Question 1: As you have seen, the error (performance) sometimes converges to a value greater than zero. Explain why this happens.

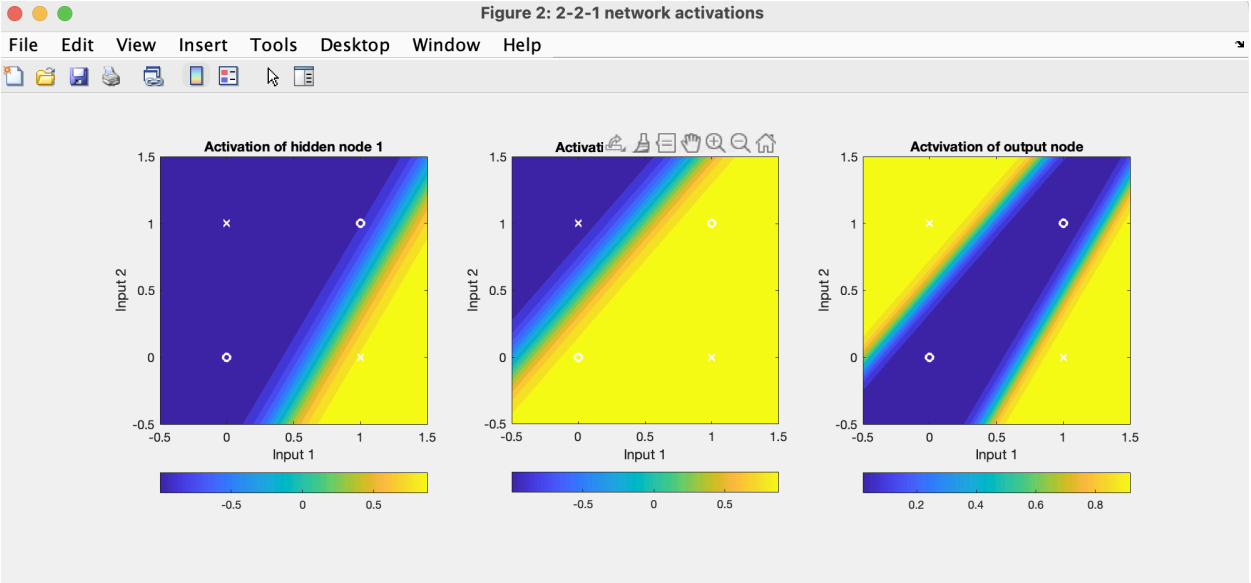
Due to different initializations, the error converges to different local minima. This means there are multiple minima in the problem and only sometimes do we reach the global minimum where the error converges to near 0.

Question 2: How does the learning rate affect the training of the network? What are the likely effects of using a too-low value? What are the likely effects of using a too high value?

The learning rate determines **how fast and how well** the neural network learns.
If the learning rate is **too low**, the weight updates are very small, causing training to take a long time. More epochs are needed to reach a satisfactory error level. Since updates are tiny, there is a risk of the network getting stuck in a local minima.
At LR 0.01, we see almost no run reaching the optimal minimum.
At LR 0.1, we see 4/10 reaching the optimal minimum.

At LR 2, we see again 4/10 runs reaching the optimal minimum. This suggests that an LR between 0.1 to 2.
If the learning rate is **too high**, weight updates **overshoot** the optimal point, making training unstable. The error fluctuates widely and does not settle. There is a high possibility that the network will never converge.
At LR 20, we see the loss fluctuating rapidly and only 2/10 runs converging properly.

Save the figures drawn by plot_xor for two different solutions you found to solve the xor problem appropriately. Motivate why they are solutions.



