6/11/25

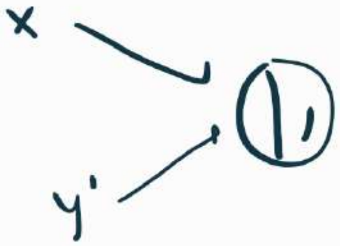
*Modified

- Goodwill hunting (movie)

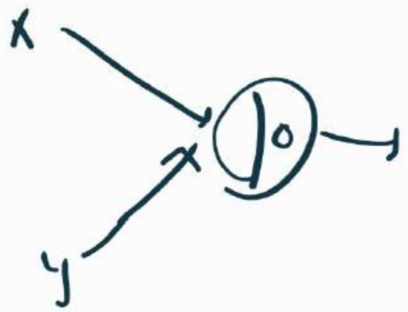
$$f(x) = 1\left(\sum_i x_i \geq \theta\right)$$

- Inputs can be excitatory or inhibitory
- when inh. input is 1 O/P → 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

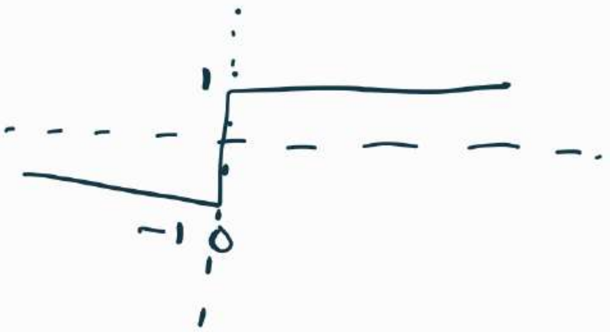
★ Perceptron.

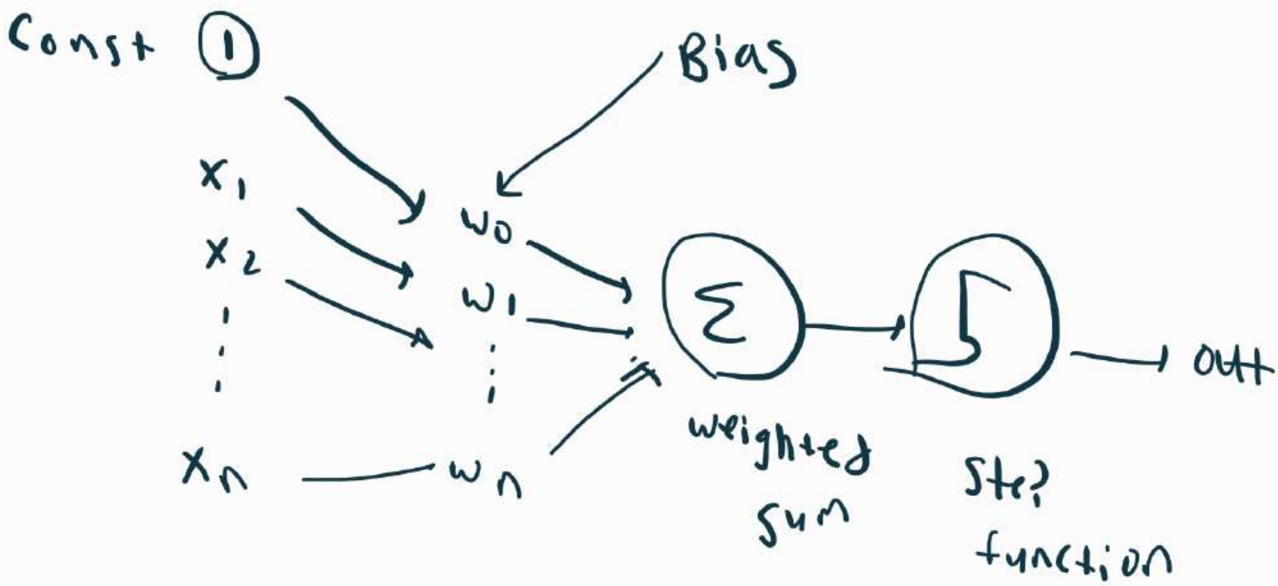
Inputs can real, weights can be diff. for diff. i/p components.

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

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

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





* Perceptron learning

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

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

while $\exists i \in \{1, 2 \cdots N\}$ 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 used for ML learning?

* Polynomial exp.

Bias-variance decomp.

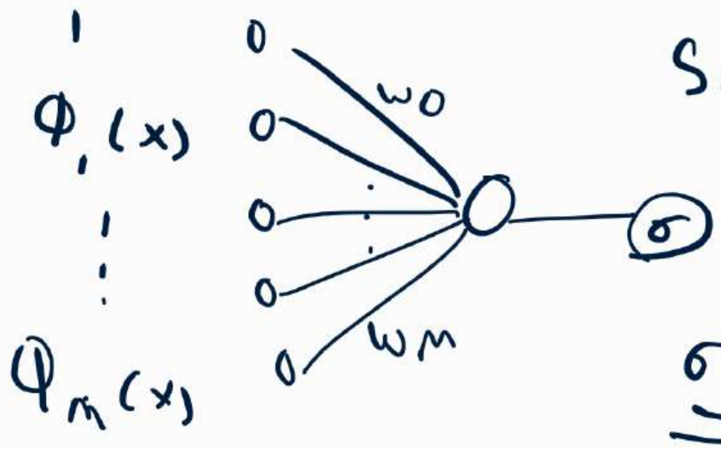
pre processing. (feature extraction)

Regularization

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

$\uparrow$

regression.



Single neuron can do logistic reg.

$$\sigma(w(x))$$

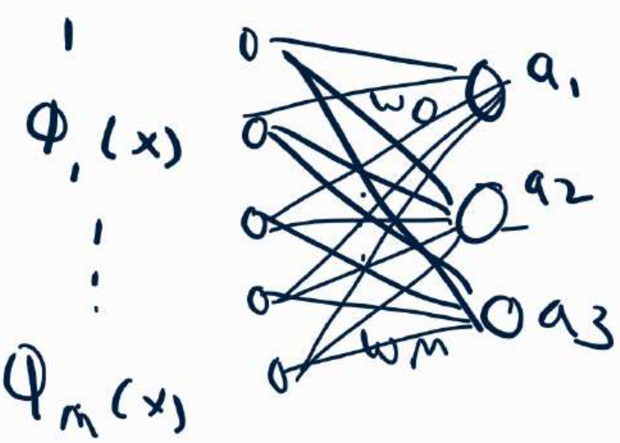
$P(C_k|x) \rightarrow$ Disc. function

$P(X|C_k) \rightarrow$ gen. function

Logistic reg $\rightarrow$ $\sigma(W^T x)$

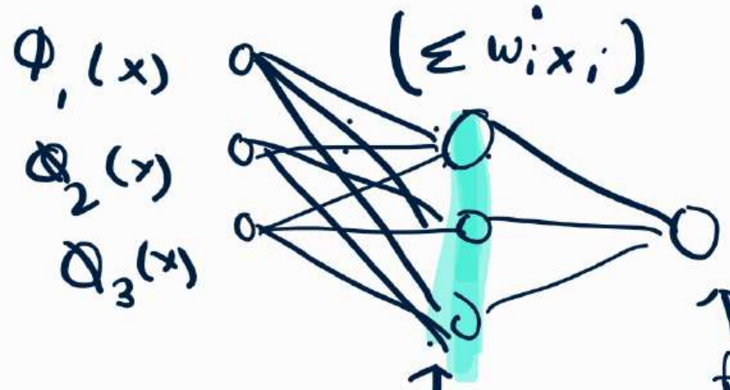
(result is $-\infty \rightarrow \infty$)

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



+ Feed Forward NN

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



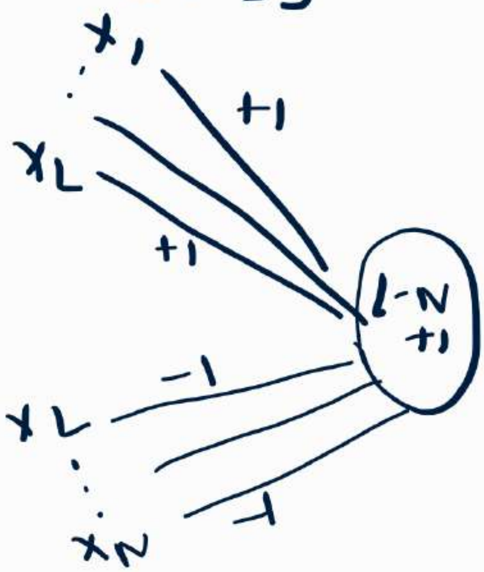
for this i/p is not adaptive as input is fixed

for this i/p is adaptive as we can change weights from 1st layer

* 9/1/25

[L-N 0]
[0 L]

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



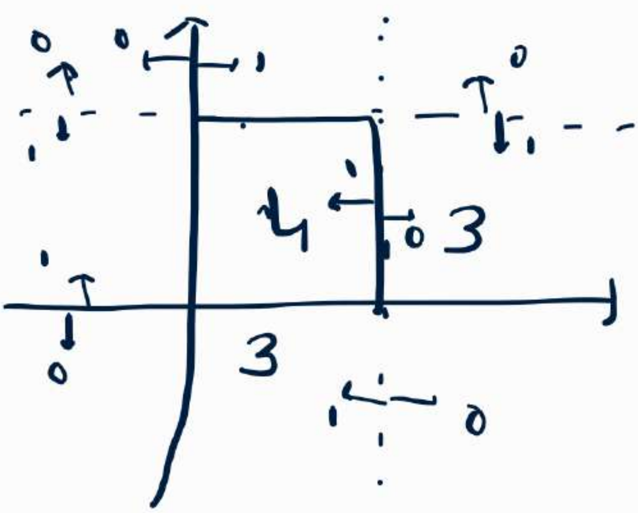
$f(A,B,C,D) = f\,A\bar{B}\bar{C}D + \ldots$

