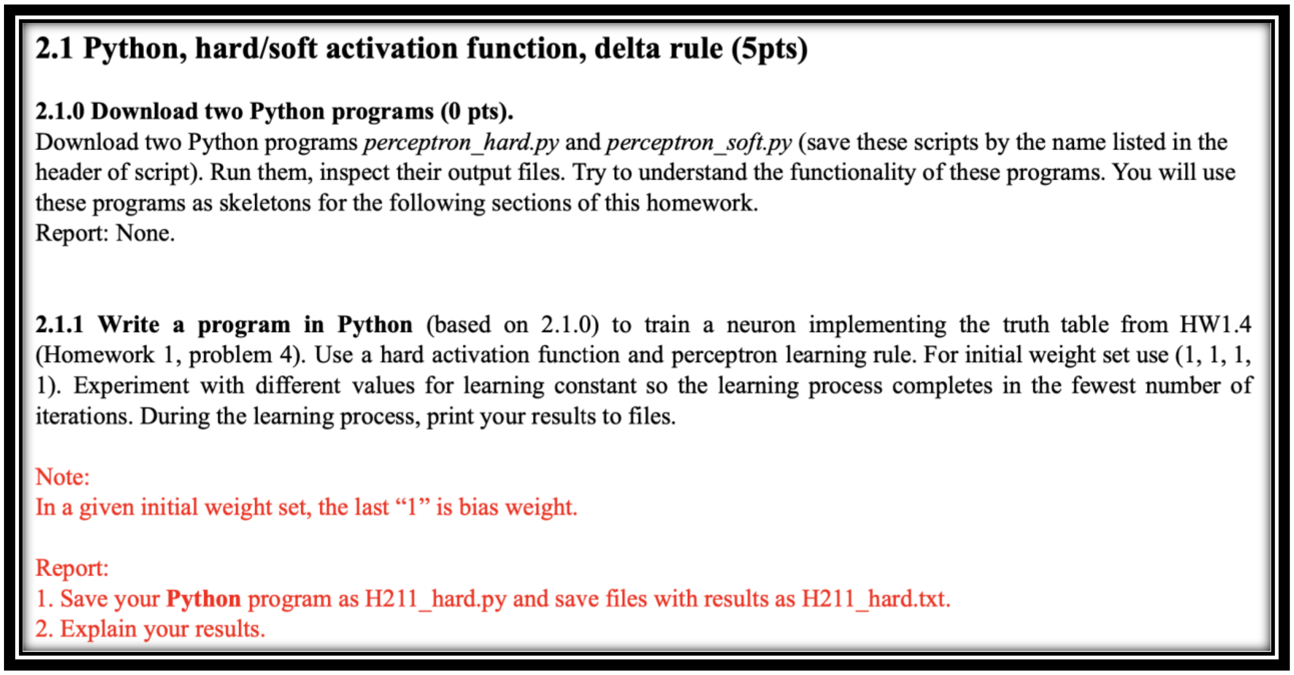
CMSC 636 Neural Nets and Deep Learning
Spring 2022, Instructor: Dr. Milos Manic, *http://www.people.vcu.edu/~mmanic*
Homework 2

Student certification:

Team member 1:
Print Name: Md Touhiduzzaman *Date: 3 March 2022*
I have contributed by doing the following: 2.1.1, 2.1.2, 2.1.3
Signed: Touhid

Team member 2:
Print Name: Maher Al Islam *Date: 3 March 2022*
I have contributed by doing the following: 2.2, 2.5
Signed: Maher

Team member 3:
Print Name: Samah Ahmed *Date: 3 March 2022*
I have contributed by doing the following: 2.3, 2.4
Signed: Samah



2.1 Python, hard/soft activation function, delta rule (5pts)

2.1.0 Download two Python programs (0 pts).
Download two Python programs *perceptron_hard.py* and *perceptron_soft.py* (save these scripts by the name listed in the header of script). Run them, inspect their output files. Try to understand the functionality of these programs. You will use these programs as skeletons for the following sections of this homework.
Report: None.

2.1.1 Write a program in Python (based on 2.1.0) to train a neuron implementing the truth table from HW1.4 (Homework 1, problem 4). Use a hard activation function and perceptron learning rule. For initial weight set use (1, 1, 1, 1). Experiment with different values for learning constant so the learning process completes in the fewest number of iterations. During the learning process, print your results to files.

Note:
In a given initial weight set, the last "1" is bias weight.

Report:
1. Save your **Python** program as H211_hard.py and save files with results as H211_hard.txt.
2. Explain your results.

Answer to 2.1.1:
From the truth table of the previous homework's problem 1.4, we had 3 boolean inputs as A, B & C and so there were 8 patterns as [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0] & [1, ,1, 1]. Additionally, there was a bias input as always 1. So, the designed neuron has 4 inputs and two outputs as binary 0 & 1.
We have defined our hard activation function like the following two ones:

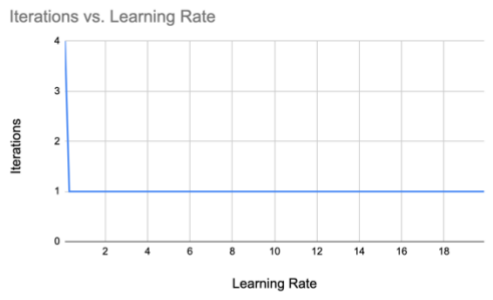
```
def unipolar_hard_activation_function(x):
        return 1 if x > 0 else 0


def bipolar_hard_activation_function(x):
        return 1 if x > 0 else -1
```

Then the provided "*perceptron_hard.py*" script is modified to run for 100 number of learning rates ranging from 0.1 to 19.9 with 0.2 increment at each level, i.e., α = [0.1, 20). The updated code is saved in the "*H211_hard.py*" file, as instructed in the question.

Discussion of Results:

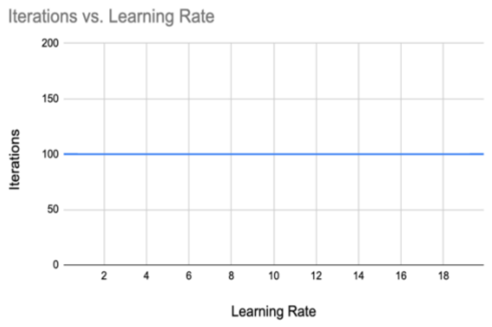
For the unipolar hard activation function, the neuron converged with 0 error after 4 iterations with α=0.1 and after only 1 iteration with all the other higher values of learning rates. The required number of iterations versus the learning rate and a few samples of the results are shown in the table below:

Learning Rate (α)	Iterations	Final Error	Final Weight
0.1	4	0.0	0.4, 0.1, 0.4, -0.7
0.3	1	0.0	0.4, 0.1, 0.4, -0.7
0.5	1	0.0	0.4, 0.1, 0.4, -0.7
1.1	1	0.0	0.4, 0.1, 0.4, -0.7
19.9	1	0.0	0.4, 0.1, 0.4, -0.7



But for the bipolar hard activation function, the neuron never converged to 0 error, even after running for preset max 100 iterations with any of the aforementioned 100 values of the learning rate. A few sample results for this case with the iteration vs learning rate graph are shown below:

Learning Rate (α)	Iterations	Final Error	Final Weight
0.1	100	10.0	0.2, 0.0, 0.2, -0.2
0.3	100	6.00	0.8, 0.0, 0.8, -0.8
0.5	100	10.0	1.8, -1.0, 1.8, -1.8
1.1	100	10.0	4.0, -1.0, 1.8, -1.8
19.9	100	6.0	32.8, 32.0, 42.0, -74.8



So, with an optimal value of learning rate 0.3, we can train the neuron with the fewest number of iterations. which is 1, using unipolar hard activation function.