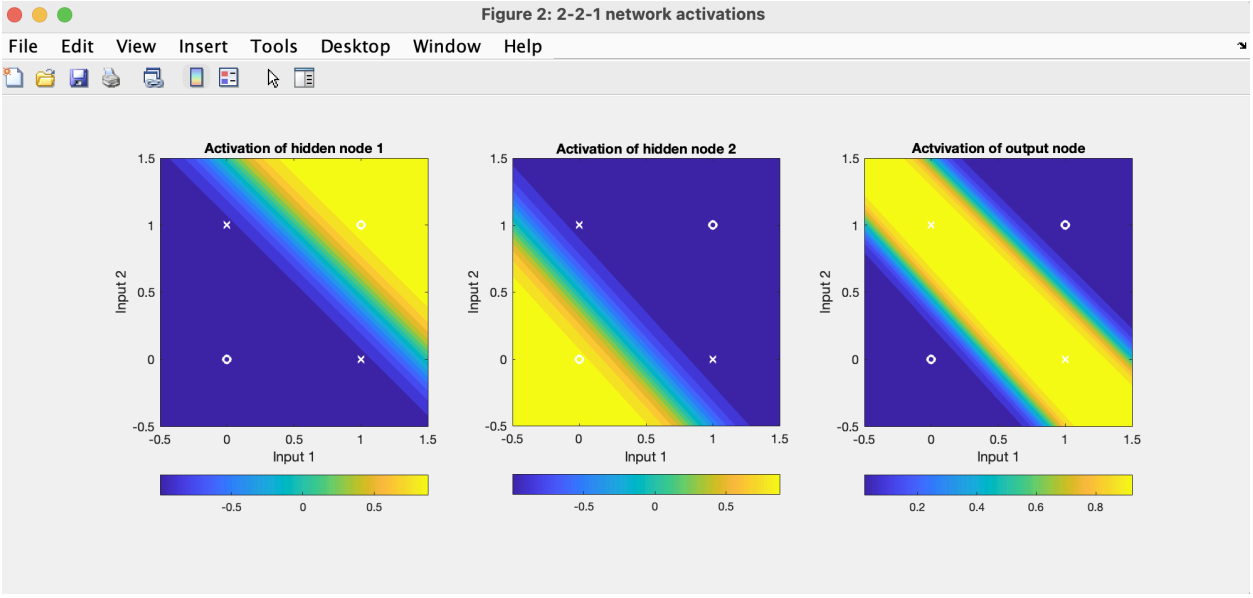
Node 1:

X1	X2	Not X2	X1 (AND) Not X2
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Node 2:

Not X1 (AND) X2

Output Node: XOR = Node1 (OR) Node2



Node 1: AND
Node 2: NOR
Ouput Node: XOR = AND (NOR) NOR

The above plots point to the two solutions. The first one separates the (0,1) and (1,0) in blue from (0,0) and (1,1) in yellow. This implies that the final neuron is differentiating between them as intended. The second figure is just a mirror reflection of the first, with the final neuron activating on opposite values.

Question 3:
Why are the activations of the hidden nodes in the range [-1,1], but the output of the network in the range [0,1]? Hint: remember how the network was created

The hidden layer uses the tansig activation function, which outputs values in the range [-1,1].

$$\text{tansig}(x) = \frac{2}{1+e^{-2x}} - 1 \quad \Rightarrow \quad [-1, 1]$$

The output layer uses the logistic sigmoid (logsig) activation function, which outputs values in the range [0,1].

$$\text{logsig}(x) = \frac{1}{1+e^{-x}} \quad \Rightarrow \quad [0, 1]$$

Question 4:

Observing the curves, why does the training not always end up with the same solution, even when the same training parameters are used?

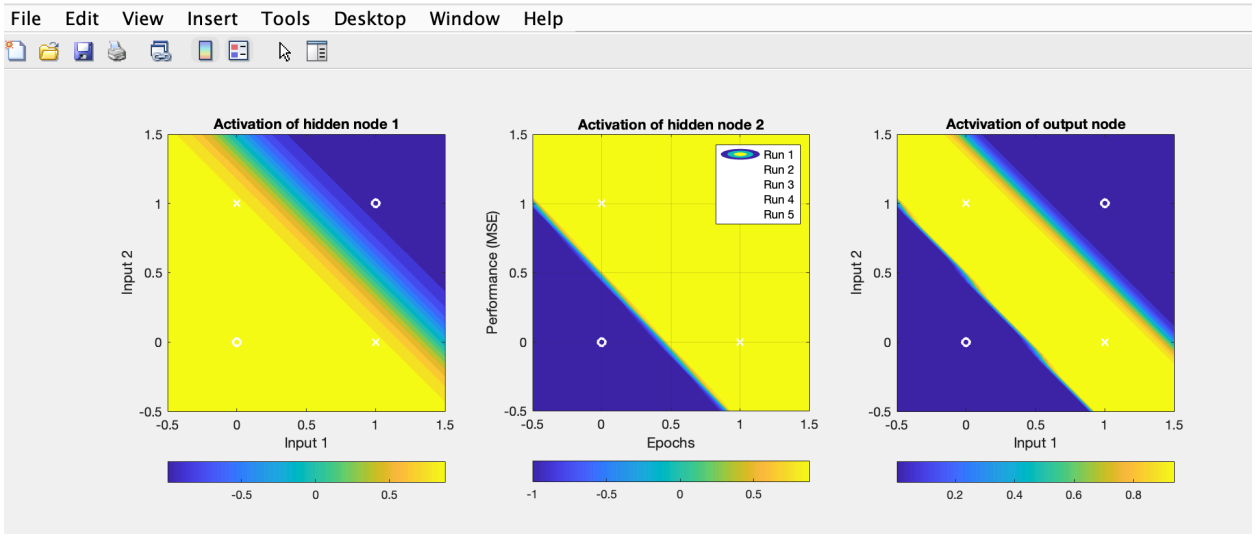
Each time we call init(net), MATLAB **randomly initializes** the weights. Thus, different starting weights lead to **different learning techniques**, causing the network to find **different local minima**.

