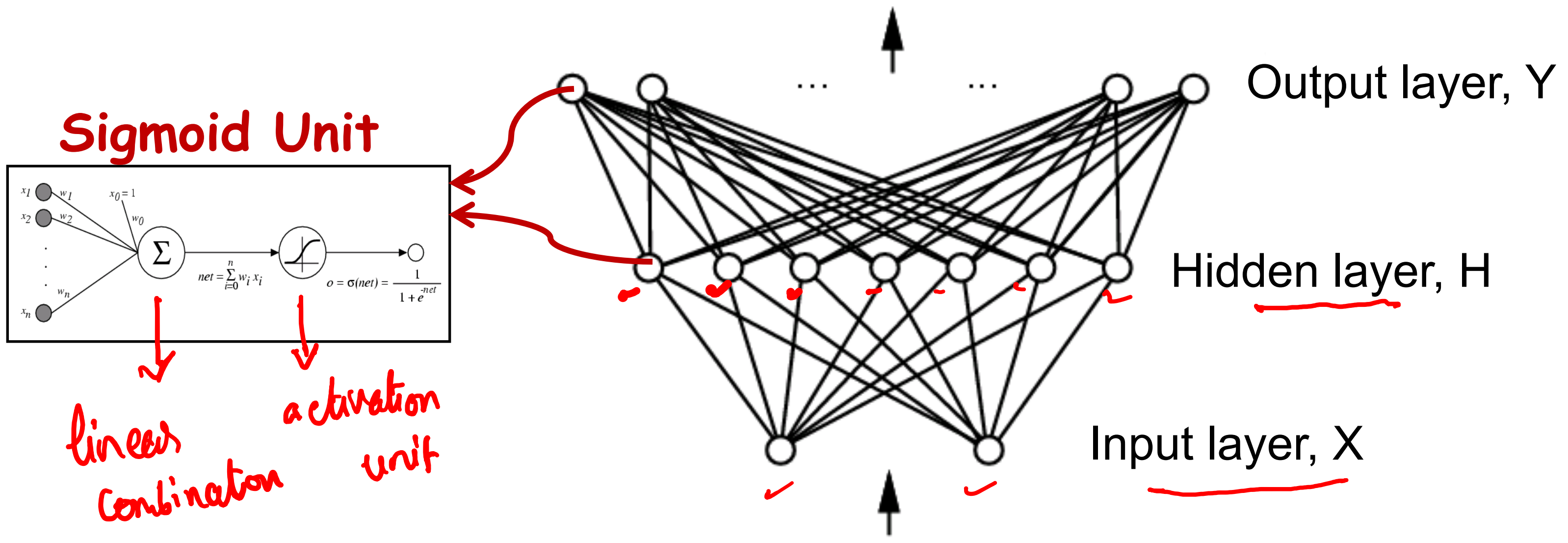
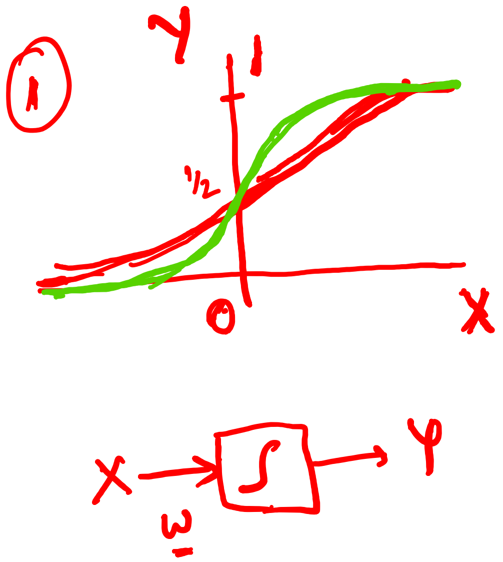
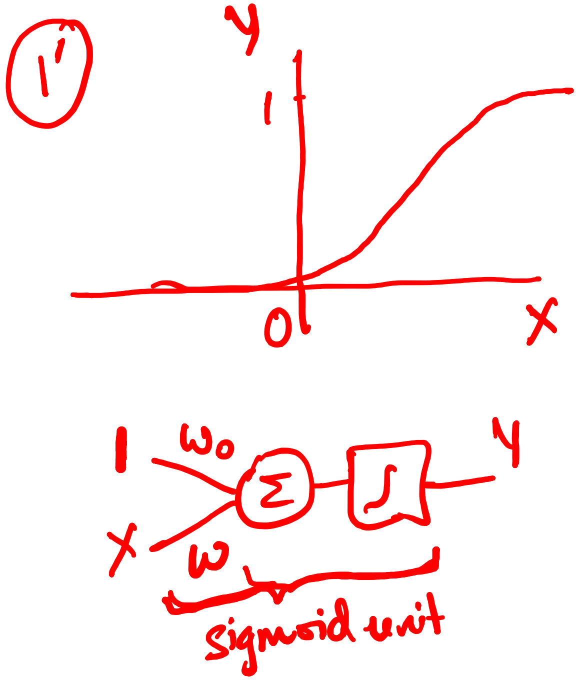
Neural Networks to learn f: X → Y

