Seminar Week 4: Answers

1 Question 1

The vector **x** has already been implemented. Each component $y(i)$ of the vector **y** can be calculated according to:

$$y(i) = 0.6 + \frac{1}{4}sin(2\pi x(i)) + \frac{1}{4}cos(4\pi x(i))$$

NB: When we do the calculation on a component basis, we have x(i) within the definition of y(i). Matlab (and many other languages) make it possible to make the calculation on a vector basis in which case, the code becomes:

$$y = 0.6 + \frac{1}{4}sin(2\pi x) + \frac{1}{4}cos(4\pi x)$$

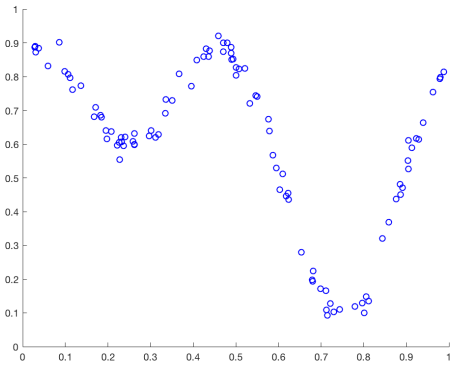
These two formulations do exactly the same thing but the latter is much more efficient. Now, we need to add some noise. We use the *randn()* function of Matlab. If you use the help function of Matlab (and I strongly advise you to do it every time you use a Matlab function, namely, type: help randn in this case), you would see that *randn(n,m)* gives a matrix of size (n,m), where each value is taken from a Gaussian distribution centred at 0 and with standard deviation 1. Therefore to make it a Gaussian noise of standard deviation 0.025, we simply need to multiply by 0.025 (or, equivalently, divide by 40). Therefore we use:

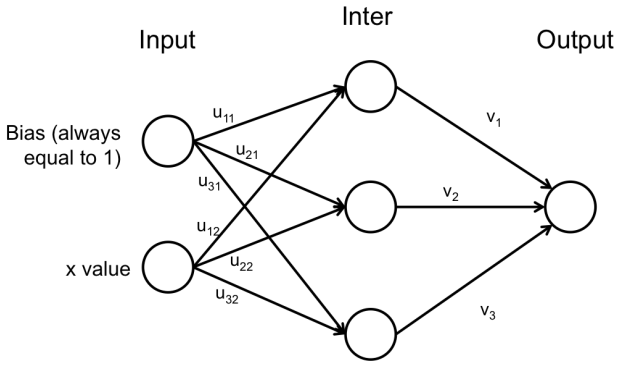
$$y = y + randn(1, 100)/40$$

If you hit control + enter, you should get a figure that looks like the following (it will be slightly different obviously as you will have different random numbers).

2 Question 2

Now that we have the curve, we want the MLP to approximate it. You need to think of MLP as a black box. You give it one input and it gives you one output. In our case the input is the x-coordinate of the point and we want to get the y-coordinate of the point. To gain more precision, we also added a bias. This is why you can see that we wrote Input=[1;x(i)]. Take a look at the structure of the MLP.





For each index i, we have a x(i) and we want the MLP to give us a value: Output(i), which is not too different from y(i). This is why the error is:

$$Err(i) = \sum_i (y(i) - Output(i))^2$$

y(i) is the target, the value that we want to have, whereas Output(i) is the value that the MLP produces. We square the difference between target and output so that we always have a positive error. Remember that if you accept negative errors, then the sum of errors could sum up to zero even though all points are incorrectly predicted. Now, how does it work? Because there are two layers, the MLP is working in two steps. The first step computes Inter (the hidden layer), and the second step computes Output. This is the forward phase. We go from the left to the right. There are three Inter (hidden) states, the three values of the vector Inter. So we have:

$$\begin{cases} Inter(1) & = g(1 \times U(1,1) + x(i) \times U(1,2)) \\ Inter(2) & = g(1 \times U(2,1) + x(i) \times U(2,2)) \\ Inter(3) & = g(1 \times U(3,1) + x(i) \times U(3,2)) \end{cases}$$

which can be summarised as:

$$\forall j \in \{1,2,3\}, Inter(j) = g(dot(U(j,:), Input))$$

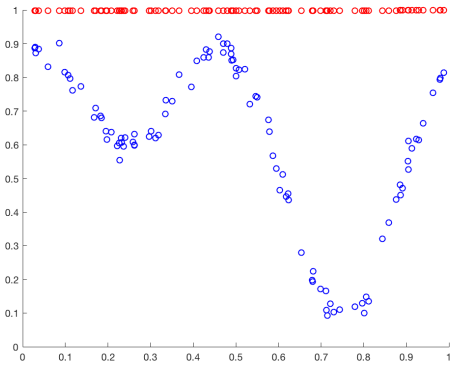
where Input = [1;x(i)] is a vector of size 2 and dot() is the dot product of two vectors. The Output is calculated using:

$$Ouput(i) = g(Inter(1) \times V(1) + Inter(2) \times V(2) + Inter(3) \times V(3))$$

which can also be summarised as:

$$Ouput(i) = g(dot(V, Inter))$$

You should understand that depending on the values of weights U and V, the output is going to be different. The idea behind the MLP is to find the right weight matrices U and V, so that the error is minimised. When we give to it a value x(i), then the value Output(i) given by the MLP should not be too different from y(i). If we had not used the g function (i.e. if we were to use $g : x \to x$), then you should realise that the MLP would have been linear. We would have given to it one input x(i) and it would have made a linear combination of this input and of the bias to output Output(i). The output would have necessarily been a linear combination of the bias and the input, x(i). Therefore, the MLP would not have been able to approximate a function that is not a line. This is why we use the g function. The g function is a sigmoid. The main thing that you should take from that, is that because g is curved, it is possible to arrange correctly many g functions together so that this combination of g functions approximate a curved function. You may not understand how it works, but you do not need to: the MLP is a black box. Now, If you hit control+enter within the second section, you would see the following:



We observe that the red dots do not match the blue dots very well. Because we have randomly initialised the matrices U and V, it is not very likely that the weights have been correctly arranged to approximate the curve, and therefore this is not surprising.

3 Question 3

Now we implement the back propagation so that the weight matrices U and V enable the MLP to approximate the curve. I am not going to demonstrate mathematically how to get the update rule, but I am going to take the rules from the lecture. Because we did not use here the same set of indices as those used within the lecture notes, some translation is needed. You can associate:

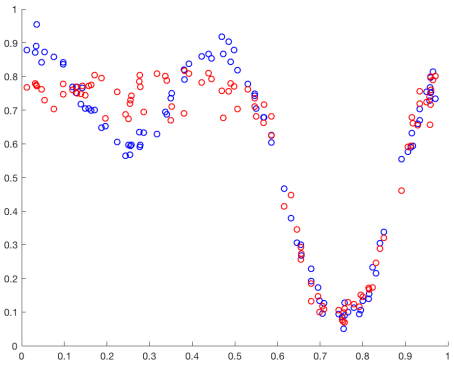
$$\begin{cases} d_k & \rightarrow & y(i) \\ y_k(t) & \rightarrow & Output(i) \\ w_{kj}(t) & \rightarrow & V(j) \\ v_{jl}(t) & \rightarrow & U(j,l) \\ x_l(t) & \rightarrow & Input(l) \\ a_k(t) & \rightarrow & \sum_{j=1}^{3} Inter(j)V(j) = dot(Inter, V) \\ u_j(t) & \rightarrow & \sum_{l=1}^{2} Input(l)U(j,l) = dot(Input, U(j,:)) \end{cases}$$

Therefore we obtain the following update rules:

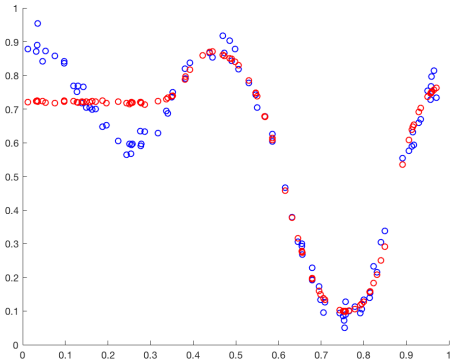
$$\begin{cases} \Delta = -g(dot(Inter, V)) \times (y(i) - Output(i)) \\ \forall j \in \{1,2,3\}, \delta(j) = g'\Big(dot(Input, U(j,:))\Big) \times V(j) \times \Delta \\ \forall j \in \{1,2,3\}, \forall l \in \{1,2\}, U(j,l) = U(j,l) - \eta \times \delta(j) \times Input(l) \\ \forall j \in \{1,2,3\}, V(j) = V(j) - \eta \times \Delta \times Inter(j) \end{cases}$$

