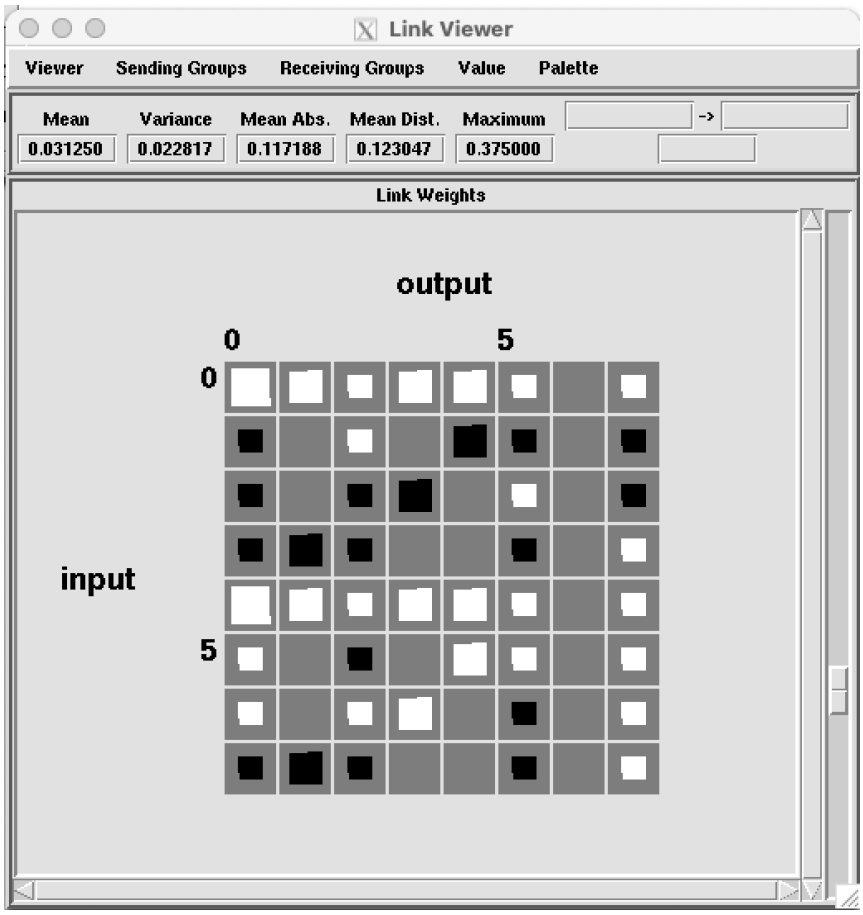
Q1 [10 pts.] Explain why the weight from input:4 to output:1 is equal to 0.25, and why the weight from input:3 to output:2 is equal to -0.125. To do this, you will have to consider the Hebb rule equation, the training patterns, the learning rate, and the fact that, as will be discussed in class, Lens applies an extra factor of 2.0 to the weight changes (from the derivative of the error function). [Note that units are numbered from 0, so, for instance, input:4 is the 5th input unit.]

The Hebb rule equation is delta wij = epsilon x ai aj. Where, e is the learning rate (0.0625), ai is activation for the input unit, aj is the activation of the output unit. Lens applies an extra factor of 2.0 to the weight changes due to the derivative of the error function so delta wij = 2 x epsilon x ai aj

Input 4 to output 1: 2 x 0.0625 x (1 x 1 + 1 x 1 + 1 x 0) = 0.25
Input 3 to output 2: 2 x 0.0625 x (-1 x 0 + -1 x 1 + 1 x 0) = -0.125



Q2 [10 pts.] If, at this point, you apply the Delta rule by clicking on "Train Network", the weights remain unchanged. Why? What would have happened if the Hebb rule had been applied a second time instead?