Furthermore, if the learning rate is **too high**, different runs might **oscillate** and not settle on the same solution.

Plot 3: Save a figure of performance statistics for 10 training sessions under the same conditions using Rprop, in the same way as before. What parameters did you use?

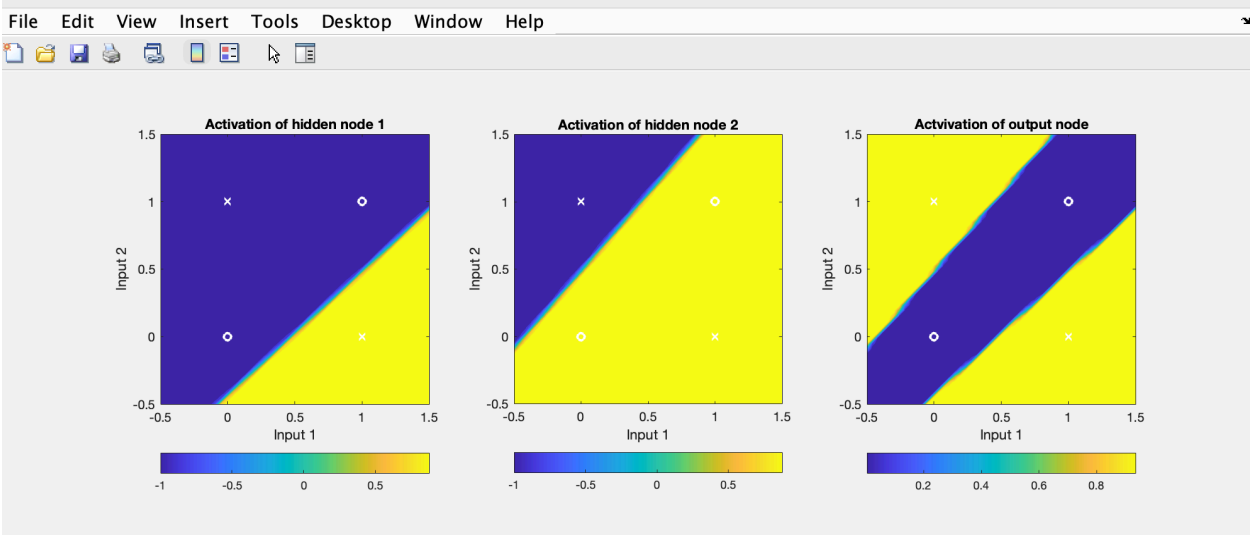
Question 5:
What are the most important differences that you can observe between the results produced by Rprop and backpropagation?
Also look at the function implemented by the network using plot_xor, and see if you notice anything different when the network is trained using Rprop.
Which of the two training algorithms do you think is most suitable for this problem, and why?

R_Prop



Node 1: NAND
Node 2: OR
Output Node: XOR = NAND (AND) OR

R_Prop (Another one)



Node 1:

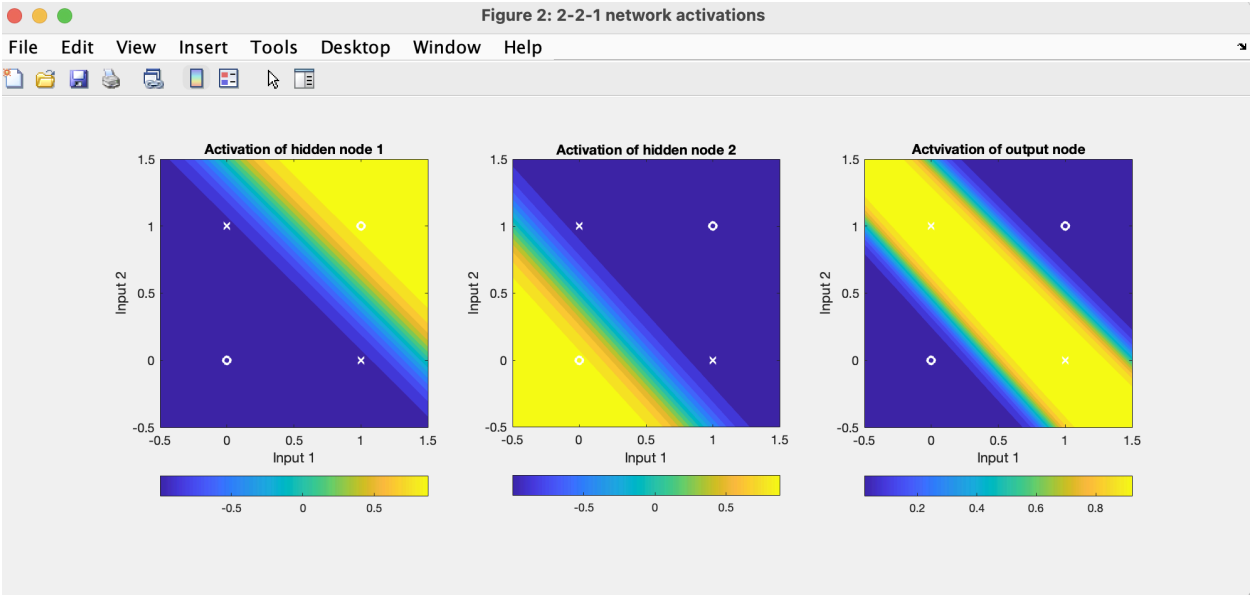
X1	X2	Not X2	X1 (AND) Not X2
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Node 2:

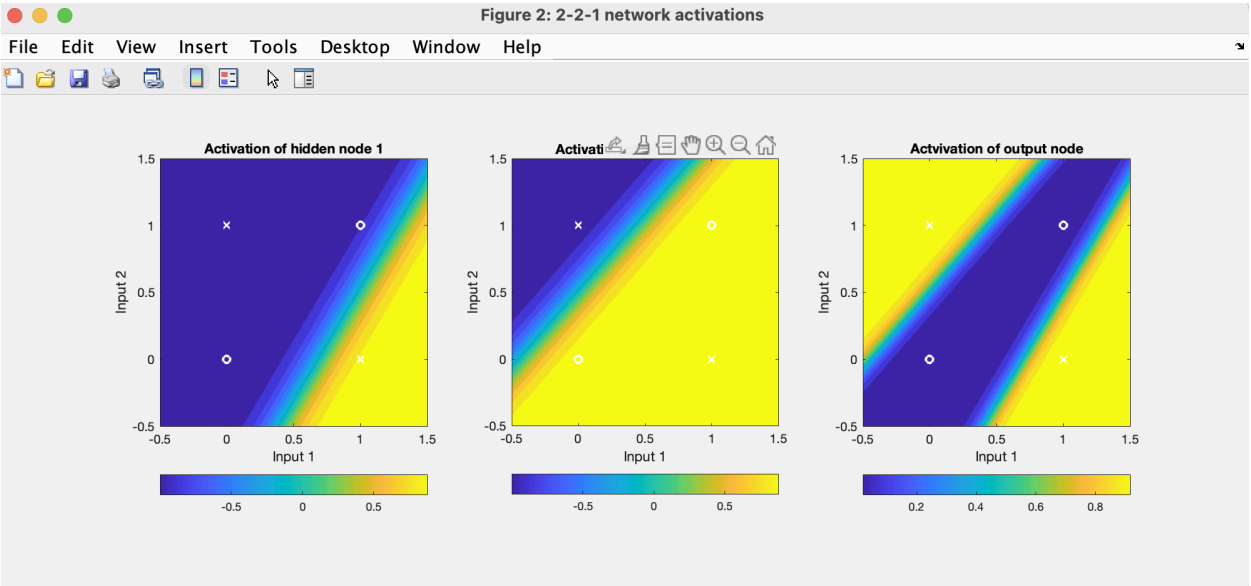
Not X1 (AND) X2

Output Node: XOR = Node1 (OR) Node2

Gradient Descent



Node 1: AND
Node 2: NOR
Ouput Node: XOR = AND (NOR) NOR



Node 1:

X1	X2	Not X2	X1 (AND) Not X2
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Node 2:
Not X1 (AND) X2

Output Node: XOR = Node1 (OR) Node2

Difference:

Gradient Descent	**R_Prop**
The decision boundary in the output layer activation plot appears smoother but slightly curved.	The decision boundary is sharper and more defined, indicating faster convergence.
Why? Weight updates were gradual and smooth.	Why? Weight updates were more aggressive and dynamic.
Why in detail: Backpropagation with gradient descent (traingd) applies small, incremental weight updates. The weight updates are proportional to both the gradient magnitude and sign. This results in slower convergence and sometimes gets stuck in local minima. Gradient descent often gets stuck in small local minima, which explains the less confident decision boundary.	Why in detail: Rprop ignores the gradient magnitude and only considers the direction of weight changes. This allows for faster adaptation, leading to sharper boundaries. When a weight consistently moves in one direction, Rprop increases the step size, making training much more efficient. Rprop avoids this by adapting step sizes, allowing it to move past local minima faster