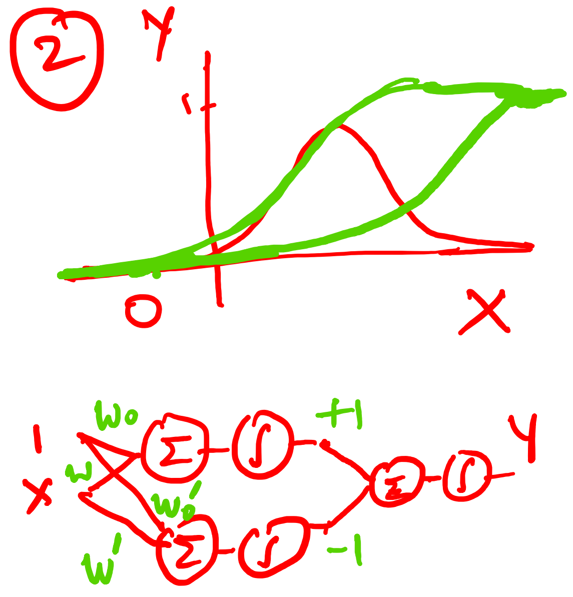
- f can be a **non-linear** function
- X (vector of) continuous and/or discrete variables
- Y (**vector** of) continuous and/or discrete variables

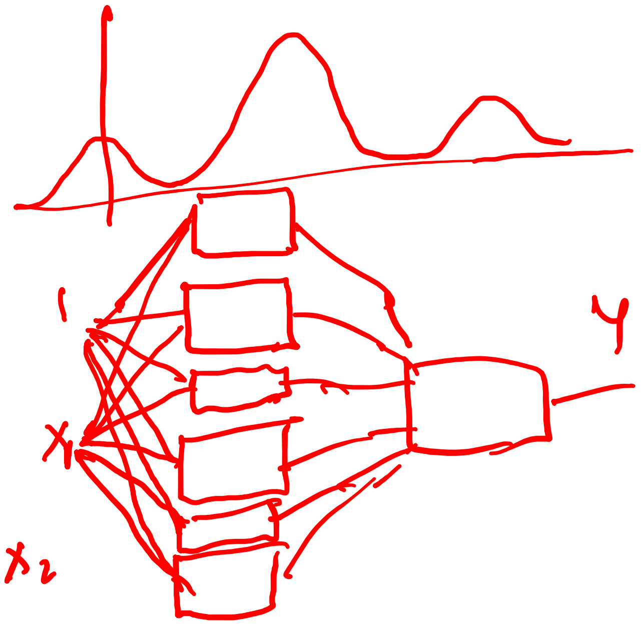
- Neural networks - Represent f by *network* of sigmoid (more recently ReLU – next lecture) units :