2.1.2 For the previous problem (problem 2.1.1), change the activation function to soft activation function and repeat your training procedure. Start with a weight set (1,1,1,1). Introduce computation of the total error in your program defined as the sum of squares of errors for each pattern. Experiment with learning coefficients to find results in the fewest number of iterations for the TE<0.01.

Report:
1. Save your Python program as name it H212_soft.py and save files with results H212soft.txt.
2. Explain & discuss your results.

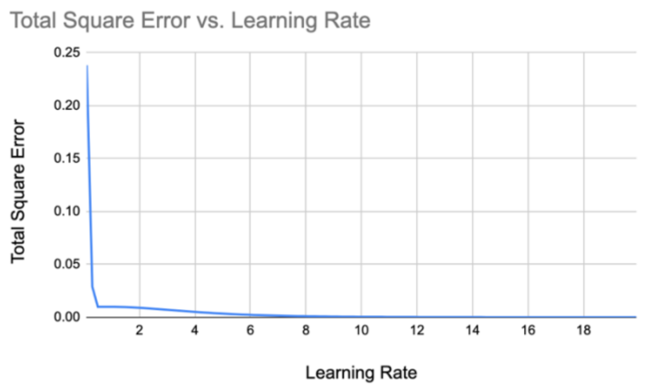
Answer to 2.1.2:

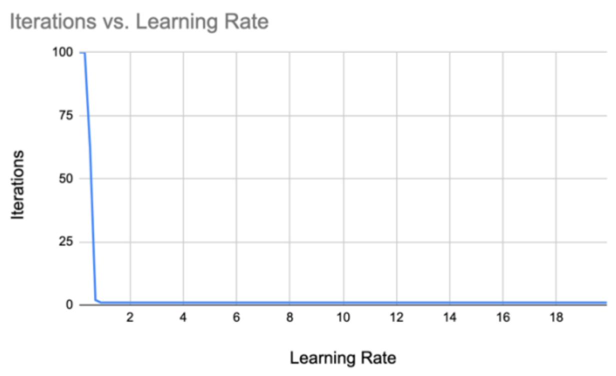
The solution to this question is almost similar to our answer to 2.1.1, but we have only replaced our activation function with the following sigmoid function, which is a soft activation function:

```
def sigmoid(x):
        k = 1
        return 1 / (1 + exp(-1 * k * x))
```

Using the soft activation function, the neuron nearly converges to 0 error after running for preset max 100 iterations with larger values of the learning rate. A few sample results achieving total error less than 0.01 or completing 100 iterations for this case are shown below:

Learning Rate (α)	Iterations	Final Error	Final Weight
0.1	100	0.23827773	2.88, -0.46, 2.85, -4.22
0.3	100	0.02900223	5.3, -0.38, 5.27, -7.85
0.5	62	0.009968878	6.42, -0.36, 6.4, -9.54
1.1	1	0.009921703	6.54, -0.35, 6.51, -9.66
19.9	1	0.000004324	68.3, -2.04, 19.79, -44.05



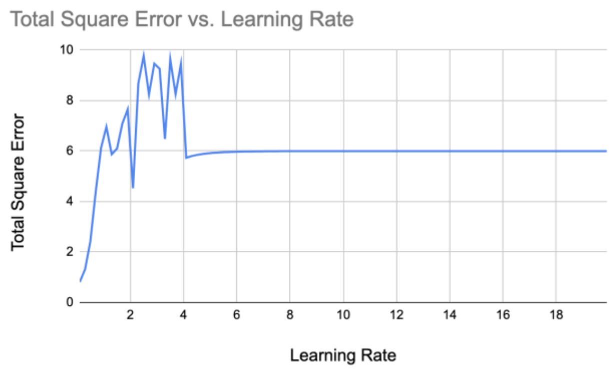


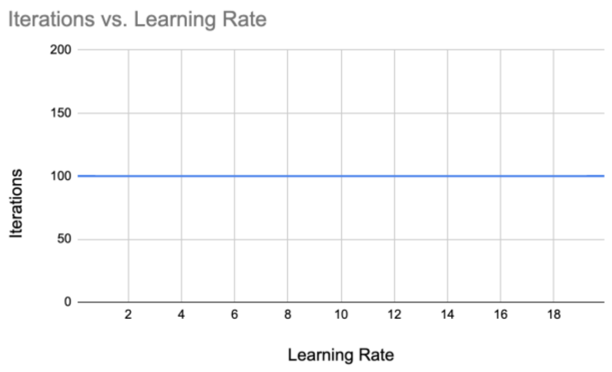
All the outputs of the program are saved inside *"H212soft.txt"* file and the code can be found inside the *"H212_soft.py"* file.

As a separate experiment, we also considered experimenting with bipolar sigmoid function, as shown in the below code block. But the error never converges to 0 for any value of learning rate out of the 100 tested values within [0.1, 20) range, which is also depicted in the following table & graphs.

```
    def sigmoid_bipolar(x):
            k = 1
            return 2 / (1 + exp(-2 * k * x)) - 1
```

Learning Rate (α)	Iterations	Final Error	Final Weight
0.1	100	0.81286948	0.66, 0.0, 0.63, -0.33
0.3	100	1.31937963	0.86, -0.0, 0.74, -0.43
0.5	100	2.4331252	1.24, -0.0, 0.97, -0.62
1.1	100	6.95935419	3.69, 0.6, 2.59, -1.72
19.9	100	6.00000	68.3, -2.04, 19.79, -44.05





2.1.3 For the problem 2.1.1, use soft activation function and modify the script to implement a DELTA training rule. Train the network using initial weights (1,1,1,1). Experiment with the learning constant to produce results in least number of iterations for the TE<0.01.

Report:
1. Save your Python program as H213_delta.py and save files with results as H213_delta.txt.
2. Explain & discuss your results.

Note: please make sure that you read the problem (data) carefully. For example, if the output values are 0 or 1, you should not be using a bipolar activation function.

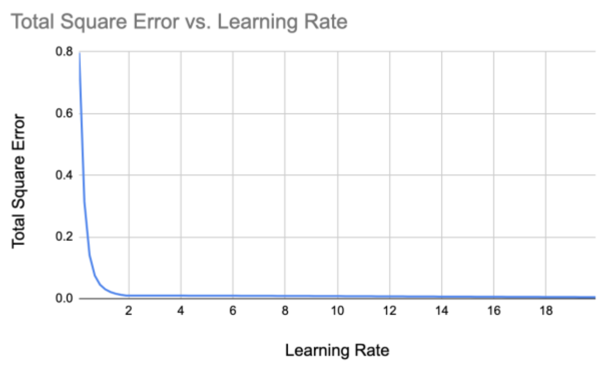
Answer to 2.1.3:

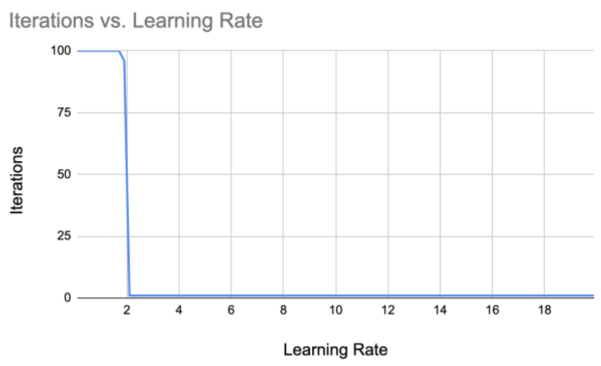
To find out the impact of delta learning in the above solution using soft activation function, we have defined the following unipolar delta & activation (sigmoid) functions:

```
    def sigmoid(x):
            k = 1
            return 1 / (1 + exp(-1 * k * x))

    def delta(x):
            k = 1
            s = sigmoid(x)
            return k * s * (1 - s)
```

