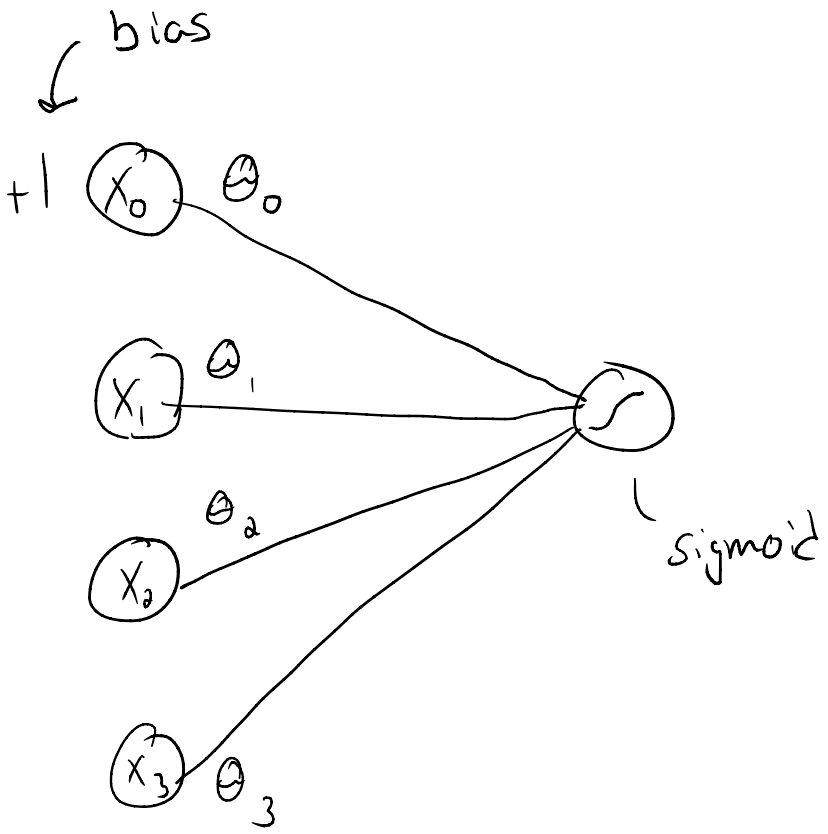
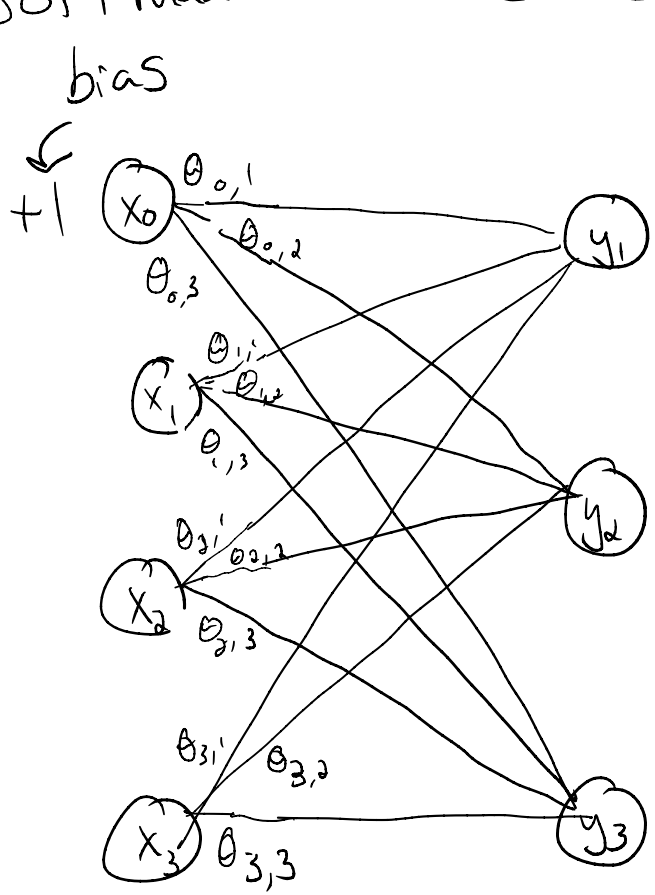
Recall from last class:

single layer neural network



$$y_i = \frac{1}{1 + \exp\left[-\sum_{j=1}^{d} \theta_j x_j\right]}$$

for $K > 2$, add more outputs and use softmax instead of sigmoid



$$y_i = \frac{\exp(\theta_i^T x)}{\sum_k \exp(\theta_k^T x)}$$

↳ gives probabilities for each

- Alternative, sigmoid of squared loss for each is option as well (this will be what you do in next project)