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

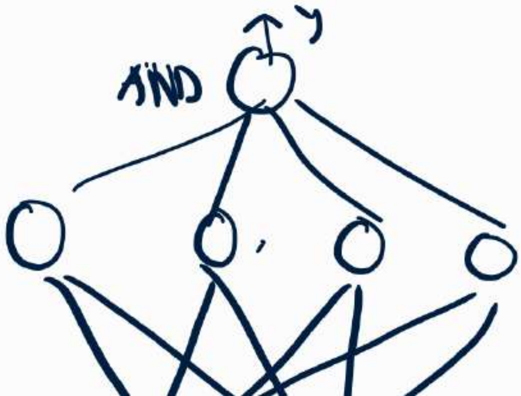
4 vari → 8

n → 2^n

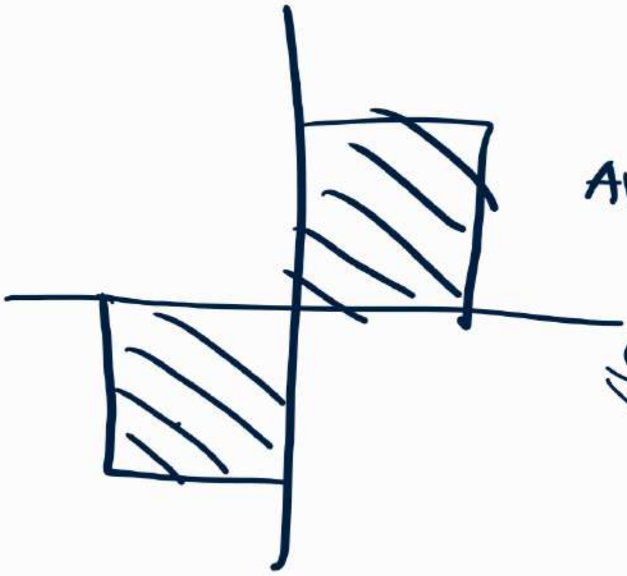
1		1	
	1		1
1		1	
	1		1

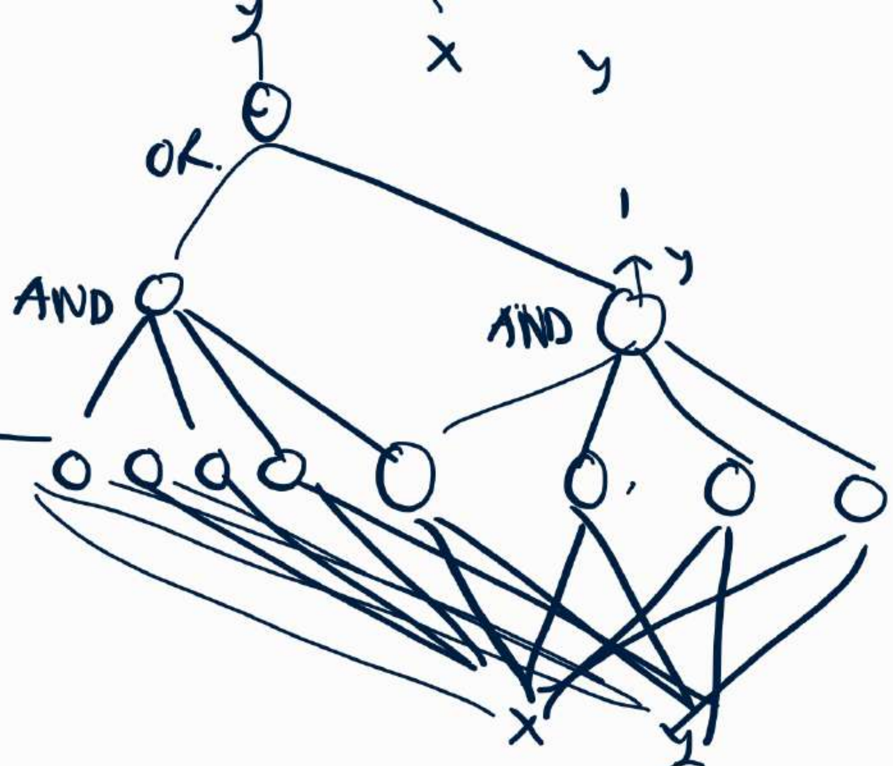


4
make perceptron which align with 4 lines & fire when it is 1



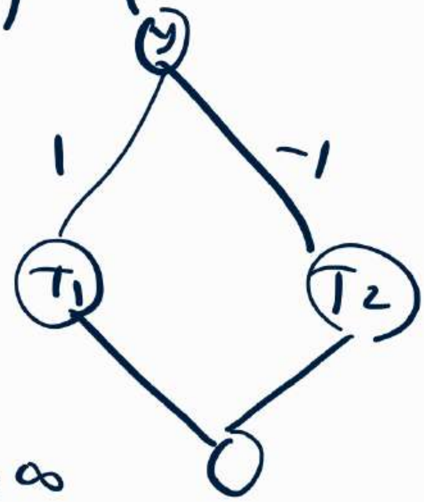
* Universal Approx Thm ?

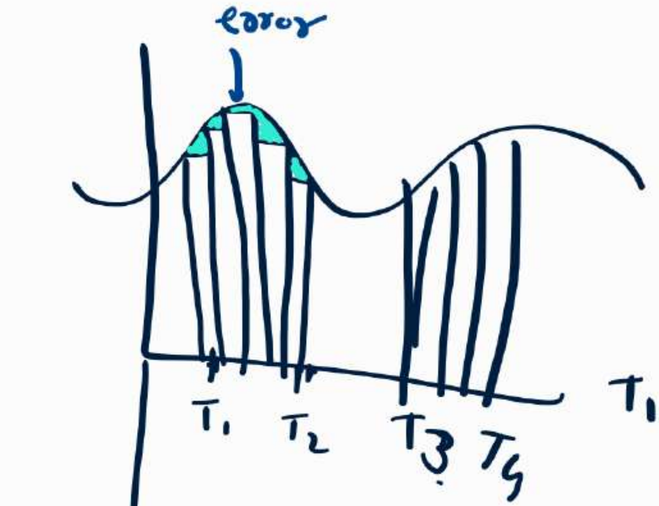




A Regression.

$(0, T_1, 1)\ T_1 - T_2$

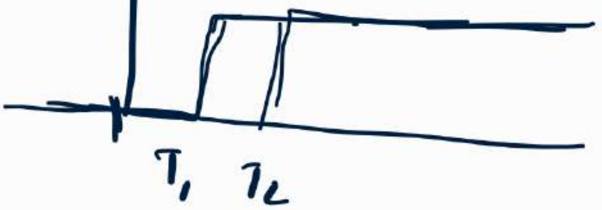


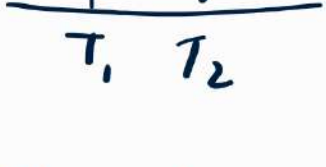


$T_1 \to -\infty \to \infty$
$\uparrow$
b

1 & -1 done

$T_1 - T_2$





T_1 fires at thr. T_1

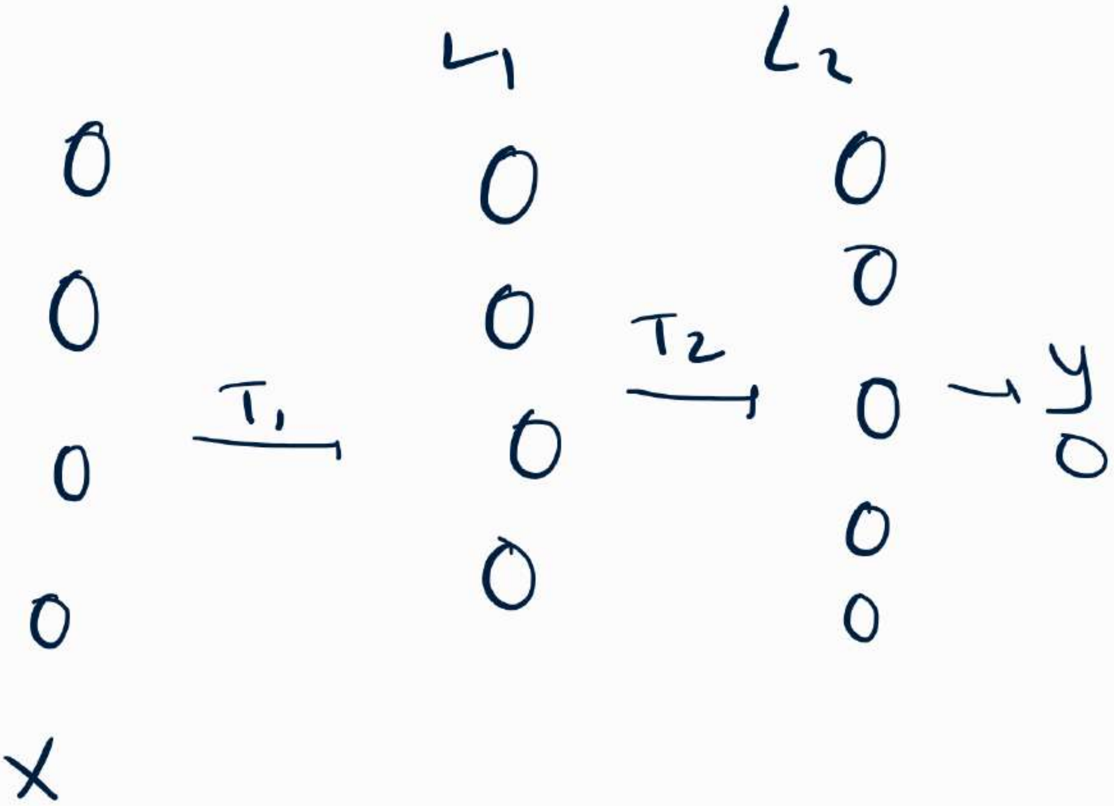
We can choose interval to reduce errors.

