Note: I did not change Prof. Parker's original perceptron file. I am not reuploading it again.

If you run the ***train-test-logic.ss*** file, you can see that it tries to train *OR*, *AND*, and *XOR* for each learning rate (10-100-1000) and generation (1e3-1e4-1e5-1e6) combination. I am also attaching the output of this training session as the ***logic-training-session.txt*** file.

Part 1: Copy perceptron.ss and perceptron-input. The program contains a single perceptron that uses a sigmoid function to determine output and the delta rule to do learning. The file perceptron-input defines a training set to learn OR. Try it out over 1000 training generations. Write a training list for AND and attempt one for XOR. If you can't get XOR, explain what might be the problem.

My training setup was able to train the perceptron for OR in 1000 generations with epsilon value of 10.

On the other hand, it was not able to train the perceptron for AND in 1000 generations with an epsilon value of 10. When I increased the training length to 10000 generations, it was able to train the perceptron for AND.

In any of my training combinations (with learning rate and epsilon) my perceptron was not able to learn to give the correct set of outputs for XOR. I think that the main reason behind this lies down in how a single perceptron works. A single perceptron is able to classify linearly separable problems like AND and OR. On the other hand, by the nature of XOR, it is not possible to separate it using a linear function. Similarly, from last week's homework, we can see that our XOR and NOT XOR neural networks had multiple layers (3 perceptrons in 2-1 alignment) rather than a single neuron structure we had for AND or OR.

Part 2: Go back to doing OR. Change epsilon, the learning rate to 10, 100, and 1000. What do you observe and why?

As you can see in my ***logic-results.txt*** file, when I keep the generation count the same and increase the learning rate, the program was not able to learn to give the correct output for OR.

In the generation count 1000, the program was able to learn OR only with the learning rate of 10. The other learning rates were not able to be trained after 1000 generations.

After the generation count 1000, for each generation count (1e4, 1e5, 1e6) the program was able to learn to give the correct output for epsilon values of 10 and 100. In any of my trials, the program was not able to fully learn OR perceptron with a learning rate of 1000.

I think that the main reason for this is, the program punishes itself significantly when there is an error. Because the magnitude of this punishment is very significant (the learning rate or the epsilon value), the program can never correctly adjust for the correct weights. I think that if we looked further into this, we could see that the perceptron weights are always around the correct range but cannot be exactly correct considering the magnitude of the learning rate.

Part 3: Write a training set (perceptron-input file) to learn passability based on 2 continuous inputs (stability and height). Let's assume to be passable the stability needs to be greater than .9 and the height needs to be less than .5. How does it do learning the concept? Explain.

If you run the ***train-test-passability.ss*** file, you can see that it tries to train using the training set I created that has 25 passable and 25 non-passable examples. I experimented with the learning rate (10-100-1000) and generation count (1e3-1e4-1e5-1e6) for each combination. I am also attaching the output of this training session as the ***passability-training-session.txt*** file. At the end of each training session, you can also see the ratio of the correct outputs. We can summarize the results as following:

- When the generation count was 1000 and the learning rate was 10, the perceptron was able to have an accuracy of 90%.
- When the generation count was 10000 and the learning rate was 10, the perceptron was able to have an accuracy of 96%.
- When the generation count was 100000 and the learning rate was 10, the perceptron was able to have an accuracy of 100%.
- When the generation count was 1000000 and the learning rate was 10, the perceptron was able to have an accuracy of 100%.
- In the other learning combinations, the perceptron had an accuracy of 50% which is basically random so it was not able to learn something enough to even have a slight accuracy. I think that, considering the conclusion from ***Part 2***, we can say that the magnitude of the learning rate significantly affects how we learn more specific networks where we have to be more precise.

I think that the learning process is the same for our logic functions. The main difference is, as we have a large training set, it takes longer for our network to fully

learn the training-set, ensuring that we have the correct output for each training-sample. In general, the learning structure is similar to our basic feed-forward, back-propagation network structure in which we get feedback each time we get an output. We update our weights and biases depending on the magnitude of our feedback and also the learning rate. As I said before, as this problem has more examples in the training set, the magnitude of the learning rate plays a crucial role in this learning process.

Part 4: Discuss how NNs with learning can be used to help us in our search problem.

I think that a well trained Neural Network with a large and detailed enough input size, can be extremely useful in our search problem. In the best case scenario, we can input the full board state and try to get a move from the neural network. I think that, in this case, we would have to train the network for a long time to be able to have a network with a comparable enough result to MCTS.

As an alternative, we can have a custom input space where we input our location and also the location of the goal. In this case, if the goal is not moving, a well trained neural network can perform well enough. As an extra, we can input the obstacle density of our 4-directions, we can make more future-proof decisions as our robot will both consider the distance from the goal and also the areas with less obstacles to be able to not get stuck in a dead end where it just tires to get close to the goal. If the goal is moving, we can input even more stuff like the obstacle density of the robot's surroundings, its distance from the edges, etc. These will all work when we are the goal and the opponent is the robot.