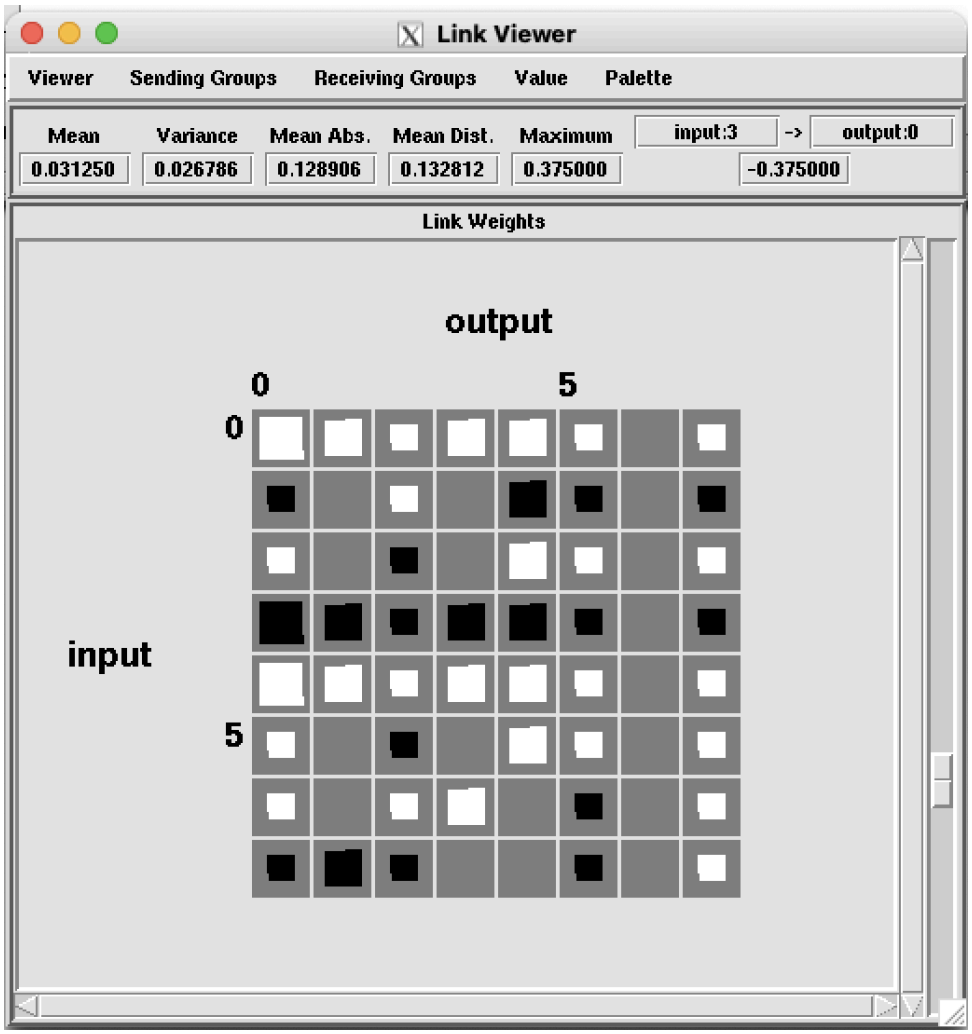
If you apply the Delta rule the weights remain unchanged because after the first pass, since the training patterns are orthogonal the Delta rule already sets the weights correctly. So for each pattern, the network's output activations aj equal to the target activations tj. The delta rule equation is delta wij = epsilon(tj - aj)ai, it adjusts weights based on the error term (tj−aj), so if tj−aj = 0 for all patterns after the first pass, delta wij = 0 meaning there are no weight changes on the second pass. Thus, the target and actual result are the same, therefore there are no interference when you train the network; since it incorporates feedback from the current weights, allowing it to converge accurately when tj equals aj. If the Hebb rule had been applied a second time instead, it wouldn't consider the current state of the weights. It would recalculate weight changes only based on the correlations between inputs and targets. As the Hebb rule equation is, delta wij = epsilon x ai aj. Thus, the same weight changes would be applied a second time which in turn will double the weights. This is because the hebb rule depends on correlations between input and output and it doesn't have the corrective feedback mechanism.

Q3 [20 pts.] Describe and explain the similarities and differences among the weights produced by the Hebb rule (`hebb-li.wt`) and those produced by the Delta rule (`delta-li.wt`) when training on the linearly independent set. (Hint: It will help to examine the actual training patterns, keeping in mind that linearly independent input patterns can have some similarity among them as long as there is something unique about each of them.)