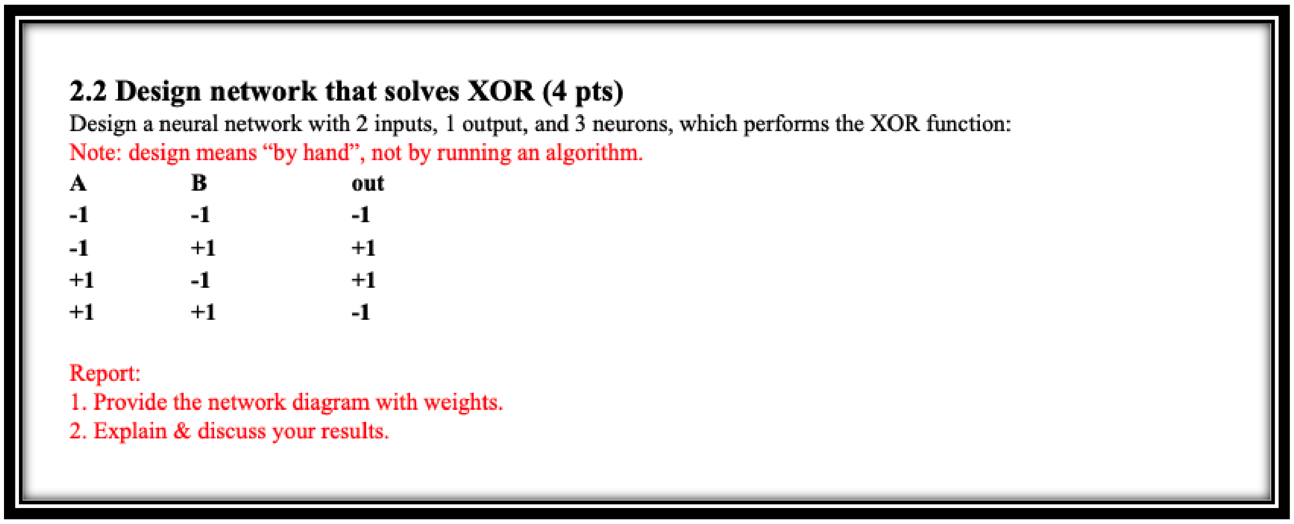
Then we have run our experiment for 100 values of learning rates ranging within [0.1, 20) and found out that lower iterations are required for higher values of the learning rate. A few samples are shown in the following table with respective graphs of total square error and required iterations against the learning rates.

Learning Rate (α)	Iterations	Final Error	Final Weight
0.1	100	0.79768623	1.12, -0.24, 1.11, -1.78
0.3	100	0.31494453	2.46, -0.28, 2.44, -3.62
0.5	100	0.14067711	3.44, -0.24, 3.43, -5.11
1.9	96	0.00998363	6.3, -0.19, 6.3, -9.44
19.9	1	0.0049191	7.1, -0.18, 7.08, -10.58





Therefore, an optimal value of learning constant can be taken as 2.1, which can deliver outputs with TE=0.00997 for only one iteration.
All the outputs of the program are saved inside *"H213_delta.txt"* file and the code can be found inside the *"H213_delta.py"* file.



2.2 Design network that solves XOR (4 pts)

Design a neural network with 2 inputs, 1 output, and 3 neurons, which performs the XOR function:

Note: design means "by hand", not by running an algorithm.

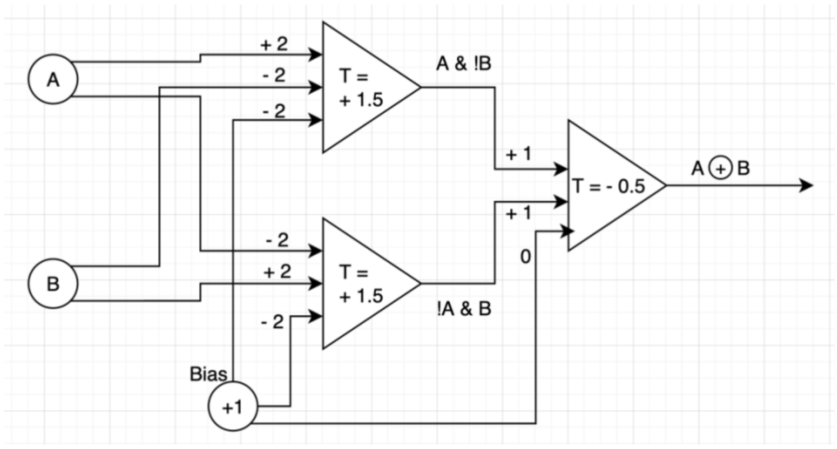
A	**B**	**out**
-1	**-1**	**-1**
-1	**+1**	**+1**
+1	**-1**	**+1**
+1	**+1**	**-1**

Report:
1. Provide the network diagram with weights.
2. Explain & discuss your results.

Answer to 2.2:

From the given truth table, it is seen that the given XOR operation is of bipolar in nature.
For boolean operation, we know that, if there are two inputs A & B, then the XOR operation can be stated as: A ⊕ B = (A & (!B)) + ((!A) & B)

So, we can have 2 AND neurons having direct and inverted inputs, i.e., with positive & negative weights, from A & B and finally an OR neuron, combining the outputs of the 2 AND neurons, can give the XOR output.
The design is shown below with respective weights and thresholds(T):