Training Perceptron:

- update weights until convergence with some form of gradient descent
- update = learning_factor (desiredout − actualout) input

$$SGD = \Delta\theta_{ij}^{t} = \alpha\left(r_j^t - y_j^t\right) x_i^t$$

← current sample considered

$$GD = \Delta\theta_{ij} = \alpha \sum_t \left(r_j^t - y_j^t\right) x_i^t$$

Power / Problems

- could implement AND, OR, NOT
- could not implement XOR
- could only represent linear functions
- solution → add intermediate ("hidden") layers between input and output layers

First thought:

let $z_h = \sum_j \theta_{jh} x_j \qquad h = 1, \dots H$

Then, let $y_i = \sum_{h=0}^{H} \psi_{h,i} z_h$

Then apply softmax ($k > 2$) or signoid ($k = 2$)

- doesn't actually gain us anything
  - ↳ linear combination of linear combination is still just a linear combination

Need output of hidden layer to be nonlinear

↳ get linear combination of nonlinear basis functions

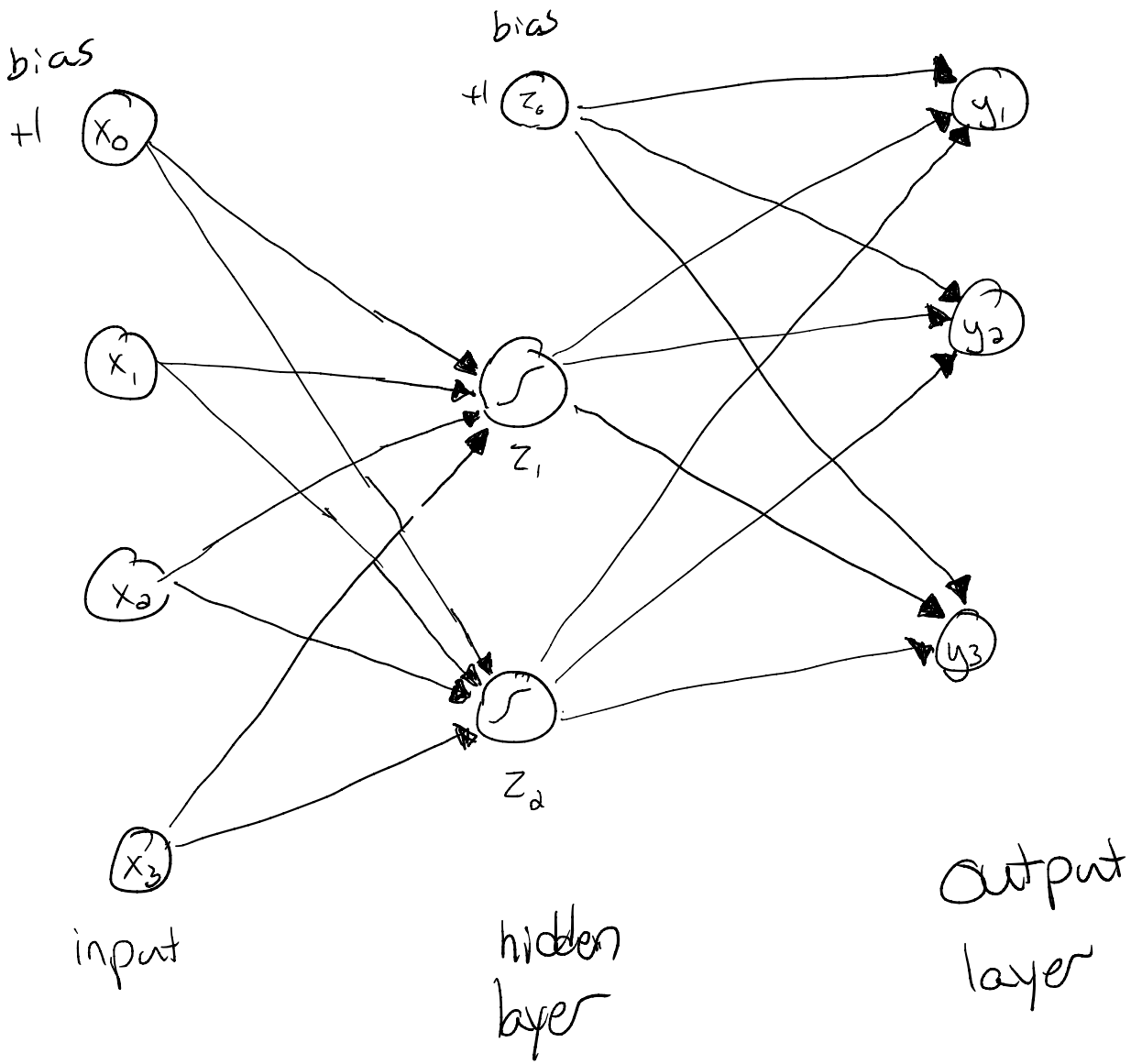
↳ nonlinear

Still want differentiability so we can used gradient based update

Options: sigmoid, tanh, etc.

Hidden units → nonlinear transformation from d-dimensional input space to H-dimensional hidden layer space

Example: 2-layer



- Can add multiple hidden layers

- Training a multilayer neural network
  - error propagates from output y back to inputs
    $\Rightarrow$ back propagation
  - gradient descent $\rightarrow$ negative gradient points downhill
  - partial derivatives w/respect to weights

$$\frac{\partial E}{\partial \theta_{jh}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial \theta_{jh}}$$

chain rule