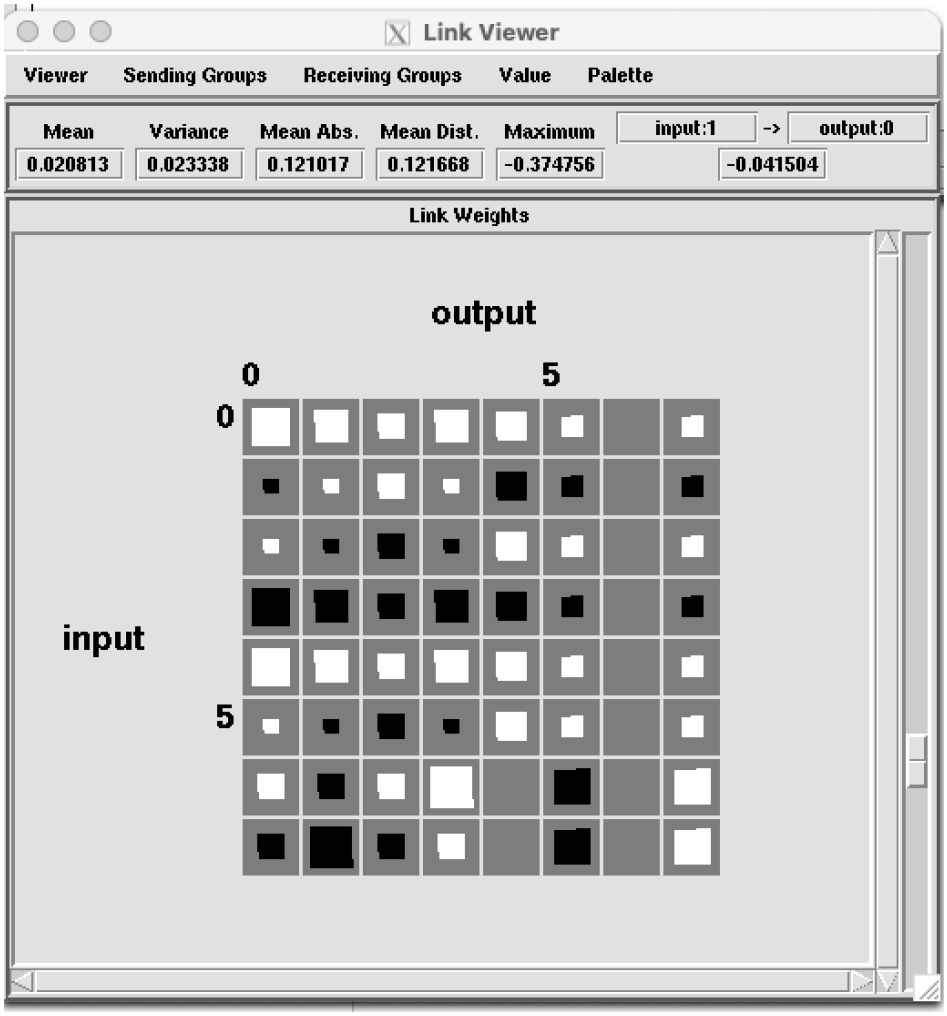
Based on the actual training patterns, patterns a and c share similarities mostly but their inputs for 6 and 7 are different and their targets for 1, 3, 5, and 7 are different. Pattern b seems different from both pattern a and c. The weights learned under the Hebb rule for 1 epoch are based on direct co-activation of the input and output units, resulting in overlap between patterns a and c since they share features and also leads to some interference in their shared weights. When training for 10 epochs with Delta Rule, weight updates depend on the error, so the required changes happen decreasing the interference leading to the biggest weight changes for patterns a and c, for the weights between input connections 6 and 7 then the respective outputs are 1, 3, 5, and 7. This change is evident in the Link Viewer screenshots where at those specific weights. In addition, in Epoch 1, weights seem more strong and distributed, more evident positive(white) and negative(black) weight connections are formed. While in Epoch 10, there is an overall decrease in weight magnitude, such as the mean weight value decreases from Epoch 1 → 10 (0.3125 → 0.020813), maximum weight values (0.375 → 0.374756), etc.. So the delta rule minimized the squared error by reducing unnecessary weight changes and it saved the critical weight differences to be able to differentiate the training patterns. Something else from Link Viewer screenshots is that in Epoch 10, weights are more evenly distributed with less variance, Epoch 1 → Epoch 10 (0.026786 → 0.023338), so the delta rule balanced the strengths of the weight across the network. Regarding pattern b, Patterns a and c are orthogonal to pattern b so it

remains unaffected by the processing over 10 epochs. Since pattern b was already accurately identified from its orthogonality, it is learned independently and is not affected. The main difference is that the Hebb rule fails to account for classification accuracy, causing overlap and weakening the distinctions between comparable patterns through the reinforcement of input/output relationships. By focusing on the error between actual and target outputs, the Delta rule effectively changes weights to reduce interference and improve differentiation, increasing the network's ability to differentiate between non-orthogonal and similar patterns, as supervised learning improves distinctions by altering representations.