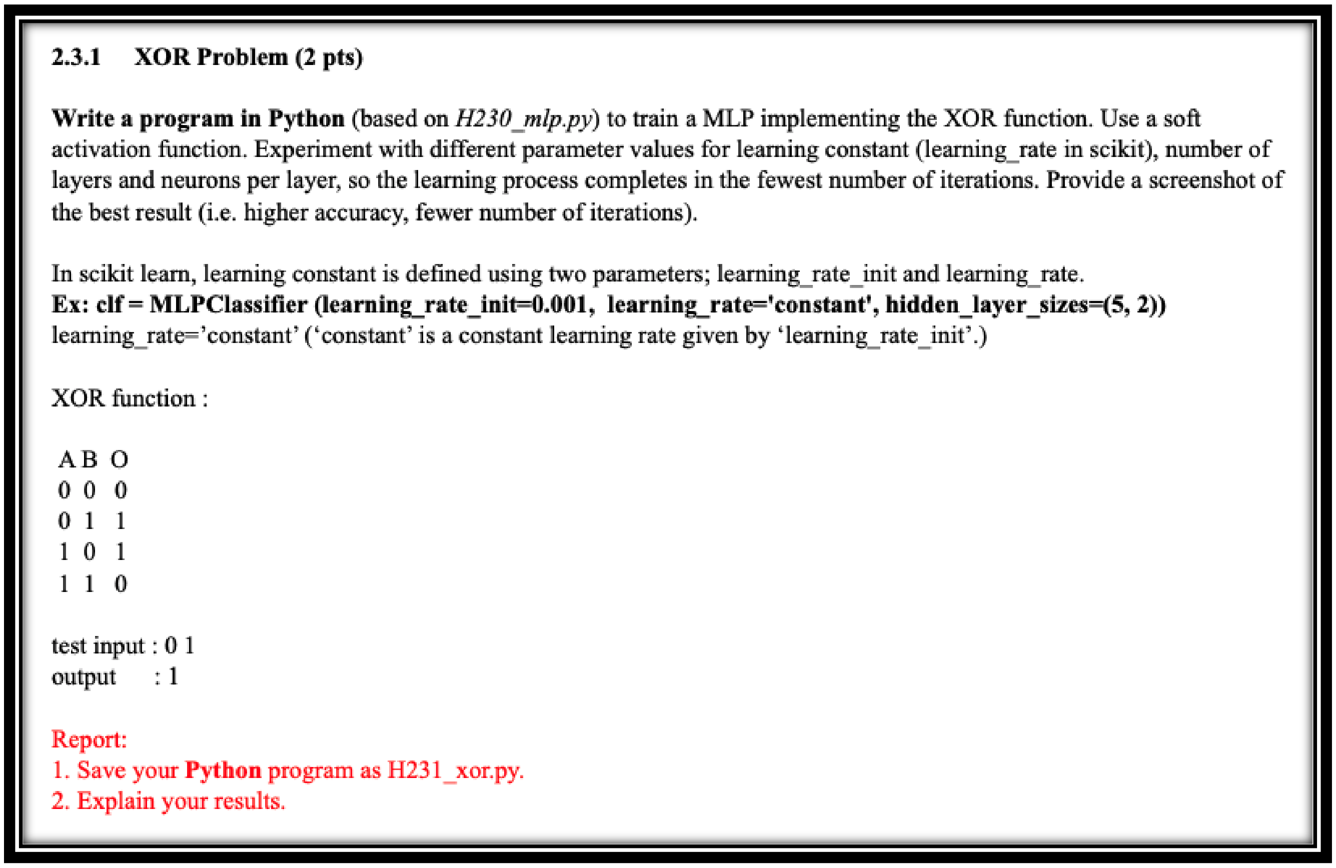
All the given 4 cases are discussed in details below for the above designed neural network.
<u>First Case:</u>
For A=-1 & B=-1, the first neuron output is: -1*2 + -1*-2 - 2= -2 < T
& the second neuron output is: -1*-2+ -1*2 - 2 = -2 < T
So, the first and second neuron do not fire.
The third neuron output gets to be: -1*1 + -1*1 = -2 < T.
Thereby, the final output neuron does not fire and so the output becomes -1.

<u>Second Case:</u>
For A=-1 & B=+1, the first neuron output is: -1*2 + 1*-2 -2 = -6 < T
& the second neuron output is: -1*-2 + 1*2 - 2 = +2 > T
So, the first neuron does not fire, but the second one does.
The third neuron output gets to be: -1 + 1 = 0 > T. ⊕
Thereby, the final output neuron fires and so the output become +1.

<u>Third Case:</u>
For A=+1 & B=-1, the first neuron output is: 1*2 + -1*-2 - 2 = +2 > T
& the second neuron output is: 1*-2 + -1*2 - 2 = - 6 < T
So, the first neuron fires, but the second one does not fire.
The third neuron output gets to be: 1 + (-1) = 0 > T.
Thereby, the final output neuron fires and so the output become +1.

<u>Fourth Case:</u>
For A=+1 & B=+1, the first neuron output is: 1*2 + 1*-2 - 2 = -2 < T
& the second neuron output is: 1*-2 + 1*2 - 2 = -2 < T
So, the first and second neuron do not fire.
The third neuron output gets to be: (-1) + (-1) = -2 < T.
Thereby, the final output neuron does not fire and so the output becomes -1.

So, the given bipolar XOR gate is properly handled by the designed neural network.



2.3.1 XOR Problem (2 pts)

Write a program in Python (based on *H230_mlp.py*) to train a MLP implementing the XOR function. Use a soft activation function. Experiment with different parameter values for learning constant (learning_rate in scikit), number of layers and neurons per layer, so the learning process completes in the fewest number of iterations. Provide a screenshot of the best result (i.e. higher accuracy, fewer number of iterations).

In scikit learn, learning constant is defined using two parameters; learning_rate_init and learning_rate.
Ex: clf = MLPClassifier (learning_rate_init=0.001, learning_rate='constant', hidden_layer_sizes=(5, 2))
learning_rate='constant' ('constant' is a constant learning rate given by 'learning_rate_init'.)

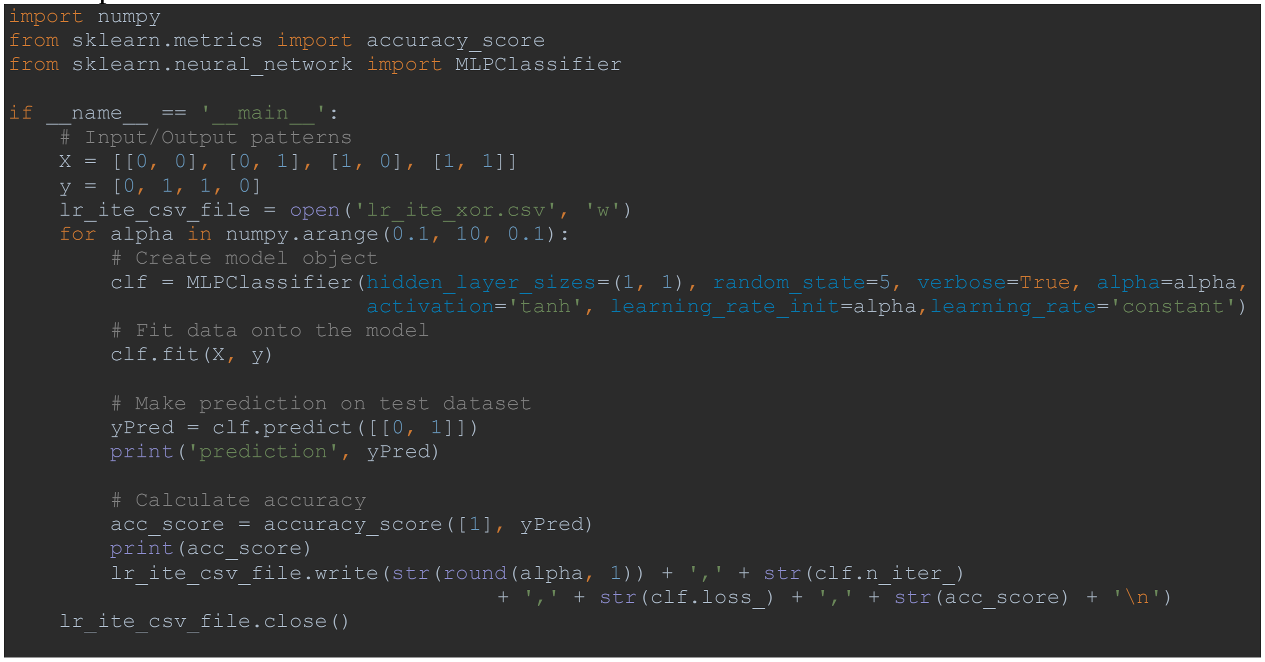
XOR function :

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

test input : 0 1
output : 1

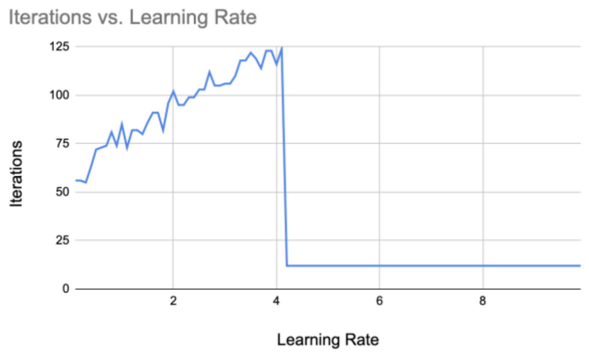
Report:
1. Save your **Python** program as H231_xor.py.
2. Explain your results.