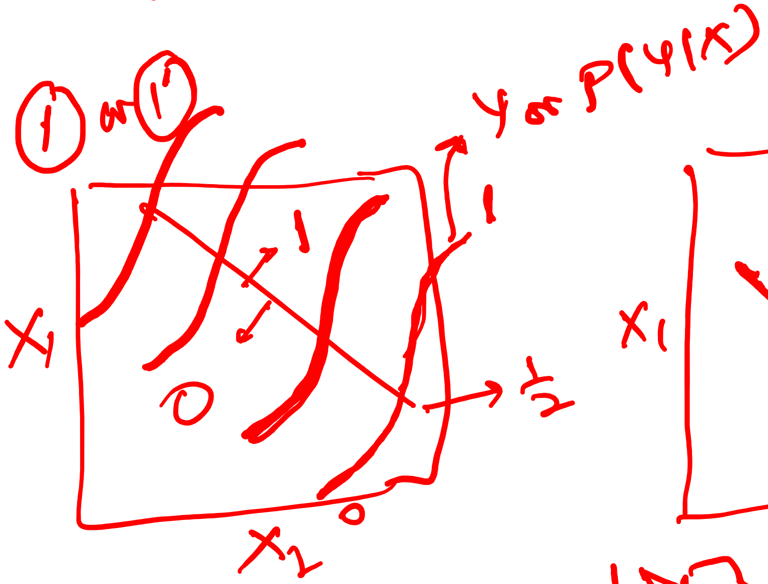


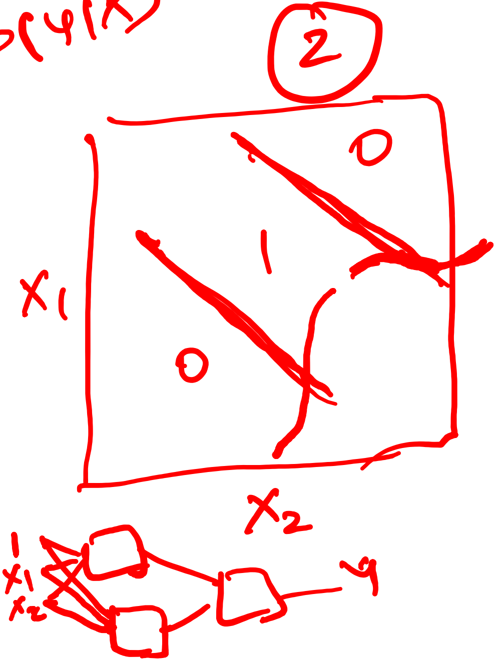

1
y
1
1/2
0
x
x
w
y


1'
y
1
0
x
1
w0
x
w
y
sigmoid unit


2
y
1
0
x
1
x
w0
w
w0'
w'
+1
-1
y


1
x1
x2
y


1 or 1'
y or P(y|x)
1
1
0
1/2
x1
x2
0


2
0
1
0
x1
x2
1
x1
x2
y

1 hidden layer NN demo on 2D inputs

- https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Training Neural Networks – l2 loss

$$W \leftarrow \arg\min_W E[W]$$

$$W \leftarrow \underline{\arg\min_W} \sum_l (\underline{y^l} - \hat{f}(x^l))^2$$

Learned neural network

Where $\widehat{f}(x^l) = o(x^l)$, output of neural network for training point x^l

Train weights of all units to minimize sum of squared errors of predicted network outputs

Minimize using Gradient Descent

For Neural Networks,
***E*[*w*] no longer convex in w**

Gradient

error

$$\underline{\nabla E[\vec{w}]} \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