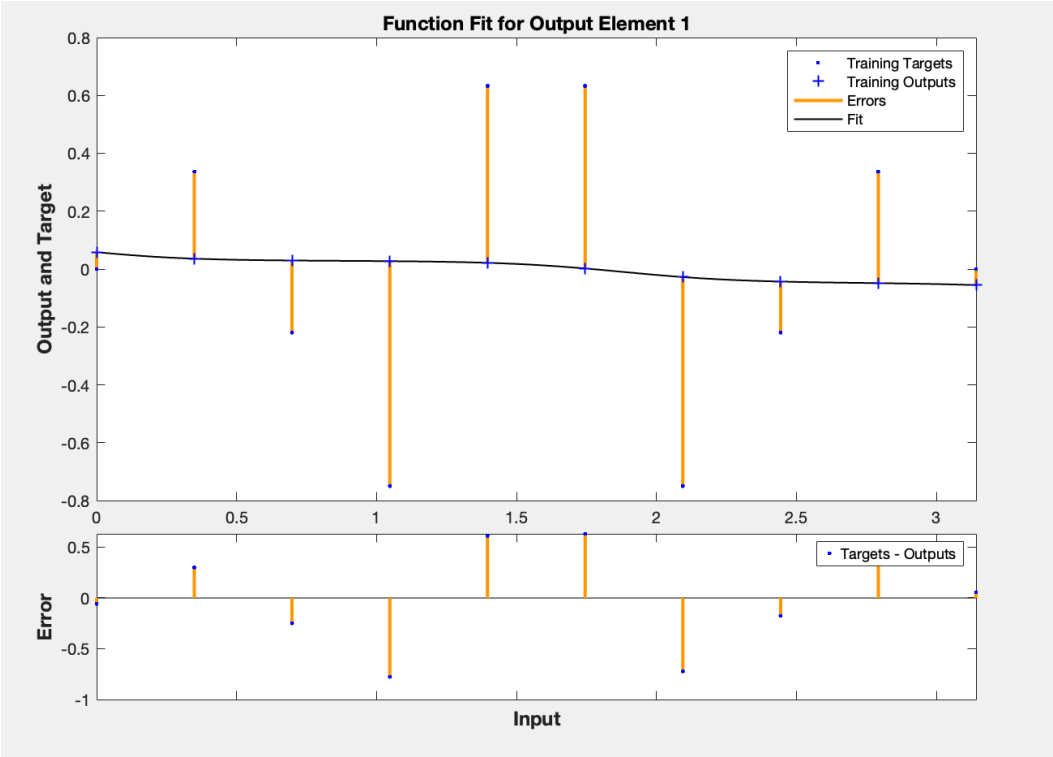
Rprop is better.
Rprop ignores the gradient magnitude and only considers the direction of weight changes.
This allows for faster adaptation, leading to sharper boundaries.
When a weight consistently moves in one direction, Rprop increases the step size, making training much more efficient.
Rprop avoids this by adapting step sizes, allowing it to move past local minima faster

Task 2: Function approximation -

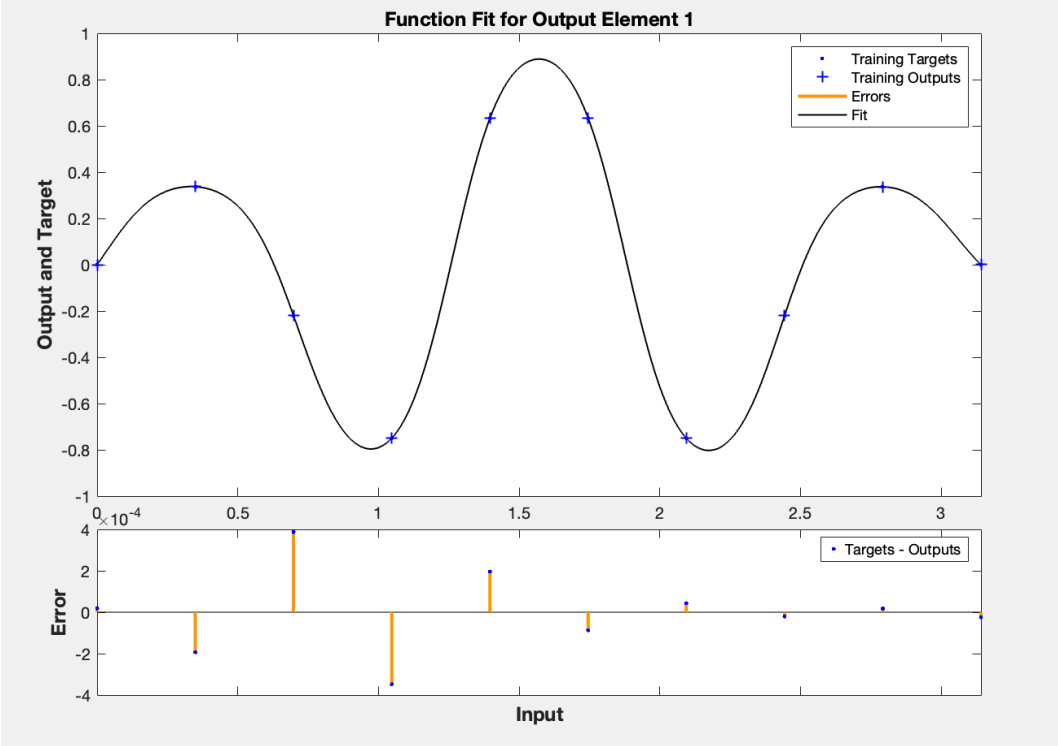
Include figures of the function (as given by the Fit button) implemented by networks with 3, 6, 10 and 20 hidden nodes, after typical training sessions with good parameter values. Make sure it is clear which figure is which, and what training parameters you used. Also include the mse of the trained networks on the training data.