Epoch 1



Epoch 10

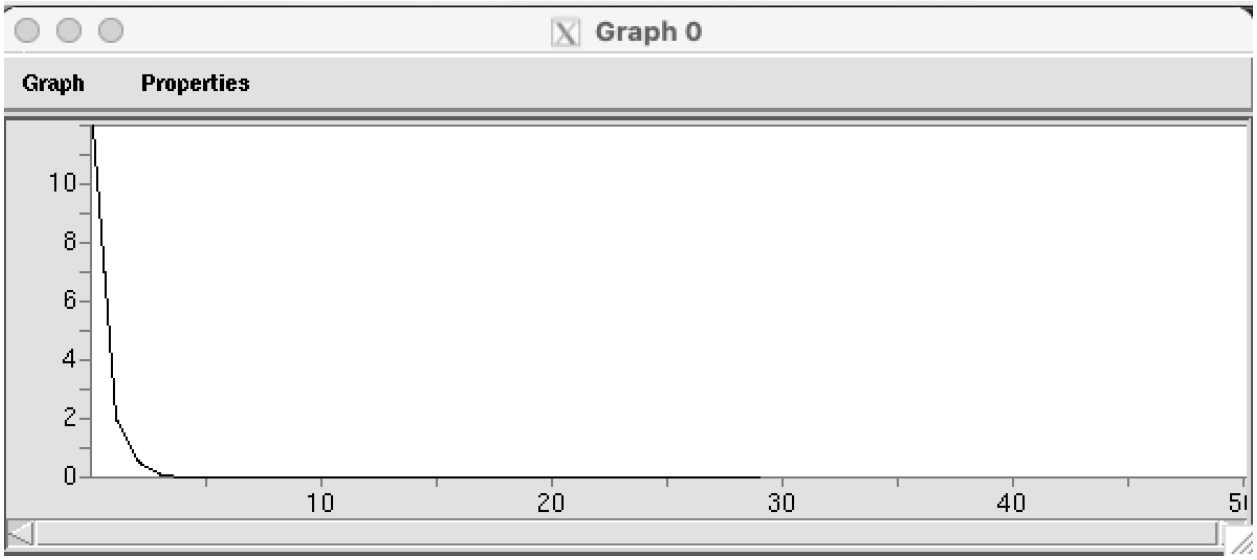


Q4 [20 pts.] Why was training with the intermediate learning rate more effective than either the higher or lower rate? Include the error graph in your response. [Hint: Consider the Delta rule equation, and recall that learning was fine with the large learning rate when there was no noise.]

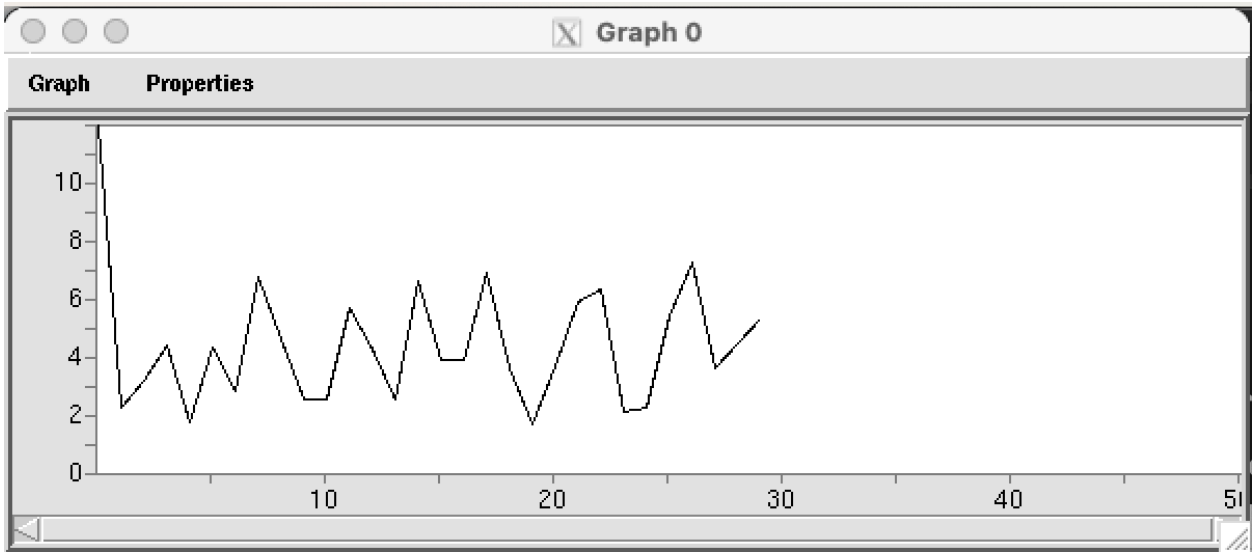
Training with the intermediate learning rate of 0.005 was more effective than either the higher (0.0625) or lower (0.001) rate because of the effect of noise on weight updates and the balance needed for stable convergence. The Delta rule delta wij = epsilon(tj - aj)ai adjusts weights based on the difference between the actual and target activations, but when noise is added to both inputs and targets, the updates become inconsistent and unstable, and fluctuation can cause them to reverse sign. Also, the weight changes may not always decrease error, they can also increase error. When the learning rate was high (0.0625), as portrayed in the Noise On graph, the error oscillated a lot, meaning that weight updates were too large, resulting in the network to overcorrect for optimal values for noise on variations so it doesn't gradually improve