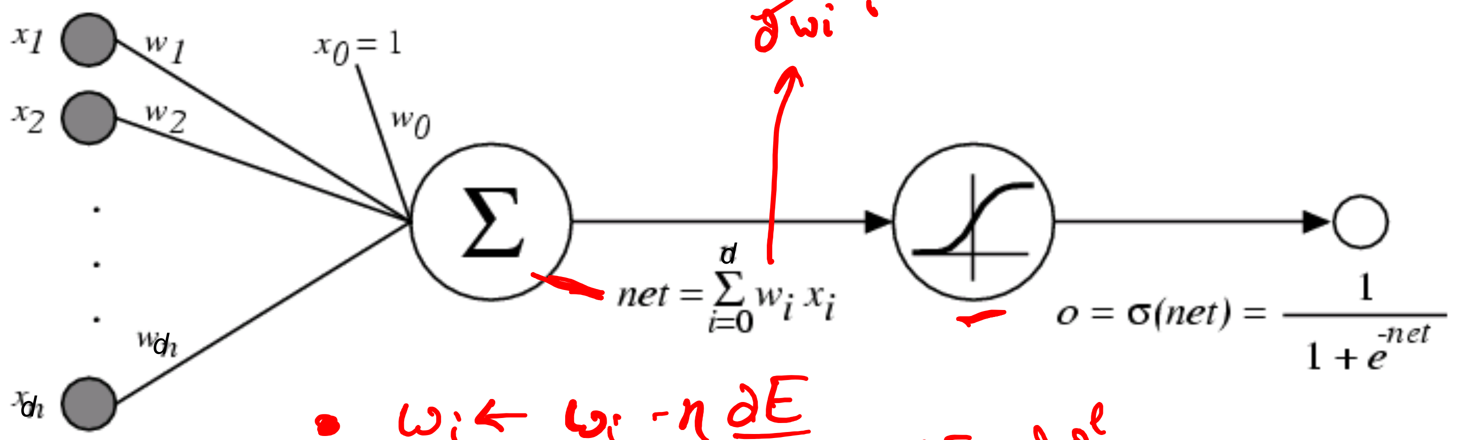
Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent for 1 sigmoid unit



x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
w_d
x_d
Σ
$net = \sum_{i=0}^{d} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

$\frac{\partial}{\partial w_i} \sum_i w_i x_i = x_i$

• $w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}$

$\frac{\partial E}{\partial o^l} \cdot \frac{\partial o^l}{\partial w_i}$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \underbrace{\sum_{l \in D} (y^l - o^l)^2} = \sum_l (y^l - o^l) \left(-\frac{\partial o^l}{\partial w_i}\right)$$

$\frac{\partial \sigma(t)}{\partial t} = \sigma(t)(1-\sigma(t))$

Gradient of the sigmoid function output wrt its input

$$\frac{\partial \sigma(net)}{\partial net} = \sigma(net)(1 - \sigma(net)) = o(1 - o)$$

Gradient of the sigmoid unit output wrt input weights

$$\frac{\partial o}{\partial w_i} = \frac{\partial o}{\partial net} \cdot \frac{\partial net}{\partial w_i} = o(1 - o)x_i$$

Incremental (Stochastic) Gradient Descent

SGD

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$

Using all training data *D*

2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \underline{\sum_{l \in D}} (y^l - o^l)^2$$

Incremental mode Gradient Descent:
Do until satisfied

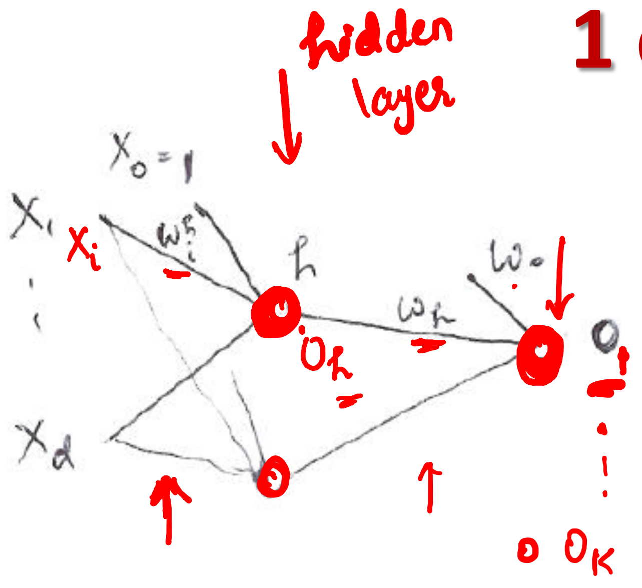
- For each training example l in D
  1. Compute the gradient $\nabla E_l[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_l[\vec{w}]$

