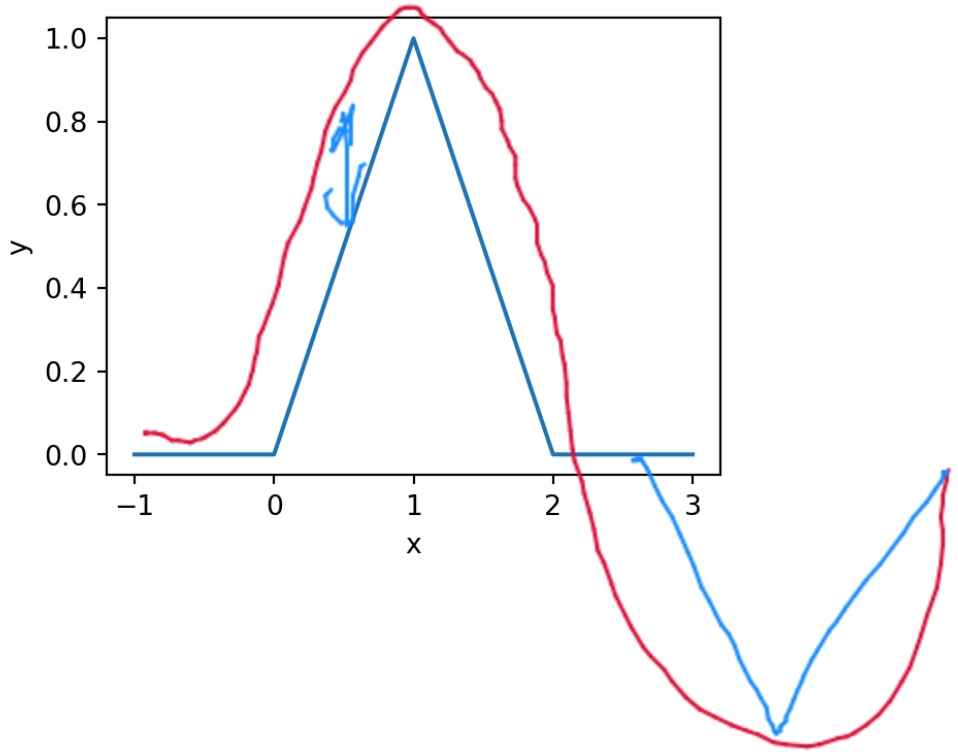
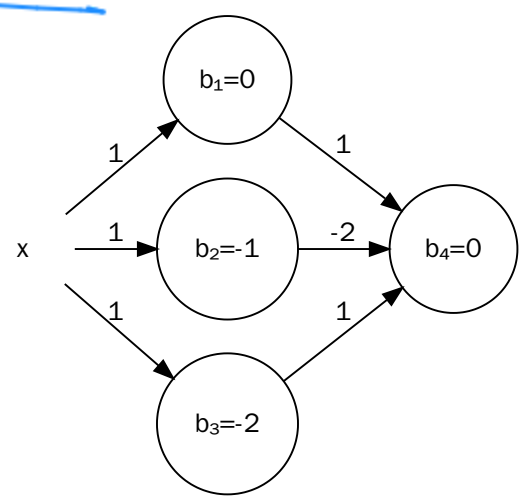
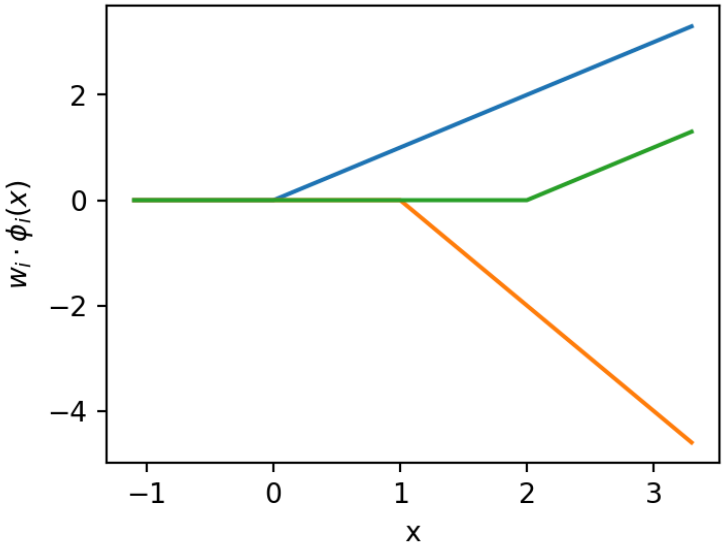
CSCI 1010 Class 9