performance which prevents convergence. With a very low learning rate (0.001), the network didn’t make much progress to improve performance within 30 epochs, as shown in the learning rate 0.001 error graph, the learning was very slow and the error decreased very little. The intermediate rate (0.005) was the most effective regarding balance, as shown in the learning rate 0.005 graph, where the network effectively reduced error over 30 epochs even with noise on. The 0.005 learning rate is both small enough for proper stability and high enough for appropriate weight changes. There are a few increases in error, but they are miniscule and are reversed quickly by the following weight changes. The interaction between learning rate, input noise, and the convergence of weight updates in gradient descent determines learning stability. In gradient descent, the intermediate rate is most effective when step sizes are sufficient to provide constant convergence without unpredictable spikes or slow adaptation.

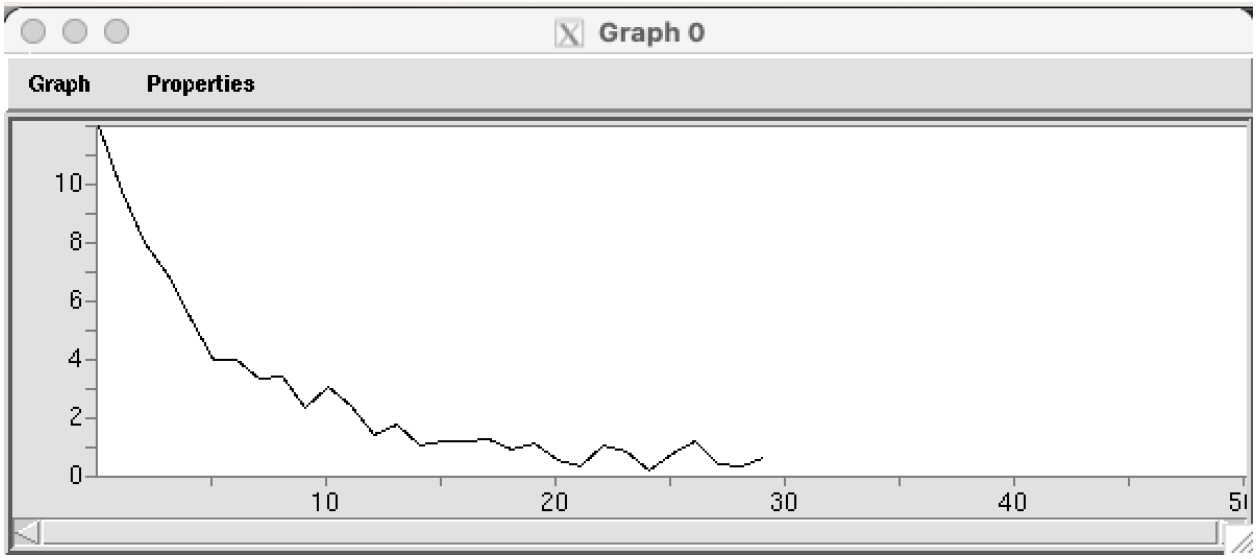
NoiseOff



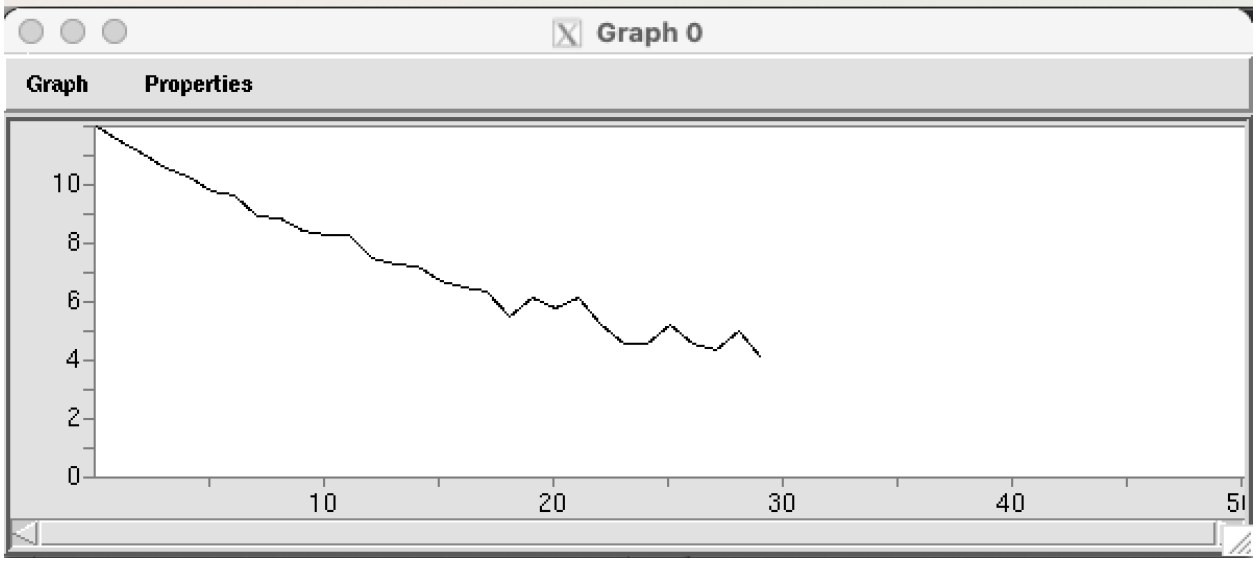
Noise On



Learning Rate 0.005

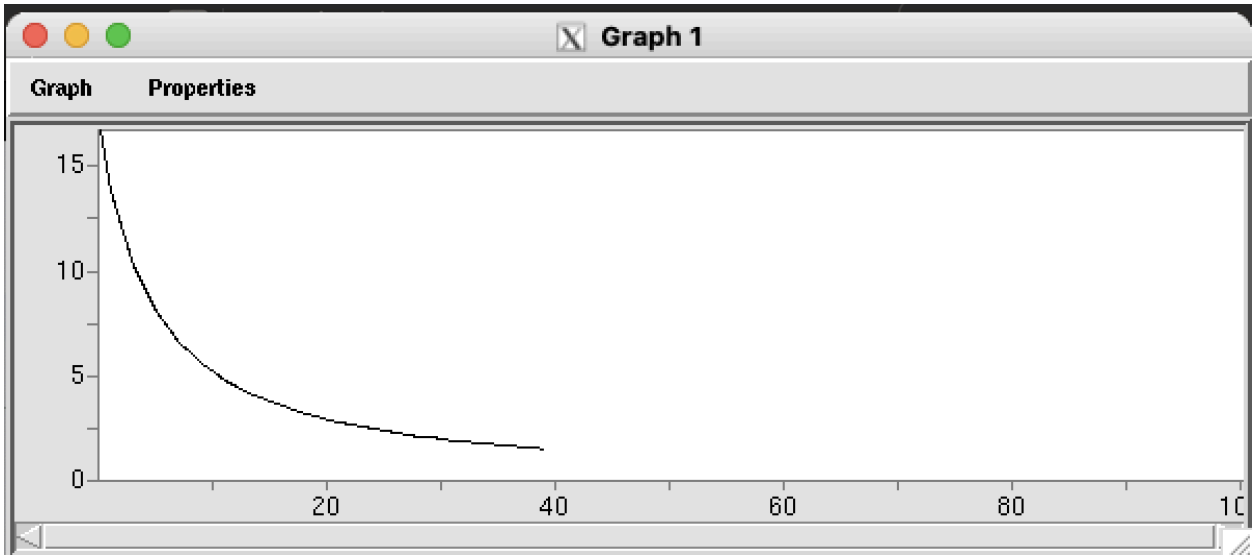


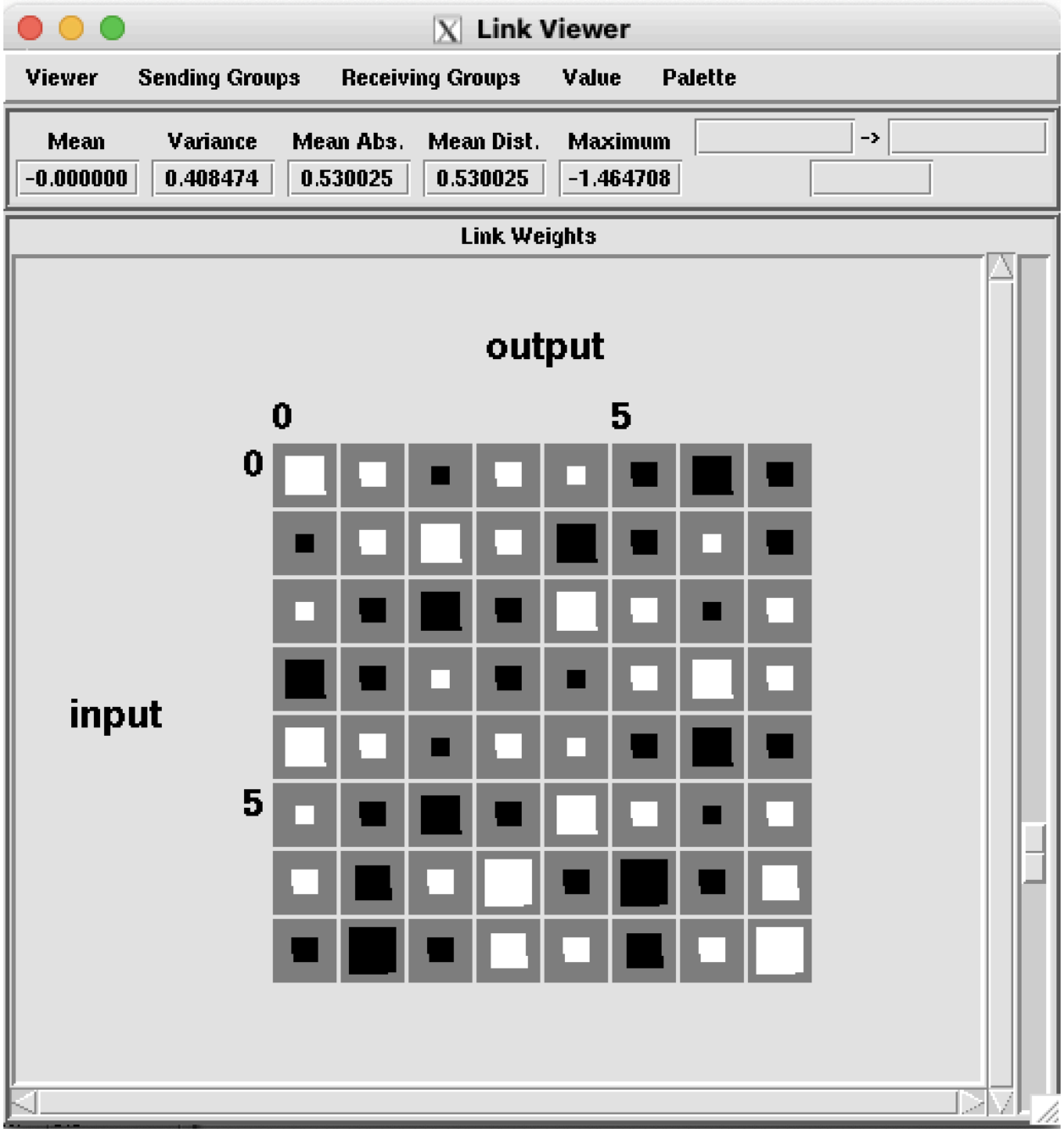
Learning Rate 0.001



Q5 [20 pts.] Why is learning so much slower using sigmoid units than when using linear units? [Hint: there are two contributing factors.] Describe and explain the similarities and differences in the resulting sets of weights.