* HW no. of neurons for XOR

*

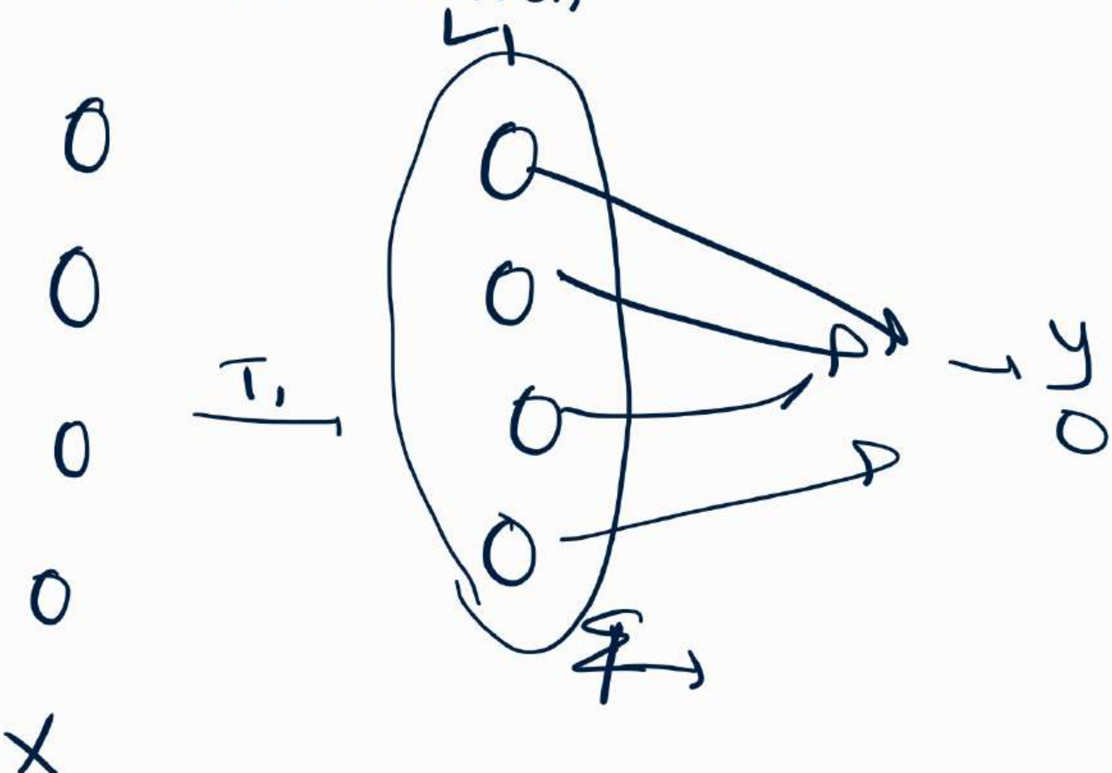
$$(\bar{b} + W\bar{x})$$

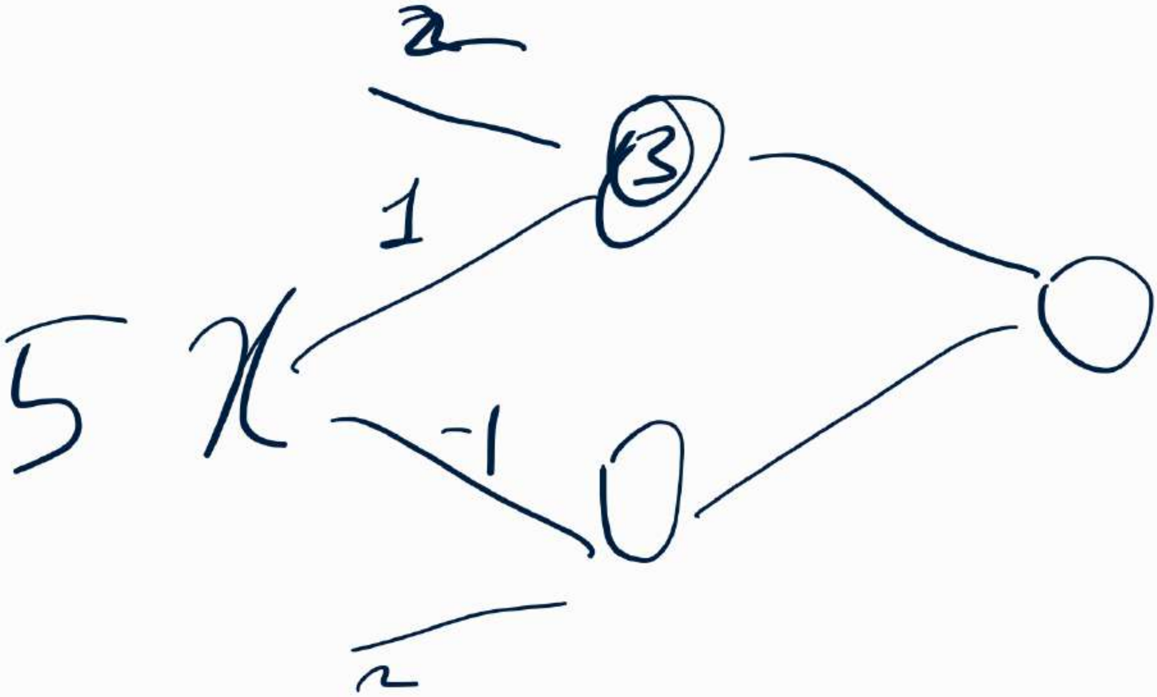
$\bar{b}$: $r \times 1$, W: $r \times h$, $\bar{x}$: $h \times 1$

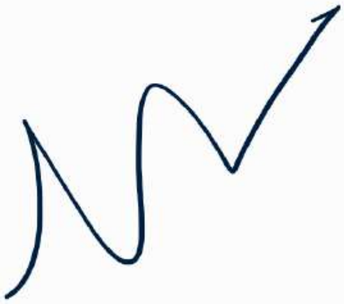


Anything beyond 2 Layers in Deep NN.

* Universal Approximation




5
x
2
1
3
-1



* 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 $R(f) = E_z(l(f,z))$

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

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

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

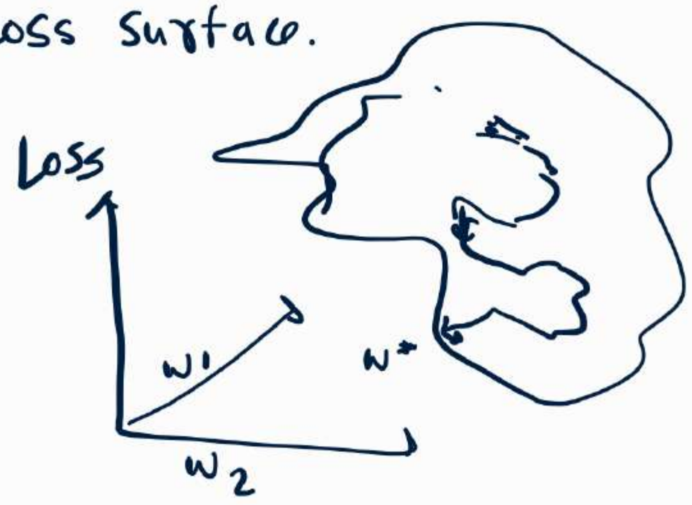
$$w^* b^* = \underset{w,b}{\operatorname{argmin}} \; l(f(w,b), D)$$

How to get optimal parameters.

linear regression.

Adhoc (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.

Derivate of fⁿ at point gives rate of change of function at that point.

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

$$\Delta f = \frac{\delta f}{\delta x} \Delta x.$$

$$f : R^D \to R.$$

$$\nabla f = R^D \to R^D \qquad x \to \left(\frac{\delta f}{\delta x_1} \cdots \frac{\delta f}{\delta x_D} \right)$$

* ∇f vector gives dir^n & rate of fastest increase for f.

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

$$L(w+u) = L(w) + \nabla_w L(w) \cdot u + \frac{1}{2!} u^T \nabla^2 L(w) u \ldots$$

$$= L(w) + \nabla_w L(w) \cdot u.$$

for $L(w+u)$ lesser than $L(w)$, need $\nabla_w L(w) \cdot u$ < 0

Goal is minimize loss.

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

* Batch Gr. descent

for i in range (nb-epoch)
  ∇L_w = evaluate (grad.(L, D, v)
  $w = w - \eta * \nabla L_v$.

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

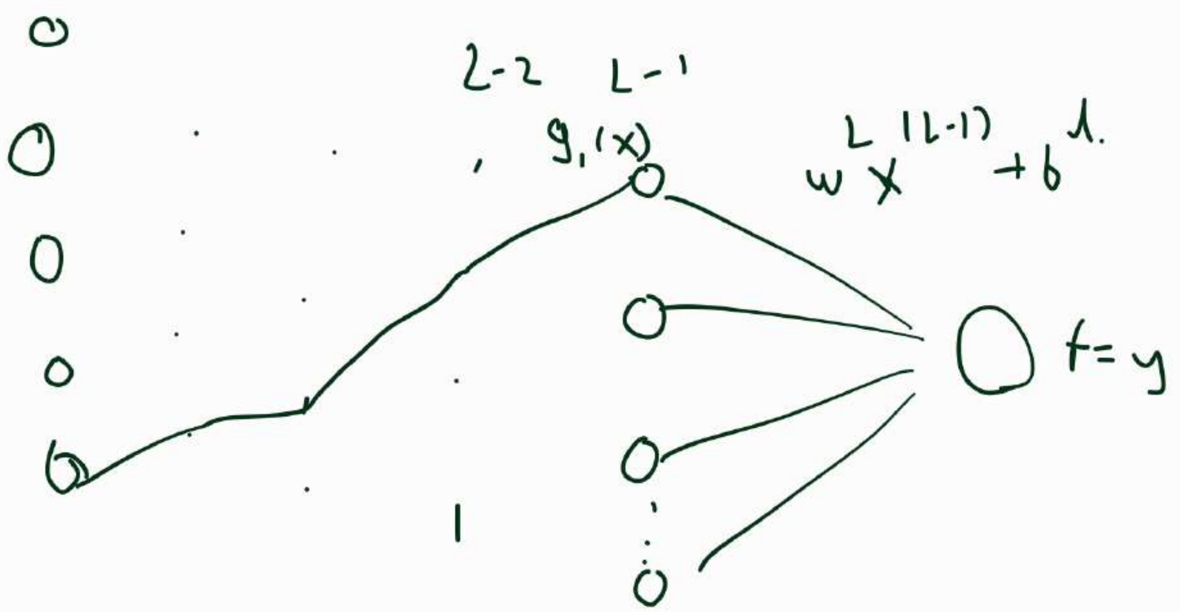
* Scohastic gradient descent.

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

* mini batch

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

* Backpropogation



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

$$\frac{\partial y}{\partial x} = \frac{df}{dg_1(x)} \frac{\partial g_1(x)}{dx} \cdots \frac{df}{dg_m(x)} \frac{dg_m(x)}{dx}.$$

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

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

22/11/25

$$\frac{\delta l}{\delta s_i^{(l)}} = \frac{\delta x^{(l)}}{\delta s^{(l)}} \cdot \frac{\delta L}{\delta x^{(l)}}$$

$$x_i = \sigma(s_i)$$

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

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

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

$$x_i = \frac{e^{s_i}}{\sum_{i}^{N} e^{s_i}}$$

$$\frac{\delta x_i}{\delta s_j} = 0$$ for hidden layer

$i \neq j$

For softmax o/p layer this not valid.

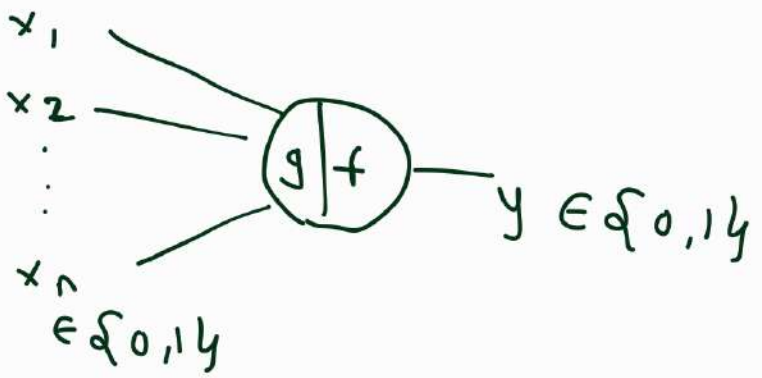
* NOTES FOR EXAM (24/11/25)

The Neuron.

Threshold Logic unit → 1st Math. model for neuron.

MP neuron.

Boolean i/p.



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

I/p can be excitatory & Inhibitory

↑ more likely fire

↑ Reduce Likelihood.

- when Inhibitory i/p set (1) then o/p → 0
- count no. of ON signals on exit. i/p vs Inhib.

* xy'

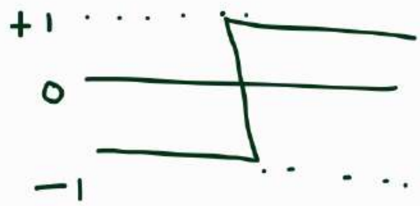
* Learn linear separation.
- No learning.

* Perceptron.