Training parameters -
learning rate = 0.02
Max Epochs = 5000

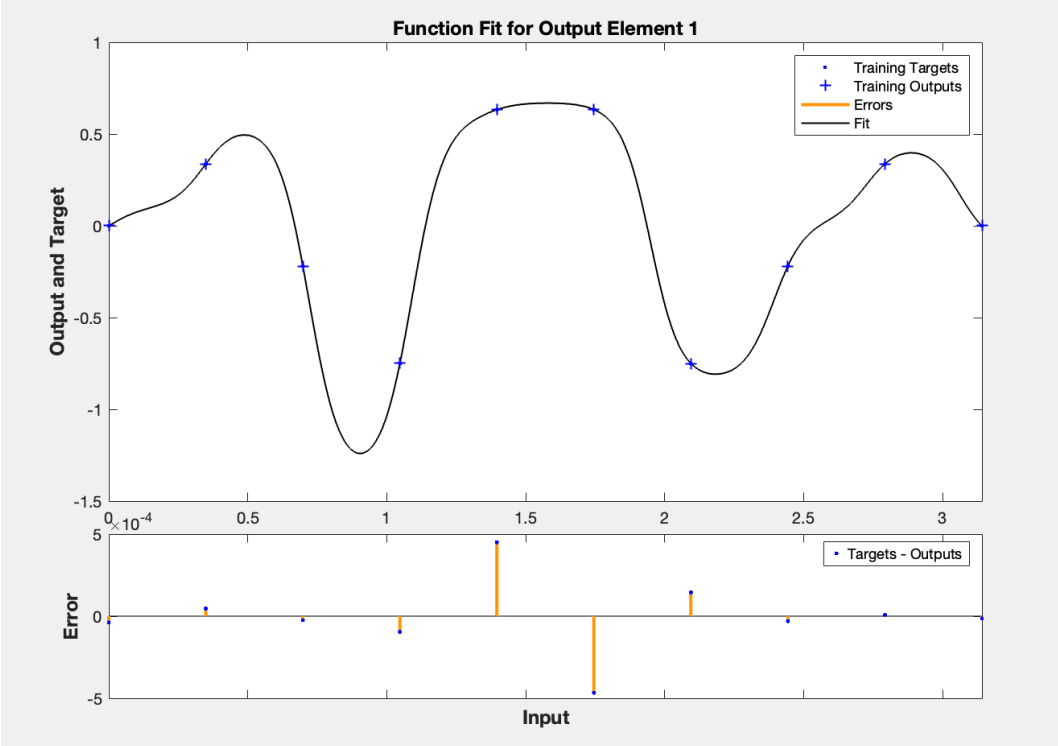
3 Nodes - MSE = 0.191



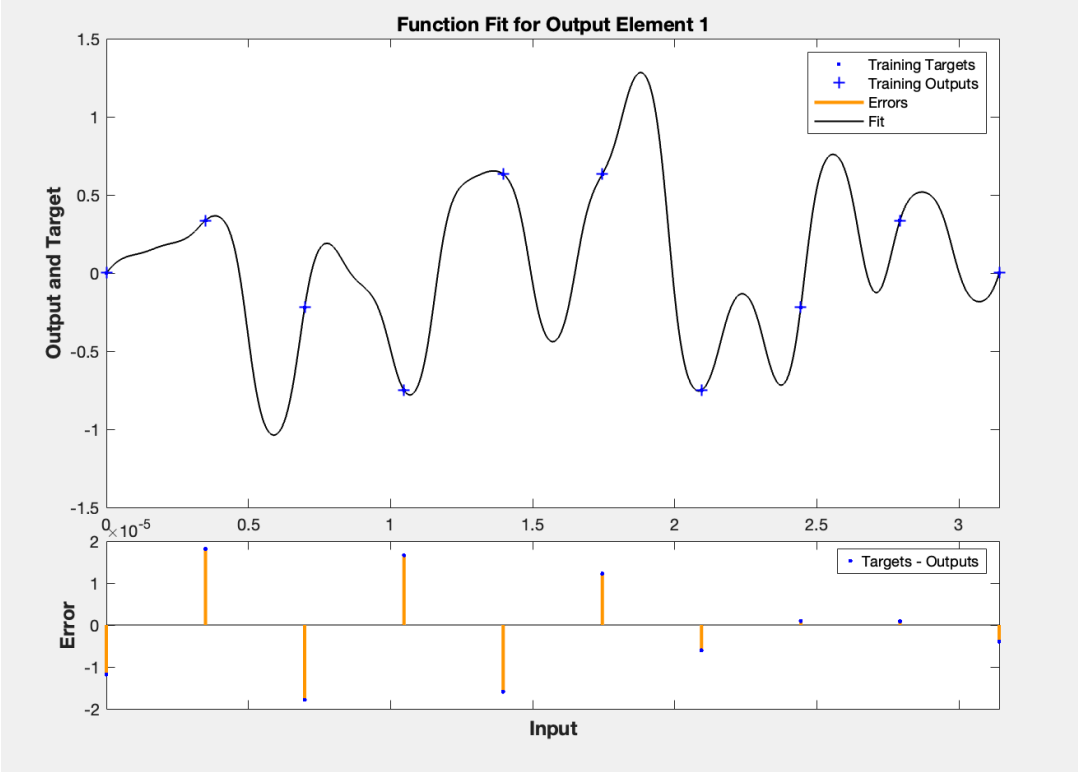
6 Nodes - MSE =2.17e-05



10 Nodes - MSE = 2.04e-07



20 Nodes - MSE = 1.5e-10



Question 6: For what number of hidden nodes do you usually get the lowest training errors (smallest mse)? What seems to be the general relation between the number of hidden nodes and training error?

Ans: 6 nodes.

Hidden Nodes	MSE (Training Error)
3 Nodes	0.191
6 Nodes	2.17× 10^{-5}
10 Nodes	2.04×10^{−7}
20 Nodes	1.5×10^{−10}

The network with 20 hidden nodes achieves the lowest training error but according to the plot, overfitting occurs when it's above 8. More hidden nodes reduce training error, but they can also lead to overfitting, where the network memorizes the training data rather than learning the true function. In this case, 6 is giving the optimal plot after fitting. **As you increase the nodes, the MSE is decreasing exponentially.**

Question 7: What number of hidden nodes usually gives the best approximation

to the function f (x) = sin(x) · sin(5x), in your opinion? Motivate your answer.

Visual Comparison of Function Approximation

- **3 Nodes:**
  - **Poor approximation; the network is underfitting.**
  - **The fit does not capture the oscillations of sin(x) · sin(5x).**
- **6 Nodes:**
  - **Very good approximation; smooth and follows the function well.**
- **10 Nodes:**
  - **Overfitting**
- **20 Nodes:**
  - **Overfitting (capturing noise rather than the true function).**