Profs. Michael Linderman and Phil Chodrow
Department of Computer Science
Middlebury College

Non-linear features and the *universal approximation theorem*







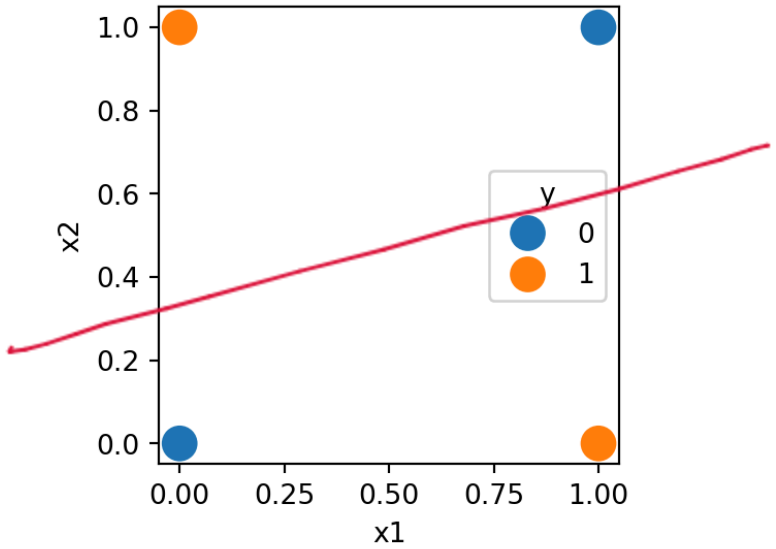
Slide 1 Notes

Consider the this simple piece-wise linear linear function (essentially a triangular "bump"). How can we represent this function using a neural network with a single hidden layer and ReLU activations? [click] If we experiment for a bit, we can construct the following network with 3 hidden neurons. The weighted features are shown below. As we trace from left to right, we can see how these features combine to form the desired output function.

But what if I wanted to do something more complex, like a sine wave? At first approximation we could think of it as two triangles [draw sine wave on top left triangle]. By adjusting the biases and weights we can shift the triangle left or right, scale it up or down, and invert it. We add a second triangle (by adding 3 more neurons/features to the hidden layer) shifted and inverted. This is still not a sine wave, but it's closer. We can shrink these errors with more triangles, and more triangles, and more triangles. I hope you can get the sense how by increasing the number of neurons in the hidden layer, we can approximate any continuous function to arbitrary accuracy. This is the essence of the *universal approximation theorem*.

Adapted from https://math.stackexchange.com/a/3796812.

x1	x2	XOR$(x1, x2)$
0	0	0
0	1	1
1	0	1
1	1	0



$W_1 X_1 + W_2 X_2 + b$

True or false? We can learn the XOR function using a single linear layer, e.g., just `nn.Linear(2, 1)`?

a. True

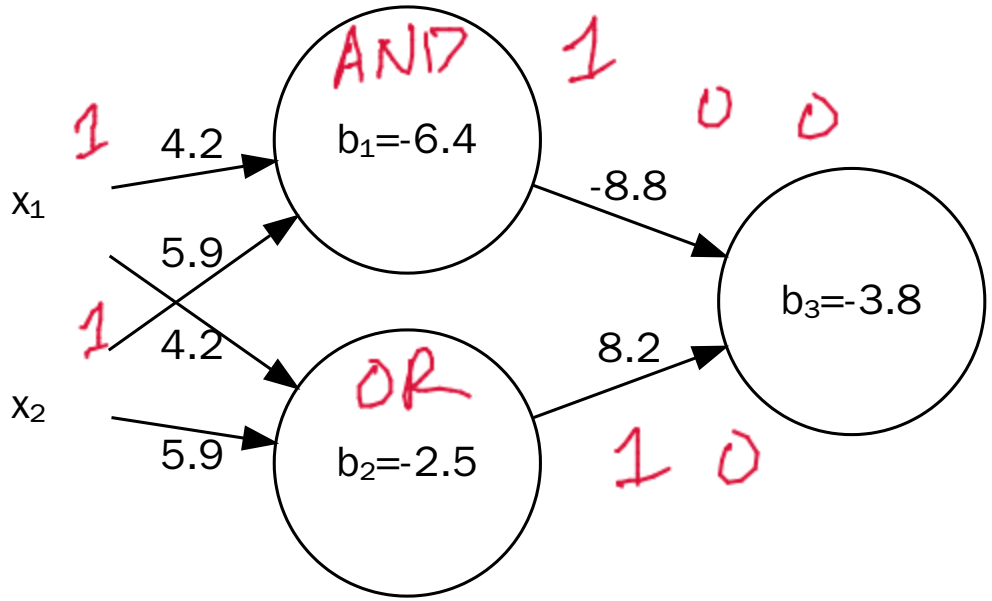
b. False

Slide 2 Notes

Answer: B

Recall from the reading, XOR is not linearly separable. i.e., there is no linear decision boundary that will separate these classes. How can we solve this? By using multiple layers, we can create non-linear decision boundaries that can represent functions like XOR.