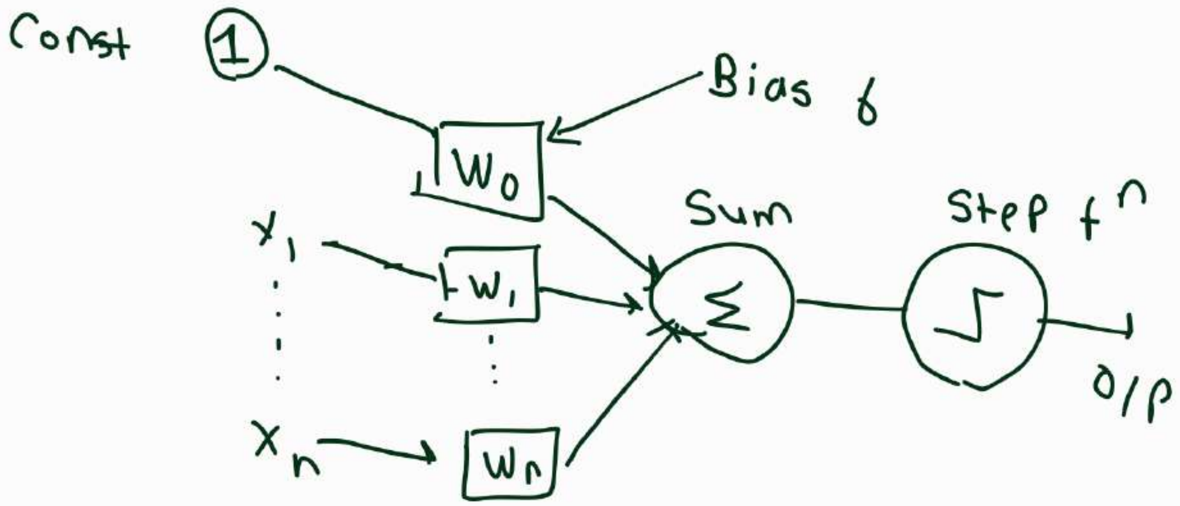
- Same like MP.
  - Input can be real,
  - weight can be diff. for diff. I/P.

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

- 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 $\exists\, i \in \{1 \cdots N\}$ 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 I/P for Perceptron.
extra Dim for Bias.

Even row D_1 } Assign.
Odd row D_2

y empty → labels.
even → 1.
odd → -1.

```
W = torch.zeros(X.size(1)) . --initialize w by 0.
def trainperceptron(x, y, w, epoch)
  for e in range(epoch)
    change =0
    for (i = 0 → x.size(0))
      if X[i].dot(w) * y[i] <= 0 ∵ wx+b<0
        w = w + y[i] * X[i]                  w = w + xy.
        change = change +1
    if change = = 0
      break # early stop.
  Print(no. of changes . ,change).
  return w.

w = train_perceptron(x, y, w, 5)
Print(w)
```

* DL-2.

Linear classifiers.

Sometime data is linearly seperable
Some time not
Some time specific preprocessing can make it seperable.

In XOR.

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

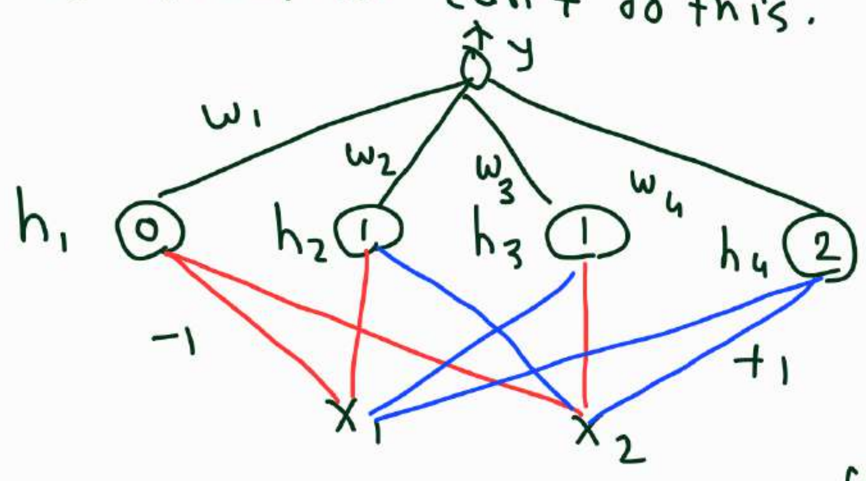
- make new dimension then you found the seperable plane.
- Increasing dimension (degree) increase capacity.
- Reduce Bias increase capacity.
- preprocessing is also may way to reduce capacity.

XOR:.

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

$w_0 < 0$
$w_2 + w_0 \geq 0 = w_2 \geq -w_0$
$w_1 + w_0 \geq 0 = w_1 \geq -w_0$
$w_1 + w_2 + w_0 < 0 = w_1 + w_2 < -w_0$

Single perceptron can't do this.



w_1, w_2 w_3 w_4

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

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

$(0,1) = -1,\ 1,\ -1,\ 1$

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

$w_1 < -w_0 \qquad w_2 \geq -w_0 \qquad w_3 \geq -w_0$

$w_4 < -w_0$

Possible Find this weights

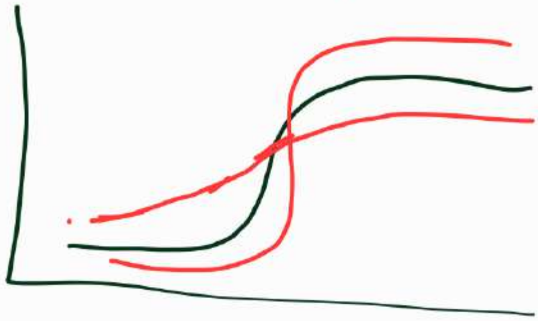
* other 2 i/p boolean f^n.

- 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.

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



* Any Boolean function with n i/p represent with one hidden layer.

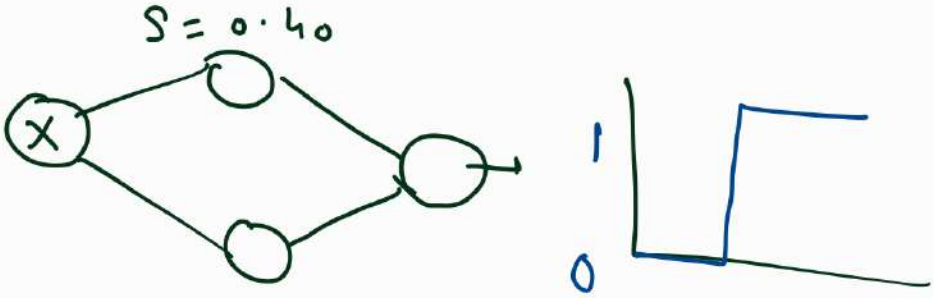
- We can represent any continuous f^n to any approximation with linear comb. of sigmoid neurons.
- Neural n/w with single hidden layer used to approx any cont. f^n to any desired precision.
- Using K map we can have 2^{N-1} perc. in hidden layer.
- $3(N-1)$ required in deep N/w. Linear in N. arranged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

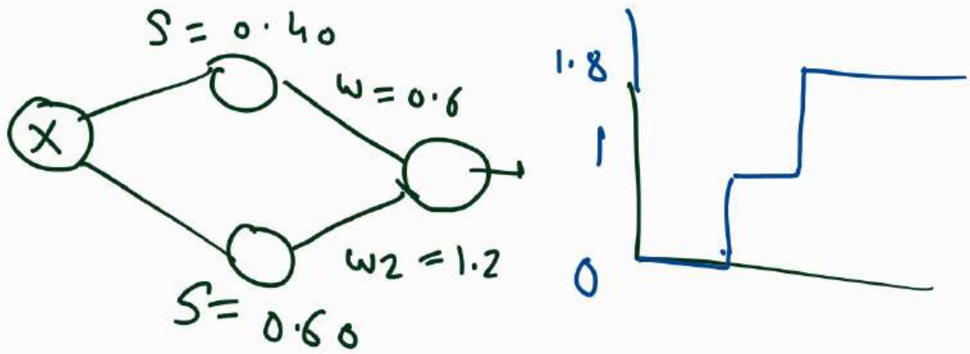
* DL-03

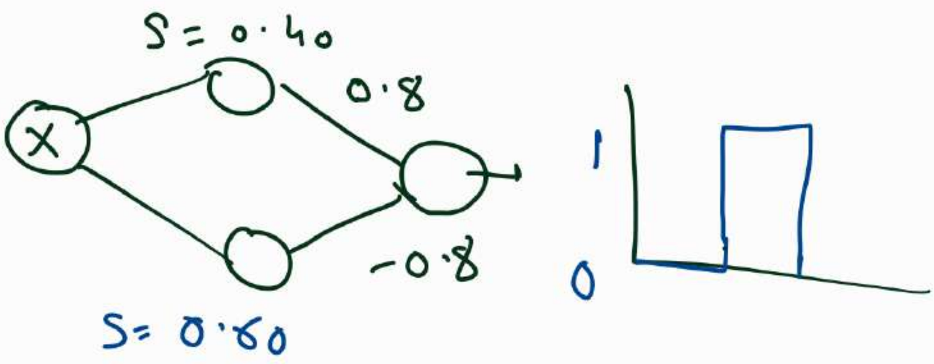
* Uniform Approximation Th^m.

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

- Sigmoid neuron is taken i/p & process & produce o/p like 'S' curve.
- NN with one hidden layer can fit any continuous function.



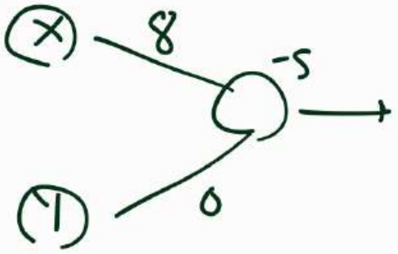


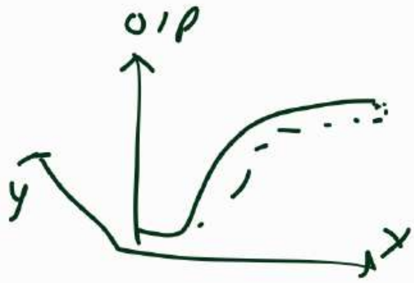


We only computed weighted sum of hidden outputs.

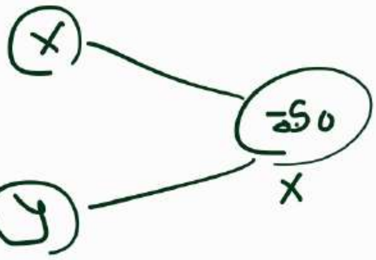
* when multiple inputs are introduced

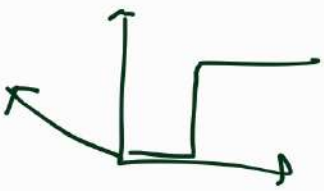
when 2 i/p we get 3 D plane.

