Connectionist Computing report

Code explanation

I created an MLP object based on the project spec provided. I used Python as it is an intuitive language. My main language would be Java but for Machine learning and my final year project I used Python because of its useful functions so I continued with it here.

Simple code overview.

The randomise() function initialises() the MLP, W1 and W2 to a small random number to begin with (between 0 and 1).

Forward() propagates the input through the system and 'O' stores the result of this.

Backward() function computes the error of the current output based on the target output and this error is back propagated through the system.

For forward and backward functions, I have an activation_type parameter that states which activation function to use; either 'tanh' or 'sigmoid' The logic for both is similar, they just use different functions and derivatives. Sigmoid is used for XOR and tanh is used for SIN and LETTER recognition.

Q1 XOR

For this test I tested several different learning rates and hidden unit values. This was also quite quick to run so I was able to run it for 1,000,000 epochs.

Here the inputs to the MLP were [0,0],[0,1],[1,0],[1,1] and the outputs were [0],[1],[1],[0]. For training, it is quite simple, forward all inputs through the system, there is no partial training set here as all are needed.

For testing I just forwarded the input array through the model one more time and calculated accuracy. I initially was rounding the numbers, but I could get a more detailed accuracy by including the exact number e.g., 0.98765 instead of 1. Not that it is important as the system performs well and learns XOR nearly perfectly. All values show the error at each epoch.

Hidden units: 3

	100	1000	10000	100000	1000000
1	0.499847146	0.09759387	0.019678046	0.005846027	0.001810651
0.75	0.500027591	0.236740911	0.023395025	0.00678556	0.002094155
0.5	0.499992559	0.394370841	0.028798388	0.008348223	0.002574011
0.25	0.500127303	0.477153087	0.043313917	0.011965232	0.003659465
0.05	0.499226647	0.499208653	0.208941011	0.028538401	0.008345091

Hidden units: 4

	100	1000	10000	100000	1000000
1	0.499287425	0.082172773	0.018422167	0.005524917	0.001713429
0.75	0.499473559	0.145907323	0.022045741	0.005915093	0.001471395
0.5	0.499357782	0.204497558	0.027411105	0.008095858	0.002529391
0.25	0.500030359	0.499870494	0.044381911	0.011401821	0.003488032
0.05	0.498605635	0.497030365	0.188225161	0.026632428	0.007895126

Hidden units: 5

	100	1000	10000	100000	1000000
1	0.500204842	0.149283625	0.0166022	0.004396904	0.001380659
0.75	0.499967127	0.130237152	0.021147538	0.005421562	0.001649038
0.5	0.500215037	0.497388282	0.027223069	0.00626665	0.001919489
0.25	0.499831646	0.426740134	0.038559374	0.00905942	0.002666852
0.05	0.498680523	0.499051069	0.175425061	0.023039256	0.006736022

For the most part, this test liked a higher learning rate. With the lowest error reported at 5 hidden units and 1,000,000 epochs

When I tested this lowest error model on the test data, I realised an accuracy of 99.86% which is very high. If you round it is 100%. Error decreased across the board and it shows that the MLP can learn XOR.

Q2 Sin()

First, I needed to create the vectors for inputs into the model. Python library numpy has an intuitive random.uniform function which gives you an array of any specified shape in a given range. I generated 4 numbers between -1 and 1 500 times. For each of these generated inputs, I calculated sin(x1 – x2 + x3 – x4) and that was the output set. I trained the system on 400 of them and tested it on the remainder.

For this test, I tested multiple hidden unit values. Accuracy seemed to get better up until 20 hidden units and the more epochs the better. This one took a bit longer to run so I only ran it 100,000 times. All values show the error at each epoch.