$$E_l[\vec{w}] \equiv \frac{1}{2} (y^l - o^l)^2$$

no sum
e

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) = \sigma\left(\sum_h w_h o_h\right)$$

$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) = \sigma\left(\sum_i w_i^h x_i\right)$$

$$\frac{\partial o}{\partial w_i^h} = \frac{\partial o}{\partial o_h} \cdot \boxed{\frac{\partial o_h}{\partial w_i^h}}$$

$$o_h(1-o_h)x_i$$

$$o(1-o)w_h$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{l\in D}(y^l - o^l)^2 = \sum_l \overset{\frac{\partial E}{\partial o^l}}{(y^l - o^l)}\left(-\frac{\partial o^l}{\partial w_i}\right)$$

Gradient of the output with respect to w_h

$$\boxed{\frac{\partial o}{\partial w_h}} = o(1-o)o_h$$

Gradient of the output with respect to input weights w^h_i

$$\frac{\partial o}{\partial w_i^h} = \overset{(y-o)\;\delta}{\boxed{o(1-o)}}\,\overbrace{o_h(1-o_h)w_h}^{\delta_h}\,x_i$$

Backpropagation Algorithm (MLE) using Stochastic gradient descent

$\boxed{\text{1 final output unit}}$

```
Initialize all weights to small random numbers.
Until satisfied, Do
  • For each training example, Do
      1. Input the training example to the network
         and compute the network outputs
      2.
            output   $\delta \leftarrow o(1-o)(y-o)$
      3. For each hidden unit $h$
            $\delta_h \leftarrow o_h(1-o_h)w_h\delta$
      4. Update each network weight $w_{i,j}$
            $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$
         where
            $\Delta w_{i,j} = \eta \delta_j o_i$
```

→ Using Forward propagation

y = label of current training example

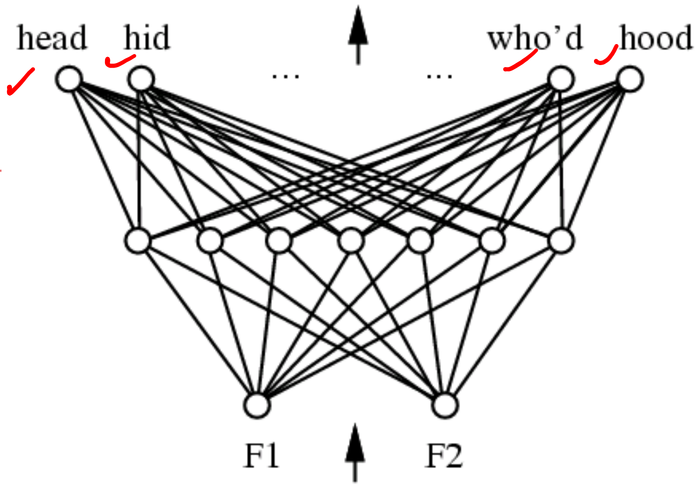
$o_{(h)}$ = unit output (obtained by forward propagation)

w_{ij} = wt from i to j

$$\begin{tikzcd} j \\ i \arrow[u, "w_j^i = w_{ij}"'] \end{tikzcd}$$

Note: if i is input variable, $o_i = x_i$

Backpropagation Algorithm (MLE) using Stochastic gradient descent



Initialize all weights to small random numbers.
Until satisfied, Do

- For each training example, Do

  1. Input the training example to the network and compute the network outputs $\longrightarrow$ Using Forward propagation
  2. For each output unit k