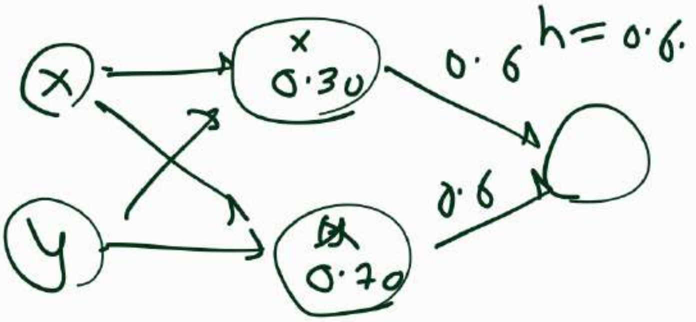


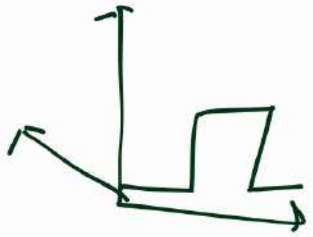


$$S = \frac{-b}{w}$$

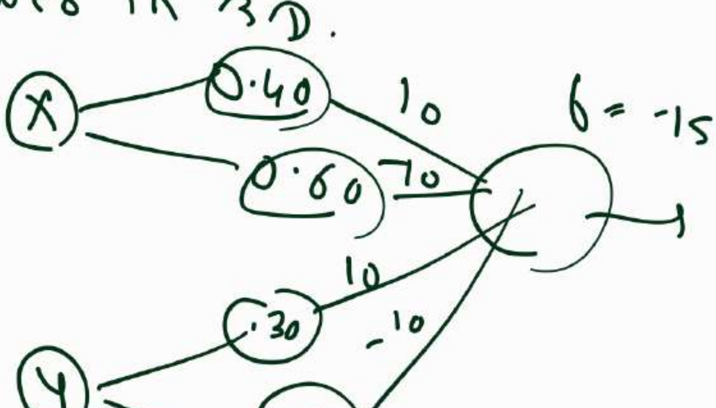


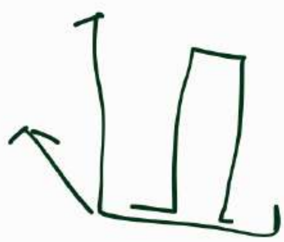






* Tower in 3D.





Only fire when above 15

0.40 → 0.60

* Target f^n lie on space other than hypercube.
  Discontinued 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 infeasible size for hidden layer.
  - May not generalize well.
  - Not enable hierarchical learning.

* DL - 4

Learning → Finding f^+ from F.
- For good function we use Loss.
- $l : F \times Z \to R$.
- Such that value $l(f, z)$ increases with wrong of f on z. (diff. bet^n 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 $R(f) = E_z(l(f, z))$
- $f^* = \underset{f \in F}{\operatorname{argmin}} R(f)$

- Emperical 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.

$$w^* = \underset{w}{\operatorname{argmin}} L(w)$$

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

- How to find optimal parameters.
    - close from 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 stiffest dir^n, make brief progress
- Repeat.
- GRADIENT DESCENT.

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

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

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

in higher dimensions

$$\nabla f : R^D \to R^D$$

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

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

$$L(w+u) = L(w) + \nabla_w L(w) \cdot u$$

To minimize loss ($L(w+u) < L(w)$)

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

- diff. would be less if u is in opp. dir^n of $\nabla_w L(w)$, the gradient.

$L(w) \rightarrow$ loss at w

$\nabla_w L(w) \rightarrow$ gradient of loss at w.

$u \rightarrow$ step

$\nabla^2 L(w) \rightarrow$ Hessian Matrix (2^{nd} dev.)

* GRADIENT DESCENT

- Goal to minimize the error.
- Determine parameter w that minimize $L(w)$
- 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 dir^n that produce steepest descent along with error

* Numerically for each w.

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

Slow & approximate.

* Analytical methods.

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

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

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

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

$$\nabla L_{iw} = \nabla \text{Hingeloss} + \nabla \text{Regularization.}$$

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

* Batch Gradient descent.

for i in range (nb_epoch):

∇L_w = evaluate_gradient (L, D, w)

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

- In terms of convex function you guaranteed get global minima.

* Storchastic Gr. De.

weight update parameter for each training exampl

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

- In case of large DS Batch GD compute redundent gradients for similar example.
- SGD does away with redunten(y & faster can be used to learn online.
- frequent updates with high variance cause objective f^n to fluctuate heavily.
- SGD fluctuation enable it to jump new & better local minina.
- complicate convergenco.

```
for i in range(ep_epoch)
    np. random .shuffle(D)
    for xi ∈ D
        ∇Lw = evaluate gradient (L, xi, w)
        w = w - η * ∇Lw.
```

* MiniBatch 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 of param. updates

used highly optimized matrix optimization.

Batch size 32 → 1024.

```
for i in range (nb epoch)
    np.random.shuffle(D)
    for batch in get bathes (D, batchsize=128)
        ∇w = evaluate gradient (L, batch, w)
        w = w - η * ∇w.
```

* Some challanges

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

* DL-5

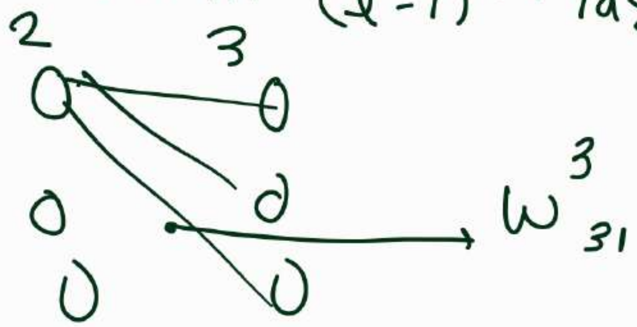
Gradient of scaler f^n $f(x) = x \rightarrow \left\{ \frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial y_n} \right\}^T$.

Gradient of vector valued f^n called Jac.

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

w^l_{jk} :

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



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

x^l_j o/p of j^{th} neuron in l^{th} layer.

$$x^l_j = \sigma\left(\sum_k w^l_{jk} x^{l-1}_k + b^l_j \right)$$

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

$W^l \rightarrow$ weight vector of layor 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 f^n.

* Loss $\ell(W,b) = \sum_n \ell(f(x_n : W, b), y_n) =$

$$= \sum_n \ell(x^L, y_n).$$

$$x^0 = x \xrightarrow{W(1),\ b^{(1)}} s^{(1)} \xrightarrow{\sigma} x^{(1)} \xrightarrow{W^{(2)}\ (b)^2} s^{(2)} \longrightarrow \cdots x^L.$$

$$x^0 = x,\ f(x : W, b) = x^{(L)}.$$

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

* Chain rule.

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

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

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

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

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

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

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

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

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

$$\frac{dy}{dx} = \frac{df}{dg_1(x)} \frac{dg_1(x)}{dx} + - + \frac{df}{dg_m(x)} \frac{d\,g_m(x)}{dx}.$$

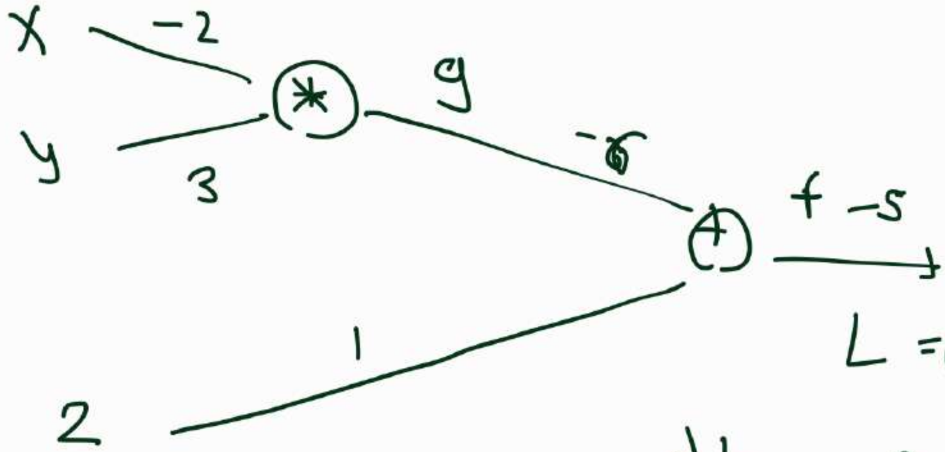
$$g_i(x) = z_i \rightarrow y = f(z_1 \; --- \; z_m)$$

$$\Delta y = \frac{df}{dz_1} \Delta z_1 + \frac{df}{dz_2} \Delta z_2 \; --- \; \frac{df}{dz_m} \Delta z_m$$

$$\Delta y = \frac{df}{dz_1} \frac{dz_1}{dx} \Delta x + \frac{df}{dz_2} \frac{dz_2}{dx} \Delta x \; ---$$

$$\Delta y = \frac{df}{dg_1(x)} \frac{d\,g_1(x)}{dx} \Delta x \; - - \; -$$

$$\Delta y = \left(\frac{df}{dg_1(x)} \frac{dg_1(x)}{dx} \; ---- \right) \Delta x.$$