Question 8 ⋆: Function approximation suffers both when you have few hidden nodes and when you have too many nodes. Explain, respectively, why that is.

Few hidden nodes - The function is unable to generalize to the given data with a small number of hidden nodes and gives high MSE (underfitting - high bias and low variance)

Many hidden nodes - The function generalizes too well and starts capturing noise of the data and gives a very small MSE but overfits to the given data (low bias and high variance)

Question 9: If you knew beforehand that the function to approximate looks something like the one in Figure 2, could you have used that knowledge to make an educated guess at a suitable number of hidden nodes to use (without trial-and-error)? Explain how you would come up with that number. Is this number close to the number that you found to be best by trial-and-error?

Given that we have 10 input data points and the optimal number of nodes is around 5-8, we can create a simple formula to get the number of required nodes as -

$N = D/2 + 1$

Where D are the number of data points.

Question 10: What differences can you observe in the results compared to those achieved with backpropagation? Based on your observations, which of the two training algorithms do you think is most suitable for this problem? Motivate your

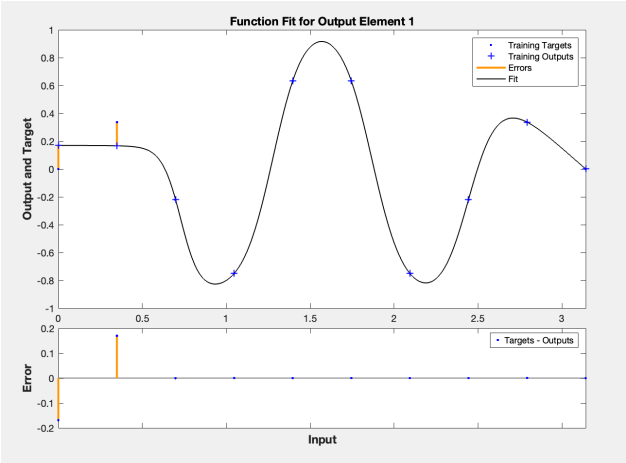
Answer.

MSE is an average measure. So Rprop may have found a smoother global fit that reduces overall error but does not interpolate through the training points as closely as GD.