Torch learned the following weights (shown on the edges) and biases (in circles) for neurons with the step activation function show right.



$$h(x) = \begin{cases} 1 & \boldsymbol{w} \cdot \boldsymbol{x} + b \geq 0 \\ 0 & \boldsymbol{w} \cdot \boldsymbol{x} + b < 0 \end{cases}$$

$-8.8 + 8.2 - 3.8 < 0$

0	0	0
0	1	1
1	0	0
1	1	0

NOT(a) AND (b)

a. (x1 AND NOT x2) OR (NOT x1 AND x2)

b. NOT (x1 AND x2) AND (x1 OR x2)

c. NOT (NOT (x1 AND NOT x2) AND NOT (NOT x1 AND x2))

d. NOT (NOT x1 AND NOT x2) AND NOT (x1 AND x2)

Slide 3 Notes

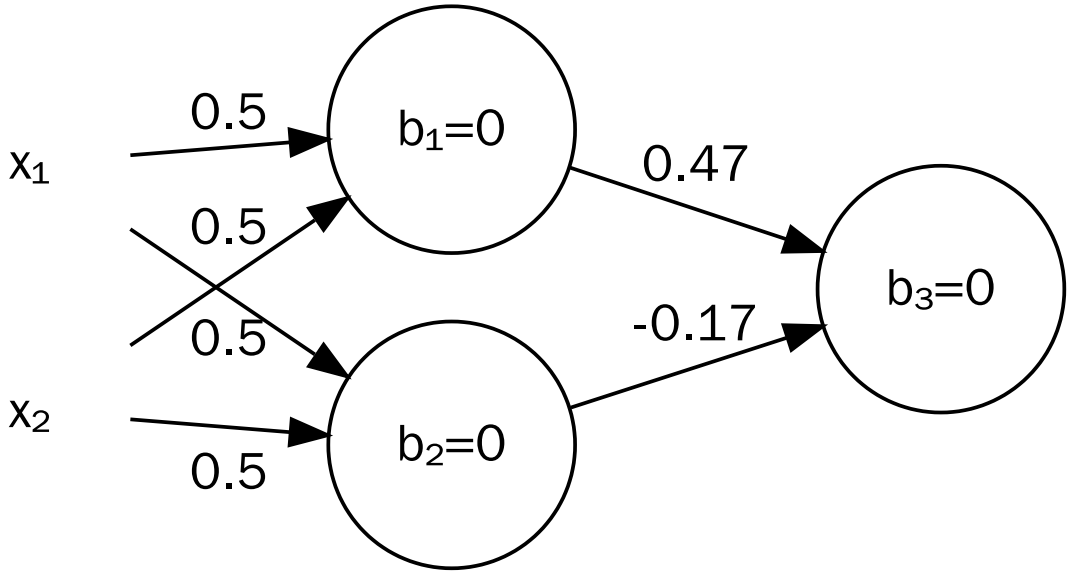
Answer: B

The first neuron is an AND (notice both inputs have to be 1, to be greater than 0), the second is an OR (only a single input needs to be 1, to be greater than 0), the 3rd is (not a AND b).

To convince yourself of the latter,

$$0 * -8.8 + 0 * 8.2 + -3.8 < 0(0)$$
$$0 * -8.8 + 1 * 8.2 + -3.8 > 0(1)$$
$$1 * -8.8 + 0 * 8.2 + -3.8 < 0(0)$$
$$1 * -8.8 + 1 * 8.2 + -3.8 < 0(0)$$

Assume the following initial weights and biases. For an initial batch of $x_1 = 0, x_2 = 0, y = 0$, the model predicts 0.54. The gradient δb_3 (i.e., $\frac{\partial \text{Loss}}{\partial b_3}$) is 0.27. How will b_3 be adjusted?



a. b_3 will increase

b. b_3 will stay the same

c. b_3 will decrease

Slide 4 Notes

Answer: C

Recall that the general expression is $w' \leftarrow w - \alpha \times \delta w$. More intuitively, we want to the loss to be smaller. Since the gradient is positive, the bias will need to decrease (be more negative) to reduce the loss. Alternately, we observe that the output is too large (we want it to be 0) so we need to make the output smaller, i.e., make the bias parameter smaller.