Basically two steps for each training instance

1) feed forward
   plug in inputs and compute outputs

2) update weights based on how far off outputs were
   - via GD, SGD, MBGD
   - multilayer must use back propagation
     - need to update weights in earlier layers, not just weights for last layer

Multilayer: Squared error (Quadratic Loss) with sigmoid on each output

$$E(w \mid x^t, r^t) = \frac{1}{2}(r^t - y^t)^2$$

derivative of sigmoid

$$y = \sigma(g) = \frac{1}{1+\exp^{(-g)}} \qquad g = \theta^T x$$

$$\partial\sigma/\partial g = \sigma(g)(1-\sigma(g))$$

For each output i: ← because of chain rule

$$E_i = (r_i - y_i) y_i (1 - y_i)$$

$$\Delta v_{hi} = \alpha E_i z_h$$

For each hidden h:

$$E_h = z_h (1 - z_h) v_{h,i} E_i$$

$$\Delta w_{ih} = \alpha E_h x_j$$

Multilayer: Two Classes - Cross Entropy Cost

- still use one output with sigmoid activation function

$$y^t = \text{sigmoid}\left(\sum_{h=0}^{H} \psi_h z_h^t\right)$$

$$E(\theta, \psi \mid X) = -\sum_t r^t \log(y^t) + (1-r^t)\log(1-y^t)$$

$\Rightarrow$

weight update equations

$$\begin{cases} \Delta\psi_h = \alpha \sum_t (r^t - y^t) z_h^t \\ \Delta\theta_{jh} = \alpha \sum_t (r^t - y^t)\, \psi_h z_h^t (1 - z_h^t) x_j^t \end{cases}$$

Multilayer: >2 classes cross entropy loss

$$y_i^t = \frac{\exp\left(\sum_{h=0}^{H} \psi_{hi} z_h^t\right)}{\sum_k \exp\left(\sum_h \psi_{hk} z_h\right)}$$

$$E(\theta, \psi \mid X) = -\sum_t \sum_i r_i^t \log(y_i^t)$$

update equations:

$$\Delta \psi_{hi} = \alpha \sum_t (r_i^t - y_i^t) z_h^t$$

$$\Delta \theta_{jh} = \alpha \sum_t \left[\sum_i (r_i^t - y_i^t) \psi_{hi}\right] z_h^t (1 - z_h^t) x_j^t$$