     $$\delta_k \leftarrow o_k(1 - o_k)(y_k - o_k)$$

  3. For each hidden unit h

     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

  4. Update each network weight $w_{i,j}$

     $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

     where

     $$\Delta w_{i,j} = \eta \delta_j o_i$$

y_k = label of current training example for output unit k

$o_{k(h)}$ = unit output (obtained by forward propagation)

w_{ij} = wt from i to j

Note: if i is input variable, $o_i = x_i$

HW2

$$\begin{bmatrix} o_1 \\ \vdots \\ o_k \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.5 \\ 0.05 \\ 0.05 \end{bmatrix}$$

$o_1 = P(y = o_1 \mid x)$

$\vdots$

$o_k = P(y_k = o_k \mid x)$

- Classification – $\boxed{\text{cross-entropy}}$ error metric

$$\ell_2 : E = \frac{1}{2}(y-o)^2$$

$$\hookrightarrow E : -y \log o$$

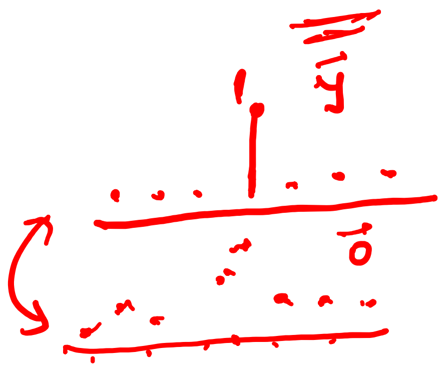
$$\underbrace{-\sum_k y_k \log o_k}$$

$$E_y[-\log o] \leftarrow$$

$$-y_i \log o_i$$

$$\begin{array}{c} y_1 \\ \vdots \\ y_k \end{array} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i^{th} \text{ class}$$

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial o} \cdot \frac{\partial o}{\partial w}$$



- Can implement backpropagation with matrix-vector products – uses matrix-vector calculus heavily

Poll 1:

- $y = f(\boldsymbol{z})$
- $z_i = g_i(\boldsymbol{x})$
- $\frac{\partial y}{\partial \boldsymbol{x}} = \cdots$

A. $\cancel{\frac{\partial y}{\partial \boldsymbol{z}}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$

B. $3 \times 4 \quad \frac{\partial \boldsymbol{z}^T}{\partial \boldsymbol{x}} \frac{\partial y}{\partial \boldsymbol{z}} \cdot 4 \times 1$

C. $\cancel{\frac{\partial y}{\partial \boldsymbol{z}}} \frac{\partial \boldsymbol{z}^T}{\partial \boldsymbol{x}}$

D. $\cancel{\frac{\partial y}{\partial \boldsymbol{z}}}^T \frac{\partial \boldsymbol{z}^T}{\partial \boldsymbol{x}}$

E. $\left(\frac{\partial y}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right)^T$

F. None of the above

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \quad 4\times1 \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad 3\times1 \qquad \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \partial y / \partial x_2 \\ \partial y / \partial x_3 \end{bmatrix} \quad 3\times1$$

$$\frac{\partial z^T}{\partial x} = \frac{\partial \overset{1\times4}{[g_1(x) \ldots g_4(x)]}}{\underset{3\times1}{\partial x}}$$

$$= \begin{bmatrix} \frac{\partial g_1(x)}{\partial x_1} & \cdots & \frac{\partial g_4(x)}{\partial x_1} \\ \vdots & & \\ \frac{\partial g_1(x)}{\partial x_3} & \cdots & \frac{\partial g_4(x)}{\partial x_3} \end{bmatrix} \quad 3\times4$$

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)

Objective/Error no longer convex in weights

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast