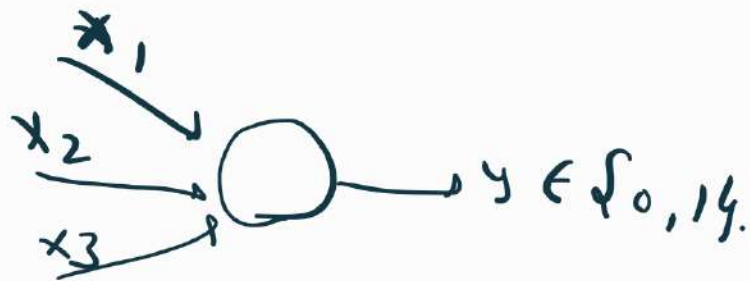


6/11/25

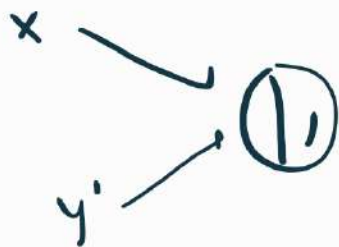
- Goodwill hunting (movie)



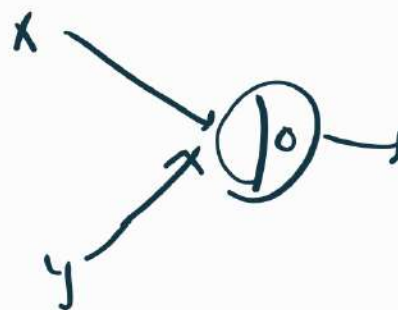
$$f(x) = 1 \left(\sum_i x_i \geq \theta \right)$$

- Inputs can be excitatory or inhibitory
- when inh. input is 1 o/p $\rightarrow 0$
- count no. of 'ON' signal on the excitatory

xy'



NOR



inhibitory inputs" refer to signals that decrease the likelihood of a neuron firing, while "excitatory inputs" are signals that increase the likelihood of a neuron firing

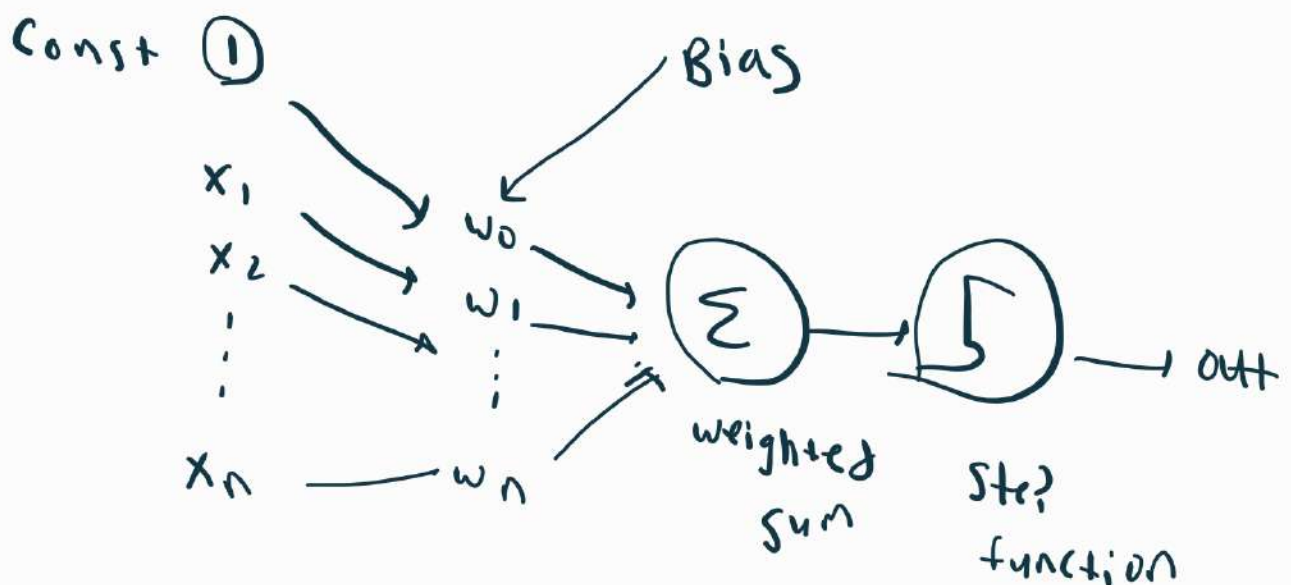
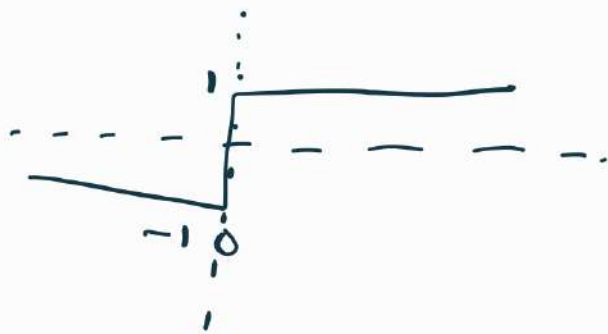
★ Perceptron.

Inputs can be real, weights can be diff. for diff. i/p components.

$$f(x) = \begin{cases} 1 & \text{when } \sum w_i x_i + b \geq 0 \\ 0 & \text{else.} \end{cases}$$

$$\sigma(x) = \begin{cases} 1 & \text{when } x \geq 0 \\ -1 & \text{else.} \end{cases}$$

$$f(x) = \sigma(w^T x + b)$$



* Perceptron learning

* Training data (x^i, y^i)

Start with $k \leftarrow 1$ & $w_k = 0$

while $\exists i \in \{1, 2, \dots, N\}$ such that $y_i (w^T \cdot x^i) \leq 0$.

$$w_{k+1} = w_k + y^i x^i$$

$k++$

HW Perceptron learning can we use for MLP learning?

* Polynomial exp.

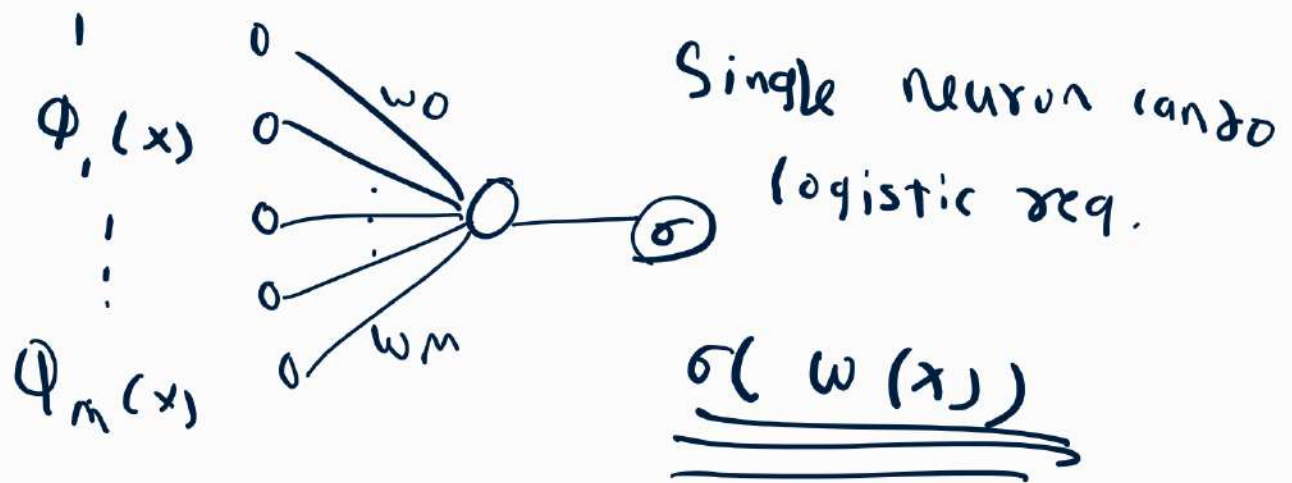
Bias-variance decomp.

pre processing. (feature extraction)

Regularization.

$$t = w^T (\phi(x))$$

\uparrow
regression.



$P(k|x) \rightarrow$ Disc. function

$P(x|k) \rightarrow$ gen. function

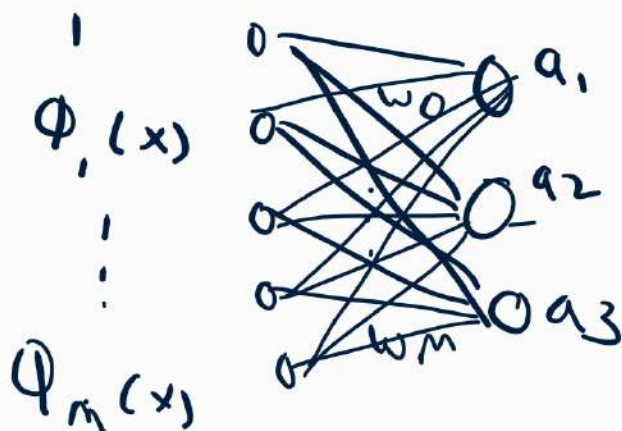
logistic reg \rightarrow

$$\sigma(w^T x)$$

\uparrow

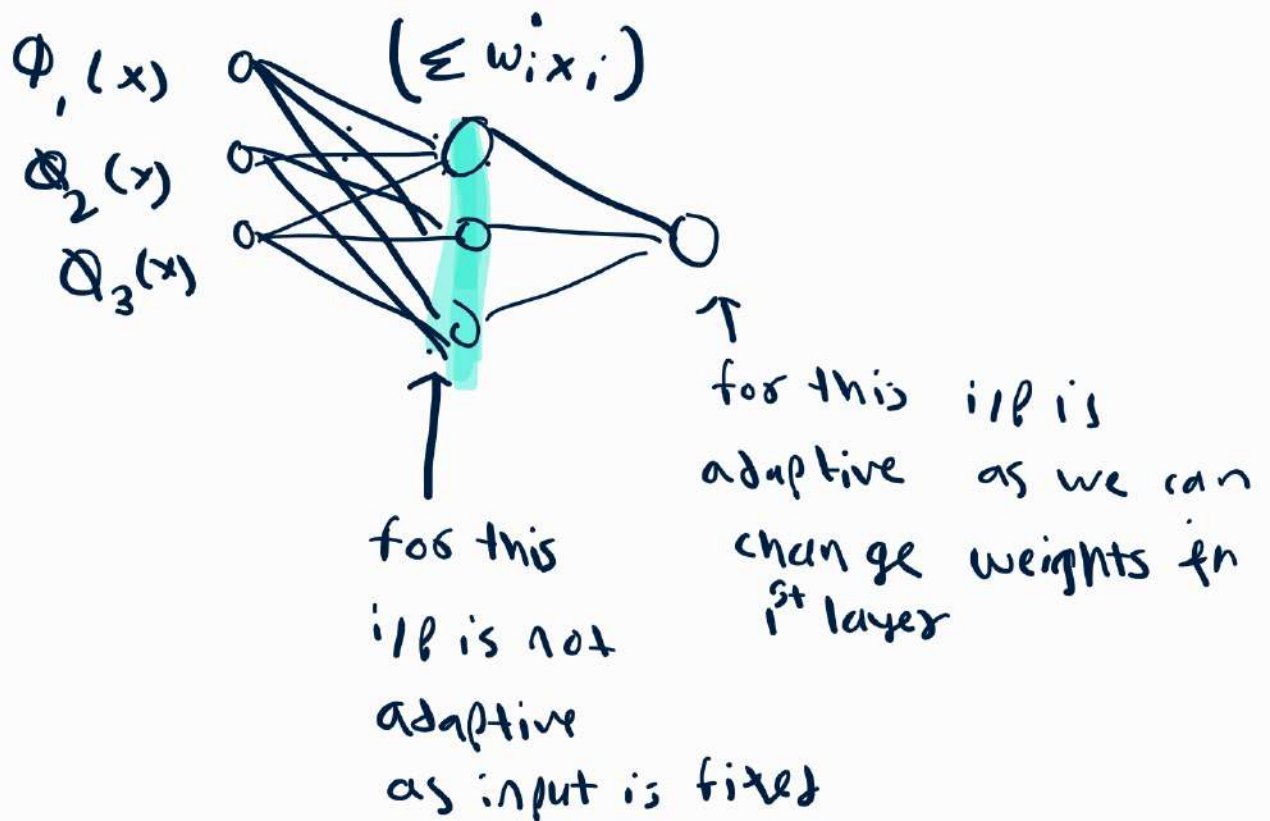
(result is $-\infty \rightarrow \infty$)

(Sigmoid
work is to convert it into $0 \rightarrow 1$)



+ Feed Forward NN

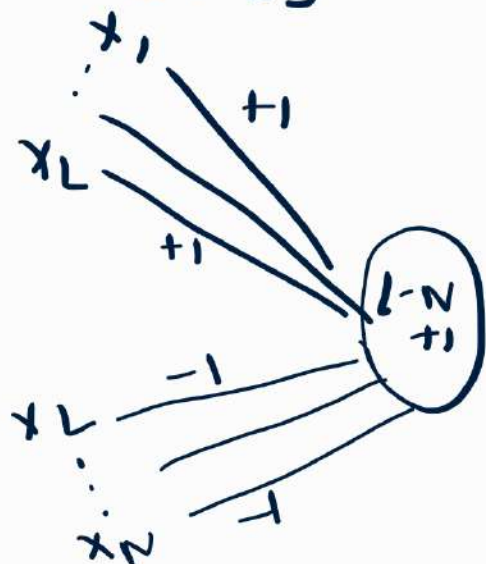
- linear comb. of nonlinear basis f^n .
- With linear we can not do much
- we have to fix no. of f^n but make it adaptive
- where ANN comes in.



* 9/1/25

$$\begin{bmatrix} L-N & 0 \\ 0 & L \end{bmatrix}$$

$$\begin{pmatrix} -1(N-L) \\ L-N \end{pmatrix}$$



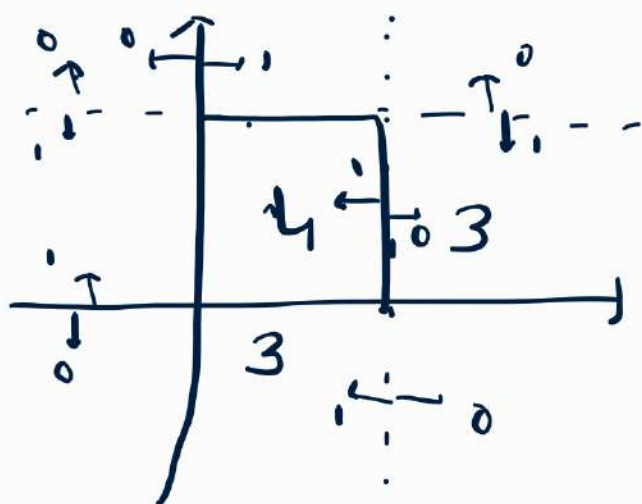
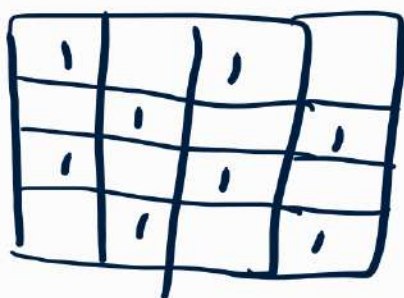
$$f(A, B, C, D) = + A \bar{B} \bar{C} D + \dots$$

$$\left(\bigvee_{i=1}^L x_i \right) \vee \left(\bigvee_{l=1}^N \bar{x}_l \right)$$

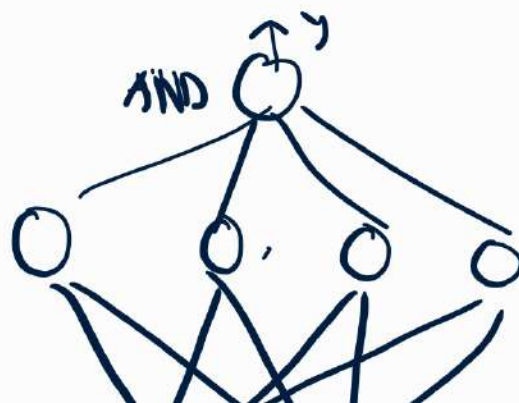
4 vari → 8

n ~

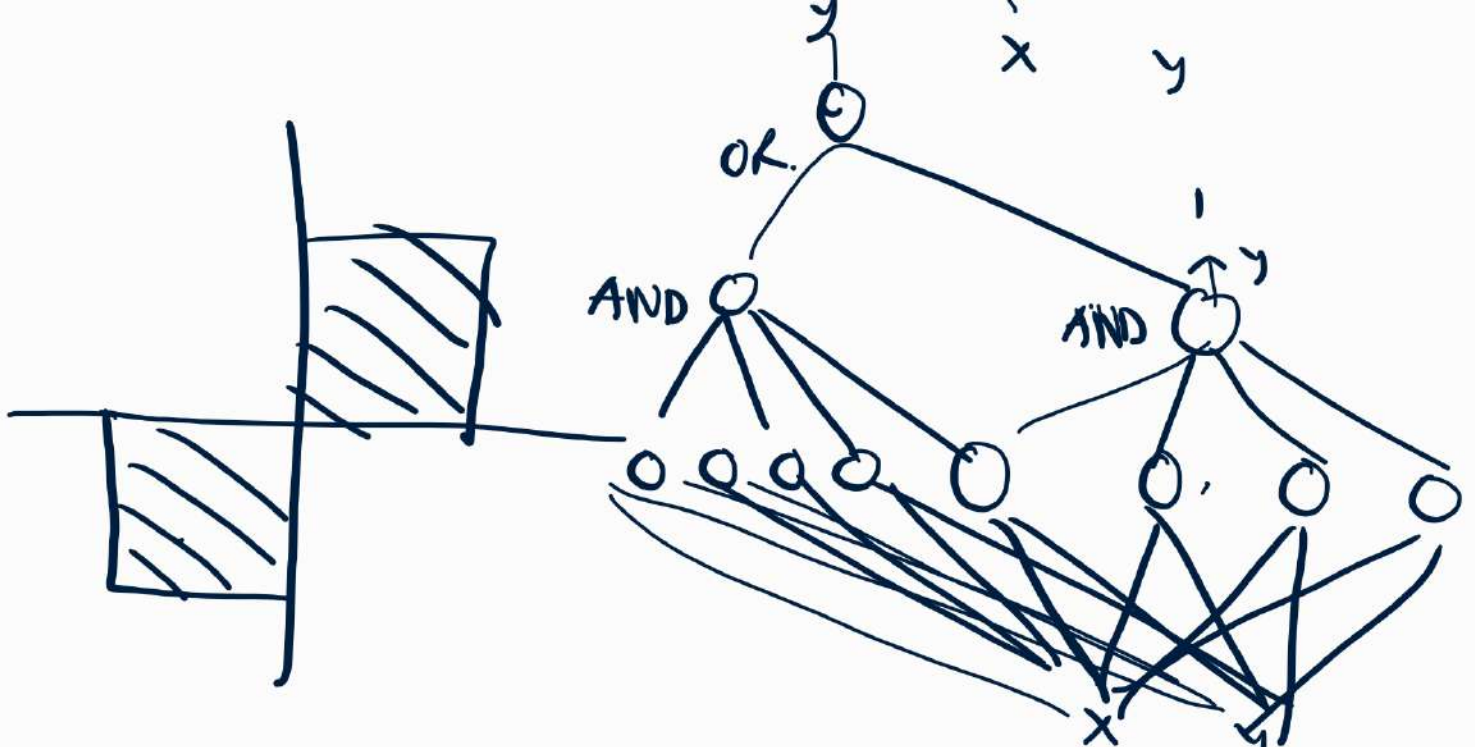
$$(2^n)$$



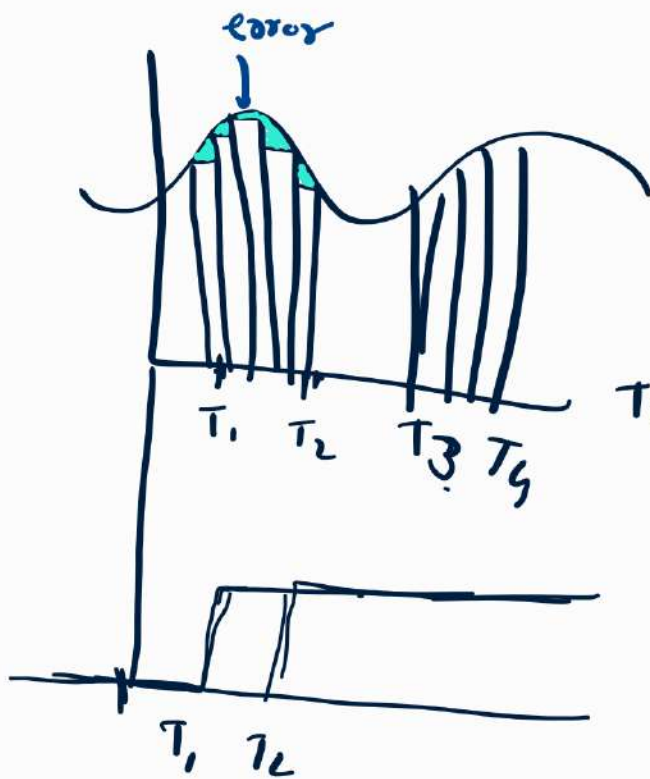
4
make perceptron which
align with 4 lines & fire
when it is 1



* Universal Approx Th^m?



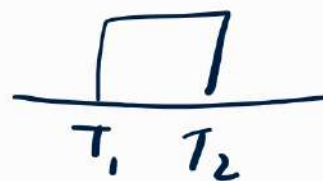
A Regression.



$$T_1 \rightarrow -\infty \rightarrow \infty$$

done

$$T_1 - T_2$$



T_1 fixes at thr. T_1

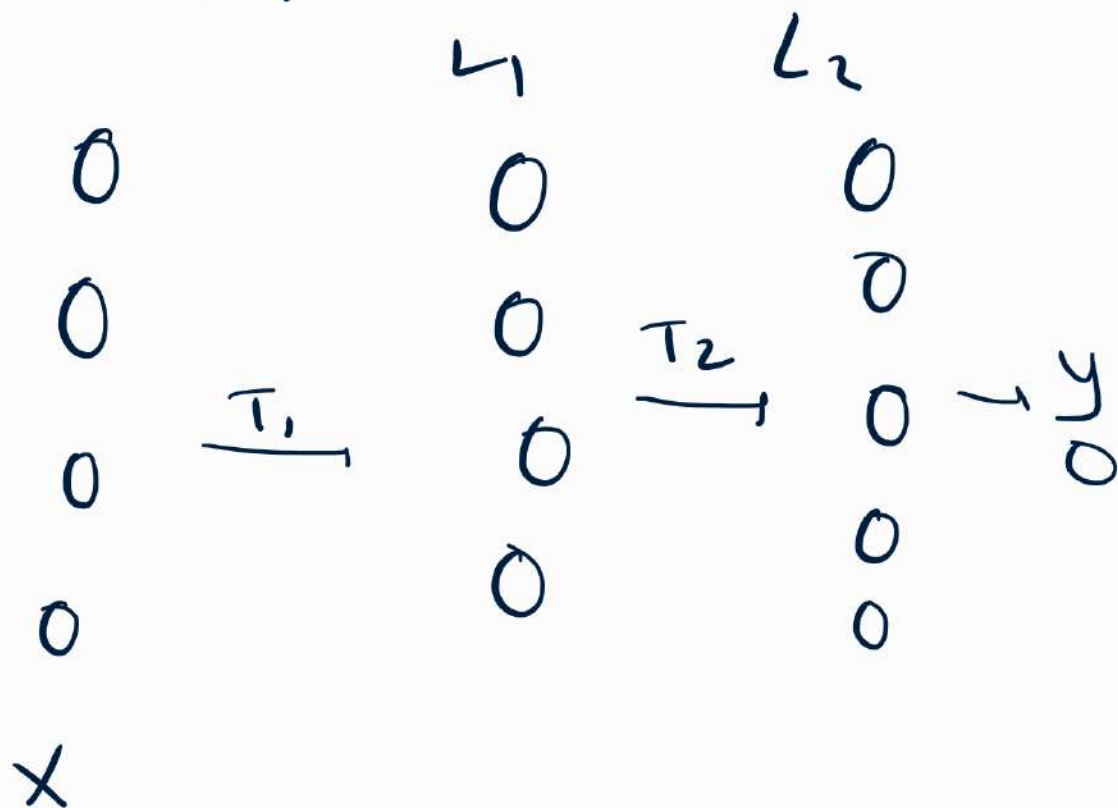
We can choose interval to reduce errors.

* HW no. of neurons for XOR

*

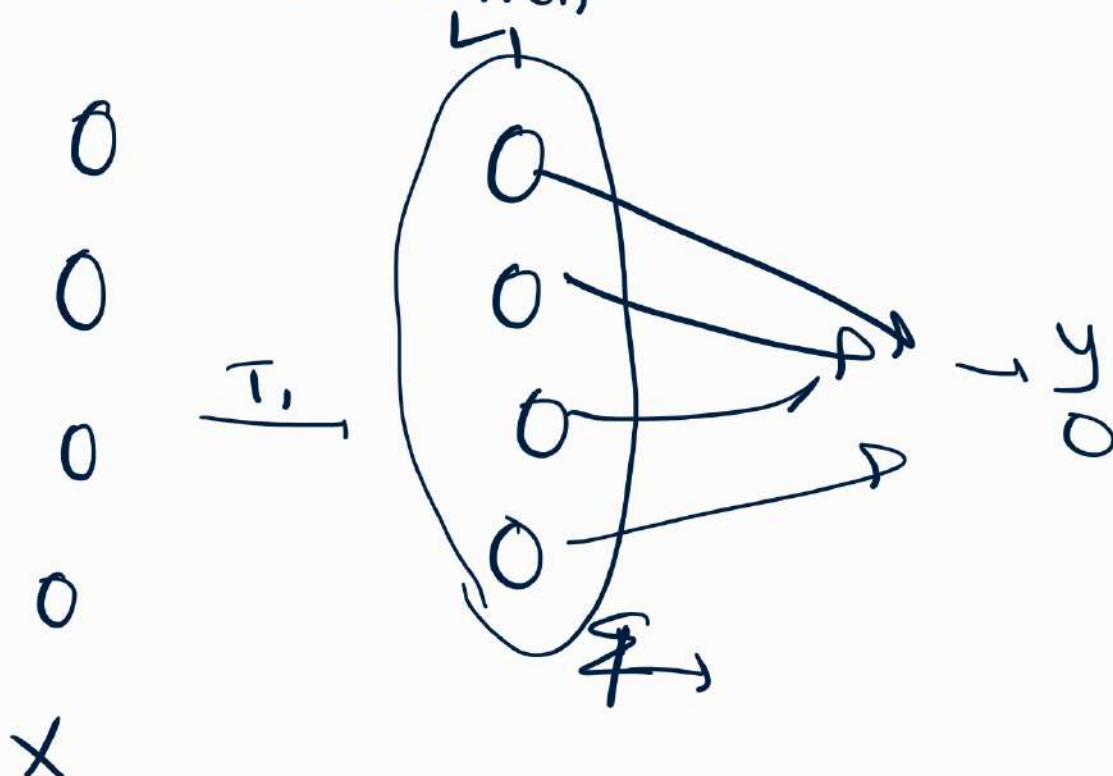
$$\begin{pmatrix} \bar{b} + w\bar{x} \end{pmatrix}$$

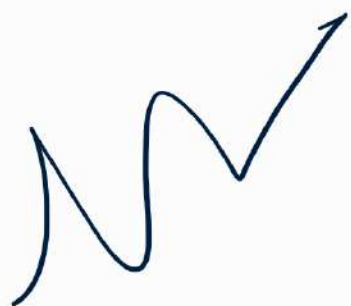
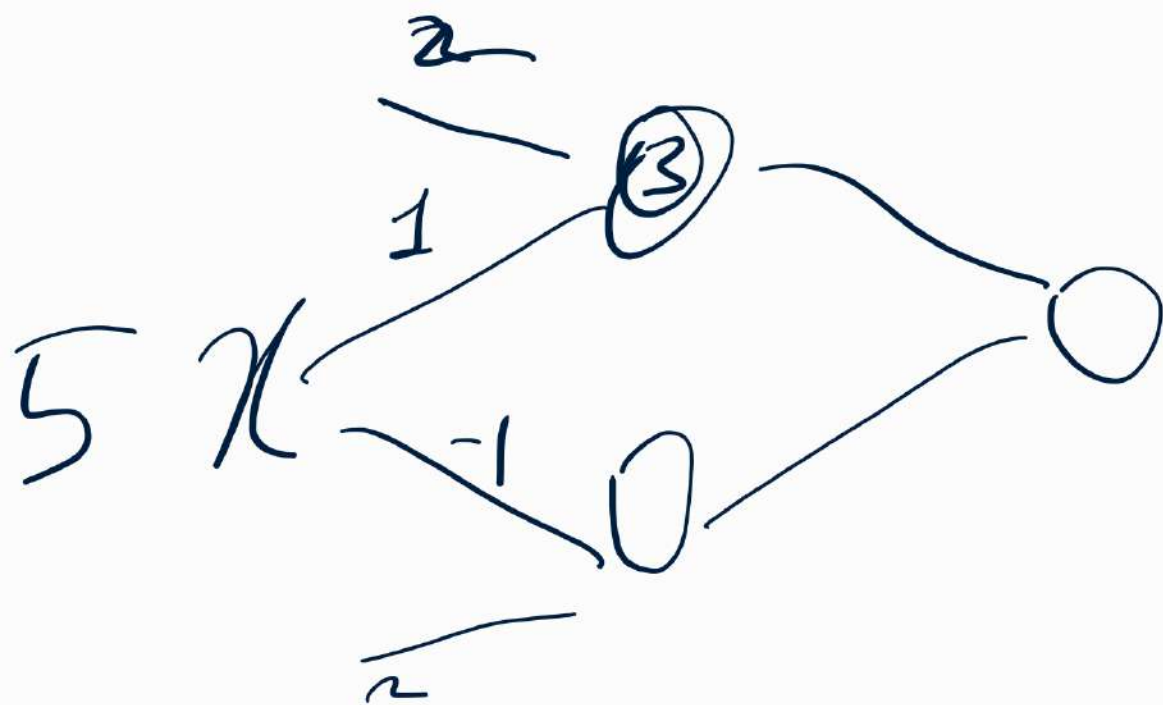
$\uparrow \quad \uparrow \quad \uparrow$
 $\delta \times 1 \quad \delta + 1 \quad 1 + 1$



Anything beyond 2 layers in Deep NN.

* Universal Approximation





* 16/11/25

Gradient Descent.

good f^n f^* from F

find goodness of f^n f .

loss : $F \times L = R$.

L is increase with wrongness of f on Z .

Possible loss f^n :-

Regression
classification.

...

f with small risk $R(f) = E_Z(l(f, z))$

$$f^* = \underset{f \in F}{\operatorname{argmin}} R(f)$$

$$\hat{R}(f; D) = \hat{E}_D(l(f, z)) = \frac{1}{N} \sum_{i=1}^N l(f, z_i).$$

$$w^* = \underset{w}{\operatorname{argmin}} L(w) \rightarrow \text{optimize loss function.}$$

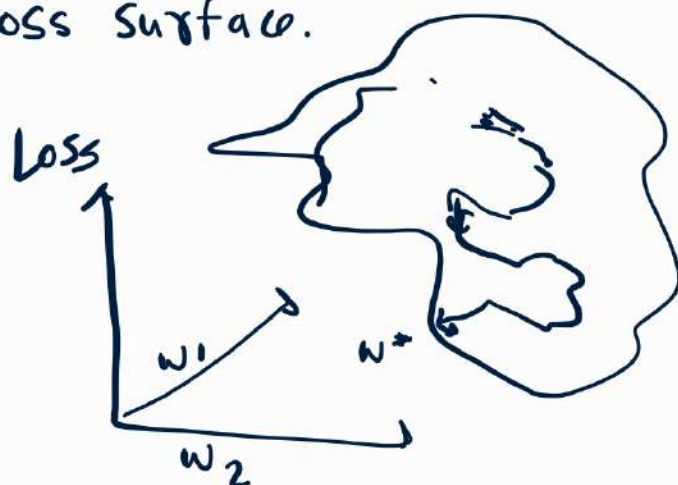
$$w^* b^* = \underset{w, b}{\operatorname{argmin}} L(f(w, b), D)$$

How to get optimal parameters.

linear regression.

Adaboost (KNN)

* Loss Surface.



Probe Random direction

Progress if you find useful direction

Repeat.

Optimal looking :-

Sense the slope around feet

Identify steepest dirⁿ & make brief progress

Repeat until convergence.

That is gradient descent.

Derivative of f^n at point gives rate of change of function at that point.

$$\frac{df}{dx} = \lim_{\delta \rightarrow 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

$$\Delta f = \frac{df}{dx} \Delta x.$$

$$f: \mathbb{R}^D \rightarrow \mathbb{R}.$$

$$\nabla f = \mathbb{R}^D \rightarrow \mathbb{R}^D \quad x \mapsto \left(\frac{df}{dx_1}, \dots, \frac{df}{dx_D} \right)$$

* ∇f vector gives dirⁿ & rate of fastest increase for f .

$$\Delta f = \nabla f \cdot \Delta x. \text{ (dot product)}$$

$$\begin{aligned} L(w+u) &= L(w) + \nabla_w L(w) \cdot u + \frac{1}{2!} u^T \nabla^2 L(w) u + \dots \\ &= L(w) + \nabla_w L(w) \cdot u. \end{aligned}$$

for $L(w+u)$ lesser than $L(w)$, need $\nabla_w L(w) \cdot u$
Goal is minimize loss.

$$\frac{\delta f}{\delta x} = \lim_{\delta \rightarrow 0} \frac{f(x+\delta) - f(x)}{\delta}$$

* Batch Gr. descent

for i in range (nb-epoch)

$$\nabla L_w = \text{evaluate_grads}(L, D, v)$$

$$w = w - \eta * \nabla L_w.$$

Guaranteed to converge global minima in case of convex fn & local minima in case of non convex function.

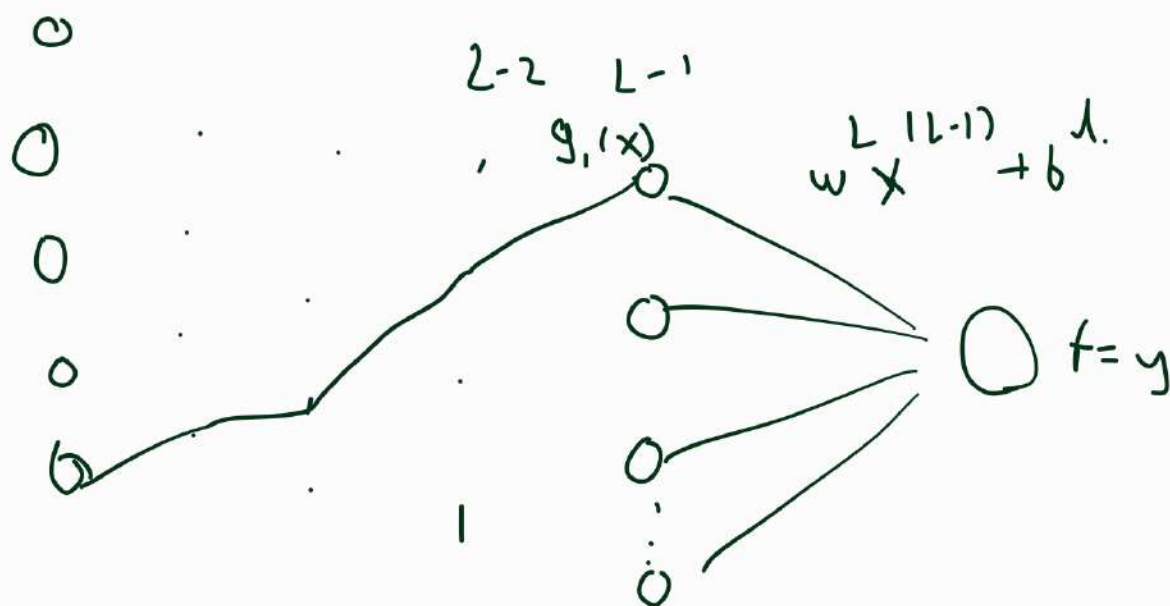
* Stochastic gradient descent.

$$w = w - \eta \nabla_w L(w, x^i, y^i)$$

* mini batch

$$w = w - \eta \nabla_w \mathcal{L}(w, x^{i:i+n}, y^{i:i+n})$$

* Backpropagation



$$y = f(g_1(x), g_2(x), \dots, g_m(x))$$

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g_1(x)} \frac{\partial g_1(x)}{\partial x} + \dots + \frac{\partial f}{\partial g_m(x)} \frac{\partial g_m(x)}{\partial x}$$

$$f(x) = e^{\sin(x^2)}$$

$$= e^{\sin(x^2)} \cdot \cos(x^2) \cdot 2x$$

22/1/25

$$\frac{\partial \mathcal{L}}{\partial s_i^{(l)}} = \frac{\partial \mathcal{L}}{\partial s^{(l)}} \cdot \frac{\partial s^{(l)}}{\partial x_i^{(l)}}$$

$$x_i = \sigma(s_i)$$

$$= \frac{\partial \mathcal{L}}{\partial x_i} \sigma'(s_i^{(l)})$$

$$s_i = \sum_j w_{ij}^l x_j^{l-1} \rightarrow b_i^{(l)}$$

$$P(y_i | x) = \frac{e^{x_i}}{\sum_{i=1}^N e^{x_i}}$$

Softmax Non linearity.

$$x_i = \frac{e^{s_i}}{\sum_i^N e^{s_i}}$$

$$\frac{\partial x_i}{\partial s_j} = 0 \quad \text{for hidden layer}$$

$i \neq j$

For softmax o/p layer this not valid.

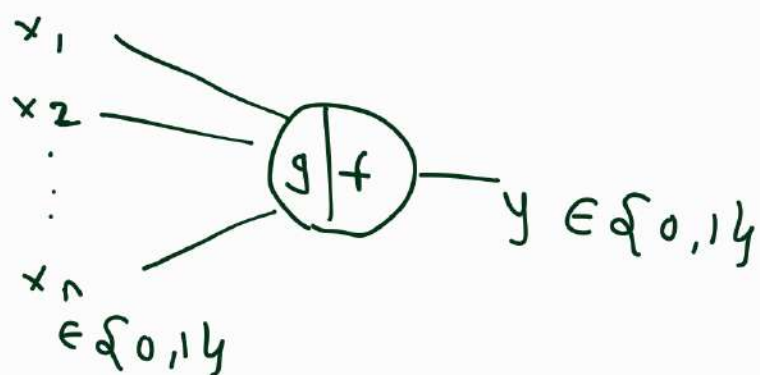
* NOTES FOR EXAM (24/11/25)

The Neuron.

Threshold Logic unit \rightarrow 1st math. model for neuron.

MP neuron.

Boolean i/p.

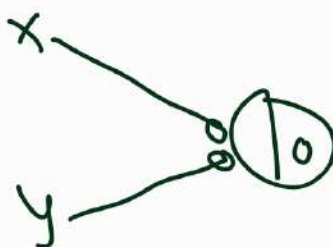
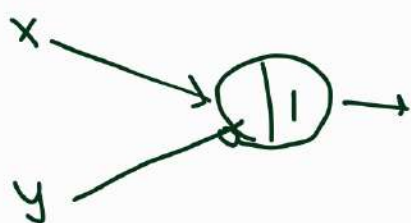


$$f(x) = 1 \left(\sum_i x_i \geq \theta \right)$$

I/P can be excitatory & Inhibitory
↑
more likely fire ↑
Reduce Likelihood.

- When Inhibitory i/p set (1) then o/p \rightarrow 0
- Count no. of ON signals on excit. i/p vs Inhib.

* xy'



* Learn linear Separation.

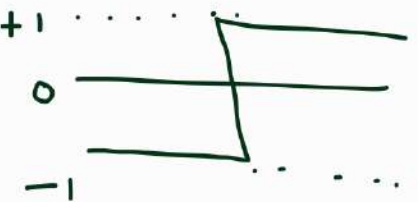
- No learning.

* Perceptron.

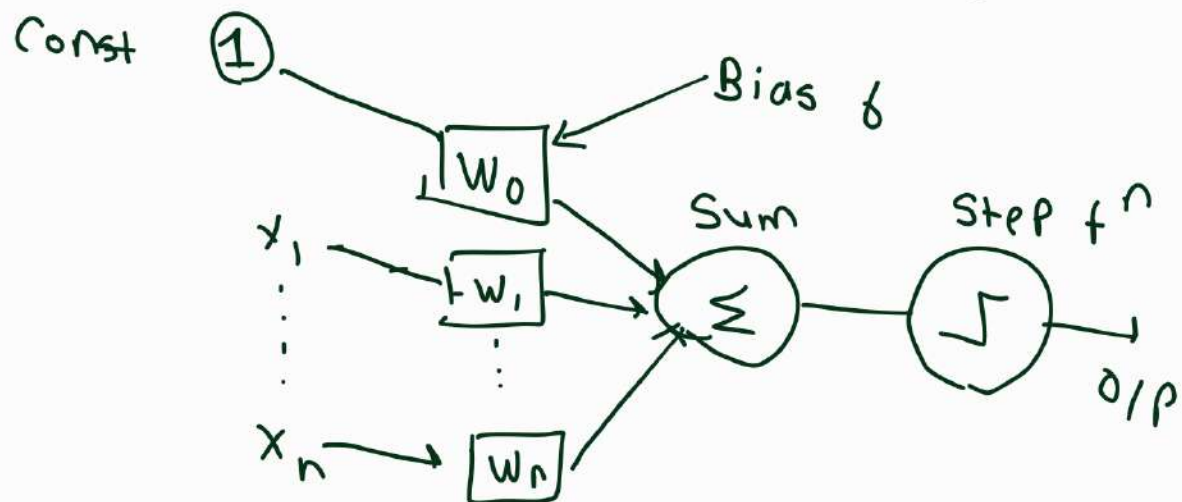
- Same like MP.
- Input can be real,
- weight can be diff. for diff. I/P.

$$f(x) = \begin{cases} 1 & \text{when } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{else.} \end{cases}$$

- Sigmoid $\rightarrow \begin{cases} 1 & \text{when } x \geq 0 \\ -1 & \text{else} \end{cases}$



- $f(x) = \sigma(w^T x + b)$ - - Activation function.
- Mechanism for learning weights.



- Training Data. $(x^i, y^i) \in \{-1, 1\}$.
Start $k = 1$ & $w_k = 0$
while $\exists i \in \{1 \dots N\}$ s.t. $y^i (w_k^T \cdot x^i) \leq 0$.
update $w_{k+1} = w_k + y^i x^i$
 $k \leftarrow k+1$.
- Bias can be appended with 1 suitably.

* Convergence Result.

- For linear sep. dataset algo converges after finite iterations.
- Stop as soon as find boundary.

* Perceptron learning code.

```
import torch.  
D1 = torch.normal ( 0.0 , 1.0 , size [100,2])  
D2 = torch.normal ( 3.0 , 1.0 , size=[100,2])  
# Generate 100 data point from 2 D with mean  
(0 & 3) & S.D of 1 & store in D1, D2.  
plt.scatter ( D1[:,0], D1[:,1] )  
plt.scatter ( D2[:,0], D2[:,1] )  
X = torch.empty (200, D1.size (1)+1, fill_(1))  
X[:, :2, :2] = D1  
X[1::2, :2] = D2  
print (X.shape)  
y = torch.empty (200, 1)  
y[:, :2, :2] = 1.0  
y[1::2, :2] = -1.0  
print (y.shape)  
# [200,3] [200,1]  
(create x with (200,3) fill with 1  
Store 1/p for perceptron.  
Extra Dim for Bias.
```

Even row D_1 } Assign.
Odd row D_2

y empty \rightarrow labels.

even $\rightarrow 1$.

odd $\rightarrow -1$.

$W = \text{torch.zeros}(X.\text{size}(1))$... initialize W by 0.

```
def train_perception(x, y, w, epoch)
```

```
    for e in range(epoch)
```

```
        change = 0
```

```
        for (i = 0  $\rightarrow$  x.size(0))
```

```
            if  $x[i].\text{dot}(w) * y[i] <= 0$  ...  $w \cdot x + b \neq 0$ 
```

```
                 $w = w + y[i] * x[i]$ 
```

```
                change = change + 1
```

$w = w + x \cdot y$

```
            if change == 0
```

```
                break # early stop.
```

```
    print("No. of changes.", change).
```

```
    return w.
```

```
w = train_perception(x, y, w, 5)
```

```
print(w)
```

* DL-2.

Linear classifiers.

Sometime data is linearly Seperable

Sometime not

Sometime specific preprocessing can make it Seperable.

In XOR.

$$\phi(x) = \phi(x_u, x_v) = (x_u, x_v, x_u x_v)$$

- make new dimension then you found the Seperable plane.
- Increasing dimension (degree) increase capacity.
- Reduce Bias increase capacity.
- preprocessing is also may way to reduce capacity.

XOR:.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

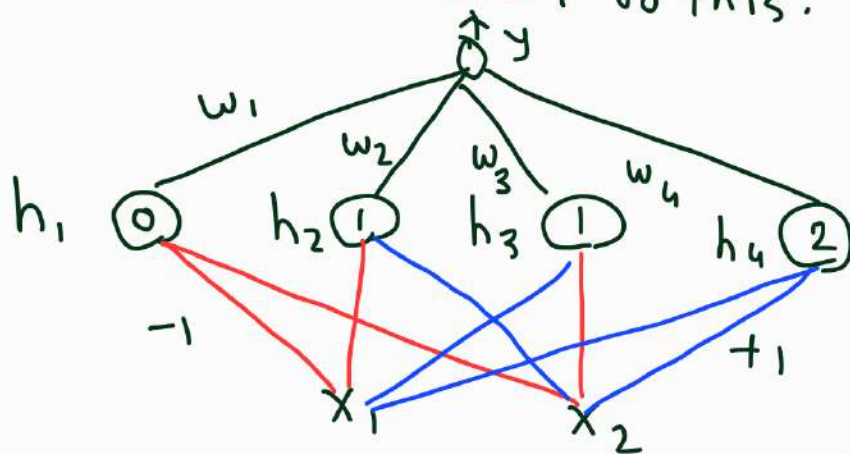
$$w_0 < 0$$

$$w_2 + w_0 \geq 0 = w_2 \geq -w_0$$

$$w_1 + w_0 \geq 0 = w_1 \geq -w_0$$

$$w_1 + w_2 + w_0 \leq 0 = w_1 + w_2 \leq -w_0$$

Single perceptron can't do this.



$$w_1, w_2, w_3, w_4$$

$$(0,0) = (-1) \cdot 0 + (+1) \cdot 0.$$

$$= (h_1 = 0) \quad h_2 = 0 \quad h_3 = 0 \quad h_4 = 0$$

$$(0,1) = -1, \textcircled{1}, -1, 1$$

↑
this fire.

x_1	x_2	h_1	h_2	h_3	h_4	y	XOR
0	0	1	0	0	0	$w_1 + w_0$	0
0	1	0	1	0	0	$w_2 + w_0$	1
1	0	0	0	1	0	$w_3 + w_0$	1
1	1	0	0	0	1	$w_4 + w_0$	0

$$w_1 \leq -w_0 \quad w_2 \geq -w_0 \quad w_3 \geq -w_0$$

$$w_4 \leq -w_0$$

Possible Find this weights

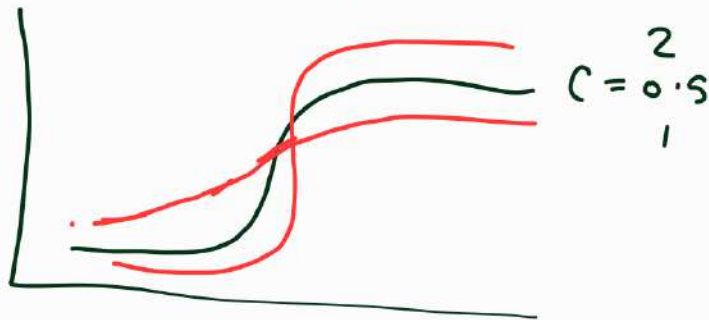
* Other 2 i/p boolean f^n .

- Can possible with 2^n perceptron in hidden layer & 1 in o/p. layer.
- $2^n + 1$ is sufficient but not necessary
- MLP (multi layer N/w of perceptron)

- * Many real world problem have non-binary o/p.
Perceptron only gives 2 o/p.

* Sigmoid neuron.

$$f(x) = \frac{1}{1 + e^{-wTx}}$$



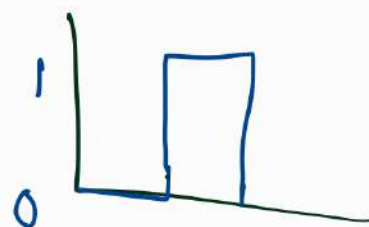
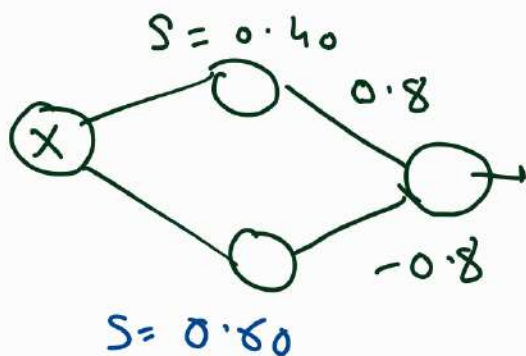
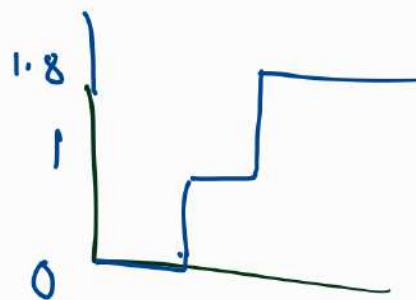
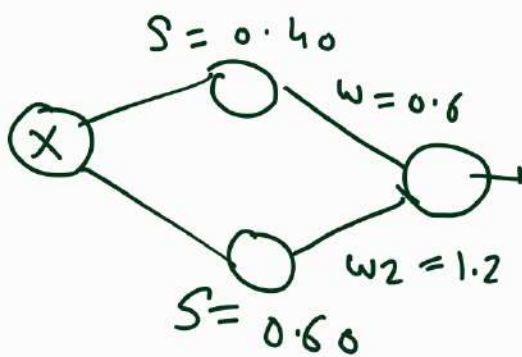
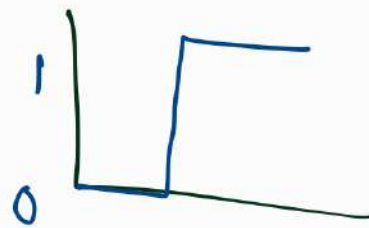
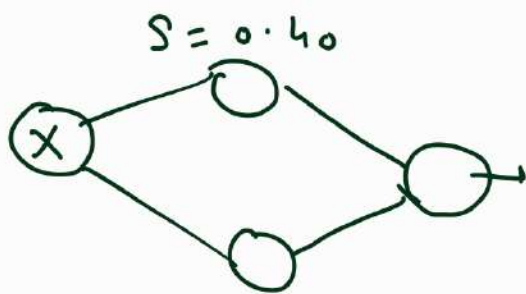
- * Any Boolean function with n i/p represent with one hidden layer.
- We can represent any continuous f^n to any approximation with linear comb. of sigmoid neurons.
- Neural N/w with single hidden layer used to approx any cont. f^n to any desired precision.
- Using Kmap we can have 2^{N-1} perc. in hidden layers.
- $3(N-1)$ required in deep N/w. Linear in N . arranged in $2 \log_2(N)$ layers.

* DL-03

* Uniform Approximation Th^m.

Using simple model (with 1 hidden layer) you can approximate any continuous function as closely as you want.

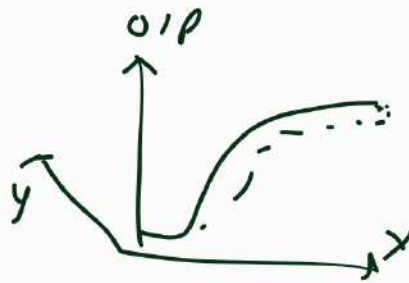
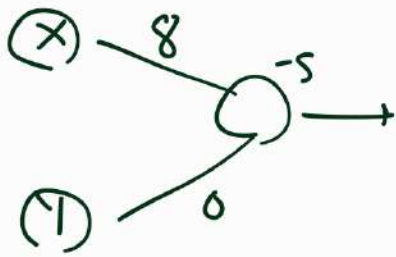
- Sigmoid neuron is taken in & process & produce out like S-curve .
- NN with one hidden layer can fit any continuous function.



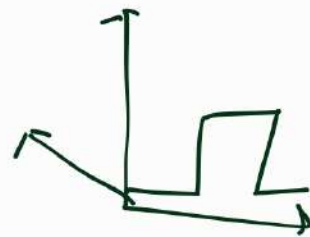
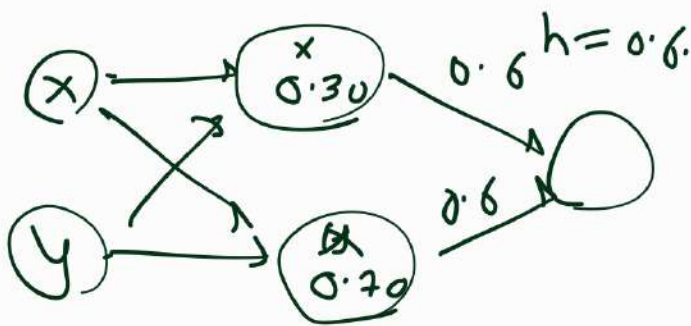
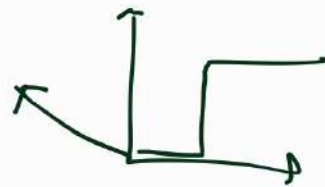
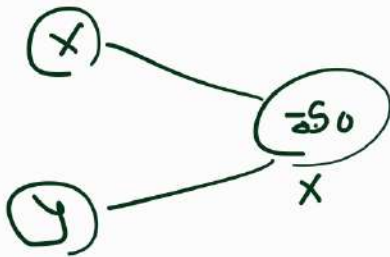
We only computed weighted sum of hidden outputs.

* When multiple inputs are introduced

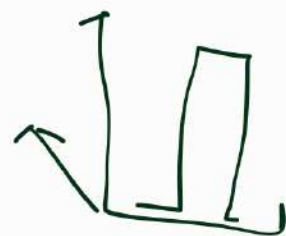
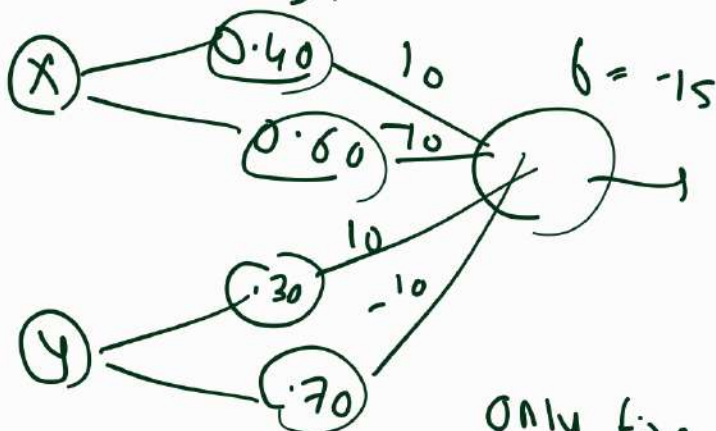
when 2 i/p we get 3 D plane.



$$S = \frac{-b}{w}$$



* Towers in 3D.



only fire when above 15

$$0.40 \rightarrow 0.60$$

- * Target f^* lie on space other than hypercube.
- Discontinuous target can be approximately arbitrary well
- σ is general as non poly f^n

- * If one layer is good then why we need deep N/w
 - may require inteseable size for hidden layers.
 - May not generalize well.
 - Not enable hierarchical learning.

* DL-4

Learning \rightarrow Finding f^* from F .

- For good function we use loss.
- $l : F \times Z \rightarrow R$.
- Such that value $l(f, z)$ increases with wrong of f on z . (diff. betⁿ expected & predicted)
- Regression: $l(f, (x, y)) = (f(x) - y)^2$
- classification: $l(f, (x, y)) = 1 (f(x) \neq y)$
- Density est.: $l(q, z) = -\log(q(z))$
- we want f with expected risk $R(f) = E_Z(l(f, z))$
- $f^* = \underset{f \in F}{\operatorname{argmin}} R(f)$
- Empirical Risk.

$$\hat{R}(f; D) = \hat{E}_D(l(f, z)) = \frac{1}{N} \sum_{i=1}^N l(f, z_n)$$

- model param. that minimize loss f^n .
- $w^* = \underset{w}{\operatorname{argmin}} L(w)$

$$- w^* b^* = \underset{w, b}{\operatorname{argmin}} (f(\cdot; w, b); \mathcal{D})$$

- How to find optimal parameters.
 - close form solution. (linear Reg.)
 - Ad hoc recipes (Perceptron, kNN classifier)

* How to find minimum loss surface.

- Probe random directions
- Progress if you find useful direction
- Repeat.
- Ineffective.

Better :- Follow the slope.

- Sense the slope around feet.
- Identify steepest dirⁿ, make brief progress
- Repeat.
- GRADIENT DESCENT.

* Derivative of f^n at given point gives rate of change of f^n at that point.

$$\frac{df}{dx} = \lim_{\delta \rightarrow 0} \frac{f(x+\delta) - f(x)}{\delta}$$

$$\Delta f = \frac{\partial f}{\partial x} \Delta x$$

in higher dimensions

$$\nabla f : \mathbb{R}^D \rightarrow \mathbb{R}^D$$

$$x \rightarrow \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_D} \right) \quad \dots \text{ gives dir}^n \& \text{ rate of inc. for } f.$$

$$\Delta f = \nabla f \cdot \Delta x.$$

$$\mathcal{L}(w+u) = \mathcal{L}(w) + \nabla_w \mathcal{L}(w) \cdot u$$

$\mathcal{L}(w) \rightarrow$ loss at w

To minimize loss $(\mathcal{L}(w+u) < \mathcal{L}(w))$: $\nabla_w \mathcal{L}(w) \rightarrow$ gradient of loss at w .

$u \rightarrow$ step

$$\nabla_w \mathcal{L}(w) \cdot u < 0.$$

- diff. would be less if u is in opp. dirⁿ of $\nabla_w \mathcal{L}(w)$, the gradient.

$\nabla^2 \mathcal{L}(w) \rightarrow$ Hessian matrix
(2nd der.)

* GRADIENT DESCENT

- Goal to minimize the error.
- Determine parameters w that minimize $\mathcal{L}(w)$
- Gradient points uphill \rightarrow -ve of grad. points downhill.

* Start with arbitrary initial parameter w_0 .

- Repeatedly modify it via updating in small steps
- At each step modify dirⁿ that produce steepest descent along with error

* Numerically for each w .

$$\frac{\partial f}{\partial x} = \lim_{\delta \rightarrow 0} \frac{f(x+\delta) - f(x)}{\delta}$$

Slow & approximate.

* Analytical methods.

- we use this as numerical method is slow & approx
- Analytical is exact & precise (calculas) use to calculation in backprop.

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \rightarrow \text{hinge loss}$$

$$L = \frac{1}{N} \sum_i L_i + \sum_k w_k^2 \rightarrow \text{reg. loss}$$

$$s = f(x, w)$$

$$\nabla L_w = \nabla \text{Hingeloss} + \nabla \text{Regularization.}$$

$$\begin{cases} \frac{\partial s_j}{\partial w} - \frac{\partial s_{y_i}}{\partial w} + 2\lambda w. & \text{if } s_j - s_{y_i} + 1 > 0 \\ 0 & \text{otherwise} \end{cases} + 2\lambda w.$$

* Batch Gradient descent.

for i in range(nb_epochs):

$$\nabla L_w = \text{evaluate_gradient}(L, D, w)$$

$$w = w - \eta * \nabla L_w$$

- If 1000s of convex function you guaranteed get global minima.

* Stochastic Gr. De.

weight update parameter for each training example

$$w = w - \eta \nabla_w L(w, x_i, y_i)$$

- In case of large DS Batch GD compute redundant gradients for similar example.
- SGD does away with redundancy & faster can be used to learn online.
- frequent updates with high variance cause objective f^* to fluctuate heavily.
- SGD fluctuation enable it to jump now & better local minima.
- Complicate convergence.

```
for i in range(ep_epochs)
```

```
    np.random.shuffle(D)
```

```
    for  $x_i \in D$ 
```

```
         $\nabla L_w = \text{evaluate\_gradient}(L, x_i, w)$ 
```

```
         $w = w - \eta * \nabla L_w$ 
```

* Mini Batch Gr. De.

Best of both & update param every mini batch.

$$w = w - \eta \nabla_w L(w, x_{i:i+n}, y_{i:i+n})$$

Reduce variance & param. updates

used highly optimized matrix optimization.

Batch size 32 \rightarrow 1024.


```

for i in range (NB epoch)
    np.random.shuffle(D)
    for batch in get_batches(D, batchsize=128)
         $\nabla w = \text{evaluate\_gradient}(L, \text{batch}, w)$ 
         $w = w - \eta * \nabla w$ 

```

* Some challenges

- choose proper learning rate
- make learning rate applies to all param.
- Avoid numerous sub-optimal local minima.

* DL-5

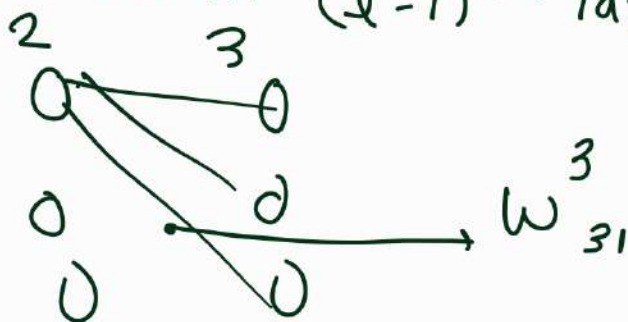
Gradient of scalar f^l $f(x) = x \rightarrow \left[\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n} \right]^T$.

Gradient of vector valued f^l called Jac.

$$J = \left[\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n} \right] = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

w_{jk}^l

connecting j^{th} neuron in l^{th} layer & k^{th} neuron in $(l-1)^{\text{th}}$ layer.



b_j^l bias of j^{th} neuron in l^{th} layer.

x_j^{l-1} op of j^{th} neuron in l^{th} layer.

$$x_j^l = \sigma \left(\sum_k w_{jk}^l x_k^{l-1} + b_j^l \right)$$

* vector of bias in layer $l \rightarrow b^l$.

$w^l \rightarrow$ weight vector of layer l .

s_j^l weighted i/p for j^{th} neuron in l^{th} layer.

$$s_j^l = \sum_k w_{jk}^l x_k^{l-1} + b_j^l$$

$$s^l = w^l x^{l-1} + b^l.$$

σ is activation fⁿ.

$$\begin{aligned} \star \text{Loss } \ell(w, b) &= \sum_n \ell(f(x_n; w, b), y_n) \\ &= \sum_n \ell(x^L, y_n). \end{aligned}$$

$$x^0 = x \xrightarrow{w^{(1)}, b^{(1)}} s^{(1)} \xrightarrow{\sigma} x^{(1)} \xrightarrow{w^{(2)}, b^{(2)}} s^{(2)} \rightarrow \dots \rightarrow x^L.$$

$$x^0 = x, f(x; w, b) = x^L.$$

$$\forall l = 1 \dots L \begin{cases} s^l = w^l x^{l-1} + b^l \\ x^l = \sigma(s^l) \end{cases}$$

\star Chain rule.

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x)$$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

$$y = f(g(x))$$

$$\frac{dy}{dx} = \frac{df}{d(g(x))} \cdot \frac{d(g(x))}{dx}.$$

$$\Delta y = \frac{dy}{dx} \cdot \Delta x.$$

$$z = g(x) \rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x.$$

$$y = f(z) \rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dz} \cdot \frac{dg(x)}{dx} \Delta x$$

$$= \frac{df}{d(g(x))} \frac{dg(x)}{dx} \Delta x.$$

$$y = f(g_1(x), g_2(x) \dots g_m(x))$$

$$\frac{dy}{dx} = \frac{df}{dg_1(x)} \frac{dg_1(x)}{dx} + \dots + \frac{df}{dg_m(x)} \frac{dg_m(x)}{dx}.$$

$$g_i(x) = z_i \rightarrow y = f(z_1, \dots, z_m)$$

$$\Delta y = \frac{\partial f}{\partial z_1} \Delta z_1 + \frac{\partial f}{\partial z_2} \Delta z_2 + \dots + \frac{\partial f}{\partial z_m} \Delta z_m$$

$$\Delta y = \frac{\partial f}{\partial z_1} \frac{dz_1}{dx} \Delta x + \frac{\partial f}{\partial z_2} \frac{dz_2}{dx} \Delta x + \dots$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} \Delta x + \dots$$

$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} \dots \right) \Delta x.$$

$g = xy$
 $f = g + z$
 $f(x, y, z) = xy + z$

$L = (f - t)^2$ (for this)

$$\frac{\partial L}{\partial f} = 2(f - t) = -2$$

$\frac{\partial L}{\partial g} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} = (-2) \cdot (-2) = -2$

$f = g + z$

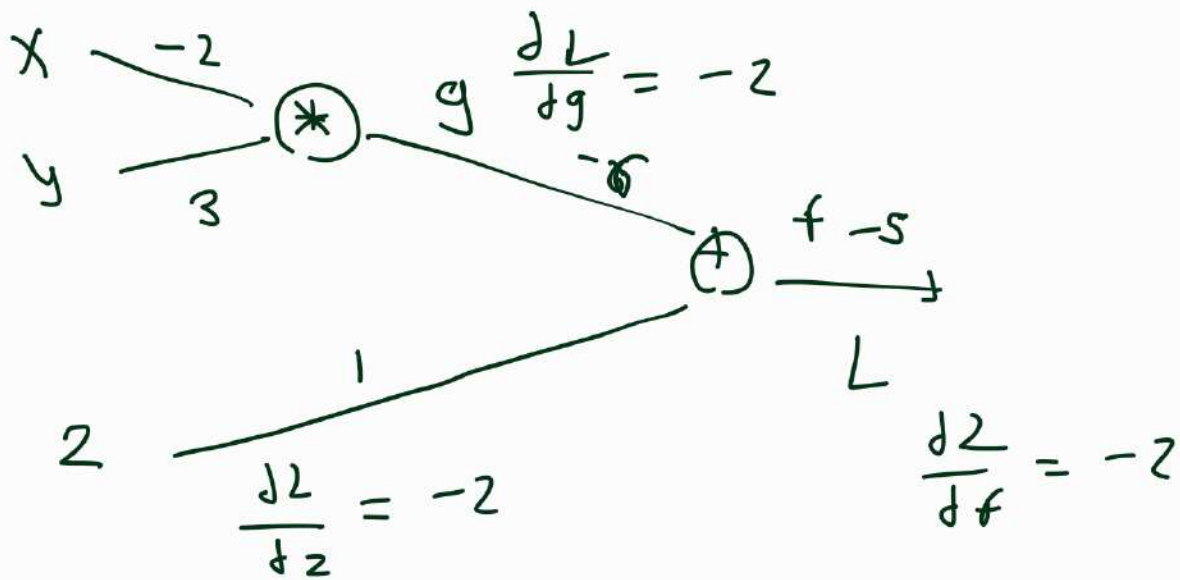
$$L \quad \frac{\partial L}{\partial f} = -2$$

$\frac{\partial L}{\partial g} = -2$

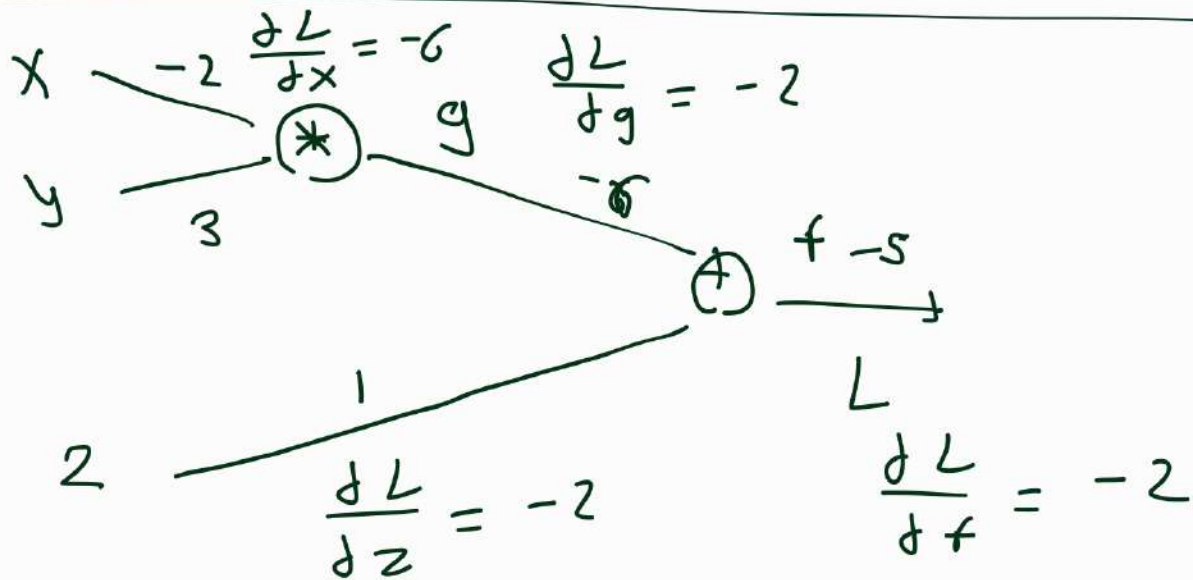
$L \quad \frac{\partial L}{\partial f} = -2$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial z} = (-2) \cdot 1 = -2$$

$f = g + z$



$$\begin{aligned} \frac{\partial L}{\partial x} &= \frac{\partial L}{\partial g} \cdot \frac{\partial g}{\partial x} \\ &= (-2) * y \\ &= (-6) \quad - (y = 3) \end{aligned} \quad g = xy$$



$$\begin{aligned} \frac{\partial L}{\partial y} &= \frac{\partial L}{\partial g} \cdot \frac{\partial g}{\partial y} \\ &= (-2) * x \\ &= \underline{\underline{4}} \end{aligned}$$

* Gradient Flow

Down
stream
grad.

$$\frac{dL}{dx} = \frac{dL}{dg} \frac{dg}{dx}$$

Upstream → as we take from forward pass
local → we calculate locally

Diagram: A circle with a cross inside, representing a node. An arrow labeled $\frac{dL}{dx}$ points to the circle from the left. An arrow labeled $\frac{dL}{dg}$ points away from the circle to the right. A line labeled y enters the circle from the bottom left. A line labeled g exits the circle to the bottom right.

$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1(x)} = J_{f_N}(f_{N-1}(\dots f_1(x))) \dots J_{f_{N-1}}(\dots f_1(x)) \dots J_{f_1}(x)$$

$J f(x)$ is Jacobian of f computed at x .

* Backpropagation.

Take any layer.

$$x^{(l-1)} \xrightarrow{w^{(l)}, b} s_i^l \xrightarrow{\sigma} x^l$$

$$x_i^l = \sigma(s_i^l)$$

s^l influence loss \mathcal{L} through $x^{(l)}$ only.

$$\frac{\partial \mathcal{L}}{\partial s_i^{(l)}} = \frac{\partial \mathcal{L}}{\partial x_i^{(l)}} \cdot \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

$$= \frac{\partial \mathcal{L}}{\partial x_i^{(l)}} \sigma'(s_i^l)$$

$$s_i^{(l)} = \sum_j w_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)}$$

Since $x^{(l-1)}$ influence loss \mathcal{L} only through $s^{(l)}$

$$\frac{\partial \mathcal{L}}{\partial x_j^{(l-1)}} = \sum_i \frac{\partial \mathcal{L}}{\partial s_i^{(l)}} \cdot \frac{\partial s_i^{(l)}}{\partial x_j^{(l-1)}}$$

$$= \sum_i \frac{\partial \mathcal{L}}{\partial s_i^{(l)}} \cdot w_{ij}^{(l)}$$

-- dev. of is w.

$(wx+b)$

w^l & b^l influence loss via s^l .

$$s_i^{(l)} = \sum_j w_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial s_i^{(l)}} \cdot \frac{\partial s_i^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial s_i^{(l)}} \cdot x_j^{(l-1)}$$

$$\frac{\partial \ell}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i} \frac{\partial s_i^{(l)}}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}}$$

* From definition of loss obtain $\frac{\partial \ell}{\partial x_i^{(l)}}$

Recursively compute loss dev. w/o activations.

$$\frac{\partial l}{\partial s_i(l)} = \frac{\partial l}{\partial x_i} \sigma'(s_i; l)$$

$$\frac{\partial \ell}{\partial x_j^{(l-1)}} = \sum_i \frac{\partial \ell}{\partial s_i^l} \cdot w_{ij}^l \quad \} \text{Gradient}$$

then wrt parameters.

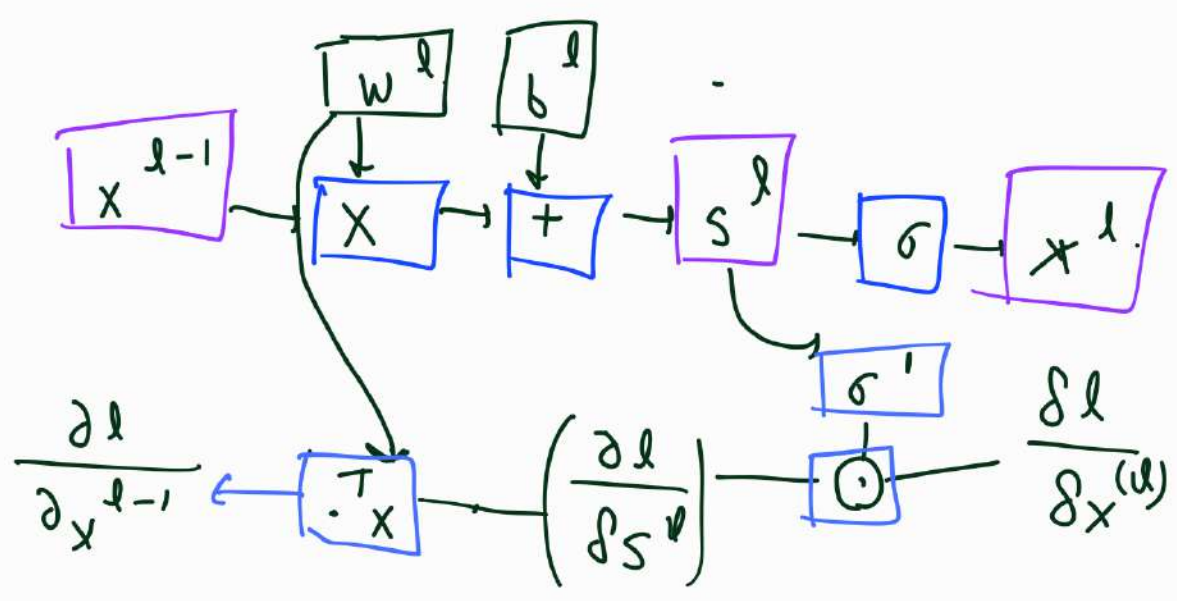
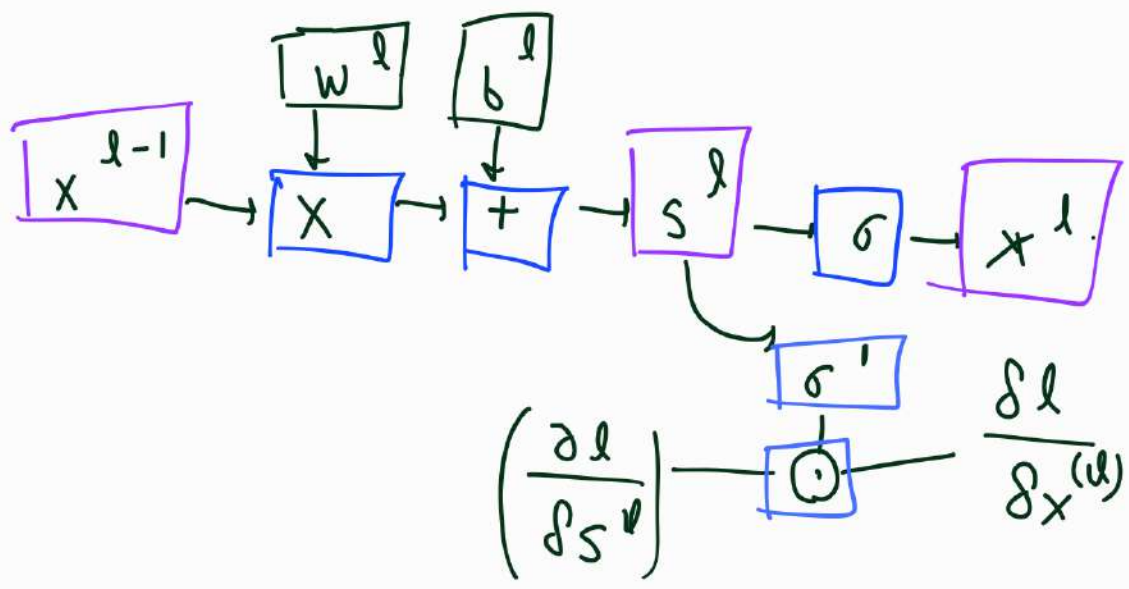
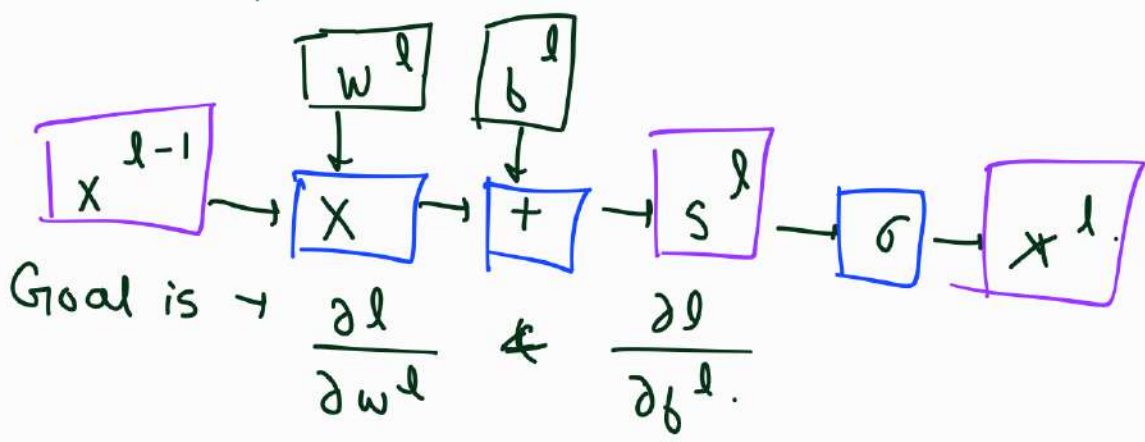
$$\frac{\partial \mathcal{L}}{\partial w_{ij}^l} = \frac{\partial \mathcal{L}}{\partial s_i^l} \cdot x_j^{(l-1)} \quad \frac{\partial \mathcal{L}}{\partial b_i^l} = \frac{\partial \mathcal{L}}{\partial s_i^l} \quad \left. \vphantom{\frac{\partial \mathcal{L}}{\partial s_i^l}} \right\} \text{Param.}$$