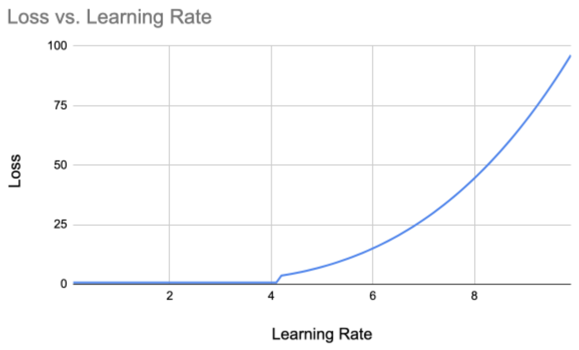
Answer to 2.3.1:
We have modified the given H230_mlp.py to generate H231_xor.py Python script, which delivers the XOR neural network with several hidden layers, node &, learning constants using the hyperbolic-tan as a soft activation function.
The implemented code is:



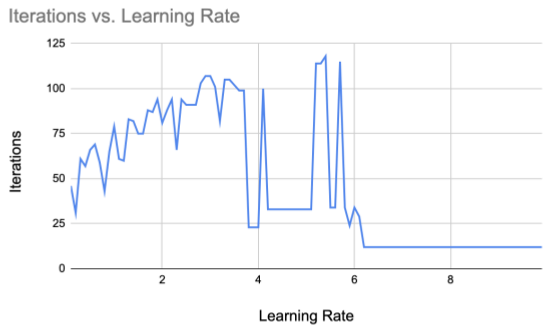
```
import numpy
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier

if __name__ == '__main__':
    # Input/Output patterns
    X = [[0, 0], [0, 1], [1, 0], [1, 1]]
    y = [0, 1, 1, 0]
    lr_ite_csv_file = open('lr_ite_xor.csv', 'w')
    for alpha in numpy.arange(0.1, 10, 0.1):
        # Create model object
        clf = MLPClassifier(hidden_layer_sizes=(1, 1), random_state=5, verbose=True, alpha=alpha,
                            activation='tanh', learning_rate_init=alpha,learning_rate='constant')
        # Fit data onto the model
        clf.fit(X, y)

        # Make prediction on test dataset
        yPred = clf.predict([[0, 1]])
        print('prediction', yPred)

        # Calculate accuracy
        acc_score = accuracy_score([1], yPred)
        print(acc_score)
        lr_ite_csv_file.write(str(round(alpha, 1)) + ',' + str(clf.n_iter_)
                              + ',' + str(clf.loss_) + ',' + str(acc_score) + '\n')
    lr_ite_csv_file.close()
```

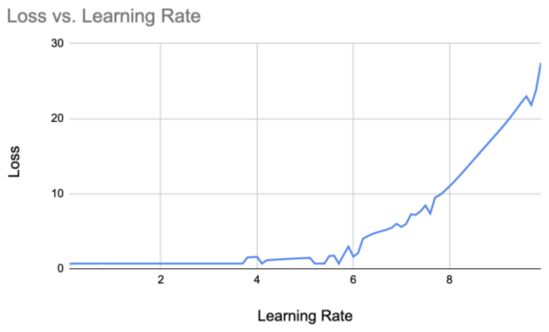
For a neural network setup with 2-nodes in each of 2 hidden-layers, we have achieved the following iteration and loss curves against the learning rates in range [0.1, 10):



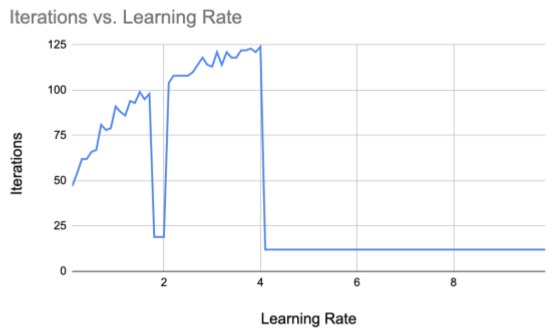


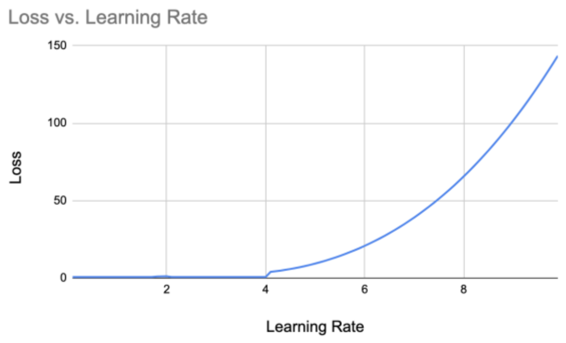
Similarly, for 1 node in a single hidden layer, the following graphs are obtained:



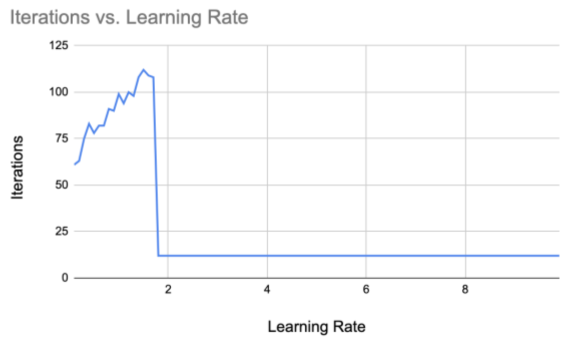


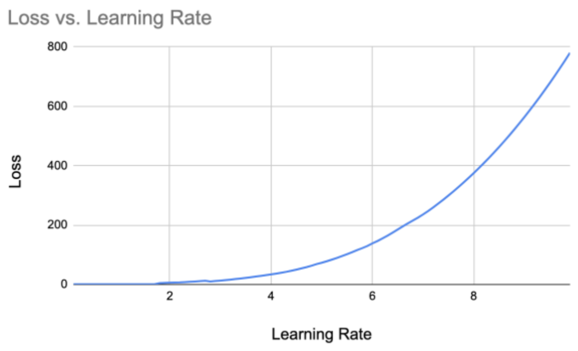
The following graphs are obtained using 2 nodes in each of 3 hidden layers:



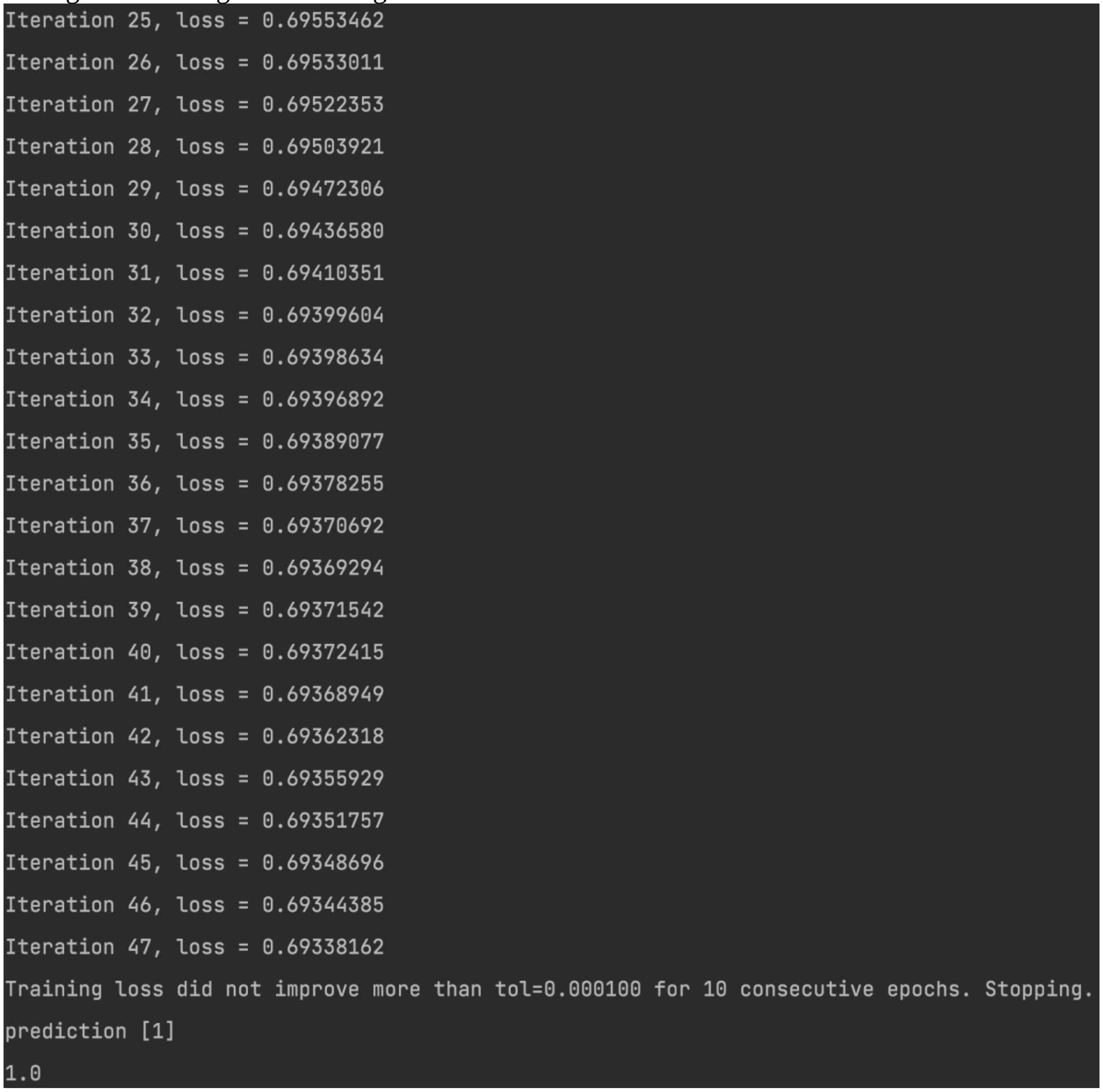


Finally, the following graphs are found using 5 nodes in each of 3 hidden layers:





In general, it is seen that **lower** values of the **learning** rate generate **less loss** and **less nodes** in the hidden layers require **less iterations** to let the network converge on some optimal state. Comparing all the run experiments, we can consider the setup of 2-nodes in each of the 3 hidden layers with learning constant 0.1 to be a good candidate. A screenshot of that specific run's ending situation is given in the figure below:



```
Iteration 25, loss = 0.69553462
Iteration 26, loss = 0.69533011
Iteration 27, loss = 0.69522353
Iteration 28, loss = 0.69503921
Iteration 29, loss = 0.69472306
Iteration 30, loss = 0.69436580
Iteration 31, loss = 0.69410351
Iteration 32, loss = 0.69399604
Iteration 33, loss = 0.69398634
Iteration 34, loss = 0.69396892
Iteration 35, loss = 0.69389077
Iteration 36, loss = 0.69378255
Iteration 37, loss = 0.69370692
Iteration 38, loss = 0.69369294
Iteration 39, loss = 0.69371542
Iteration 40, loss = 0.69372415
Iteration 41, loss = 0.69368949
Iteration 42, loss = 0.69362318
Iteration 43, loss = 0.69355929
Iteration 44, loss = 0.69351757
Iteration 45, loss = 0.69348696
Iteration 46, loss = 0.69344385
Iteration 47, loss = 0.69338162
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
prediction [1]
1.0
```

2.4 (4 pts)

2.4.1 Write a program in Python (based on *H230_mlp.py*) to train a MLP implementing the truth table from HW1.4 (homework 1, problem 4). Use a soft activation function. Experiment with different parameter values for learning constant (learning_rate in scikit), number of layers and neurons per layer, so the learning process completes in the fewest number of iterations. Provide a screenshot of the best result (i.e. higher accuracy, fewer number of iterations).

In scikit learn, learning constant is defined using two parameters; learning_rate_init and learning_rate.
Ex: clf = MLPClassifier (learning_rate_init=0.001, learning_rate='constant', hidden_layer_sizes=(5, 2))
learning_rate='constant' ('constant' is a constant learning rate given by 'learning_rate_init'.)

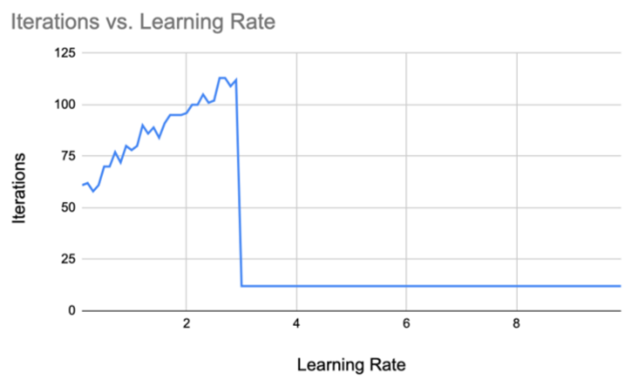
Report:
1. Save your **Python** program as H241.py.
2. Explain your results.

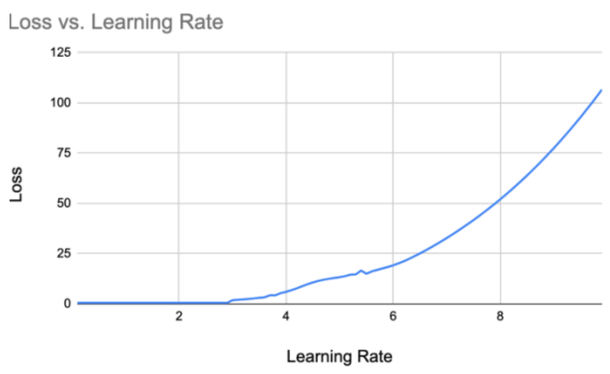
<u>Answer to 2.4.1:</u>

The required code implementation is like the following:

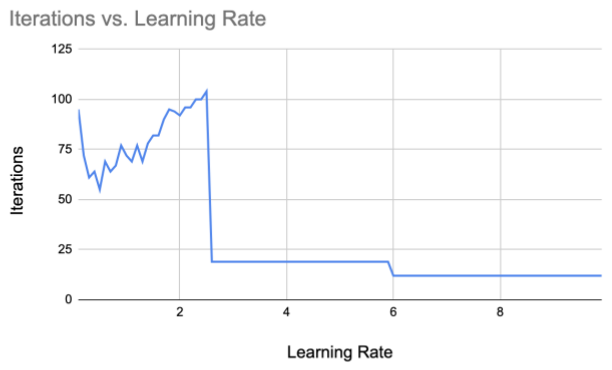
```python
import numpy
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier

if __name__ == '__main__':
    # Input/Output patterns
    X = [  # Patterns as a 2-dimensional list
        [0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 0, 1], [0, 1, 1, 1],  # Each item = [A, B, C, Bias]
        [1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 1]]
    y = [0, 0, 0, 0, 0, 1, 0, 1]

    lr_ite_csv_file = open('lr_ite_hw14_222.csv', 'w')
    for alpha in numpy.arange(0.1, 10, 0.1):
        # Create model object
        clf = MLPClassifier(hidden_layer_sizes=(2, 2, 2), random_state=5,verbose=True,
          alpha=alpha,activation='tanh', learning_rate_init=alpha, learning_rate='constant')
        # Fit data onto the model
        clf.fit(X, y)

        # Make prediction on test dataset
        yPred = clf.predict([[1, 0, 1, 1], [1, 1, 0, 1], [0, 1, 0, 1], [1, 1, 1, 1]])
        print('prediction', yPred)

        # Calculate accuracy
        acc_score = accuracy_score([1, 0, 0, 1], yPred)
        print(acc_score)
        lr_ite_csv_file.write(str(round(alpha, 1)) + ',' + str(clf.n_iter_) + ','
                  + str(clf.loss_) + ',' + str(acc_score) + '\n')
    lr_ite_csv_file.close()
```

