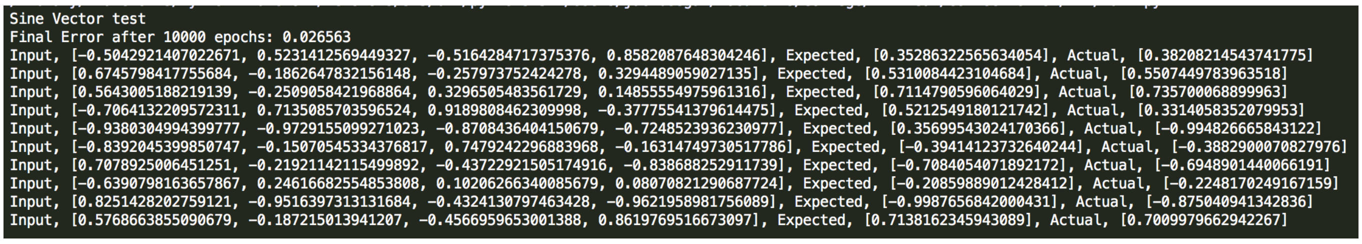
For this example, an MLP was created with 2 input units, 2 hidden units and 1 output unit.
Training was undertaken over 10,000 epochs with initial random weights in the range [-0.2, 0.2] and a learning rate of 0.09. These values were found through empirical observation of the error rate, with the aim of minimising this rate.
The training set was a simple list of all possible XOR combinations over a vector of length 2.
The output of error rate throughout training can be found in “outputs/xor20170501_020203.csv”
The tests I ran on this consisted of testing each combination of binary vectors of length 2, that were then compared to the expected output value from the known XOR function. The result can be seen in the image below. A low error rate has been achieved (0.001453), and the actual values closely matches the expected output for a given pair of inputs.
I felt this was a very successful attempt at training an MLP for XOR functionality.



```
XOR test
Final Error after 10000 epochs: 0.001453
Input (0,0), expected 0, actual [0.0]
Input (0,1), expected 1, actual [0.9620378570290287]
Input (1,0), expected 1, actual [0.9620380875332032]
Input (1,1), expected 0, actual [0.004846164366254565]
```

## Sine of Vector Combinations

For this example, an MLP was created with 4 input units, 5 hidden units and 1 output unit.
Training was undertaken over 10,000 epochs with initial random weights in the range [-0.5, 0.5] and a learning rate of 0.05. Again, these values were found through empirical observation of the error rate as these values were altered, with the aim of minimising this rate after a given number of epochs.
The training set consisted of 40 randomly generated 4 component vectors as input, with the output being the sine of the sum of these components.
The trend in the error rate throughout training can be found in “outputs/sin_vector20170501_021156.csv”
The tests I ran on this consisted of using 10 unseen 4-component vectors as inputs with the output compared to the sine of the sum of the components. The result can be seen in the image below. A low error rate has been achieved (0.026563), and the actual values closely matches the expected output in most cases, with the 5th test proving to be the least successful by a significant amount.
I felt this was relatively successful at training an MLP for sine functionality, given that even for unseen data, as in the test cases, the network could still perform to a satisfactory level.



```
Sine Vector test
Final Error after 10000 epochs: 0.026563
Input, [-0.5042921407022671, 0.5231412569449327, -0.5164284717375376, 0.8582087648304246], Expected, [0.35286322565634054], Actual, [0.38208214543741775]
Input, [0.6745798417755684, -0.1862647832156148, -0.257973752424278, 0.3294489059027135], Expected, [0.5310084423104684], Actual, [0.5507449783963518]
Input, [0.5643005188219139, -0.2509058421968864, 0.3296505483561729, 0.14855554975961316], Expected, [0.7114790596064029], Actual, [0.735700068899963]
Input, [-0.7064132209572311, 0.7135085703596524, 0.9189808462309998, -0.37775541379614475], Expected, [0.5212549180121742], Actual, [0.3314058352079953]
Input, [-0.9380304994399777, -0.9729155099271023, -0.8708436404150679, -0.7248523936230977], Expected, [0.35699543024170366], Actual, [-0.994826665843122]
Input, [-0.8392045399850747, -0.15070545334376817, 0.7479242296883968, -0.16314749730517786], Expected, [-0.39414123732640244], Actual, [-0.38829000708279776]
Input, [0.707892500645125l, -0.21921142115499892, -0.4372292150517491 6, -0.838688252911739], Expected, [-0.7084054071892172], Actual, [-0.6948901440066191]
Input, [-0.6390798163657867, 0.24616682554853808, 0.10206266340085679, 0.08070821290687724], Expected, [-0.20859889012428412], Actual, [-0.2248170249167159]
Input, [0.8251428202759121, -0.9516397313131684, -0.4324130797463428, -0.9621958981756089], Expected, [-0.9987656842000431], Actual, [-0.875040941342836]
Input, [0.5768663855090679, -0.187215013941207, -0.4566959653001388, 0.8619769516673097], Expected, [0.7138162345943089], Actual, [0.7009979662942267]
```