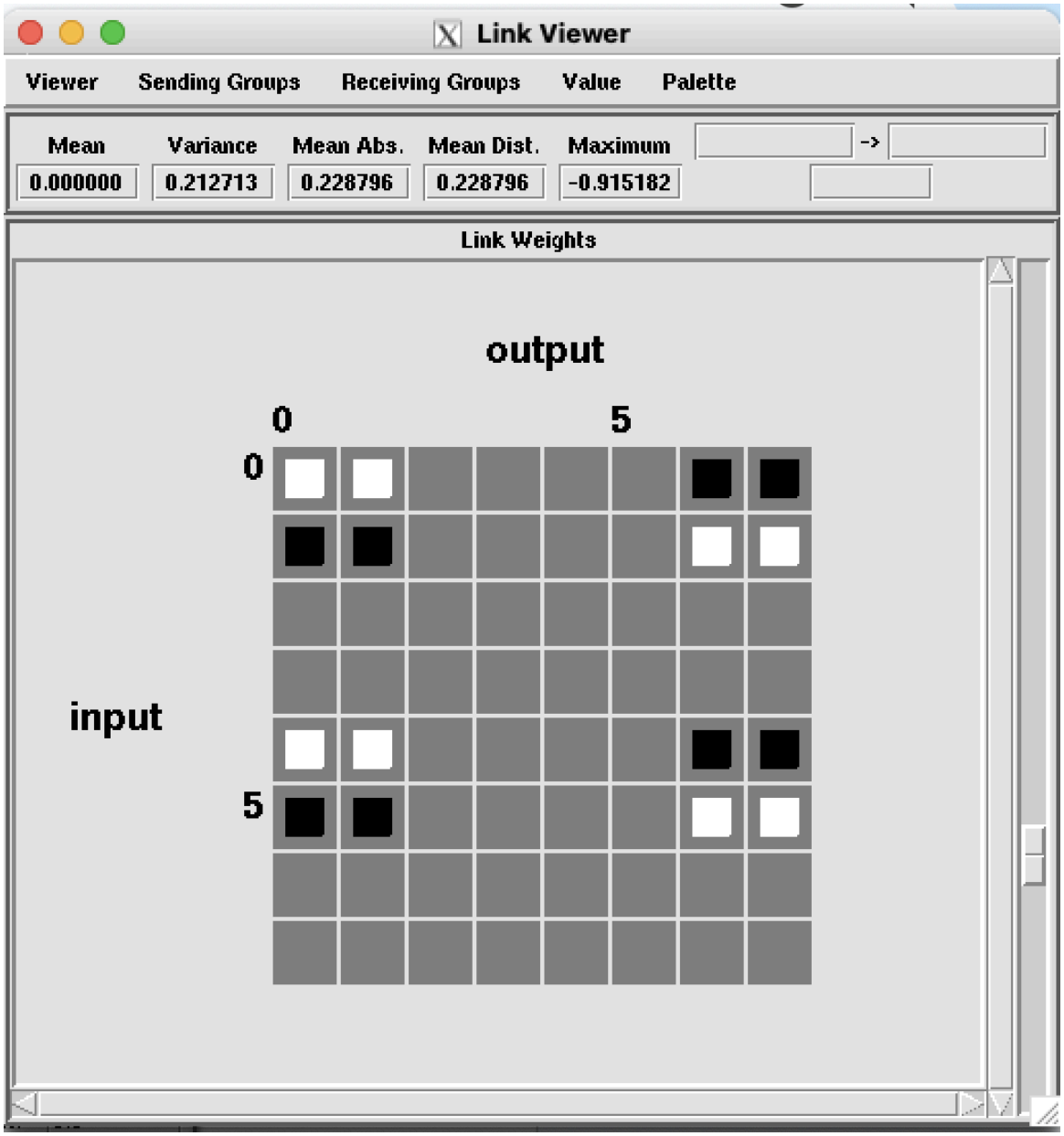
Learning is so much slower using sigmoid units than linear units because of the asymptotic shape of the sigmoid activation function shown in the graph, which reduces the range of weight updates as inputs vary. With linear units, output activations change proportionally to weight updates, so learning is more efficient and direct. Because the sigmoid curve is decreasing and its gradient is becoming more flat, weight updates gradually get smaller as activations get closer to 0/1. This is because with sigmoid units, a change in net input leads to a change in output activation, which decreases as the net input gets larger in magnitude. Since the network must overcome barriers to accumulate sufficient weight changes to increase outputs to target values close to 1.0 or 0.0, convergence is slower. From the Link viewer screenshot of the weights, in linear networks, many weights are 0 when the corresponding target values are always 0, the error is also 0, so some specific connections are neglected effectively. Sigmoid trained networks have no weights that are 0, so their activations don't deactivate entirely, and each weight gets updated, slowing convergence, so when the target is zero, there would be a weight shift when the input unit is turned on in the training patterns. This is based on the delta rule equation, where weight updates are proportional to input activations and error signals. The effective error signal is limited by the limited range of sigmoid outputs, resulting in more minor and gradual weight changes. Compared to linear units, sigmoid based learning needs more iterations to reach similar results. This is due to gradient descent as learning rates are a critical adjusting parameter in training efficiency since non-linearity in activation functions impacts weight space exploration.


Graph 1
Graph
Properties
15
10
5
0
20
40
60
80



Q6 [20 pts.] Explain why learning fails here, even with the Delta rule (where "fails" here means that learning cannot produce weights that cause activations to equal targets for all examples). To answer this, you will have to examine the patterns carefully, noting that some input values are "redundant" with each other---that is, they provide no additional information---and also the relationship of these input values to the targets

Learning fails here even with the Delta rule because the target values for possibly one of the output units aren't linearly separable based on the input patterns. With redundancies in the input patterns, some inputs provide no additional information to distinguish outputs. Also some specific input values can also be identical or direct negations of each other, so then the network can't learn distinct weight updates to correctly separate all target values. When inputs aren't linearly separable, a weight vector can't be positioned to divide the input space into distinct output regions. From the input patterns, 0, 1, 4, and 5 are consistent for all four training patterns, so they don't contribute new information to be able to differentiate outputs. Inputs 2 and 3 are redundant as they are direct negations of each other so they also don't contribute independent information to learning. Due to this redundancy, there are two unique input patterns (6/7), that form an exclusive relationship with the targets. XOR functions aren't linearly separable, thus the Delta rule cannot accurately change the weights to reach a solution. When input patterns are linearly independent so when no pattern can be formed by recombining scaled versions of the others, the delta rule is most effective. Many of the weights in the link viewer diagram are low or zero since redundancy causes weight changes to cancel out, making it unable to develop proper associations. Only the weights that reinforce the constant output activations from the constant inputs are there.


Link Viewer
Viewer
Sending Groups
Receiving Groups
Value
Palette
Mean
Variance
Mean Abs.
Mean Dist.
Maximum
0.000000
0.212713
0.228796
0.228796
-0.915182
->
Link Weights
output
0
5
0
5
input