$g = xy$

$f = g + z.$

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

$L = (f-t)^2. \quad \left(t = -4 \text{ for this} \right)$

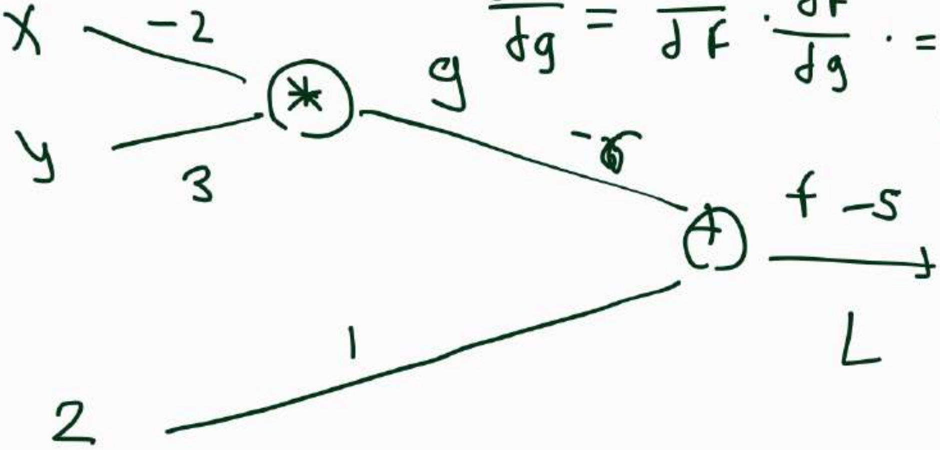
$$\frac{dL}{df} = 2(f-t)$$

$$= -2$$

$$\frac{dL}{dg} = \frac{dL}{df} \cdot \frac{df}{dg} = (-2) * 1$$

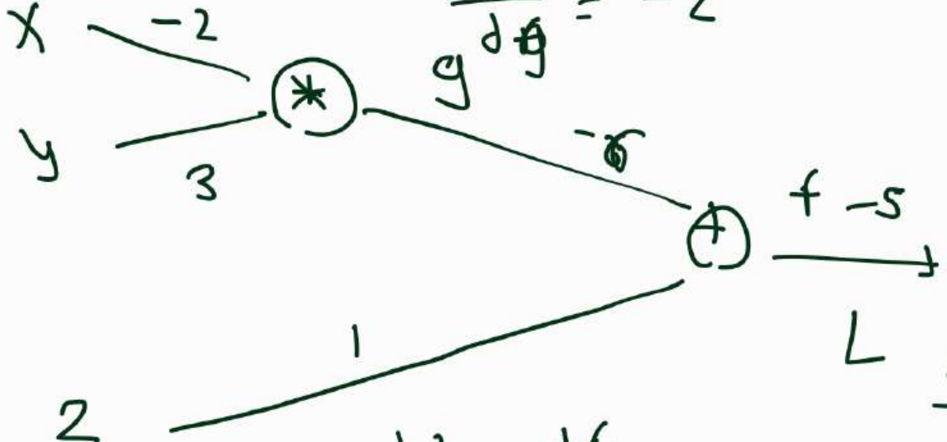
$$= -2$$

$f = g + z.$



$$\frac{dL}{df} = -2$$

$$\frac{dL}{dg} = -2$$



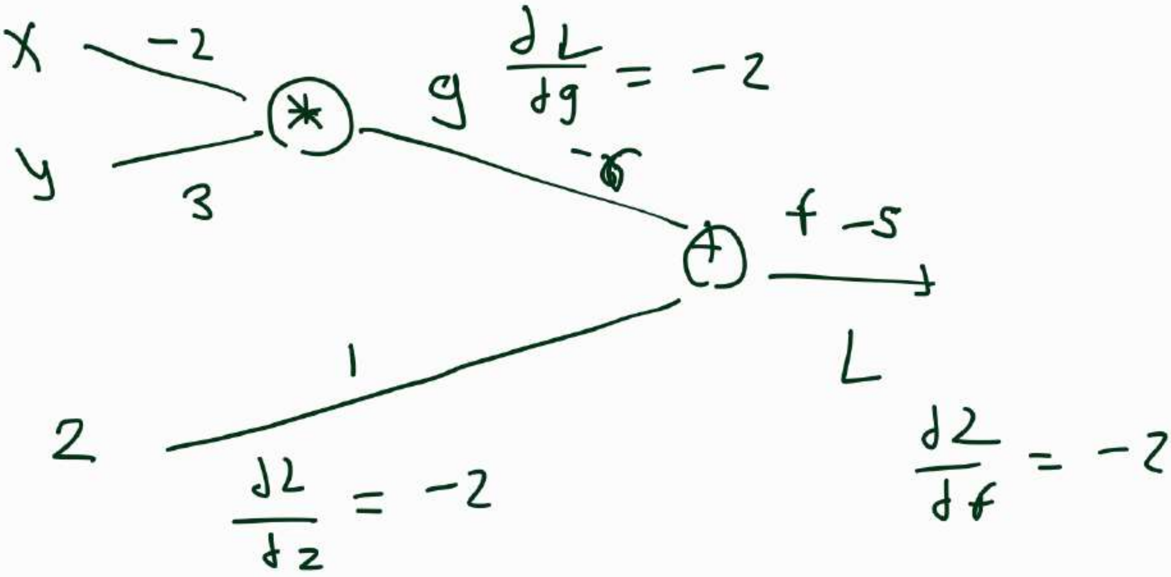
$$\frac{dL}{df} = -2$$

$$\frac{dL}{dz} = \frac{dL}{df} \cdot \frac{df}{dz}.$$

$f = g + z$

$$= (-2) * 1$$

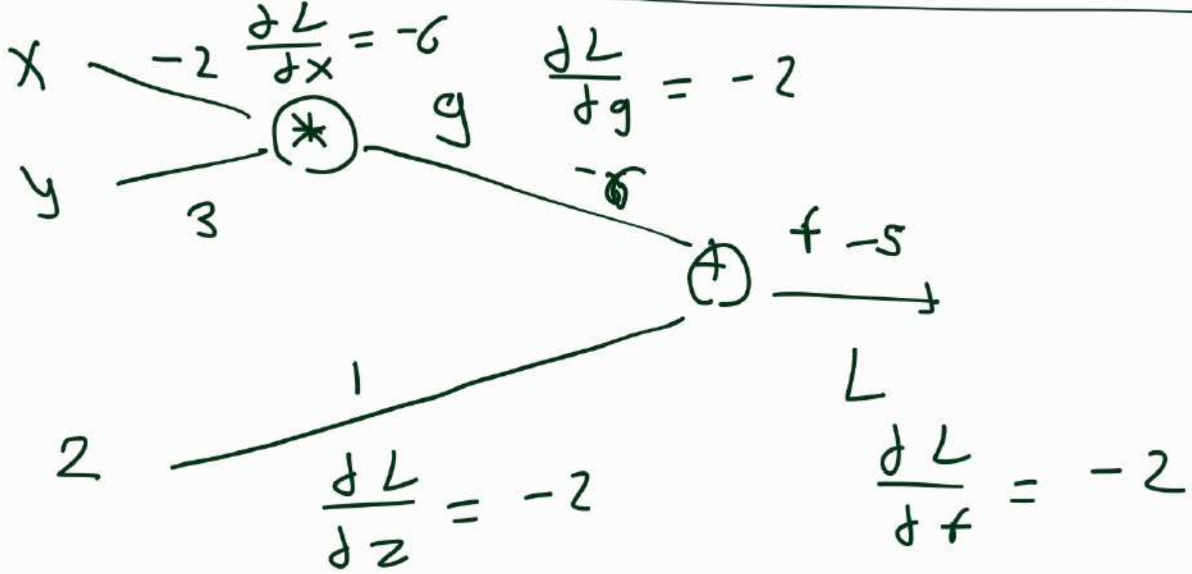
$$= \underline{\underline{-2}}$$



$$\frac{dL}{dx} = \frac{dL}{dg} \cdot \frac{dg}{dx} \qquad g = xy$$

$$= (-2) * y$$

$$= (-6) \quad -(y = 3)$$



$$\frac{dL}{dy} = \frac{dL}{dg} \cdot \frac{dg}{dy}$$

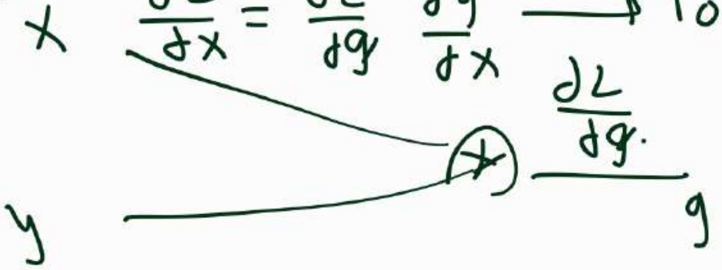
$$= (-2) * x$$

$$= \underline{\underline{4}}$$

* Gradient Flow

Down stream grad. → $\frac{dL}{dx} = \frac{dL}{dg} \frac{dg}{dx}$ → local ↳ we calculate locally

upstream → as we take from forward tensor



$$J_{f_N \circ f_{N-1} \circ \cdots f_1}(x) = J_{f_N}(f_{N-1}(\cdots f_1(x))) \cdot J_{f_{(N-1)}}(f_{N-2} \cdots f_1(x)) \cdots J_{f_1}(x).$$

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

* Backpropagation.

Take any layer.

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

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

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

$$\frac{\delta l}{\delta s_i^{(l)}} = \frac{\delta l}{\delta x_i^{(l)}} \cdot \frac{\delta x_i^{(l)}}{\delta s_i^{(l)}}$$

$$= \frac{\delta l}{\delta x_i^{(l)}} \sigma'(s_i^l)$$

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

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

$$\frac{\delta l}{\delta x_j^{(l-1)}} = \Sigma_i \frac{\delta l}{\delta s_i^{(l)}} \cdot \frac{\delta s_i^{(l)}}{\delta x_j^{(l-1)}}$$

$$= \Sigma_i \frac{\delta l}{\delta s_i^{(l)}} \cdot W_{ij}^{(l)}$$ -- derv. of $(wx+b)$ is w.

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

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

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

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

* From defination of loss obtain $\frac{\delta l}{\delta x_i^{(l)}}$

Recursively compute loss dev. w r t activations.

$$\left.\begin{aligned} \frac{\delta l}{\delta s_i^{(l)}} &= \frac{\delta l}{\delta x_i^l} \sigma'(s_i^l) \\ \frac{\delta l}{\delta x_j^{(l-1)}} &= \sum_i \frac{\delta l}{\delta s_i^l} \cdot w_{ij}^l \end{aligned}\right\} \text{Gradient}$$

then wrt parameters.

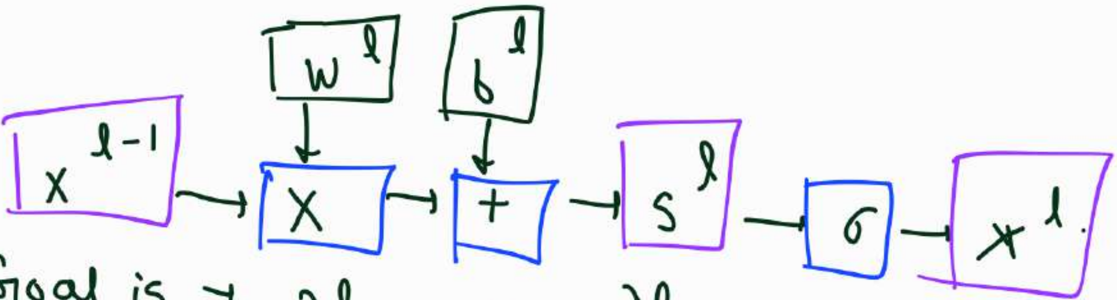
$$\frac{\delta l}{\delta w_{ij}^l} = \frac{\delta l}{\delta s_i^l} \cdot x_j^{(l-1)} \qquad \left.\frac{\delta l}{\delta b_i^l} = \frac{\delta l}{\delta s_i^l}\right\} \text{Param.}$$

* Jacobian in Tensorial Form.