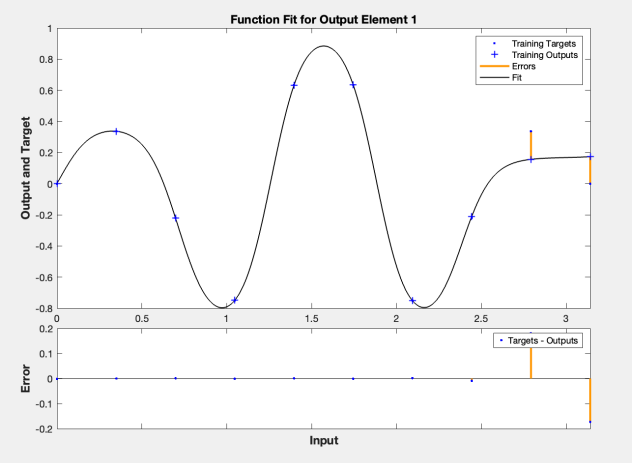
The better point-wise fit in GD might indicate that it is closely following the data points, potentially leading to overfitting.

Rprop, due to its sign-based approach, may not tweak the weights as precisely as GD does for small changes, leading to slightly worse pointwise fit.

Rprop



GD



11. What settings did you use to get good results? What was the smallest number of hidden nodes that you needed? Is this number reasonable with respect to the size of the dataset? Approximately how many percent of the wines are placed in the right class after training this network?

R_PROP:
trainrp, tansig + softmax, crossentropy, 7 hidden nodes, 2000 epochs
The smallest number of hidden nodes is 7.
It gives an accuracy of 98%-100% every time. I guess, the dataset pretty small as we have only 178 samples.

GD:
Traingd, tansig, softmax, mse, 10 hidden nodes, 1000.
The smallest number of hidden nodes is 10.

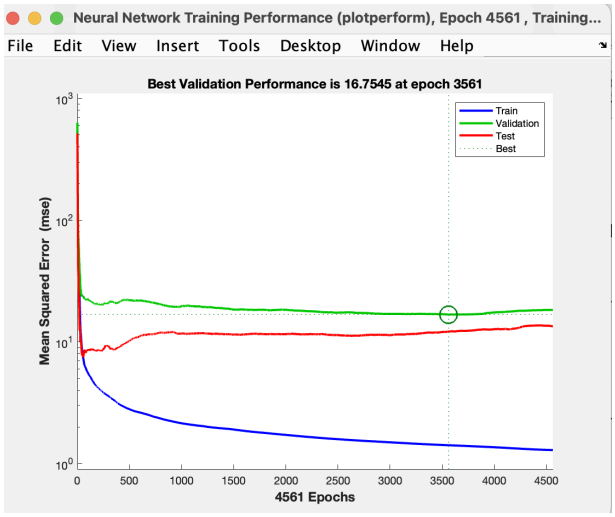
It gives an accuracy of 48%-50% on an average. I guess, the dataset pretty small as we have only 178 samples.

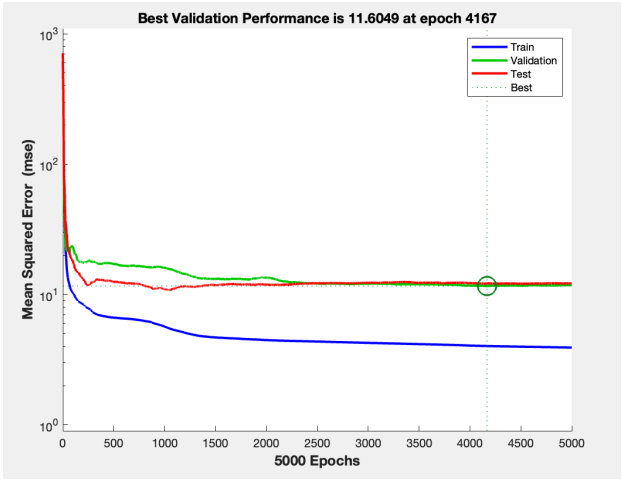
Question 12: Did the normalization have any significant impact on the results of the training? How many hidden nodes do you need now in order to get good results?

Yes, it is improving accuracy to 70-80%. It is working pretty well when the hidden nodes are set to 6 or 7.

Task 4:

Question 13 ⋆: As the training goes along, the three error curves usually behave differently. Describe these trends, and explain why the curves behave in such a way.





Question 13 ⋆: As the training goes along, the three error curves usually behave differently. Describe these trends, and explain why the curves behave in such a way.

Validation error is initially higher than test error because it is still in the training process.
Test error increases with more epochs because the model starts to overfit.
Test error is higher than Training error because it has not seen the data beforehand and the data on which it is testing is completely new. .

Question 14: In light of your answer to the question above, what should one have in mind (in terms of training epochs) when training a neural network?

The epochs should not be too high, which will lead to overfitting (high variance).
It should not be to low, which will lead to underfitting (high bias).
If the test set error and the training set error are very high, then you are probably underfitting.
When the training error gets smaller and smaller, but your test set error starts increasing, you are probably overfitting.