4 Question 4

If you implement those rules and you hit control + enter a sufficient number of times, you should see that the points are converging. In the following Figure, we used $\eta = 1$:



We observe that the red points are stuck in a poor approximation of the first part of the curve (i.e. at x = 0, x = 0.2 and x = 0.4). There is a local minima: if all the points were to go up, the error would be decreased for the points that are close to the x-coordinate of the maxima of the blue curve, while it would increase for the points that are close to the x-coordinate of a local minimum of the blue curve. The opposite is true if we were to make all the points go down. Now if I decrease the learning rate to $\eta = 0.01$, I observe that the red points are stuck within this local minimum of the error.

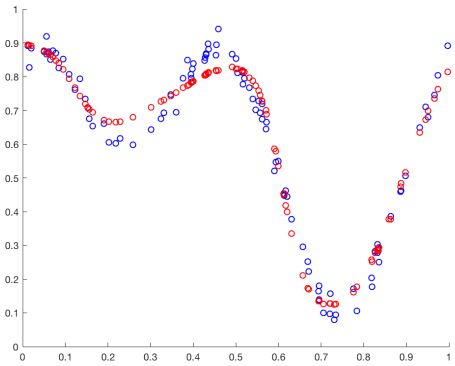


Indeed if the learning rate η is too small, then we will not be able to move out from the local minima. Therefore we need to start with an η that is big enough to be sure that we will not fall within a local minimum. On the opposite, if η is too big then we will make steps that are too large and we will never be able to get close to the function we want to approximate. Try with $\eta = 10$ for instance, you should see that the points are not converging. Therefore a strategy is to find the right η to start with (e.g. 0.5) and decrease it to use steps that are small enough to smooth the red points (e.g. until 0.01). If you did it correctly, you should be able to observe the following kind of behaviour.

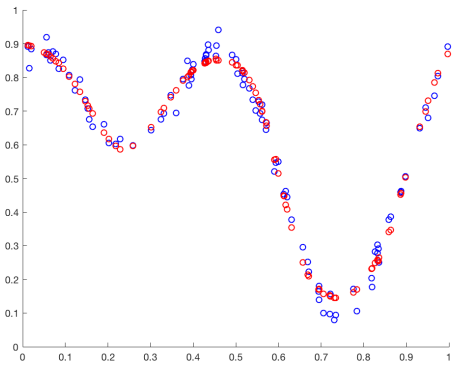
NB: The use of $\eta > 1$ above is for the sake of illustration. In general, it does not make any mathematical sense to use $\eta > 1$. This is because the gradient is only accurate for small changes. It is possible for values of $\eta > 1$ to occasionally lead to interesting results but this should not relied upon.

5 Question 5

You should not be satisfied with the above behaviour, because it clearly does not fit correctly the y function ($y = 0.6 + \frac{1}{4}sin(2\pi x) + \frac{1}{4}cos(4\pi x)$). This is because you used only 3 states. If you were to increase the number of states to 4 for instance, then using the same procedure as before, you should be able to



obtain the following kind of plot:



This is because the number of hidden states corresponds more or less to the complexity of the MLP that you are using. The more states you have, the more combinations there are and therefore the more accurate your approximation can be. Be careful not to use too many states, however, because then it takes much more time to compute and you may overfit the data (i.e., learning the noise!).

6 Question 6

Because we update the weights for each i, it is a sequential gradient descent algorithm. The batch version would consist in updating the weights according to the mean of the update of each of the individual i updates. It is implemented in the last section of the code. You should notice that in that case it is harder, it takes more time, to have the red points converging. This is because the batch version will update the weights hundred times slower than the sequential

version. However, you should note that whatever the value of η you are taking, the red points still make a smooth line. This is because the weights are changed for all points at the same time.