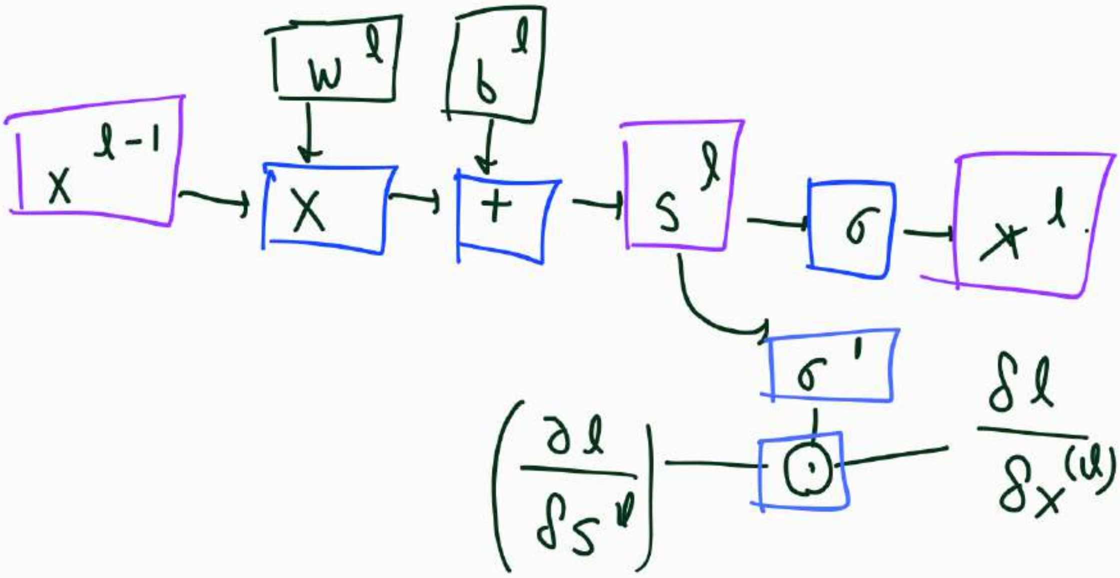
$$\psi : R^N \to R^M \text{ then } \left[\frac{\partial \psi}{\partial x}\right] = \begin{bmatrix} \frac{d\psi_1}{dx_1} & \cdots & \frac{d\psi_1}{dx_N} \\ \vdots & \ddots & \vdots \\ \frac{d\psi_M}{dx_1} & \cdots & \frac{d\psi_M}{dx_N} \end{bmatrix}$$

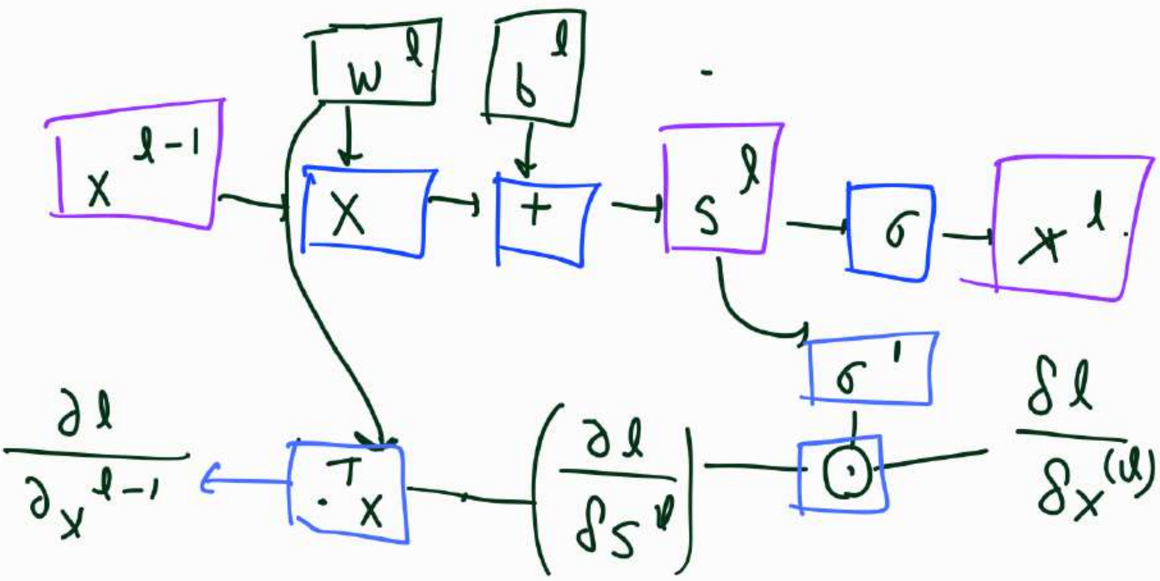
$$\psi \; R^{N\times M} \to R \text{ then } \left[\left[\frac{d\psi}{dx}\right]\right] = \begin{bmatrix} \frac{d\psi}{dw_{11}} & \cdots & \frac{d\psi}{dw_{1M}} \\ \vdots & \ddots & \vdots \\ \frac{d\psi}{dw_{N1}} & \cdots & \frac{d\psi}{dw_{NM}} \end{bmatrix}$$

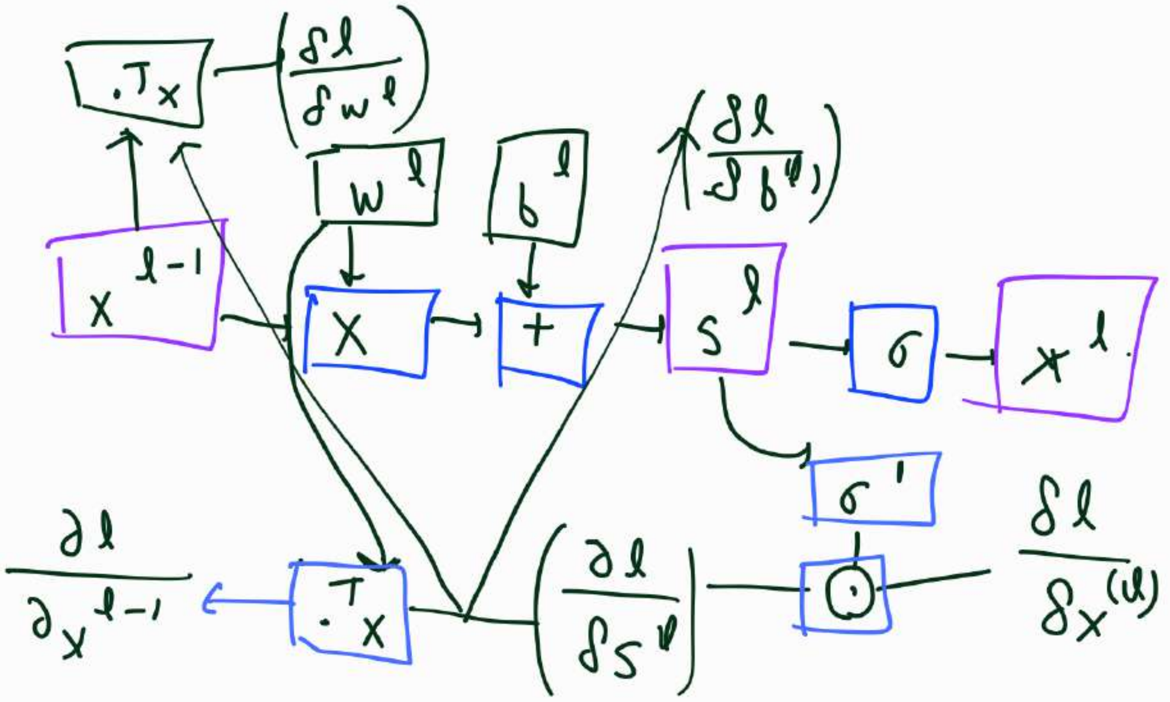
* Forward pass



Goal is → $\frac{\partial l}{\partial w^l}$ & $\frac{\partial l}{\partial b^l}$.





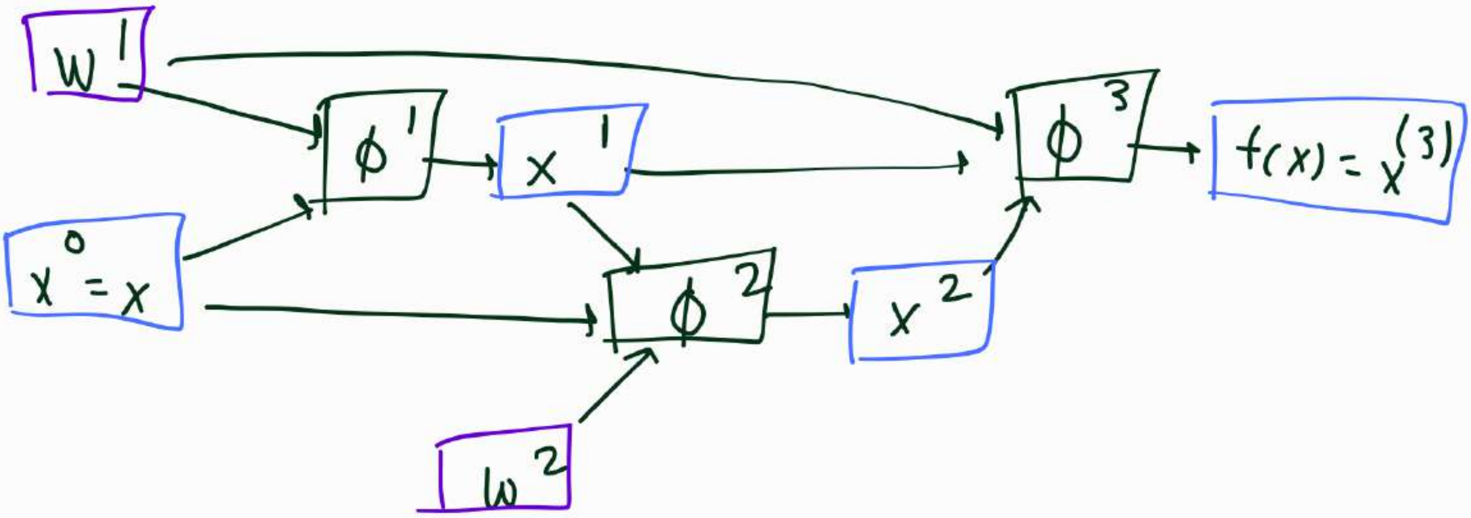


update params

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

BP is applying chain rule iteratively.
expressed in tensorial form
Heavy computation with linear ops.
Non-lin. goes with simple element wise ops.
BP needs all int. layer result to be in memory
Take twice the computations of FW pass.

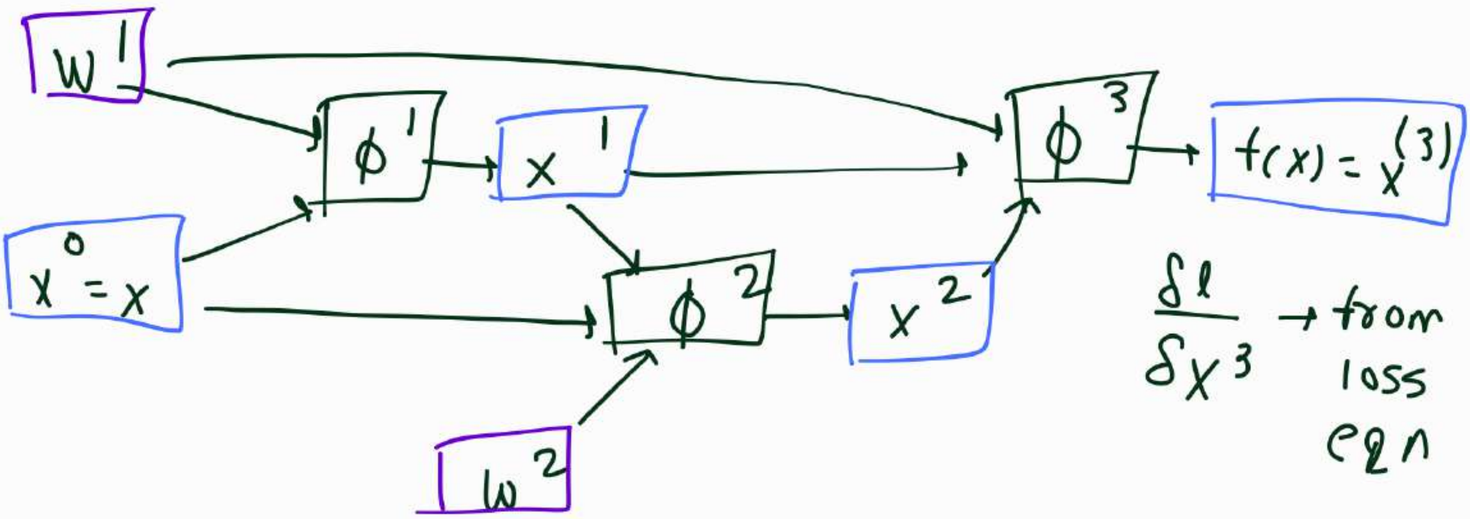
* For an Example.