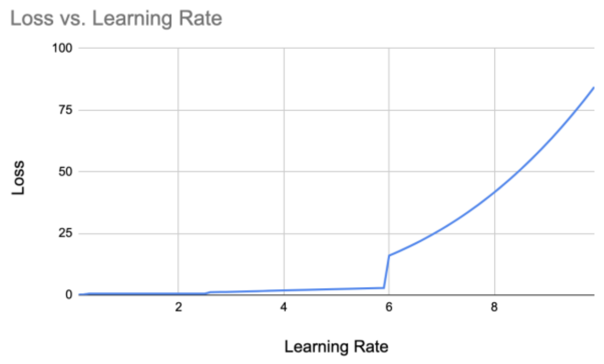
For a neural network setup with 2-nodes in each of 3 hidden-layers, we have achieved the following iteration and loss curves against the learning rates in range [0.1, 10):



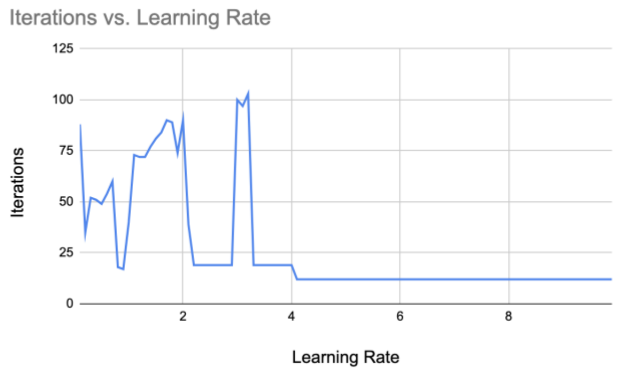


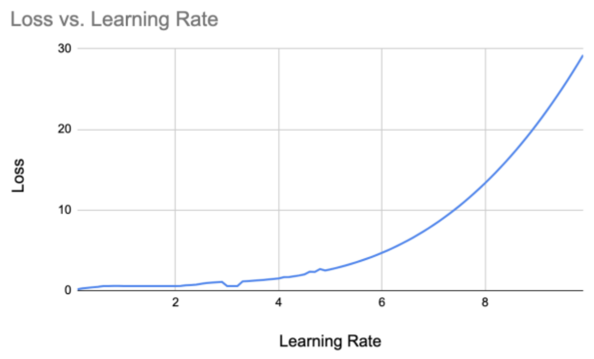
The following graphs are obtained using 2 nodes in each of 2 hidden layers:



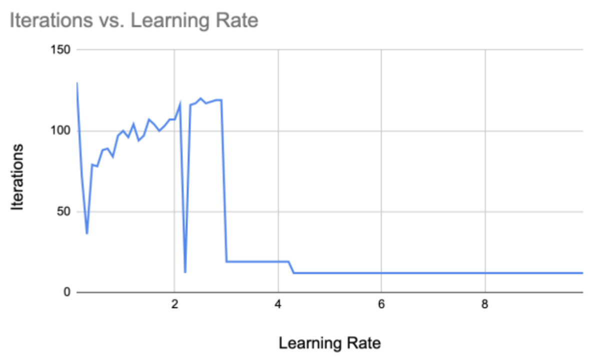


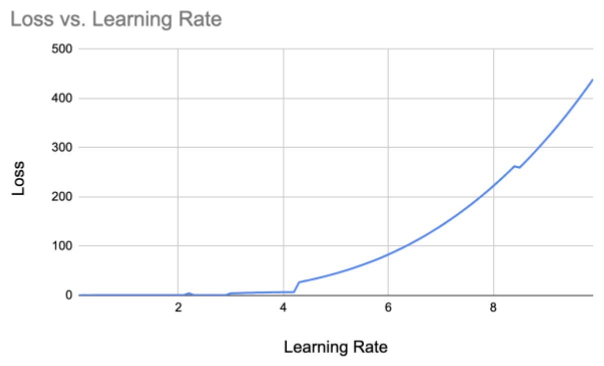
Similarly, for a single node in just a single hidden layer, we get the following graphs:



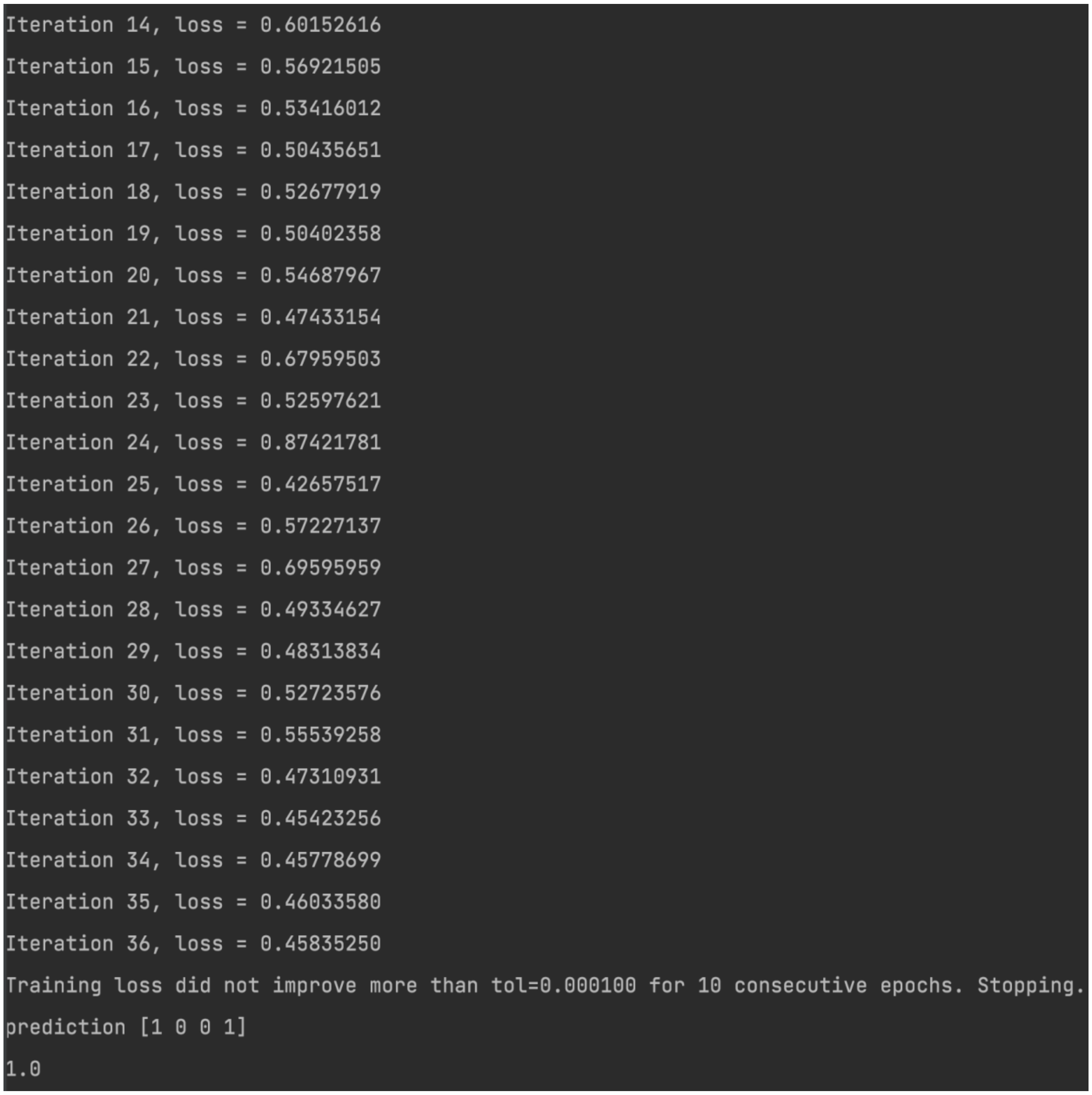


As a final setup, we got the following graphs by running the experiment on a neural network with 5 nodes each in 3 hidden layers:

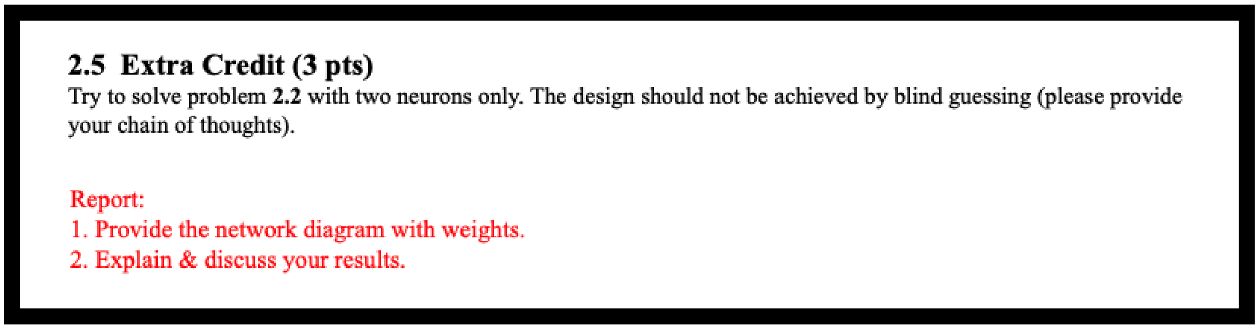




For the 1node-1layer setup, the required number of iterations is seen much lower, but the loss is comparatively higher for this setup. The least amount of loss is seen for 2node-3layer and 5node-3layer setups, but the latter one required much lower iterations to converge. So, we can take 5-node and 3 layers setup with learning rate as 0.3 as an optimal setup, which converged within 36 iterations of our program.
A screenshot of this specific run is given below:



```
Iteration 14, loss = 0.60152616
Iteration 15, loss = 0.56921505
Iteration 16, loss = 0.53416012
Iteration 17, loss = 0.50435651
Iteration 18, loss = 0.52677919
Iteration 19, loss = 0.50402358
Iteration 20, loss = 0.54687967
Iteration 21, loss = 0.47433154
Iteration 22, loss = 0.67959503
Iteration 23, loss = 0.52597621
Iteration 24, loss = 0.87421781
Iteration 25, loss = 0.42657517
Iteration 26, loss = 0.57227137
Iteration 27, loss = 0.69595959
Iteration 28, loss = 0.49334627
Iteration 29, loss = 0.48313834
Iteration 30, loss = 0.52723576
Iteration 31, loss = 0.55539258
Iteration 32, loss = 0.47310931
Iteration 33, loss = 0.45423256
Iteration 34, loss = 0.45778699
Iteration 35, loss = 0.46033580
Iteration 36, loss = 0.45835250
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
prediction [1 0 0 1]
1.0
```



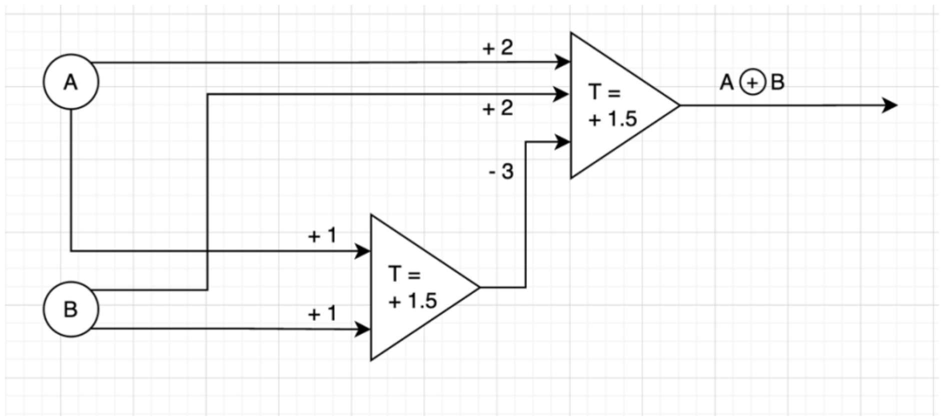
2.5 Extra Credit (3 pts)

Try to solve problem **2.2** with two neurons only. The design should not be achieved by blind guessing (please provide your chain of thoughts).

Report:
1. Provide the network diagram with weights.
2. Explain & discuss your results.

Answer to 2.5:
From the truth table and 3-neuron solution of problem number 2.2 above, we can estimate some rubrics about the weights and thresholds of a 2-neuron XOR gate, like – the 2-variable XOR operation can be rewritten as, $A \oplus B = (A + B) - (A . B)$.
Now, if we design an AND gate feeding largely negative weighted input to another OR gate for the same set of inputs, then the above negation operation can be performed using 2 neurons.

The neural network design, with respective weights and thresholds, can shown as the following:



All the 4 possible cases of XOR operation of 2 -variables are discussed in details below for the above designed neural network.
<u>**First Case:**</u>
For A=-1 & B=-1, the first neuron output is: -1*1 + -1*1= -2 < T
& the second neuron output is: -1*2+ -1*2 – 3 * -1 = -1 < T
Thereby, the final output neuron does not fire and so the output becomes -1.

<u>**Second Case:**</u>
For A=-1 & B=+1, the first neuron output is: -1*1 + 1*1 = 0 < T
& the second neuron output is: -1*2 + 1*2 – 3 * -1 = +3 > T
So, the second neuron fires, while the first one does not fire.
Thereby, the final output neuron fires and so the output become +1.

<u>**Third Case:**</u>
For A=+1 & B=-1, the first neuron output is: 1*1 + -1*1 = 0 < T

& the second neuron output is: 1*2 + -1*2 – 3 * -1 = + 3 > T
So, the second neuron fires, while the first one does not fire.
Thereby, the final output neuron fires and so the output become +1.

Fourth Case:
For A=+1 & B=+1, the first neuron output is: 1*1 + 1*1 = +2 > T
& the second neuron output is: 1*2 + 1*2 – 3 * 1 = +1 < T
So, the first neuron fires and the second neuron does not fire.
Thereby, the final output neuron does not fire and so the output becomes -1.

So, the XOR gate is properly handled by the designed neural network with only 2 neurons.

~~~~~~ [The End] ~~~~~~
~~~~~~