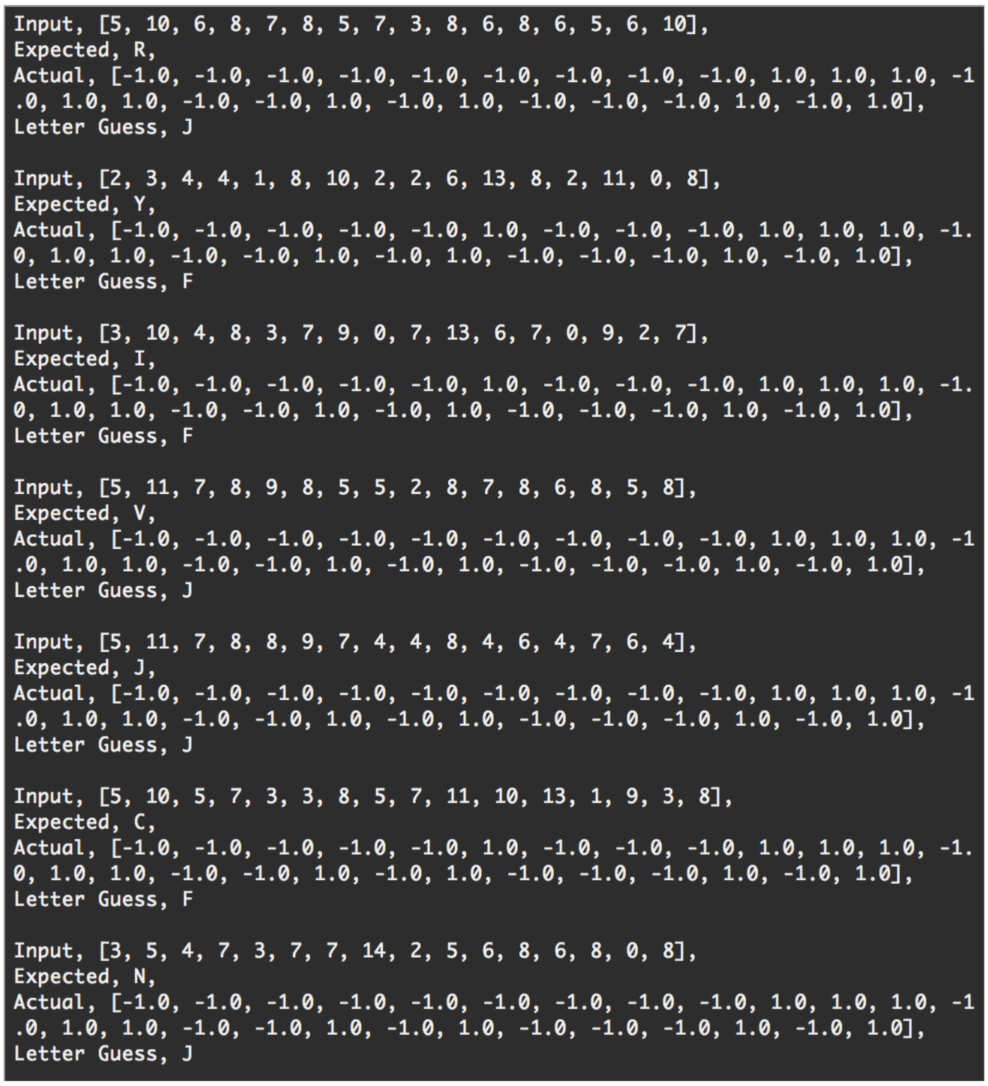
## Letter Recognition

For this example, an MLP was created with 16 input units, 10 hidden units and 26 output units. The program was set up to be able to run the learner recognising data set, when given the -w flag during execution. Training was configured to run over 1,000 epochs with initial random weights in the range [-0.5, 0.5] and a learning rate of 0.05. These values were untested and just carried over from the sine training defaults, due to the time constraints required in fine-tuning these values over such a large dataset.

The training set consisted of 16,000 vectors, each with 16 int attributes as an input, with the output being a 26-component binary vector, with only one instance on 1.0, denoting the position in the alphabet of the letter, i.e. the 0th component would be 1.0 for A, while all others would be 0.0, etc. The test set consists of using 4,000 unseen 16-component input vectors with the output again being a 26-component binary vector.

The attempted evaluation of this dataset was very poor, with no successful learning taking place. The error was at 55360.00 after 1000 epochs, with this value occurring from the second epoch onwards, with no further changes taking place. This can be seen in “letters20170507_205353.csv”

Some of the output from the test set can seen below, with each prediction based upon the first occurrence of the number 1.0 in the output vector, giving a letter value of F or J in all cases. Based on these results,
I believe such a data-set would require significantly more hidden units to function preferably, and ideally could utilise multiple hidden layers to process the data more efficiently.



```
Input, [5, 10, 6, 8, 7, 8, 5, 7, 3, 8, 6, 8, 6, 5, 6, 10],
Expected, R,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1
.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, J

Input, [2, 3, 4, 4, 1, 8, 10, 2, 2, 6, 13, 8, 2, 11, 0, 8],
Expected, Y,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1.
0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, F

Input, [3, 10, 4, 8, 3, 7, 9, 0, 7, 13, 6, 7, 0, 9, 2, 7],
Expected, I,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1.
0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, F

Input, [5, 11, 7, 8, 9, 8, 5, 5, 2, 8, 7, 8, 6, 8, 5, 8],
Expected, V,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1
.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, J

Input, [5, 11, 7, 8, 8, 9, 7, 4, 4, 8, 4, 6, 4, 7, 6, 4],
Expected, J,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1
.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, J

Input, [5, 10, 5, 7, 3, 3, 8, 5, 7, 11, 10, 13, 1, 9, 3, 8],
Expected, C,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1.
0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, F

Input, [3, 5, 4, 7, 3, 7, 7, 14, 2, 5, 6, 8, 6, 8, 0, 8],
Expected, N,
Actual, [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1
.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0],
Letter Guess, J
```