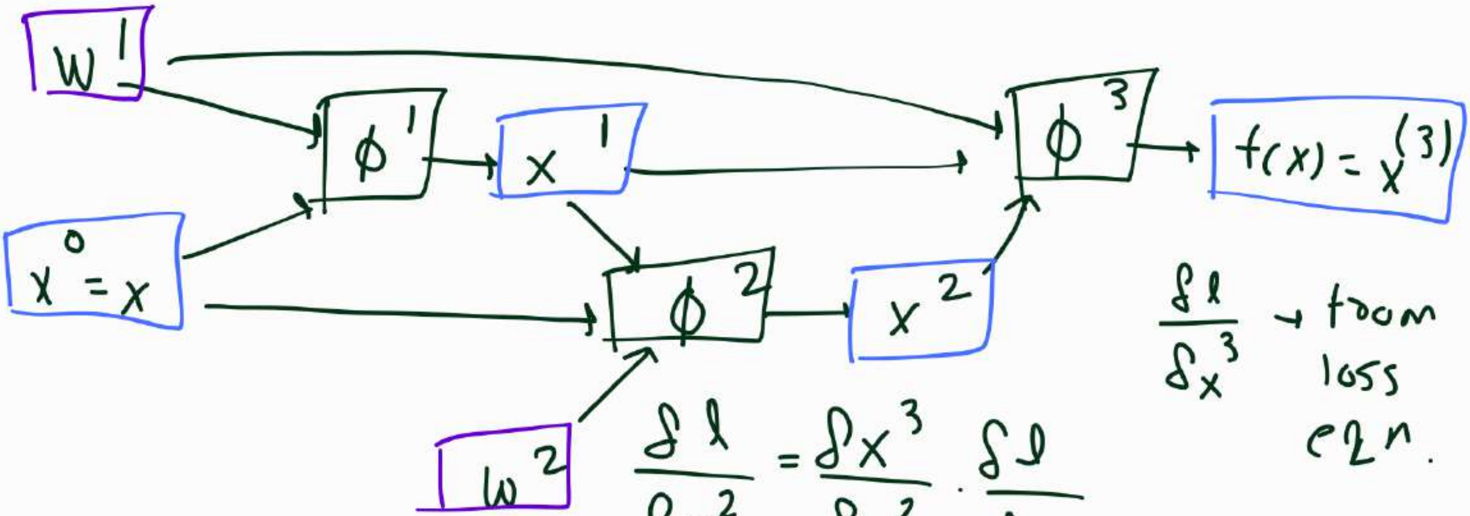
Forward pass

$$x^0 = x$$
$$x^1 = \phi^1(x^0 \cdot w^1)$$
$$x^2 = \phi^2(x^0, x^1, w^2)$$
$$x^3 = \phi^3(x^1, x^2, w^1)$$

* Back propogation.



$\frac{\delta l}{\delta x^3}$ → from loss eqn



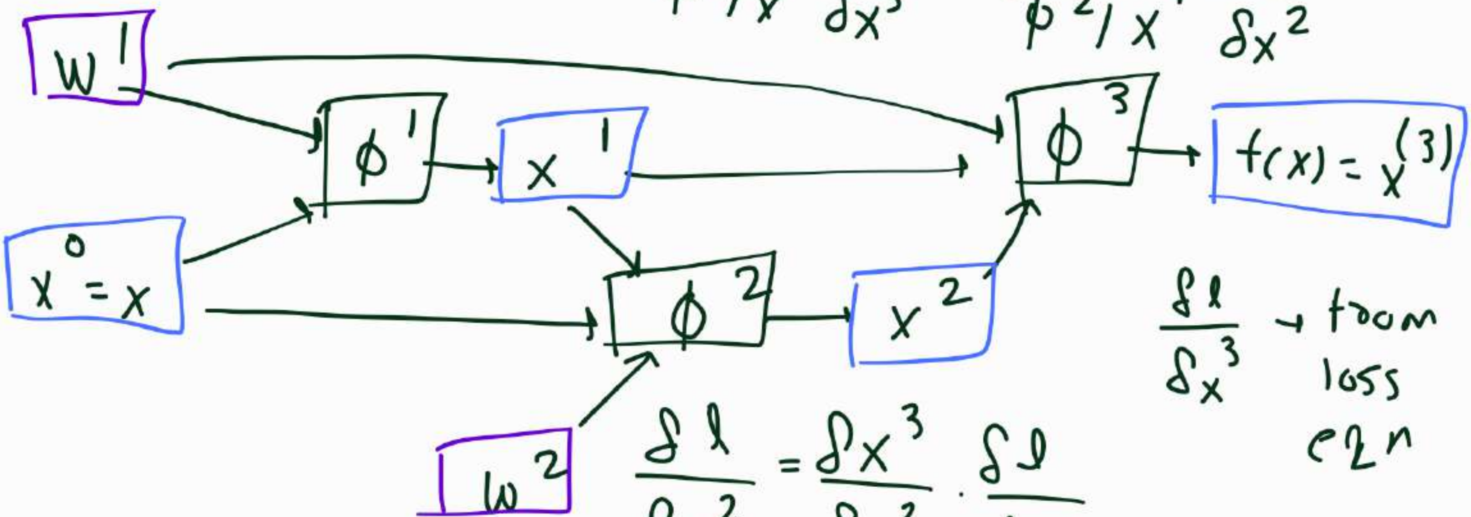
$\frac{\delta l}{\delta x^3}$ → from loss eqn.

$$\frac{\delta l}{\delta x^2} = \frac{\delta x^3}{\delta x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$= J^T_{\phi^3/x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$\frac{\delta l}{\delta x^1} = \frac{\delta x^3}{\delta x^1} \cdot \frac{\delta l}{\delta x^3} + \frac{\delta x^2}{\delta x^1} \cdot \frac{\delta l}{\delta x^2}$$

$$= J^T_{\phi^3/x^1} \frac{dl}{dx^3} + J^T_{\phi^2/x^1} \cdot \frac{\delta l}{\delta x^2}$$



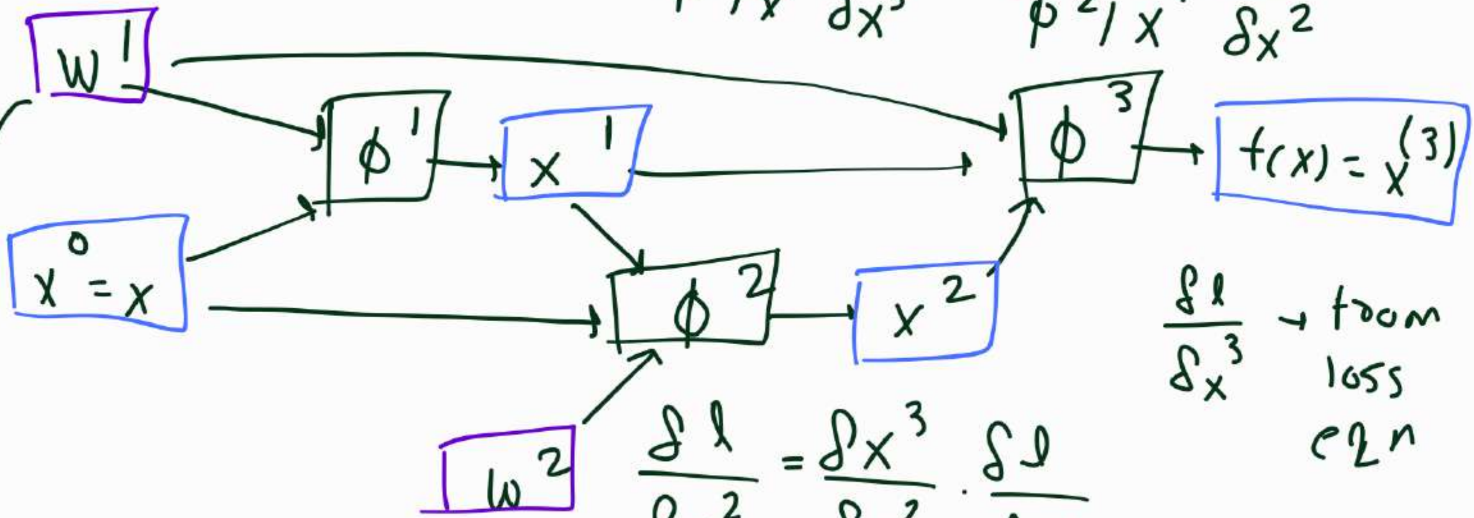
$\frac{\delta l}{\delta x^3}$ → from loss eqn

$$\frac{\delta l}{\delta x^2} = \frac{\delta x^3}{\delta x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$= J^T_{\phi^3/x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$\frac{\delta l}{\delta x^1} = \frac{\delta x^3}{\delta x^1} \cdot \frac{\delta l}{\delta x^3} + \frac{\delta x^2}{\delta x^1} \cdot \frac{\delta l}{\delta x^2}$$

$$= J^T_{\phi^3/x^1} \frac{d l}{d x^3} + J^T_{\phi^2/x^1} \frac{\delta l}{\delta x^2}$$



$\frac{\delta l}{\delta x^3}$ → from loss eqn

$$\frac{\delta l}{\delta x^2} = \frac{\delta x^3}{\delta x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$= J^T_{\phi^3/x^2} \cdot \frac{\delta l}{\delta x^3}$$

$$\frac{\delta l}{\delta w^1} = \frac{\delta x^3}{\delta w^1} \cdot \frac{\delta l}{\delta x^3} + \frac{\delta x^1}{\delta w^1} \cdot \frac{\delta l}{\delta x^1}$$

$$= J^T_{\phi^3/w^1} \left(\frac{\delta l}{\delta x^3} \right) + J^T_{\phi^1/w^1} \left(\frac{\delta l}{\delta x^1} \right)$$

$$\frac{\delta l}{\delta w^2} = \frac{\delta x^2}{\delta w^2} \frac{\delta l}{\delta x^2} = J^T_{\phi^2/w^2} \left(\frac{\partial l}{\delta x^2} \right)$$

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.