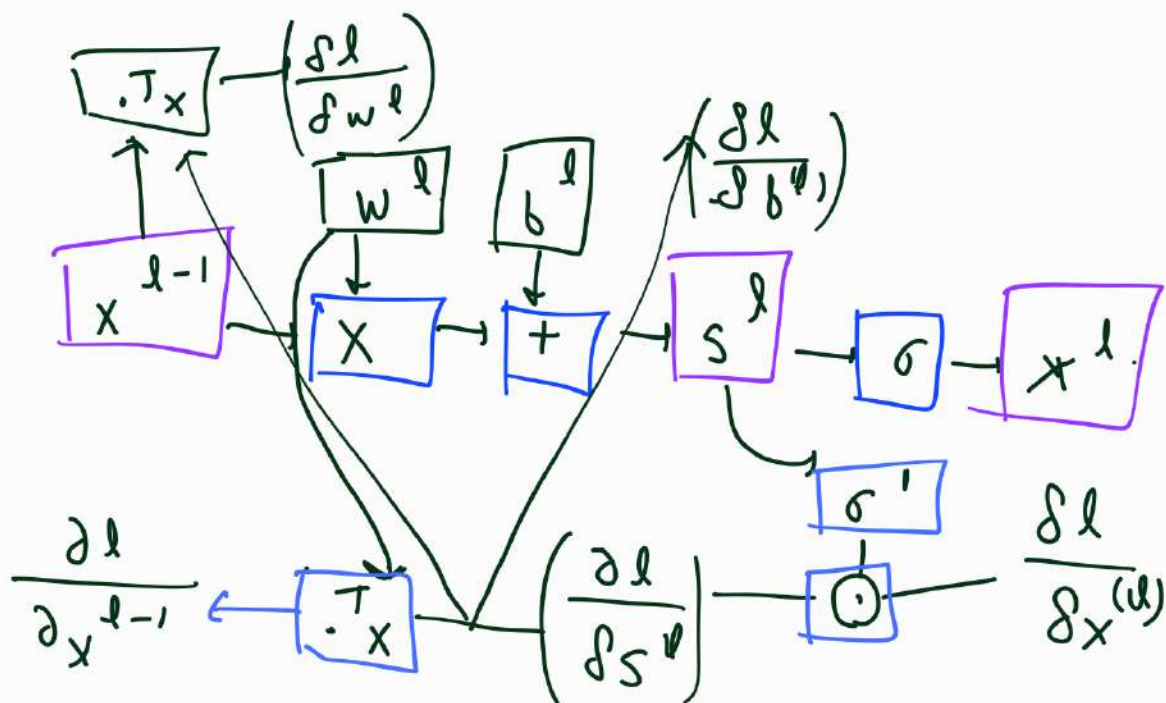
* Jacobian in Tensorial Form.

$$\Psi: \mathbb{R}^N \rightarrow \mathbb{R}^m \text{ then } \left[\frac{d\Psi}{dx} \right] = \begin{bmatrix} \frac{d\Psi_1}{dx_1} & \dots & \frac{d\Psi_1}{dx_N} \\ \vdots & \ddots & \vdots \\ \frac{d\Psi_m}{dx_1} & \dots & \frac{d\Psi_m}{dx_N} \end{bmatrix}$$

$$\psi \in \mathbb{R}^{N \times m} \rightarrow \mathbb{R}^{1 \times n} \left[\left[\frac{\partial \psi}{\partial x} \right] \right] = \begin{bmatrix} \frac{\partial \psi}{\partial w_{11}} & \dots & \frac{\partial \psi}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \psi}{\partial w_{N1}} & \dots & \frac{\partial \psi}{\partial w_{Nm}} \end{bmatrix}$$

Forward pass





update params

$$w^l = w^l - \eta \left[\left[\frac{\partial l}{\partial w^l} \right] \right] \quad \& \quad b^l = b^l - \eta \left[\frac{\partial l}{\partial b^l} \right]$$

BP is applying chain rule iteratively.

Expressed in tensorial form

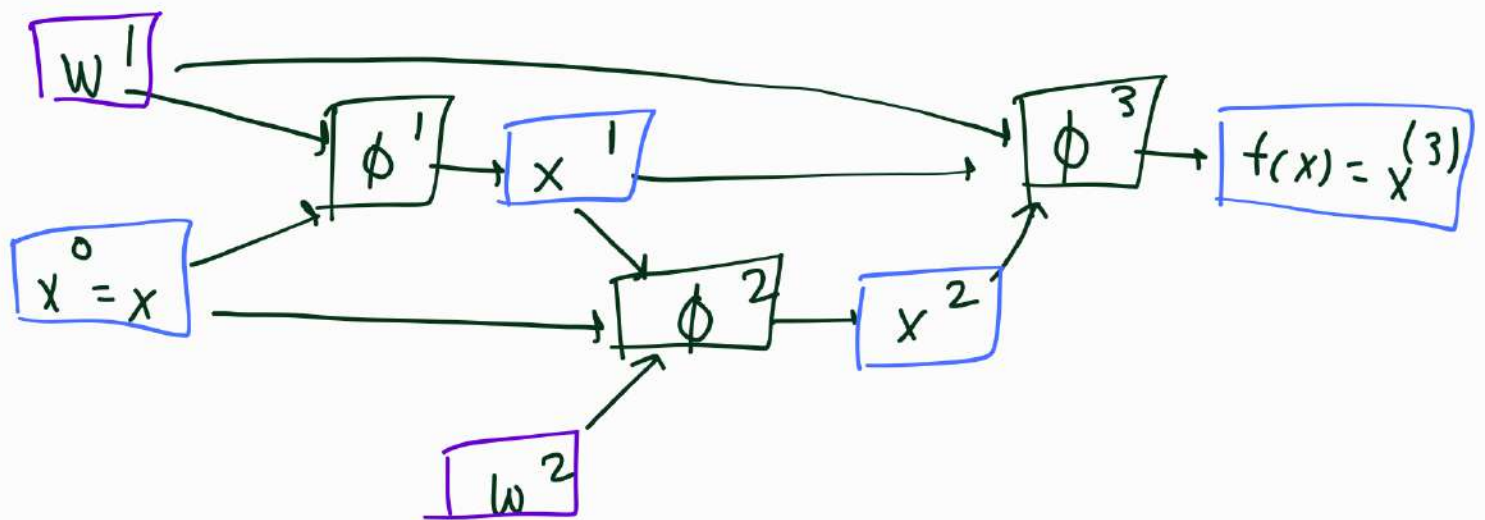
Heavy computation with linear ops.

Non-linear goes with simple element wise ops.

BP needs all int. layer result to be in memory

Take twice the computations of Fw pass.

* For an Example.