Hidden units: 10

	100	1000	10000	100000
0.1	10.00324075	47.98382381	40.72954415	40.20268718
0.01	0.080224797	0.066625561	0.060341458	0.066114
0.001	0.205062017	0.059307945	0.028382267	0.024908914
0.0001	0.373252715	0.183657245	0.059450082	0.032980124

Hidden units: 20

	100	1000	10000	100000
0.1	7.620822984	12.98342717	70.73601668	16.91509726
0.01	0.174380379	0.095628896	0.130603147	0.078942477
0.001	0.210381228	0.041048271	0.021299633	0.006440311
0.0001	2.516068305	0.169135609	0.042295982	0.021971744

Hidden units: 50

	100	1000	10000	100000
0.1	47.84730069	193.3782881	204.4374158	206.7817896
0.01	3.047350668	0.127198048	0.112353979	0.112843739
0.001	4.828196812	0.072879209	0.023055549	0.012156018
0.0001	9.750266354	6.937388815	0.047488983	0.019206255

As you can see, the lowest error was obtained with 20 hidden units and at 100,000 epochs with a learning rate of 0.001. The accuracy is calculated as 1 – difference of output and results(error) / 100 (number in test set). The accuracy here was 99.28% which is very impressive. Error for the most part decreased across the board which shows the MLP has a capacity to learn sin()

Q3 Letter-recognition

This test was more difficult. Using the dataset provided, and a csv_reader provided in the csv library in Python, I parsed the file and stored all inputs in an array. The data at index 0 is the letter output so I stored than in an output array and the rest in the input array. For training the model, my method was to generate a x by 26 array where it is 0 at all indexes except for the corresponding output. For example. 'A' had position 0 so if 'A' was output , the output array for that row would look like [1,0,0,0,0,…]. 'B' would have index of 1 and 'C' 2 and so on and so on. I stored this in a real_output array and here I also needed to split the data between training and testing with 4/5 for training and the remainder for testing. The error here would then be the difference between that generated real_output array with the model's output array.

For this question, the model took a long time to train, so to test what a decent learning rate would be, I ran it for learning rates at 100 epochs at 10 hidden units and the results are as followed. All values show the error at each epoch.

	0.5	0.05	0.005	0.0005	0.00005	0.000005
100Epoch	1914.519564	277.8489916	139.9689056	19.86113083	2.649200442	4.532510106

You can see here that 0.00005 has the lowest error of all so that is the one I went with for my further testing.

To test whether the model was learning I calculated the overall error before testing, during testing and after testing.

Hidden units: 10

	100	1000	10000	100000
0.00005	3.003924036	2.018526198	1.713598852	1.68391825

Hidden units: 15

	100	1000	10000	100000
0.00005	7.204063586	6.220235014	5.40884837	4.233244297

Hidden units: 20

	100	1000	10000	100000
0.00005	10.19086881	10.18695448	10.1330347	9.508806778

For this test I calculated the error at each epoch as before and I also tested the error before training and after training. The lowest error I recorded was with 10 hidden units on 100,000 epochs. With more hidden units the error seemed to increase for this test.

```
Epochs = 100000
Learning rate = 5e-05
Hidden units = 10


Error before training = 4.880819088527534

 Error at Epoch:    100   is    3.0039240355315746
 Error at Epoch:    1000      is    2.0185261978717564
 Error at Epoch:    10000     is    1.7135988524100825
 Error at Epoch:    100000    is    1.683918249751138

Error after training = 1.6856339122180894
```

Here you can see the decrease in the error before and after training and it has decreased which would imply learning. For training this model the more epochs the better, it would be interesting to run it for 1 million or more epochs and see how well it can perform. I also tested it with normalising the inputs but there was not much difference in learning.

Conclusion

Overall, this was a very interesting assignment, we know the capabilities of neural networks and usually see them at the cutting edge of technology and being able to create a simple one myself was enjoyable. I saw a decrease in error across the board (when things like learning rate and hidden units were respectable) from epoch to epoch which implies there is learning taking place. I realised good accuracy for the first two tests but for the letter recognition test it was definitely learning but it was not great. With more time I would increase the epochs of the letter recognition test and I would work on the prediction analysis. I learned a lot about MLPs and the different activation functions and during implementation I was doing a lot of printing to the screen to see how the algorithm worked and it is fascinating to see the backpropagation or error adjusting the weights to reduce the error. I also learned the effect different learning rates and hidden units have on certain algorithms and it was interesting to explore that. The more epochs the better is what I learned, which makes sense as it gives the model more time to reduce the error. It also showed my the general learning capabilities of MLP and it was great to implement that myself.