Forward pass

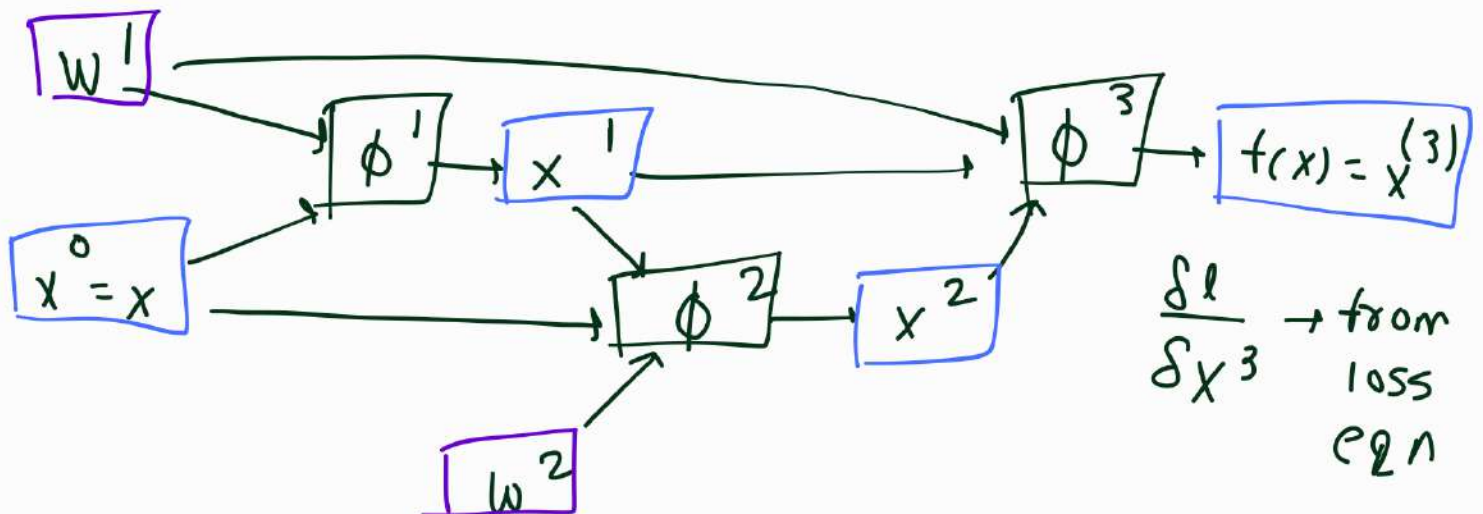
$$x^0 = x$$

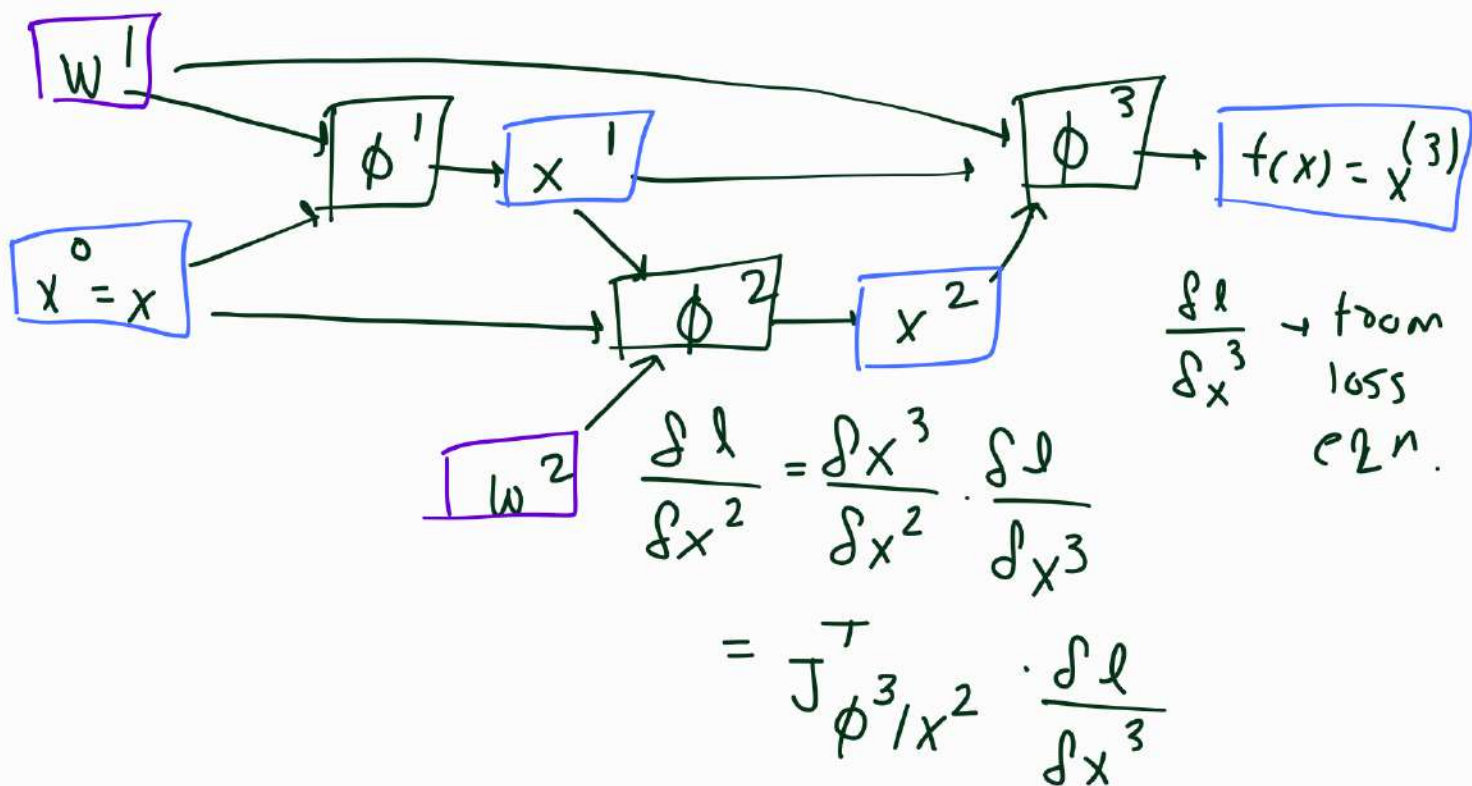
$$x^1 = \phi^1(x^0, w^1)$$

$$x^2 = \phi^2(x^0, x^1, w^2)$$

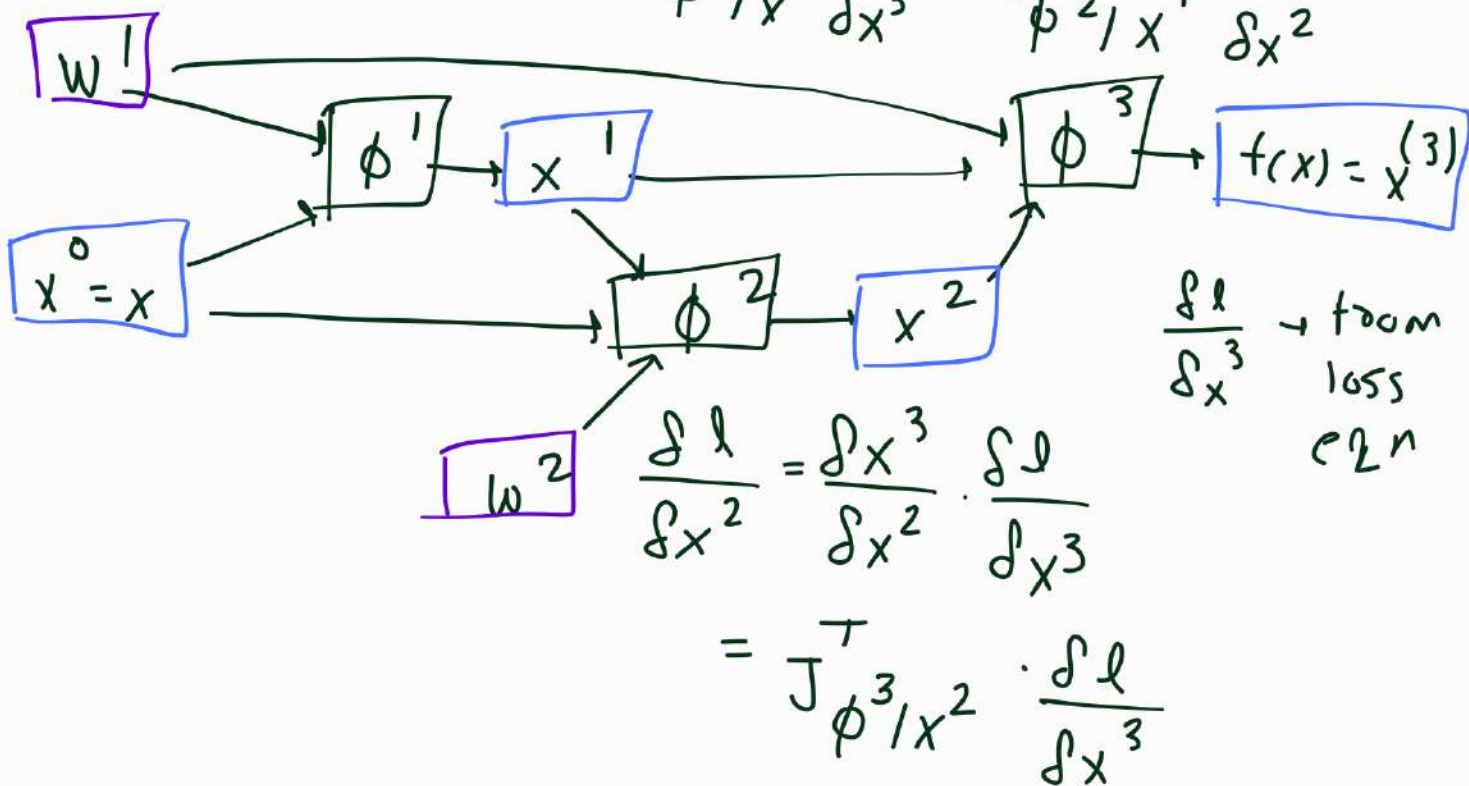
$$x^3 = \phi^3(x^1, x^2, w^1)$$

* Back propagation.

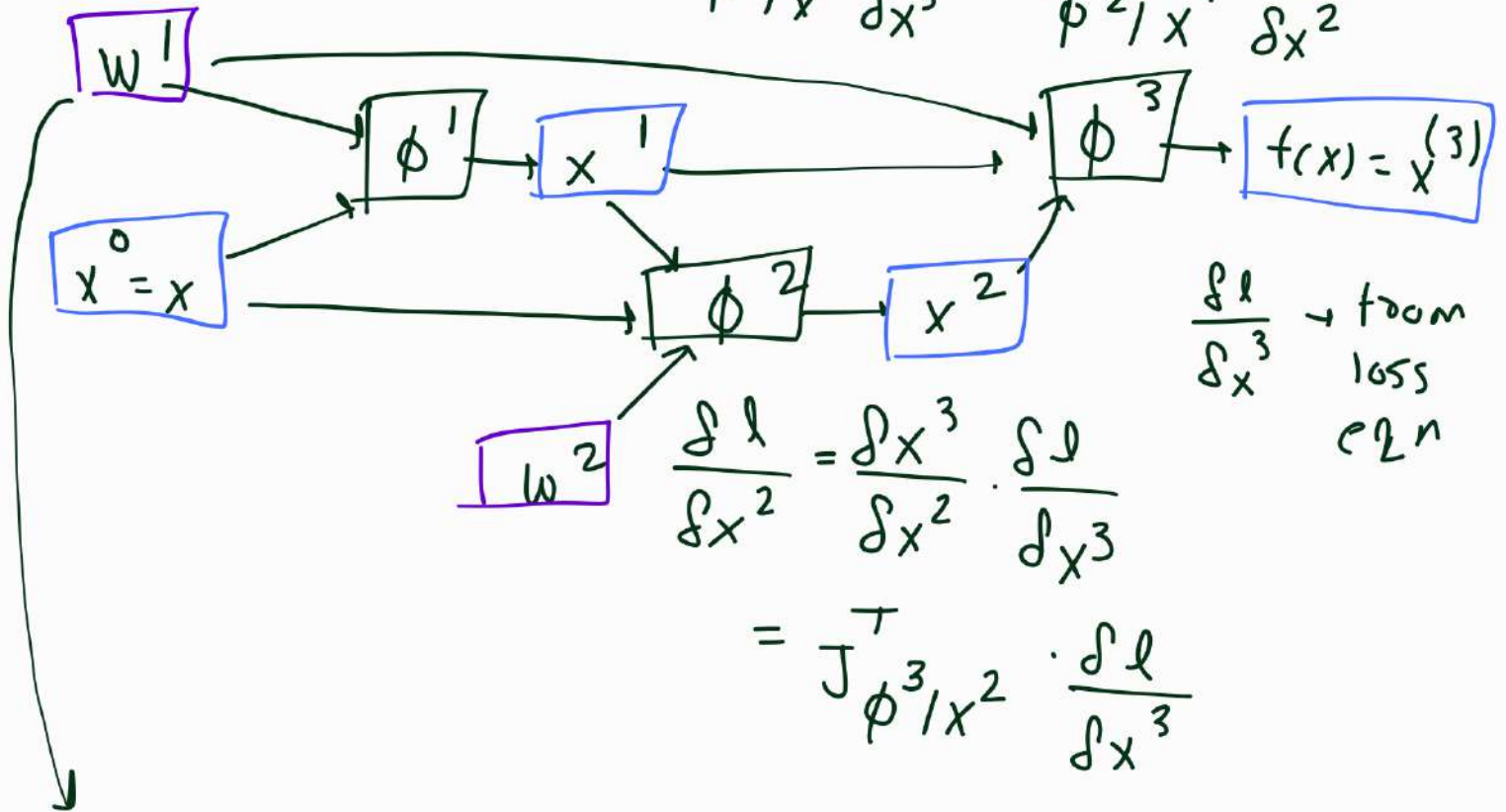




$$\begin{aligned}
 \frac{\partial l}{\partial x^1} &= \frac{\partial x^3}{\partial x^1} \cdot \frac{\partial l}{\partial x^3} + \frac{\partial x^2}{\partial x^1} \cdot \frac{\partial l}{\partial x^2} \\
 &= J_{\phi^3/x^1}^T \frac{\partial l}{\partial x^3} + J_{\phi^2/x^1}^T \frac{\partial l}{\partial x^2}
 \end{aligned}$$



$$\begin{aligned}\frac{\partial \ell}{\partial x^1} &= \frac{\partial x^3}{\partial x^1} \cdot \frac{\partial \ell}{\partial x^3} + \frac{\partial x^2}{\partial x^1} \cdot \frac{\partial \ell}{\partial x^2} \\ &= J_{\phi^3/x^1}^T \frac{\partial \ell}{\partial x^3} + J_{\phi^2/x^1}^T \frac{\partial \ell}{\partial x^2}\end{aligned}$$



$$\begin{aligned}\frac{\partial \ell}{\partial w^1} &= \frac{\partial x^3}{\partial w^1} \cdot \frac{\partial \ell}{\partial x^3} + \frac{\partial x^1}{\partial w^1} \cdot \frac{\partial \ell}{\partial x^1} \\ &= J_{\phi^3/w^1}^T \left(\frac{\partial \ell}{\partial x^3} \right) + J_{\phi^1/w^1}^T \left(\frac{\partial \ell}{\partial x^1} \right)\end{aligned}$$

$$\frac{\partial \ell}{\partial w^2} = \frac{\partial x^2}{\partial w^2} \cdot \frac{\partial \ell}{\partial x^2} = J_{\phi^2/w^2}^T \left(\frac{\partial \ell}{\partial x^2} \right)$$

New training samples may change BP minimally
Prefers consistency (low variance) over perfection (low Bias)
minimizing proxy may not minimize actual.

- Saddle points are more frequent than local minima.
- Most local minima are equivalent & close to global minima.